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

D2.2 –Final report on the characterization and modeling of the reference applications

Due date of deliverable: 31 December 2011
Actual Submission: 31 December 2011

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	**23.11.2011**	**Rosa M. Badia**	**BSC**	**Initial template**
0.2	**28.11.2011**	**Rosa M. Badia**	**BSC**	**Integration of partners input**
0.3	**05.12.2011**	**Rosa M. Badia**	**BSC**	**Further edition of sections**
0.4	**20.12.2011**	**Rosa M. Badia**	**BSC**	**Adding reviews from reviewers and final formatting**

Release Approval

Name	Role	Date
Rosa M. Badia	**Originator**	**20.12.2011**
Nacho Navarro	**WP Leader**	**20.12.2011**
Roberto Giorgi	**Project Coordinator for formal deliverable**	**30.12.2011**

TABLE OF CONTENTS

EXECUTIVE SUMMARY **5**
1 INTRODUCTION **6**
1.1 DOCUMENT STRUCTURE 6
1.2 RELATION TO OTHER DELIVERABLES 6
1.3 ACTIVITIES REFERRED BY THIS DELIVERABLE 6
2 IDENTIFICATION OF SUBSET OF THE REFERENCE APPLICATIONS **8**
3 APPLICATIONS CHARACTERIZATION **11**
3.1 ANALYTIC CHARACTERIZATION (THALES) 11
3.2 CHARACTERIZING THE MEMORY REQUIREMENTS OF MPI APPLICATIONS (BSC) 15
3.3 CPI STACK CHARACTERIZATION OF STARSS APPLICATIONS (BSC) 18
3.3.1 Instrumentation *18*
3.3.2 Clustering and CPI Stack *18*
3.3.3 Discussion of results *19*
3.4 CHARACTERISATION OF STARSS + TM APPLICATIONS (BSC) 26
3.5 TFLUX DATA-FLOW MODEL (UCY) 30
3.5.1 Characterization *30*
4 APPLICATIONS PORTING **34**
4.1 SCALA (UNIMAN) 34
4.2 IMPROVEMENTS OF THE IMPLEMENTATION OF STAP IN STARSS 37
4.2.1 Initial Results *37*
4.2.2 Optimization *39*
4.2.3 Final Results *40*
5 CONCLUSIONS **42**
REFERENCES **43**

LIST OF FIGURES

FIGURE 1 DETAIL OF ONE TASK IN THE APPLICATION GRAPH FOR IMAGE TILES OF 60X60 PIXELS. 42X101 COMPUTATIONS CA BE DONE IN PARALLEL IN ORDER TO USE THE ENTIRE INPUT IMAGE. 13
FIGURE 2 PEDESTRIAN DETECTION MAPPING ONTO A PARALLEL ARCHITECTURE 13
FIGURE 3 PEDESTRIAN DETECTION CONTROL-FLOW GRAPH 14
FIGURE 4 CHARACTERIZATION OF TRANSACTIONS TIME 27
FIGURE 5 CHARACTERIZATION OF COMMIT TIME 29
FIGURE 6 CHARACTERIZATION OF ABORT TIME 29
FIGURE 7 IPC FOR DIFFERENT INPUT DATA SETS FOR (A) DATA-FLOW AND (B) DATA-FLOW+TM APPLICATIONS 31
FIGURE 8 LLC CACHE MISSES RATE FOR DIFFERENT INPUT DATA SETS FOR (A) DATA-FLOW AND (B) DATA-FLOW+TM APPLICATIONS . 32
FIGURE 9 BANDWIDTH FOR DIFFERENT INPUT DATA SETS FOR (A) DATA-FLOW AND (B) DATA-FLOW+TM APPLICATIONS 33
FIGURE 10 PRELIMINARY RESULTS ON SCALA FOR LEE-TM (SCALABILITY WITH NUMBER OF CORES). 35
FIGURE 11 PRELIMINARY RESULTS ON SCALA FOR KMEANS (SCALABILITY WITH NUMBER OF CORES). 36
FIGURE 12 A SUBSET OF THE EXECUTED DATAFLOW GRAPH FOR KMEANS 37

LIST OF TABLES

TABLE 1 LIST OF PROJECT REFERENCE APPLICATIONS 9

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

Rosa M. Badia, Yoav Etsion, Rahul Gayatri,
Milan Pavlović, Tomasz Patejko
BSC

Daniel Goodman, Christopher Seaton,
Mikel Lujan and Ian Watson
University of Manchester

Antoni Portero, Roberto Giorgi

UNISI

Andreas Diavastos, Constantinos Christofi, Samer Arandi,
George Michael, Pedro Trancoso, Paraskevas Evripidou
UCY

Sylvain Girbal
Thales

Executive Summary

This document is the second 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 teradevice 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 second year of the project in task 2.2 and task 2.3 and also the results for milestone 2.2. The objective of milestone 2.2 was to define the set of benchmarks, kernels and applications that the partners commit to port to the project programming models. The outcome of this milestone is described in section 2; this Milestone has been successfully achieved in m18. In the elaboration of this list, the suggestions received from the reviewers (adding graph-based and Recognition, Mining, Synthesis (RMS) applications) and from the Scientific Advisory Board were taken into account.

The activities performed in task 2.2 relate to the characterization of the applications. In the first year of the project, the different characterization methodologies to be used were defined and described in deliverable D2.1. This year, the partners have been performing the characterization of the project applications using these different methodologies. Section 3 presents a selection of the results obtained by the partners, using different methodologies.

Finally, although the task 2.3 was meant to start in year three of the project, the partners have started this task since there was a need for applications to perform the characterization and as input for other WPs. Section 4 presents some of the results of these porting activities.

1 Introduction

This is the second deliverable of WP2, Benchmarks and Applications. Understanding the runtime behaviour of applications is a crucial guideline in the design of computing systems, as they are the effective consumers of the underlying compute power. As TERAFLUX explores the design of highly parallel teradevice 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 themselves, down to extensions to commodity architecture. This exploration includes:

- Identify applications that can serve as reference applications for a programming model based on data-flow principles, and that can efficiently scale to utilize teradevice system.
- Characterize the resource requirements of these highly parallel applications, in terms of memory usage, bandwidth, and latency. Identify the performance requirements from underlying interconnection network. These characteristics will assist the architectural exploration performed in WP6.
- Uncover common data-flow and data-locality patterns implicit to the reference applications that can be disseminated into the programming model (WP3) as either data-flow or transactional semantics.
- Port a few applications to the programming models chosen by WP3. The ported applications will be used by the other work packages to guide their proposed designs.
- Extract the interesting patterns and data accesses and that will assist other work packages build sensible benchmarks that can test the proposed constructs in all domains: programming model (WP3), compilation platform (WP4), reliability (WP5) and architecture (WP6).

1.1 Document structure

The deliverable is organized as follows: this section introduces the deliverable and its structure, section 2 describes the list of applications and benchmarks that have been selected by the project in milestone 2.2 to be ported to the project programming models. Section 3 presents relevant results on the characterization of project applications using different characterization methodologies. Section 4 presents results on porting applications to the project programming models, and finally section 5 concludes the document.

1.2 Relation to other deliverables

Along the document there are references to deliverable D3.3. Also D2.1 presented metrics for TM that is not repeated again.

1.3 Activities referred by this deliverable

This deliverable refers to the activities performed in Task 2.2 during the second year of the project. Task 2.3 Porting applications to include new programming models was supposed to start in the third year of the project has already started since there was a need for initial benchmarks and applications to be used in the other WPs.

2 Identification of subset of the reference applications

One of the milestones of the project, M2.2 Subset of the reference applications to be ported to the programming models defined in WP3, was due month 18 of the project. This section presents the agreement of the partners involved in WP2 Benchmarks and Applications on which benchmarks and applications will be ported to the project programming models. The table below lists the benchmarks and applications, the programming model to which will be ported and the partner responsible of this activity.

The list was elaborated taking into account the list of reference applications selected in milestone 2.1 and extending it with the recommendations from the reviewers to add graph-based and Recognition, Mining, Synthesis (RMS) applications.

Benchmark	**Responsible partner**	**Programming model**	**Comments**
Matmul	BSC INRIA UCY UNIMAN	StarSs OMP + Sync DDM Scala + TM	
Radix Sort	INRIA	OMP	
Lonestar - TBC	INRIA	OMP + Sync	
Barnes-Hut	BSC	StarSs	
Cholesky	BSC UCY	StarSs DDM	
Sparse LU	BSC INRIA UCY UNIMAN	StarSs OMP + Sync DDM Scala +TM	
FFT2D	BSC INRIA	StarSs OMP + Sync	
SPECFEM3D	BSC UNIMAN	StarSs Scala + TM	
N Queens	BSC	StarSs	
Lee's Routing (Labyrinth)	UNIMAN BSC INRIA UNIMAN + UCY	Scala +TM StarSs OMP + Sync DDM + TM	
Kmeans	UNIMAN BSC	Scala + TM StarSs	RMS (mining)
Ssca2	UNIMAN	Scala + TM	Graph algorithm
STAMP – TBC	INRIA	OMP + Sync	
FFT 1D	INRIA	OMP	
Fmradio	INRIA	OMP	
802.11a	INRIA	OMP	
SimDiasca	INRIA	Sync	

Benchmark	**Responsible partner**	**Programming model**	**Comments**
Picture-in-picture	INRIA	Sync	
Ad-hoc software radio	INRIA	OMP + Sync	
Conv2d	UCY	DDM	
IDCT	UCY	DDM	
Trapez	UCY	DDM	
Graph 500	BSC	StarSs	Graph Algorithms
Flux (object tracking)	BSC	StarSs	RMS (recognition)
Application	**Responsible partner**	**Programming model**	
GROMACS	BSC	StarSs	RMS (Synthesis)
PEPC	BSC	StarSs	RMS (Synthesis)
WRF	BSC	StarSs	
STAP (Radar)	Thales BSC INRIA	Seq. code StarSs OMP + Sync	RMS (*Recognition*)
Viola & Jones (Pedestrian detection)	THALES INRIA	Seq code OMP + Sync	RMS (*Recognition*)
HPL Linkpack	BSC	StarSs	

Table 1 List of project reference applications

Remarks

OMP = OpenMP with streaming data-flow and transactional extension

Sync = Data-flow synchronous language with transactions

RMS = Recognition, Mining, Synthesis

Other plans

The consortium plans to choose one/two applications in StarSs in order to port some kernels to lower data-flow language (i.e. DDM).

LINPACK and GRAPH 500

Between the list of applications, we propose to enrich the TERAFLUX application list with two "single figure" benchmarks:

- LINPACK (HPL-2.0) - used in top500.org
- GRAPH 500 - newly introduced [3] for several programming models (including MPI, OMP, etc.).

Brief Introduction to LINPACK (HPL-2.0)

LINPACK [1] is employed to determine performance for ranking supercomputers in TOP500 [2]. TOP500 lists the world´s fastest computers. LINPACK is a software library that uses Basic Linear Algebra Subprograms (BLAS) libraries for performing basic vector and matrix operations.
The High Performance LINPACK benchmark (HPL-2.0) solves an N^2 linear equation system. This benchmark is executed with increasing matrix sizes (N) with the purpose of searching for the size N_{max} for which the maximum performance R_{max} is attained. R_{max} represents the floating point operations per second to solve the above linear system. Another measured point is when half of the performance ($R_{max}/2$) is accomplished – the corresponding N is called $N_{1/2}$.

In the experiments carried out at UNISI, the simulated machine (provided by the COTSon simulator – see D7.1, D7.2) consists of a master node (in TERAFLUX called service node), which is the node that runs the operating system, and **1024 auxiliary cores** (64 nodes of 16 cores). Currently, we are able to run LINPACK (HPL-2.0) experiment with MPI and the highest value we got is **R_{max} 2.0 GFLOPs** (in such case: N_{max} =4000, simulated time=21.04). However, we still need to carry out more extensive experiments.

Brief Introduction to GRAPH 500

The main purpose of Graph 500 [3] is to help guide the design of software systems and hardware architectures, because graph algorithms are a core part of many analytics workloads. The Graph 500 aims to cover kernels common in domains like optimization (single source shortest path), concurrent search, and edge-oriented (maximal independent set). In the future, other graph-related computer industry areas will be covered like Medical Informatics, Data Enrichment, Cyber security, Social Networks, and Symbolic Networks.
The benchmark consists of two stages: first a synthetic graph is generated and then some searches are performed on it. The number of the Traversed Edges Per Second (TEPS) is accounted and presented at the end for the given problem size (also called 'scale'), defined as the logarithm base two of the number of vertices.

In the experiments carried out at UNISI, we executed the Graph 500 in the same simulated machine that we used for LINPACK (1 service node, which runs the Operating System, and **1024 auxiliary cores**). The executed command is: *mpiexec.hydra –np <X> ./graph_mpi_simple <Y>*, where X = 1, 2, 4, 8, 16, 32, and 64; where X represents number of processes for the MPI launcher. Y represents the scale of the problem for the Graph-500. The highest value we obtained is **17 Million TEPS** (for X=1, Y=1). However, also in this case, we still need to carry out more extensive experiments.

3 Applications characterization

This section presents results on applications characterization. Different characterization methodologies have been used; a description of these characterization methodologies can be found in deliverable D2.1.

3.1 Analytic characterization (Thales)

For the purpose of analysing the STAP and the Pedestrian Detection applications, Thales used an internal co-design environment called SpearDE. This environment allows for dataflow applications characterization in view of their parallelisation and code generation for heterogeneous distributed (embedded) architectures, by exposing relevant features both in the application and the computing architecture models.

SpearDE workflow for application characterization and modelling: The applications are modelled as a dataflow graph based on the graphical interface of PtolemyII, but following a specific multi-dimensional synchronous dataflow-like model of computation based on the ArrayOL formalism.
Along with the application graph SpearDE, enables the modeling of different heterogeneous parallel architectures via a structural view (processing units/elements, memory layout, communication paths) as well as a performance view (such as time behavioral models of the architecture elements) of the architecture.
Based on these two models a mapping strategy can be defined, addressing both task-level and data-level parallelism. The tedious operations linked to mapping, which are also error prone when done manually, such as data transfers, data reordering or model consistency in general, are handled by the SpearDE tool. These are intrinsically linked to the target hardware and different for each configuration regardless the fact that the application model might be the same. SpearDE generates a valid scheduling, including software pipelining for the mapped application, as well as the involved static memory layout. This leads to rapid performance simulation on the given platform and therefore allows the user to iteratively choose the best design according to the imposed constraints. Finally, code generation is performed using back-end, processor-specific compilers.

Application Analysis: The first step consists in describing the dataflow application as an acyclic graph whose nodes are statically affine nests of loops to be executed over some Elementary Transform (ET), whereas the edges represent the input-output flow as multi-dimensional arrays of data. Each ET is described as a C-code kernel usually operating onto a subset of the input data, and which will need to be repeated by the node until the entire input array has been consumed. SpearDE computes dynamically the needed loop bounds, provided the dimensions of the input data are known (non-parametric).

This graphical representation highlights several axes of parallelism already at functional level before even considering a given parallel platform. Task parallelism becomes obvious when considering different branches in this graph and data parallelism is highlighted inside each node through the external loops to be executed.
At this point, despite the fact that this application representation is architecture agnostic, thanks to the information describing an elementary operation, data sizes are computed by the tool on the fly, this giving a preliminary estimation of throughput and computational load.

This information will be later used in SpearDE to prepare the user-assisted architectural mapping phase, to automatically generate the communications allowing to transfer data between processing elements, or to automatically fuse several elementary tasks together to reduce the communication overhead.

Communications and the Transposition problem: For image and signal processing applications, the dataflow usually consists of multidimensional arrays, whereas the ETs consist of various filters, thresholds, etc. to be applied to some dimensions of this multi-dimensional data independently.
However an Elementary Operation makes no assumptions on the data organization applying itself systematically to the first dimension(s) of the input data. Therefore data reorganisation (e.g. matrix transpose) might be necessary prior to executing one node in the graph, which usually translates into an inserted communication (e.g. in architectures with distributed memory). To avoid extra communication costs, those transpositions are performed alongside with the communications between processing elements in the architecture, trying both to maximize data-locality and to minimize memory occupancy.

In order to test the functional validity of the application graph, sequential, executable C code can be generated for comparison with reference implementation results if any or for functional debug.

From this Application Analysis phase, the Teraflux project will take benefits from the data-flow description of the application, the associated estimated throughput, and from the sequential version of the application to validate against.
Within the Teraflux project, we will first consider those transpositions as extra elementary operations, allowing to directly use the Teraflux communication model at the cost of extra communications. In a second step we will consider embedding those transpositions into the communication scheme to reduce the communications and benefits from better data locality as presented above.

Pedestrian detection application
For the pedestrian detection application characterized in D1.1 the dataflow graph contains the nodes:

1. GrabOneFrame – reads image data
2. Integral_rows – computes the Integral Image on the row dimension
3. Integral_col – uses the results of the above task to accumulate the results on the column dimension, thus yielding the Integral Image at the end
4. ImgDotSquare – computes the pixel-wise square image, needed for the computation of a normalization factor.
5. normFactors – computes the variance of each image window, using integral images of the original image and the pixel-wise square image.
6. GrabFeatures – reads the rectangle-features at each stage in the cascade, along with their associated thresholds and classifier decision parameters, a and b.
7. ScaleFeatures – computes the new coordinates of the rectangles according to the actual scale
8. GrabThresholds – reads the stage thresholds into an array.
9. Detect – is the main task applying the detection cascade on all windows in an image at a given scale.

Parallelisation: This application is interesting because it exhibits several axes for parallelisation according to the scales, the image tiles as well as the stage filters in the classifier cascade. The SpearDE characterization immediately highlights the latter two in the corresponding nodes of the graph, for instance tile parallelism in Figure 1.

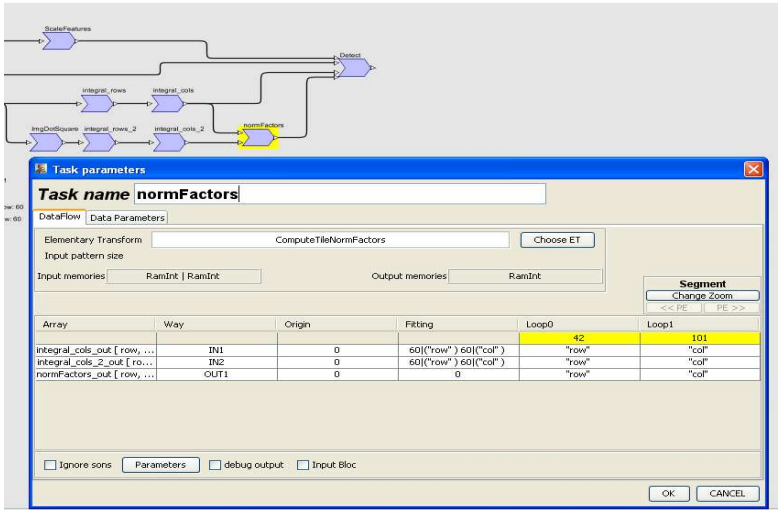


Figure 1 Detail of one task in the application graph for image tiles of 60x60 pixels. 42x101 computations ca be done in parallel in order to use the entire input image.

Then the parallelisation of the application onto a multi-cluster architecture can be done as in the example below.

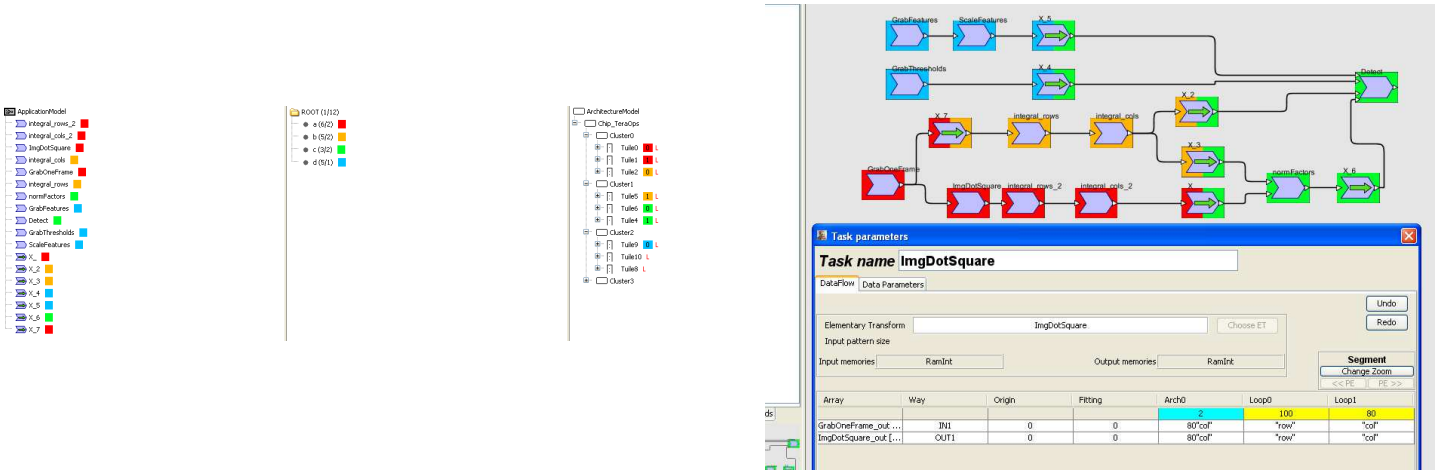


Figure 2 Pedestrian detection mapping onto a parallel architecture

Figure 2 shows in the left-hand side the allocation of tasks onto processing elements (the different colours represent so-called segments and each segment executes a subset of the application graph onto a subset of the architecture). The right-hand side show the result (i.e. mapped application) obtained

when allocation is complete for the entire graph. The lower side represents the detail of one node in the graph in which an extra-dimension related to the architecture has been added after mapping (Arch0). Here Arch0 contains two processing elements which will function in SIMD fashion onto half of the input data each.

The global control-flow graph for this application, as given in Figure 3, shows the previously mentioned parallelism axes. Note that in this algorithm there is a trade-off to be done between parallelising at filter level or at image tile level. The number of filters to apply is much smaller at early stages in the classifier cascade than at the last stages, while the number of image tiles is rapidly decreasing with the stage index.

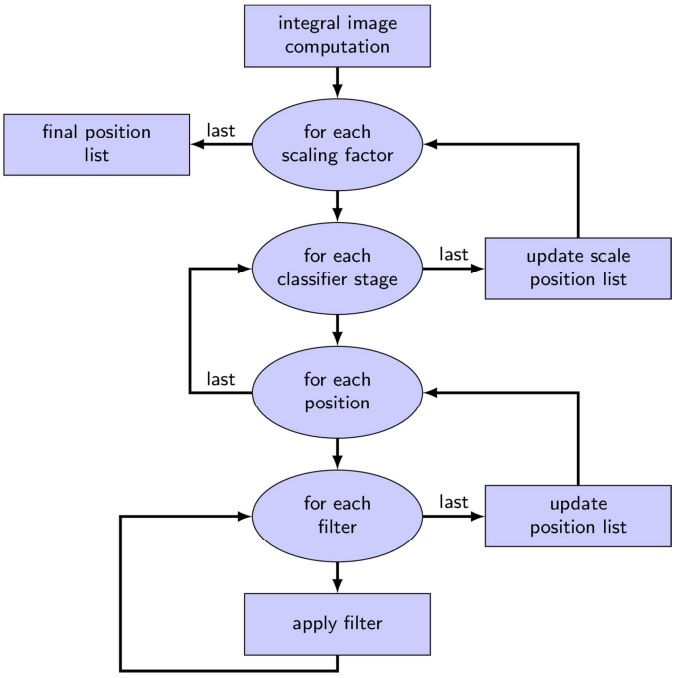


Figure 3 Pedestrian detection control-flow graph

A possible parallelization option is to use the image tile axis at the beginning of the classifying cascade and the filter level axis at the end of the cascade. The same applies for the scale axis.

According to the number of image tiles at a given scale, another option would be change the algorithm into applying the entire cascade on all image tiles without updating the position list at intermediate steps. This introduces another axis of parallelisation given by the stages in the cascade that can be applied in parallel. The extra computation might be compensated by the parallel processing power of the chosen architecture.

3.2 Characterizing the Memory Requirements of MPI applications (BSC)

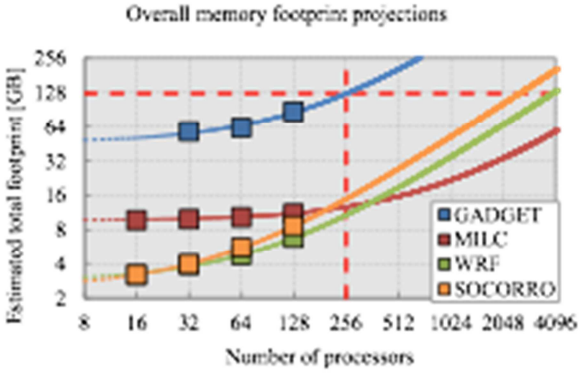
In this work, presented in [4], we characterize the memory behavior of several well-known parallel scientific applications. We project the performance required from the memory system, to adequately serve large-scale CMPs, in terms of memory size, memory bandwidth and cache size.

We base our predictions on the per-CPU memory requirements of distributed memory MPI applications. Although this methodology is imperfect (data may be replicated between nodes, which may result in pessimistic predictions when addressing shared-memory environments), we believe it provides a good indication of the requirements from a CMP memory system.

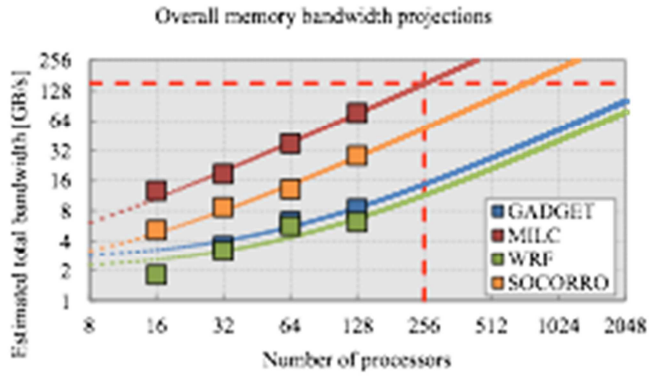
We base our analysis on a set of four applications, chosen to represent the dominant scientific domains in the supercomputing centers surveyed. More importantly, the analysis showed that each of the selected applications stress different aspects of the memory system. The selected applications include:

- GADGET (GAlaxies with Dark matter and Gas intEracT). A code for cosmological simulations of structure formation. It computes gravitational forces with a hierarchical tree algorithm, optionally in combination with a particle-mesh scheme for long-range gravitational forces. It is one of the most often used applications, representing the area of astronomy and cosmology. From the four applications that we used, GADGET had the highest requirements of memory size.
- MILC (MIMD Lattice Computation). A set of codes for doing simulations of four-dimensional SU(3) lattice gauge theory, represents the area of particle physics, and in our analysis had the highest requirements of memory bandwidth.
- WRF (Weather Research and Forecasting). A next-generation mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. It is a well-known DEISA benchmark from the area of earth and climate. In our analysis, it is characterized as the application that is bound by the system's computational resources.
- SOCORRO — self-consistent electronic-structure calculations utilizing the Kohn-Sham formulation of density-functional theory. Calculations are performed using a plane wave basis and either norm-conserving pseudopotentials or projector augmented wave functions. This application mostly stresses the memory bandwidth.

Each of the applications was executed on 16, 32, 64 and 128 processors, with the exception of GADGET — whose memory footprint could not fit on 16 MareNostrum blades. The input sets in all of the analyzed applications remained unchanged while scaling the number of processors.



The total memory footprint is calculated as the number of processors multiplied by the maximum per-processor footprint. In case that the memory system does not satisfy the maximum footprint requirements of a given application, it would crash when left out of memory space. Our experiments show that doubling the number of processors does not halve the size of the per-processor memory footprint. For both WRF and SOCORRO, doubling the number of processors only reduces the per-processor memory footprint by 20-40%. Scaling is somewhat better for GADGET, for which scaling from 32 to 64 processors reduces the per-processor footprint by 45%, while scaling further to 128 processors reduces the footprint by only 30% more. Scaling for MILC is very good, as footprint reduction is very close to 50%. Our projections show that future manycores consisting of more than 100 cores must be directly backed with a few dozen GBs of main memory in order to support scientific workloads.

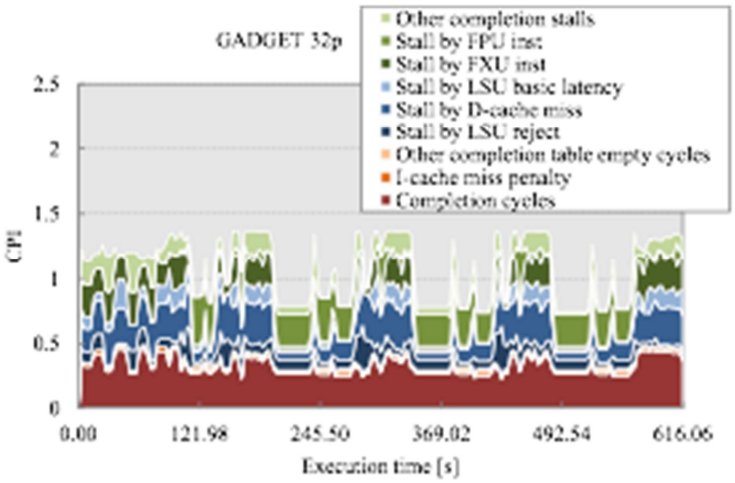


In order to predict memory bandwidth requirements, we measured the per-processor bandwidth consumed by each benchmark at three levels: the off-chip memory, L2 cache, and L1 cache. Total bandwidth is calculated as the average per-processor bandwidth multiplied by the number of processors. Our projections show that future manycore systems consisting of more than 100 cores may easily require more than 100 GB/s of main memory bandwidth. Modern architectures such as Intel Nehalem-EX or IBM Power7 employ 4 and 8 DDR3 channels respectively, peaking at 102.4 GB/s of bandwidth. Knowing that the sustained bandwidth is typically 20%–25% lower due to page and bank conflicts, we conclude that such large-scale systems will need to provide higher bandwidth to support high-performance scientific computing.

The cycles per instruction (CPI) metric is defined as the average number of processor cycles needed to complete one instruction. For any execution segment, it is calculated as a total of elapsed cycles divided by the number of completed instructions. A low CPI value means that the system resources are better utilized, and the architecture operates closer to its peak performance.

The CPI stack model, which is the breakdown of CPI value to the individual latencies contributed by different micro-architectural resources, can therefore be used to determine the key factors that impede

performance. Each component in the stack describes the average number of cycles an instruction stalled on a particular core resource (like Load/Store unit, or Floating Point unit).



GADGET's most obvious CPI stack patterns are significant fluctuations of the overall CPI value, revealing three periodic iterations. Parts of each iteration with low FPU usage have an overall high CPI value, whereas parts with high FPU usage have low CPI value. When correlating this with the frequency of memory accesses, it is noticeable that high CPI value corresponds to high memory traffic (more specifically high number of stores). This also justifies fairly large number of LSU stalls (LSU reject, D-cache miss and LSU basic latency) in this part of the iteration. Therefore, each of the three iterations can be divided into a communication phase (low FPU usage, lots of memory accesses, high bandwidth, high CPI), and a computation phase (high FPU usage, few memory accesses, low bandwidth, low CPI). We do not see much variation in GADGET graphs for 32, 64 or 128 processors.

In summary, LSU related stalls seem to be the most dominant CPI stack component in the applications tested. The exceptions are GADGET and WRF, whose computation phases are limited by FPU stalls. It indicates that memory hierarchy could have an exceptionally high impact on performance of the future manycores. It is also clear that wider superscalar approach can have a limited impact, and the only ones that could see the benefit are FPU intensive applications.

System designers often use processor's arithmetic performance measurements to dimension the required performance of other computer system components. Dimensioning memory bandwidth is often based on a ratio of maximum bandwidth and maximum theoretical rate of floating point operations. A common rule of thumb for obtaining optimal performance is to keep this ratio around 0.5 bytes per flop. This means that a processor capable of achieving 9.2 GFLOPS should rely on memory that supplies 4.6 GB/s of bandwidth. However, our analysis show that this ratio is heavily overestimated, and that it should not be taken into account at all, mostly due to the underutilization of floating point resources in real applications.

In light of our findings, we expect that current memory architectures based on on-chip memory controllers and multiple parallel DDR channels should be able to sustain multicores for the next decade. However, technology constraints are limiting its further scalability, and in order to support the

requirements of multicores consisting of several hundreds of processors, we need to explore other approaches.

3.3 CPI stack characterization of StarSs applications (BSC)

The previous section presented characterization of memory latency and bandwidth and CPI stack for MPI applications. Similarly, in deliverable D2.1 results for other MPI applications were presented. In this section, results for StarSs applications that follow a data-flow execution model are presented. More specifically, the research work has focused in characterizing the CPI stack for StarSs applications using the BSC CEPBA-Tools performance tools.

We present a method of obtaining and analyzing CPI stack information. In this case, the CPI stack is built based on runtime architectural information provided by Power PowerPC 970 Performance Monitor Units. A clustering algorithm is used to group instances of tasks into clusters with the same performance characteristics. This section presents results for the STAP application.

3.3.1 Instrumentation

The application to be analyzed in compiled by OmpSs compiler (one of the StarSs implementations) with instrumentation set to on. The instrumentation package used byt the OmpSs runtime is Extrae, which is part of the CEPBA-tools toolset. The communication between the instrumented application and Extrae is managed by an instrumentation plug-in. The plug-in is a part of the OmpSs runtime library. The compiler inserts calls to API provided by instrumentation plug-in in task's code. The calls are as follows:

- Call for registering a task as a user function for Extrae
- Calls that start and stop collecting values of performance counters.

Functions that are used in the second bullet generate a registration key, set by the call from the first bullet, to identify task's instance performance information that has been gathered for. Trace information collected by Extrae is then emitted to the trace file.

3.3.2 Clustering and CPI Stack

After the trace file containing values of performance counters mapped onto tasks' instances is generated, a clustering algorithm is applied. It identifies and groups into clusters executed instances with similar performance characteristics. The clustering algorithm we used is DBSCAN. This algorithm builds clusters based on density of neighborhood around a given point that exceeds a threshold MinPts. The neighborhood is shaped with chosen distance function dist(p, q) between points p and q. For points p and q that lie inside a cluster the following holds: dist(p, q) <= Eps. These two parameters, MinPts and Eps, are parameters passed to the algorithm.

By altering the value of Eps we can change the neighborhood of the points and by that change the shape of the clusters. Lower values of Eps make the algorithm generate smaller and dense clusters with low number of border points (points on the border of a cluster), with higher number of points that are considered noise. Higher values of Eps generate larger and sparse clusters with many border points, with lower number of points that are counted in as noise.

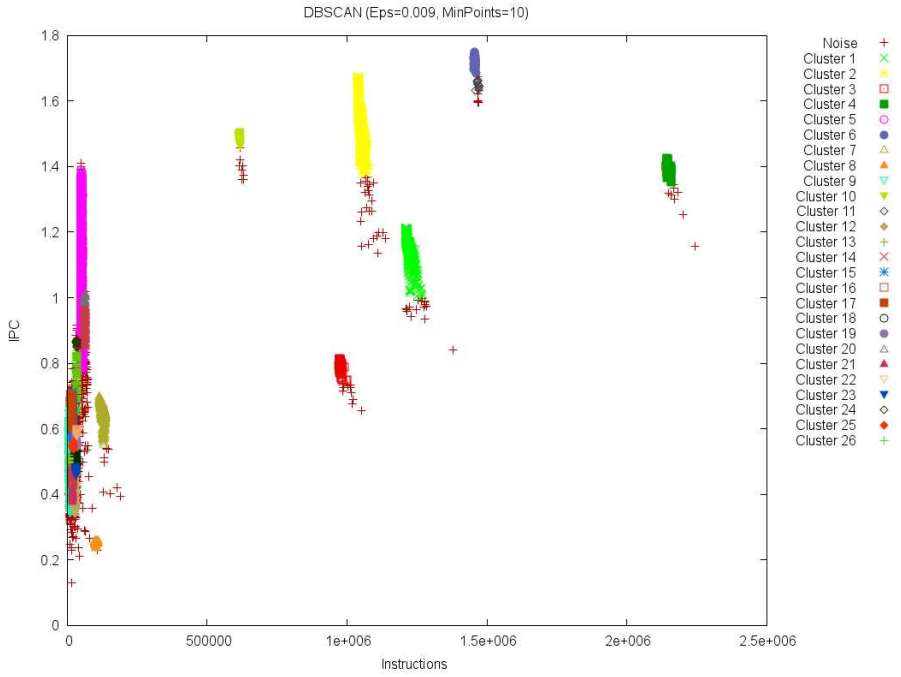
The metrics that we use are IPC (Instructions Per Cycle) combined with Instructions Completed. They form a distance function used for DBSCAN to cluster instances of tasks with similar performance characteristics.

The output of the clustering process is data that describe the clusters grouping tasks' instances with respect to the mentioned metrics, and Paraver trace file showing the tasks mapped onto clusters. Once the clusters grouping tasks with similar performance characteristics are determined, the CPI stack information is generated.

3.3.3 Discussion of results

STAP

We start our discussion of the results by analyzing the clusters that were shaped from executed tasks. The plot with clusters containing instances of tasks with similar performance characteristics is shown below:



Clusters 1, 2, 3 and 4 group instances of tasks that contribute the most into the overall performance of the application; the contribution of the clusters is at least 3% of the total execution time of the application. These clusters group the following instances of tasks:

- Cluster 1
  - `Apply_Filter_task`
- Cluster 2
  - `Calc_Filter_task`
- Cluster 3
  - `tfac_task`
- Cluster 4
  - `Mat_Invert_task`

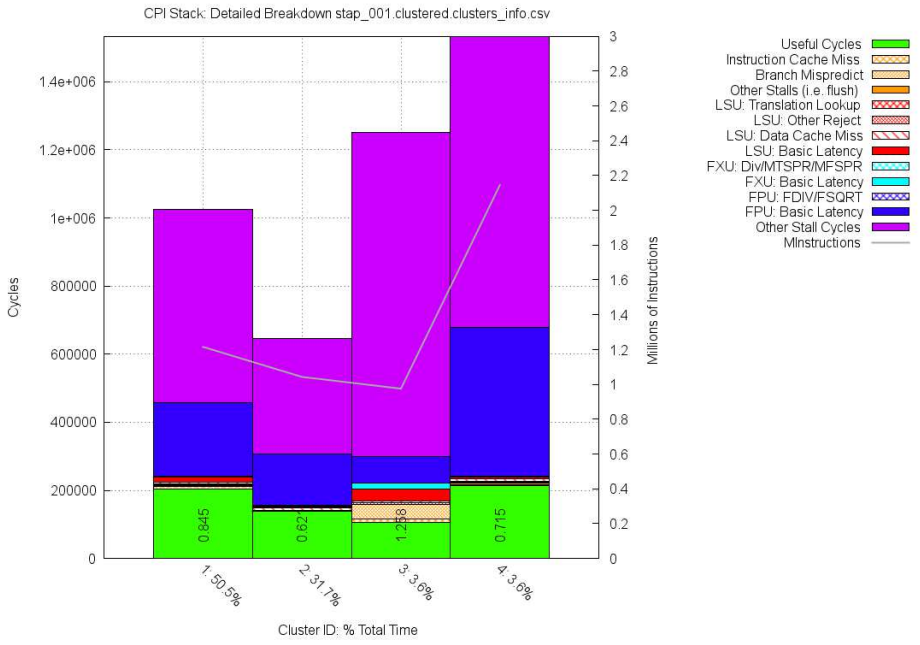
Tasks `Apply_Filter_task` and `Calc_Filter_task` are the largest in terms of duration, and with the highest number of executions. Because they have high number of instances (2880 for both tasks), clusters grouping instances of these tasks have a large variance with IPC ranging from almost 1 to more than 1.2 for cluster 1 and from almost 1.4 to more than 1.6 for cluster 2. A similar case can be observed for cluster 5 grouping instances of task `Int_Dop_task` (also with 2880 instances). The instances of this task are much smaller than instances of tasks `Apply_Filter_task` and `Calc_Filter_task`. However, cluster 5 has a large variance with respect to IPC ranging from values below 1 to 1.4.

Size and duration of instances of tasks in clusters 1 and 2 are the main factors that contribute intothe high number of instructions completed by the instances of these tasks.

The points representing instances of tasks `tfac_task` and `Mat_Invert_task` form smaller and denser clusters (clusters 3 and 4) with low variability with respect to the IPC. The number of instances is much smaller compared to tasks in clusters 1 and 2 (144 for task `Mat_Invert_task` and 180 for task `tfac_task`).

However, these clusters differ when values of instructions completed are compared: instances of task `tfac_task` complete around 1 MInstructions while instances of task `Mat_Invert_task` complete more than 2 MIntructions. This indicates that cluster 4 groups computationally intensive instances of tasks.

Now we analyze the CPI breakdown of clusters 1, 2, 3 and 4. CPI breakdown is shown on the plot below:



Tasks in all the clusters perform floating-point calculations so they all experience latency related to FPU execution unit. As it was mentioned before, task `Mat_Invert_task (cluster 4)`, although is the smallest, is the most computationally intensive: it completes the highest number of

instructions with high latency caused by the FPU. This task implements a matrix inversion with Gauss elimination method. Also two larger tasks: `Calc_Filter_task (cluster 2)` and `Apply_Filter_task` (cluster 1) are computationally intensive. They implement a filter calculation and an algorithm that reduce the clutter signal in the STAP method.

The only task that wastes cycles on FXU (fixed-point unit) is task `tfac_task` grouped by cluster 3. This task does integral calculations on array indices. Results of these calculations are used as predicates in `if` statements. This explains the high number of cycles wasted cycles on miss-predicted branches. This also causes higher ratio of wasted work with highest CPI comparing to other tasks.

SPECFEM3D

We will start analysis of CPI stack for SPECFEM3D by examining clusters' shapes and distribution of points. As it was a case for STAP, we will focus on clusters whose tasks cover more than 3% of execution time in total of the application. These clusters are:

- Cluster 1
  - `process_element`
- Cluster 2
  - `scatter`
- Cluster 3
  - `gather`

These clusters are shown on the plot below:

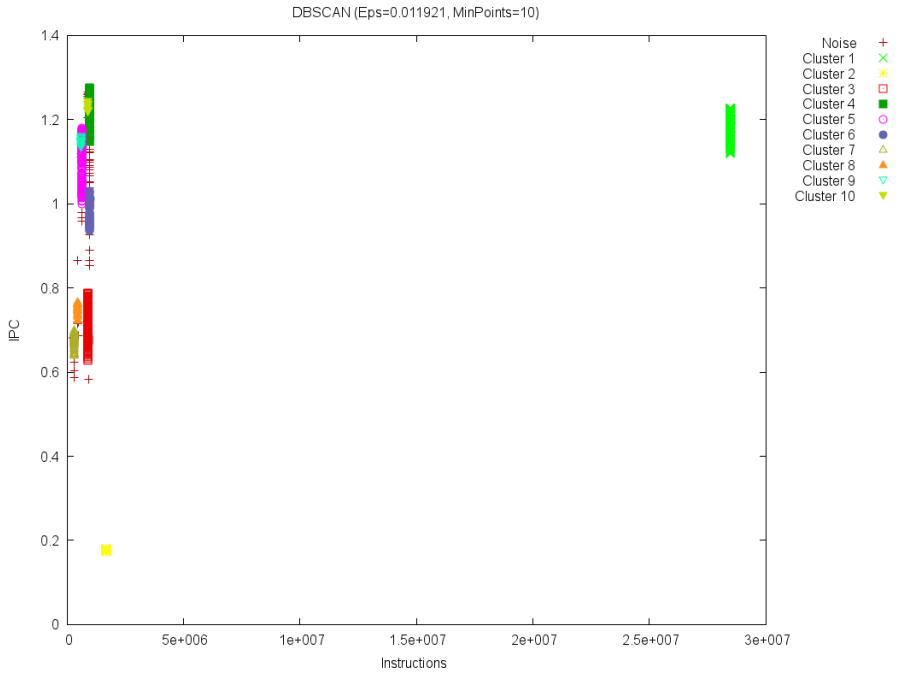


Table below shows a description of the tasks' instances grouped by aforementioned clusters.

Name of task	Number of instances	Total execution time [ns]
`process_element`	720	7.498.931.151
`scatter`	720	3.001.285.428
`gather`	720	397.080.482

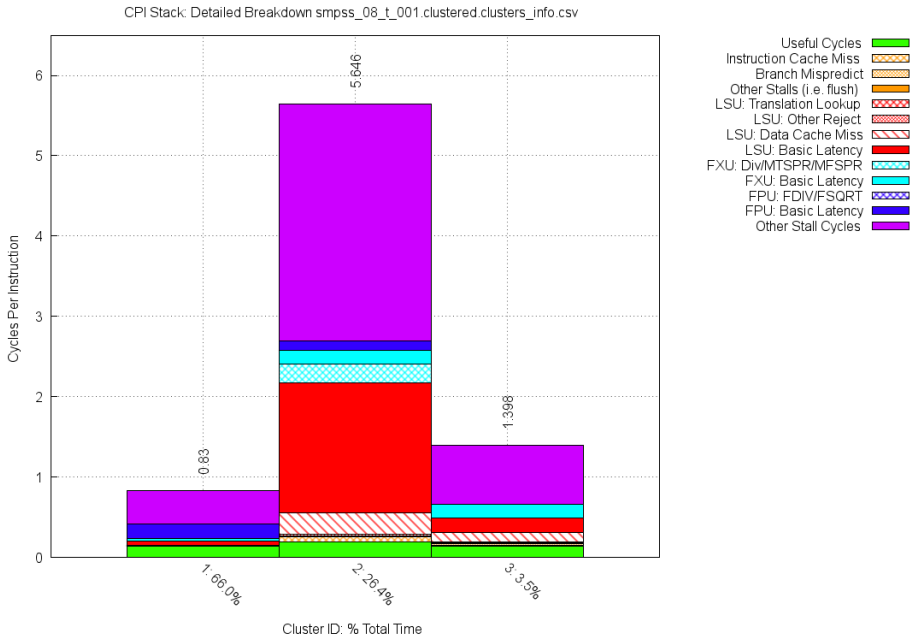
Cluster 1 groups instances of the task `process_element`. As we can see in the table above, the task is the largest in terms of duration. The IPC of the points in the cluster varies from 0.6 to 1.3. They complete the largest number of instructions among task instances grouped by clusters under consideration (2.8 MInstructions). The cluster also achieves the highest values of IPC. These factors indicate that these task instances do not suffer from major issues that degrade its performance.

Shape of cluster 3, that collects instances of task `gather`, is similar to cluster 1. The points that form the cluster are scattered in terms of IPC with values that range from 0.6 and 0.8. The value of IPC is lower than in cluster 1. Also, the task instances complete a lower number of instructions (0.9 MInstructions) and they are smaller in terms of size and duration. This indicates task `gather` encounters runtime issues that degrade the performance of its instances.

The instances of the task `scatter` form cluster 2. The cluster is dense and task instances are concentrated in one place. They share the same values of performance characteristics. Value of CPI is the lowest (less than 0.2) among all the clusters under examination, with small number of instructions completed (less than 1.7 MIntructions). That indicates that task instances suffer from the same performance problem that causes their inefficiency.

Now we will analyze CPI stack of tasks whose instances are grouped in clusters 1, 2 and 3. The plot showing CPI stack breakdown is shown on the next page.

As it was already mentioned the highest performance rate is achieved by instances of task `process_element` grouped in cluster 1. Each instance of the task operates on local mesh that has been previously collected by instances of task `gather`. This provides data distribution and isolation among task instances. Task instances of the cluster encounter small number of d-cache misses (0.04 MCycles wasted on handling this issue compared to total number of 23MCycles). This indicates good spatial and temporal locality of memory accesses each task instance performs.



Operations performed by task `process_element` are floating-point calculations (mainly addition and multiplication). We can see that large number stalls caused by FPU instructions (5 MCycles compared to total number of 23MCycles).

Cluster 3 collects instances of task `gather`. The task instances feature worse performance metrics compared to task instances from cluster 1. The task localizes data from global displacement vector and, using indirect addressing, places it in local mesh. This is the task whose instances touch their local meshes for the first time. This generates d-cache misses (97 KCycles wasted compared to total number of 1.2 MCycles) and causes stalls in LSU while data from lower levels of memory hierarchy is being fetched (0.2 MCycles). This explains poor performance of task instances. The data is then reused by task instances from cluster 1. This explains good performance metrics of instances of task `process_element`.

Cluster 2 groups instances of task `scatter`. The task iterates over array that is shared among threads that execute instances of the task. Synchronization is provided by `#pragma omp atomic` operations. The operation is compiled to assembly code that implements spin lock by emulating compare-and-set operation using instructions `lwarx` (*Load Word and Reserve Indexed*) and `stwcx` (*Store Word Conditional Indexed*). These instructions are placed in loop; they repeatedly access memory trying to acquire reservation. This puts additional pressure on LSU (3.2MCycles caused by LSU instructions compared to total number of 9.4MCycles).

There's one more reason why LSU encounters high latency while instances of task `scatter` are executed. PowerPC 970MP uses store-through with no-fetch-on-write policy. This means that whenever shared data (in our case elements of array accessed with OpenMP atomic operation) is written to by the core, all the other cores need to fetch this data from main memory. If they store copies of data being accessed to in their L1 d-cache, the cache line containing shared data is invalidated. When they access the data cache miss is generated and they have to fetch the correct data

from memory. It also explains high number of d-cache misses during `scatter` task's instances execution (0.4 MCycles compared to total number of 9.4 MCycles).

Cholesky

We will now analyze clusters and CPI stack breakdown of a kernel that implements a Cholesky factorization. Clusters whose task instances cover more than 3% of execution time of applications are as follows:

- Cluster 1
  - smpSs_sgemm_tile
- Cluster 2
  - zz_copyBlockTransposed
- Cluster 3
  - smpSs_strsm_tile
- Cluster 4
  - smpSs_ssyrk_tile
- Cluster 5
  - zz_copyBlock

Although task instances are grouped into five clusters, we will focus on clusters 2 and 5. Cluster 1, 3 and 4 group instances of tasks that call to BLAS routines that operate on tiles of a matrix. Results of the CPI stack breakdown for these tasks depend on the BLAS library being used. So analysis given in this section will obscure CPI stack results of execution of the application built against different other BLAS libraries. However, analysis of CPI stack breakdown prepared for several BLAS libraries will help choosing the library that is the most efficient one.

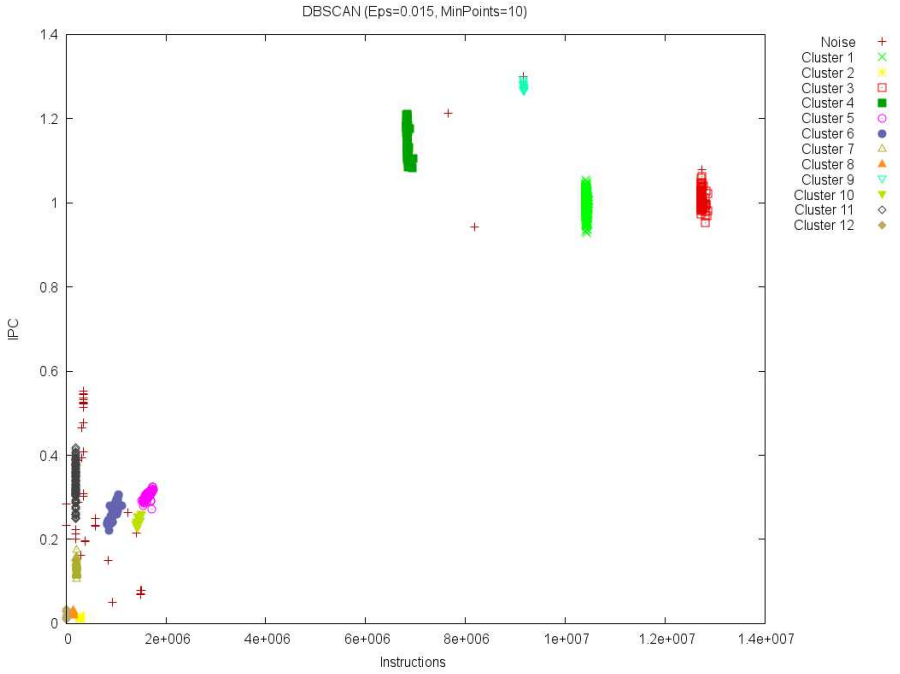
Shape of clusters 1 and 5 is shown on the next page. Both clusters 2 and 5 encounter poor performance metrics. Task instances grouped in cluster 5 feature IPC value of 0.3, IPC value of task instances in cluster 2 ranges from 0.01 to 0.02. This indicates similar performance issues related to memory accesses that we have described for SPECFEM3D's clusters 2 and 3 (grouping instances of tasks `scatter` and `gather` respectively).

Both tasks implement copying elements of one array to another. The difference between these tasks is that task zz_copyBlockTransposed realizes in the following manner: value of element a1i,j of array A1 is assigned to element a2j,i of array A2; in case of task zz_copyBlock value of element a1i,j of array A1 is assigned to element a2i,j of array A2. These accesses, in both cases, are realized inside two nested loops.

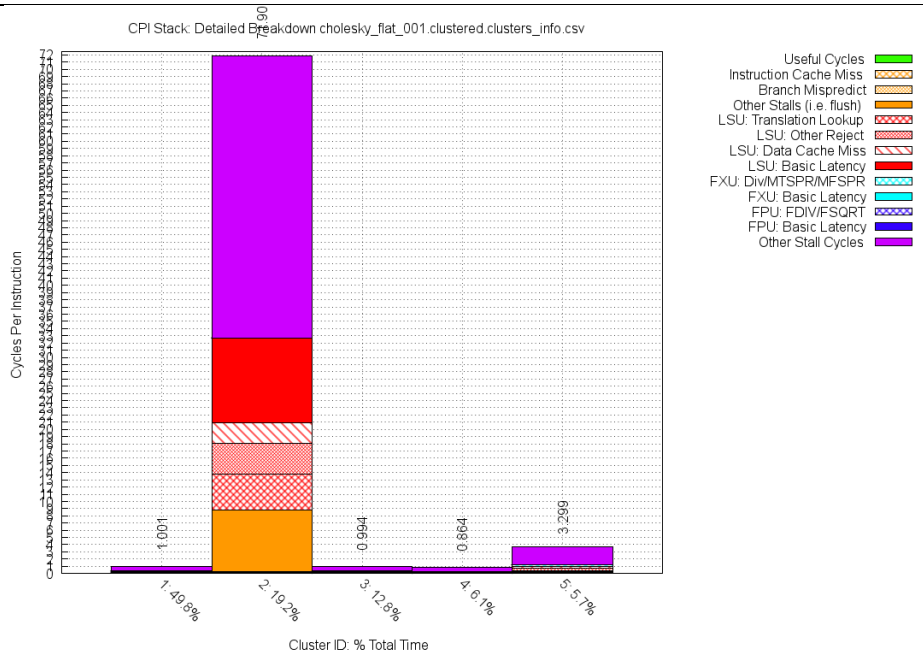
We can see that accesses to array A2 in task zz_copyBlockTransposed are realized in column-oriented manner. So instances of the task touch non-continuous pieces of memory. Accessing memory this way does not provide good spatial locality. Also instances of task zz_copyBlockTransposed touch elements of the array A2 for the first time. This causes significant number of d-cache misses.

When we look at graph showing CPI stack break down (that is shown on the next page), we see that instances of task `zz_copyBlockTransposed` encounter wide range of problems related to memory access, for example: d-cache misses (0.8MCycles compared to total number of 20MCycles)

and ERAT misses (1.3MCycyles). This in turn results in stalls LSU encounters (6.5 MCycles). These memory access phenomena are caused by scheme of memory access the task implements and result in very high value of CPI (72 cycles per instruction, with value of 0.035 MCycles of completion cycles).



Cluster 5 that contains instances of task zz_copyBlock achieves better performance characteristics than cluster 2. Cluster 5 features CPI of value 3.3 cycles per instruction. Although, as it was already mentioned, it implements straightforward functionality, the operation causes some performance degradation related to memory accesses. Instances of task zz_copyBlock access the same arrays as task instances of zz_copyBlockTransposed. These task instances are also executed just after instances of task zz_copyBlockTransposed complete (in between instances of task zz_makeBlockSymmetric touch array A1). This means that it's likely that blocks of array A2 were evicted from cache and need to be load there again. This is a reason for d-cache misses (0.5 MCycles compared to total number of 5.4 MCycles) instances of task zz_copyBlock encounter.



3.4 Characterisation of StarSs + TM applications (BSC)

As described in deliverable D3.3, one of the efforts in the project deal with the integration of Transactional Memory with the StarSs programming model, to provide an alternative mechanism for accessing shared data with mutual access or to access data atomically, but also to provide a mechanism to enable the speculative execution of tasks in conditional and while-loop structures, allowing for better performance on the execution of the StarSs applications.

While the results presented in deliverable D3.3 for a set of benchmarks are promising, we wanted to further understand the impact of software TM in the StarSs runtime. While StarSs runtime is already instrumented and Paraver tracefiles can be generated that enable to analyze both the behavior of the application and of the runtime, we further instrumented the runtime in those areas where the application was entering a transaction. The instrumentation marks when an application thread enters a transaction, when the thread performs a commit and when it aborts.

With this instrumentation, tracefiles for the benchmarks queens, matmul and specfem3D have been generated, and we have characterized the transactions for each of them. We have analysed the tracefiles with regard the impact of TM using Paraver. More specifically, we have generated histograms of the duration of the periods when each application is in a transaction, is performing the commit and when it is aborting. We have analyzed these times for different numbers of threads to understand how TM impacts when increasing the number of threads.

The figures below show the histogram of the duration of the periods when each application is in a transaction. To better understand how this time varies with the number of threads, what is shown for each value of time is the percentage of the number of instances that have a given duration.

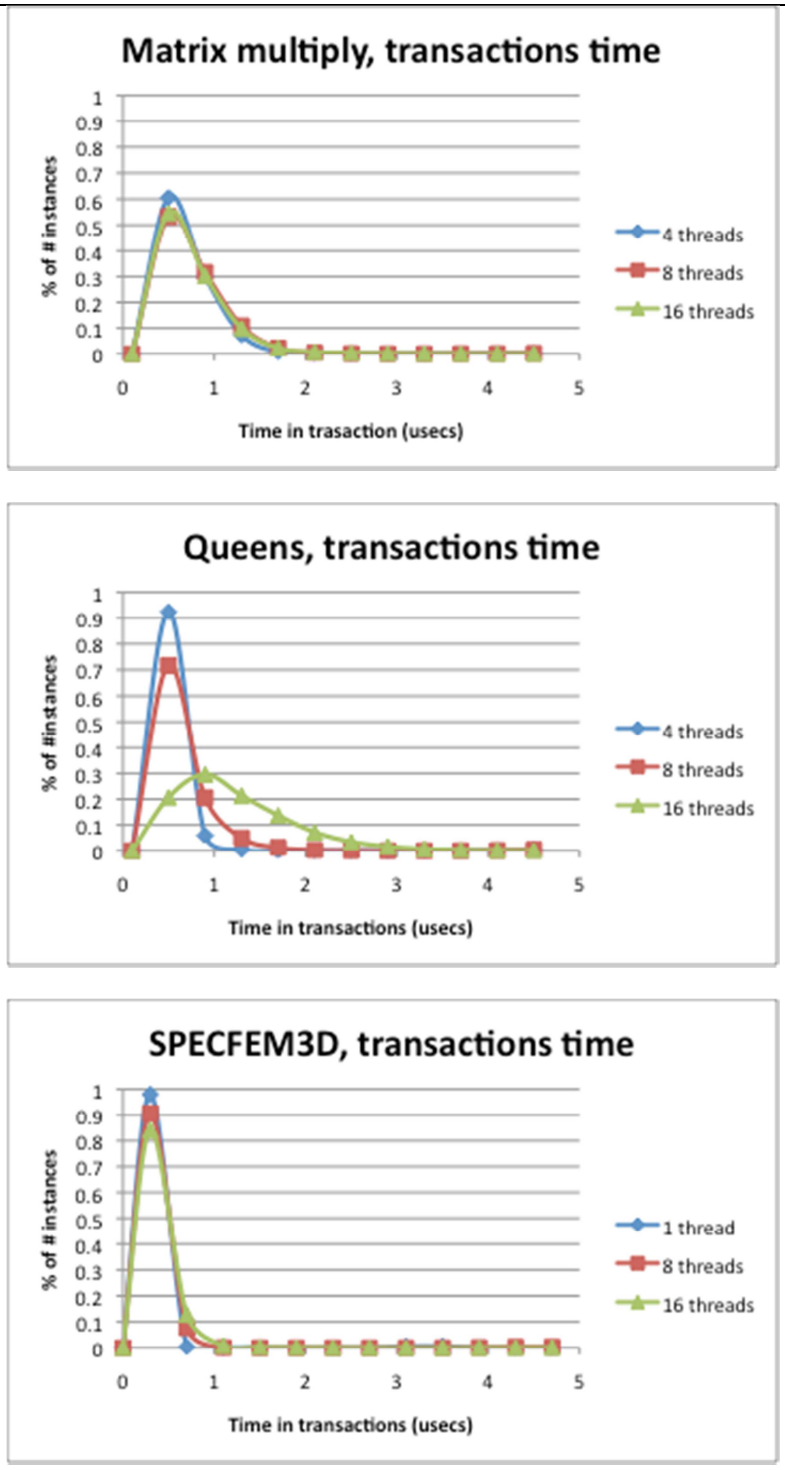
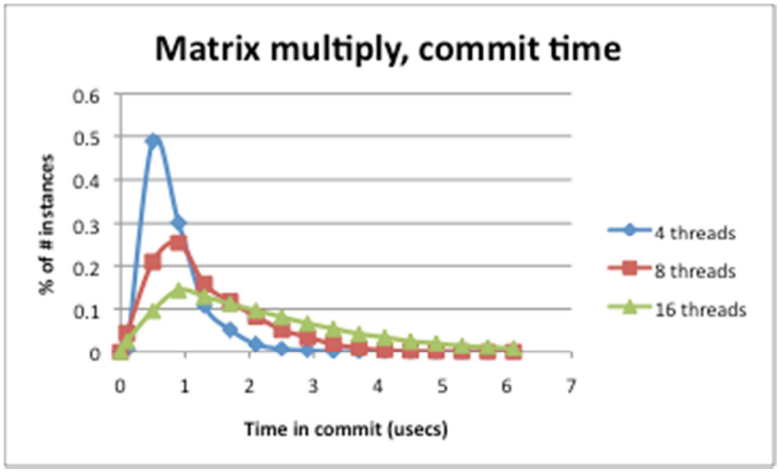


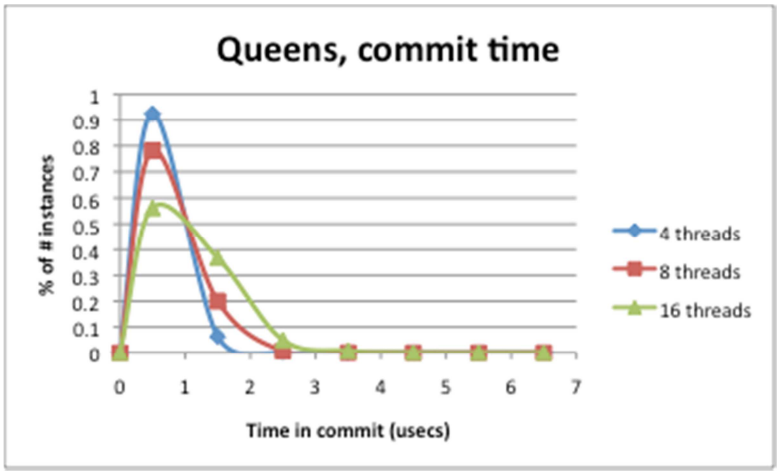
Figure 4 Characterization of transactions time

For each of the applications we observe a slightly different behavior. Both in the matrix multiply and the SPECFEM3D cases, the number of threads does not seem to affect severely the duration of the time in a transaction. It is noticeable that the duration of the time in a transaction is smaller in the SPECFEM3D case, but this is because fewer operations are performed. However, for the Queens

case, the average time in transactions increases. In all cases the time in the transaction is not very large (most of cases less than 1 microsecond), and we think that this time can be further reduced with hardware support.

Similarly, the figures below show the histogram of the duration of the periods when each application is in the commit stage. Again, what is shown for each value of time is the percentage of the number of instances that have a given duration.





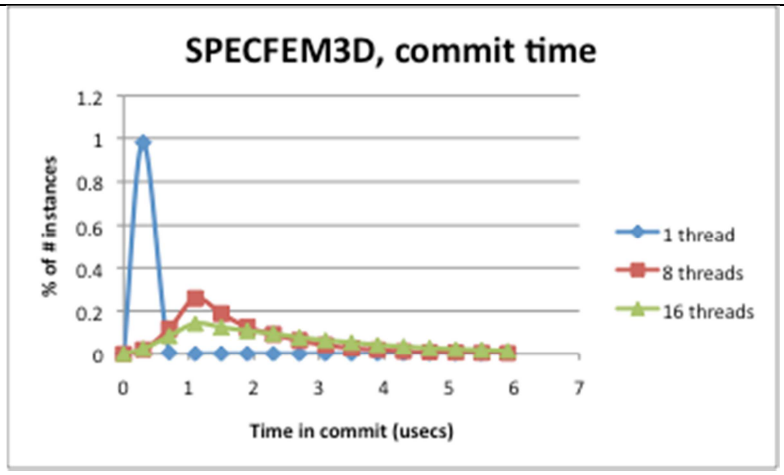


Figure 5 Characterization of commit time

For all the applications we observe a similar behavior, with the time increasing when increasing the number of threads. This may represent a problem when scaling to large number of threads. It would be interesting to understand how this time would evolve when hardware support is provided.

Finally, next figure shows the evolution of the number of aborts for the matrix multiply case. It is the only case that it is shown since for the Queens case, no aborts are observed and for the SPECFEM3D case, although the number of aborts increase with the number of threads, these are scattered without a defined trend, with values varying between 1.5 and 17 microseconds.

In this case, the histogram of the number of instances per a given period time is shown to illustrate that not only the duration of the aborts increase with the number of threads but also the number of aborts. The duration of the abort is significantly larger than the time in transaction or in abort, therefore it would be more important to improve this time when implementing hardware solutions.

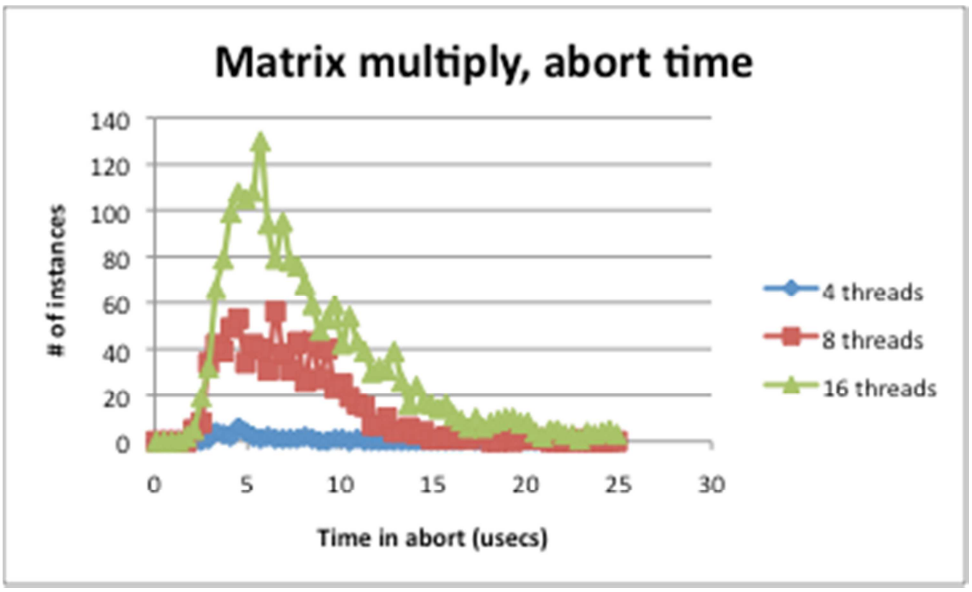


Figure 6 Characterization of abort time

3.5 TFlux Data-flow model (UCY)

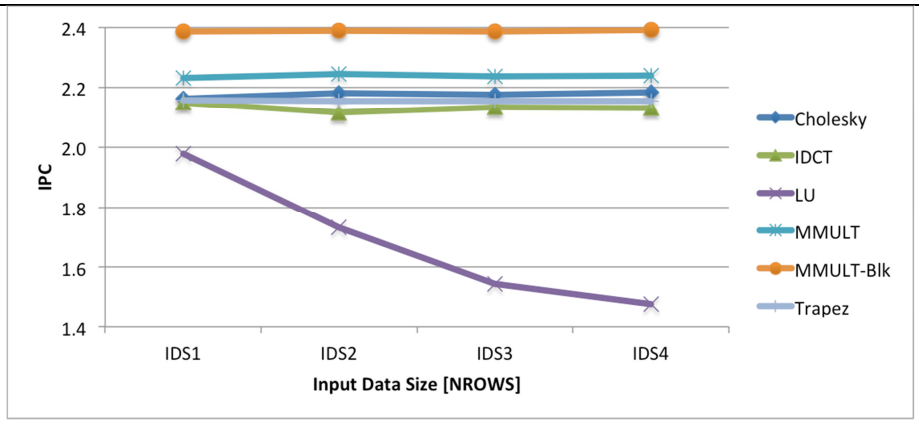
During this period we continued our effort in porting more applications to the Data-flow model. We did this by analyzing the application code and then augmenting the code with the TFlux pragma directives (see Deliverable 3.2). We have now ported the following applications:

- Cholesky: This is a blocked Cholesky decomposition. The reference code is from the SMPSs implementation (http://www.bsc.es/plantillaH.php?cat_id=425)
- IDCT: This is an implementation of the inverse discrete cosine transform algorithm. The code is based on the IDCT kernel from the mpeg library code similar to one found in: http://www.irisa.fr/master/COURS/CAPS/CoursCD/HTML/Codes/ExercicesScap/exercice9/idct.c
- LU: This is an application implementing the blocked LU decomposition algorithm. The reference code can be found in the examples from CellSs (http://www.bsc.es/plantillaH.php?cat_id=421)
- MMULT: This is a matrix multiply application. We present results for two versions, the original and the blocked.
- Trapez: This is an application that implements the Trapezoidal Rule of Integration.
- LEE: Lee's routing algorithm guarantees to find a shortest interconnection between two points using the Expansion-Backtracking technique. It is used primarily in the process of producing an automated interconnection of electronic components. The reference code can be found in the STAMP benchmark suite.

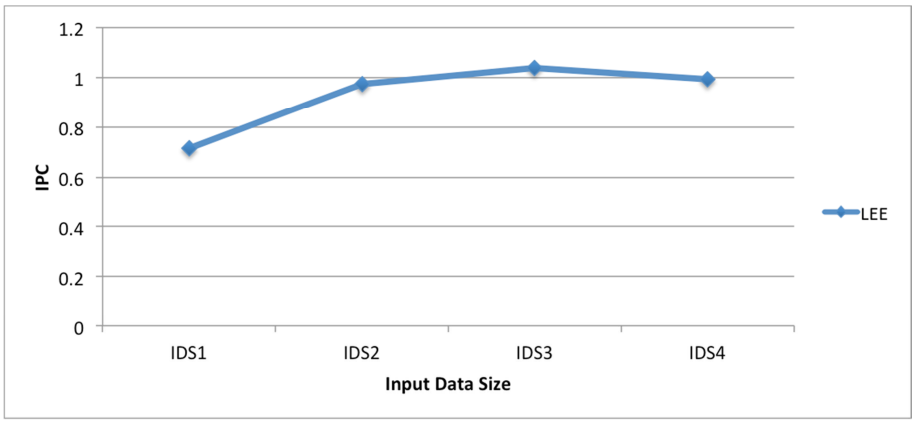
Notice that the last application, LEE, was ported to our data-flow model using the new TFlux+TM pragma directives (see Deliverable 3.3). This was a collaboration effort between UCY and UNIMAN.

3.5.1 Characterization

The applications presented in the previous section, as mentioned before, have been ported to the data-flow model using the TFlux pragma directives. The applications were then passed through our source-to-source preprocessor and then compiled with the regular *gcc* compiler (version 4.5.2). They are then linked with our Data-Driven Multithreading runtime system. They were then executed on an 8 core Intel Xeon E5320 1.86GHz system. Given that we are running natively using our software implementation of the model, we use only 7 out of the 8 cores for worker threads while the other core is reserved for the execution of the software TSU. The executions were performed with 4 different input data set sizes, which we call IDS1, IDS2, IDS3, and IDS4. For Cholesky, IDCT, LU, MMULT, and MMULT-Blk, these input data set sizes correspond to 512, 1024, 2048, and 4096 rows in the matrix, respectively. For Trapez the sizes correspond to a range of 25, 50, 100, and 150. For LEE the sizes correspond to a maze of dimension 32x32x3, 64x64x3, 128x128,x3, and 256x256x3. We collected the statistics using the *perf* system tool. This tool uses the system's program counters to measure different events. For this work we collected enough information to report the IPC, Last-Level Cache and TLB miss rate, and Bandwidth. We present the different results in the following charts.



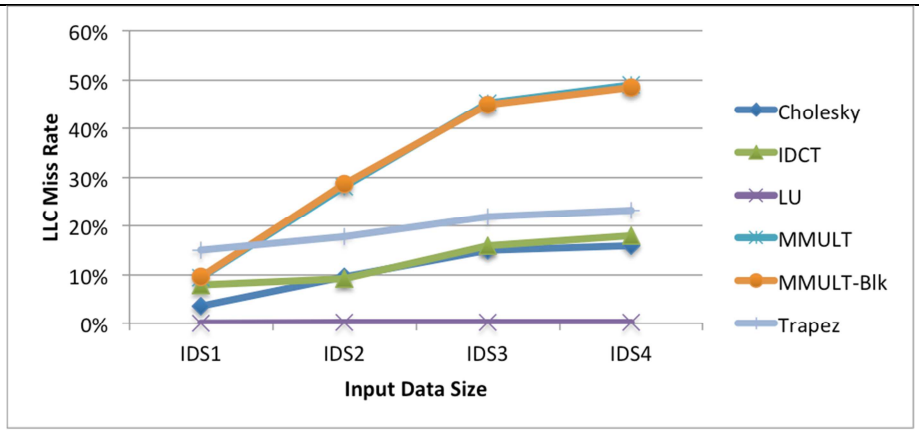
(a)



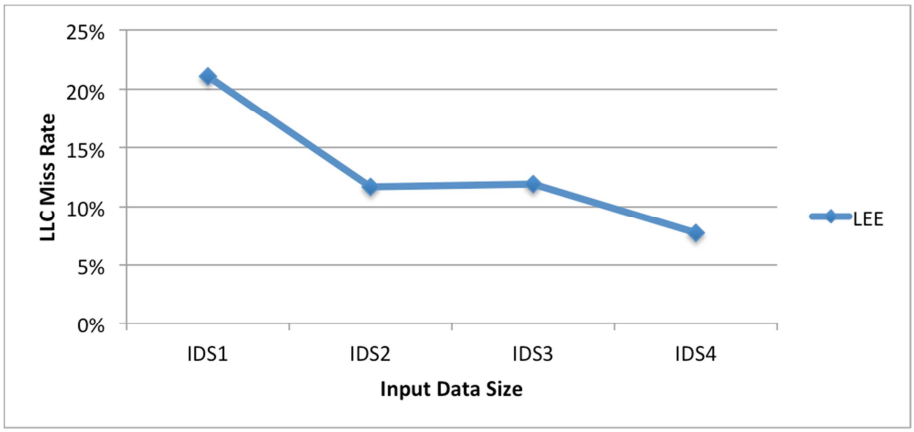
(b)

Figure 7 IPC for different input data sets for (a) data-flow and (b) data-flow+TM applications

From Figure 7, which presents the IPC for the different applications and the different input data set sizes, it is possible to observe that all applications other than LU show a near constant IPC for the different input data sets. For LU we observe a small decrease in the IPC as the input data set is increased. This shows that LU becomes less efficient as the input data set is increased. As for LEE we observe the opposite of LU with a slight increase of the IPC as the input data set sizes increases. The changes in IPC for both LU and LEE are too small to be representative.



(a)



(b)

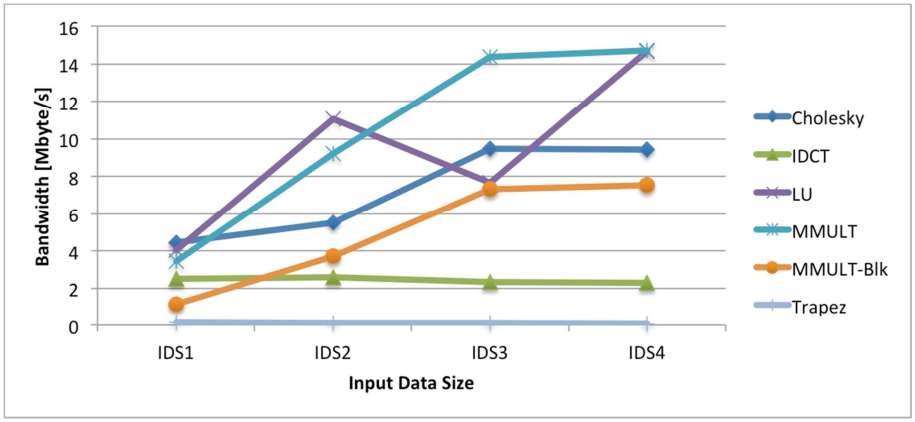
Figure 8 LLC cache misses rate for different input data sets for (a) data-flow and (b) data-flow+TM applications

In Figure 8 we present the LLC miss rate for different input data sets. From these results it is possible to observe a general increase in the miss rate as the input sets increase. This increase is observed in a larger degree for both MMULT applications. This showing that LLC cache is not able to capture the working set. The input data set size does increase more than linearly from set to set does the relative increase in the miss rate is actually relatively smaller than the rate of increase of the input data.

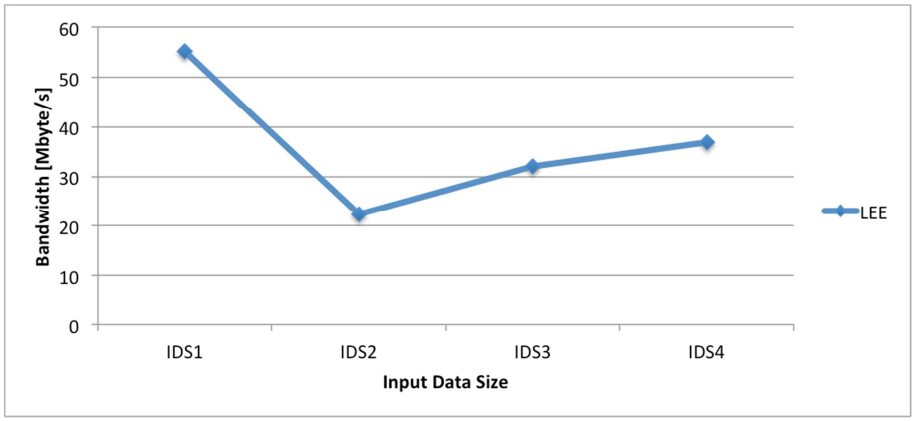
Interestingly, the behavior shown by LEE is the opposite as its miss rate decreases as the input set increases. This may be a consequence of the fact that the core of the data used by this application does not increase as the input data increases. It may also be that the cache is not used efficiently to start with for this application. We need to further study the reason for these observed behaviors.

Notice that even though we have measured also the data TLB miss rate, as the values observed were very small (<0.1%) we did not consider relevant to include in this report.

Finally, in Figure 9 we present the Bandwidth for the different input data sets.



(a)



(b)

Figure 9 Bandwidth for different input data sets for (a) data-flow and (b) data-flow+TM applications

The results in Figure 9 show that the applications in general increase their bandwidth demands as the input data set sizes are increased, as expected. The bandwidth requirement is larger than 14Mbps for both the original MMULT and LU for the larger data set size. Between the two MMULT versions it is possible to observe the blocked version as expected has much smaller bandwidth requirement, around 8Mbps instead of 14Mbps for the larger input data set size. Finally, IDCT does not show a large bandwidth requirement neither it changes for different input data set sizes. Regarding LEE, it again shows a different behavior. First, its bandwidth requirement is much larger than the rest of the applications ranging from 20 to 55 Mbps. Second, its highest point is for the smallest data set even though after IDS2 it starts to increase again as expected. Once more we need to take a further look into this application as to better understand its behavior.

4 Applications porting

Although porting of applications was planned for the second half of the project, some of the partners have already started the porting of applications to the corresponding programming models. Some of the more relevant porting efforts to Scala and StarSs are reported in this section.

4.1 Scala (UNIMAN)

This section presents the progress in porting benchmarks to Scala using dataflow and TM. Although these benchmarks are also being ported to C plus pragmas, we are exploring the use of a programming notation based on the Scala language to cater for high productivity software developers.

When writing in a dataflow style, we have found useful to have certain language features to ease dataflow (functional) composition. The ability to return tuples as function results and to match against them as arguments is probably the most important. The strong typing of Scala also facilitates the static detection of errors.

We use only a subset of Scala omitting both conventional synchronisation and arbitrary shared state manipulation. Shared state manipulation only occurs within atomic sections (TM). We have developed a new Scala-based dataflow library which we use in conjunction with a Scala-based Transactional Memory library (see more information in TERAFLUX Deliverable 3.3).

During this year we have successfully ported: matrix-matrix multiplication (no TM needed), Vacation, Lee Routing (Lee-TM, i.e., labyrinth) and KMeans. We have not done performance tuning of the libraries nor of the benchmarks yet. For Lee-TM we have attempted several approaches to stretch the infrastructure being developed.

Lee-TM is a circuit router that makes connections automatically between points. Routing is performed on a 3D grid that is implemented as a multidimensional array, and each array element is called a grid cell. The application loads connections (as pairs of spatial coordinates) from an input file, sorts them into ascending length order (to reduce ‘spaghetti’ routing), and then loads them into thread-local queues in a round-robin manner. Each thread then attempts to find a route from the first point to the second point of each connection by performing a breadth-first search, avoiding any grid cells occupied by previous routings. If a route is found, backtracking lays the route by occupying grid cells. Concurrent routing requires writes to the grid to be performed transactionally. Lee-TM is fully parallel, with conflicts at concurrent read/write or write/write accesses to a grid cell. A second version of Lee-TM has been implemented that uses early release. This version removes grid cells from the readset during the breadthfirst search. Two transactions may be routable in parallel, i.e. the set of grid cells occupied by their routes does not overlap, but because of their spatial locality, the breadth-first search of one transaction reads grid cells to which the second transaction writes its route, thus causing a read/write conflict. Removing grid cells from the readset during the breadth-first search eliminates such false-positive conflicts.

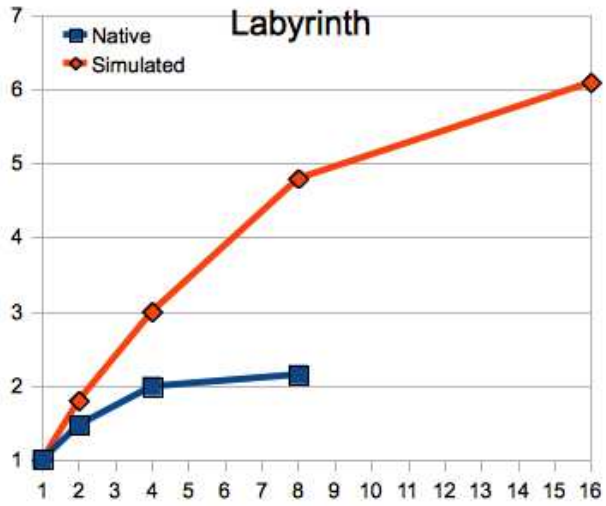


Figure 10 Preliminary results on Scala for Lee-TM (Scalability with number of cores).

Vacation simulates a travel booking database in which multiple threads transactionally book or cancel cars, hotels, and flights on behalf of customers. Threads can also execute changes in the availability of cars, hotels, and flights transactionally. Each customer has a linked list holding his reservations. The execution of Vacation is completely parallel, but available parallelism is limited by the number of relations in the database and the number of customers.

Experimental Setup: Having constructed each of these benchmarks with our Dataflow and Transactional Memory libraries, we present some performance results. The results are obtained on a 12 core machine (2x AMD Opteron (Six-Core) Model 2427), with 32 Gigabytes of RAM (8x 4GB 667MHz DDR2). Scala version 2.9 is used, with the Java SE Runtime Environment 1.6, using the Hotspot 64 bit server VM. We have also used trace based simulation to produce expected performance figures for executing these codes on a processor with support for dataflow and transactional memory. The simulation assumes idealised dataflow hardware with no latency for the passing of tokens. Threads are scheduled immediately after their inputs are ready, and a free processor is available. For TM, we selected a non-optimal scenario. When a transaction conflicts with one already running, this is handled conservatively by delaying the start of the transaction so that there is no overlap.

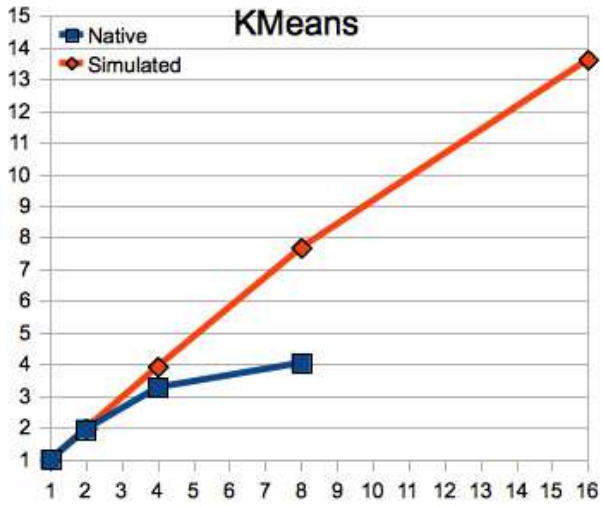


Figure 11 Preliminary results on Scala for Kmeans (Scalability with number of cores).

Figure 11 presents the first results for KMeans for a non-optimized scenario. We do not want to make any claims yet about the scalability (although it is looking promising), but simply illustrate the progress that we are making with porting applications.

Figure 12 presents a graphical representation of a subset of the dataflow graph executed by KMeans. This graph is generated automatically by the dataflow library implemented in Scala to aid software developers. **KMeans** clusters objects into a specified number of clusters. The application loads objects from an input file, and then works in two alternating phases. One phase allocates objects to their nearest cluster (initially cluster centers are assigned randomly). The other phase re-calculates cluster centers based on the mean of the objects in each cluster. Execution repeatedly alternates between the two phases until two consecutive iterations generate, within a specified threshold, similar cluster assignments. Assignment of an object to a cluster is done transactionally, thus parallelism is controlled by the number of clusters. Execution consists of the parallel phase assigning objects to clusters, and the serial phase checking the variation between the current assignment and the previous.

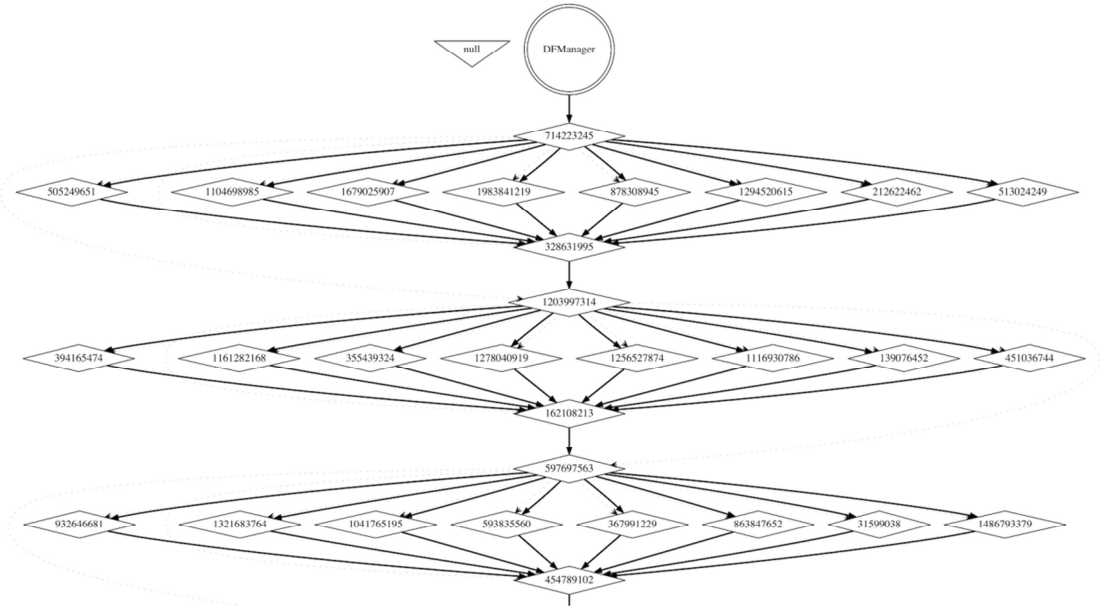


Figure 12 A subset of the executed dataflow graph for KMeans

4.2 Improvements of the implementation of STAP in StarSs

The STAP (Space Time Adaptive Processing) application comes from airborne radar domain. It provides simplified implementation of MTI (Moving Target Indication). The original sequential implementation was provided by Thales in C, and a first implementation in StarSs was also written. This section analyzes the performance of this initial implementation and how it was optimized.

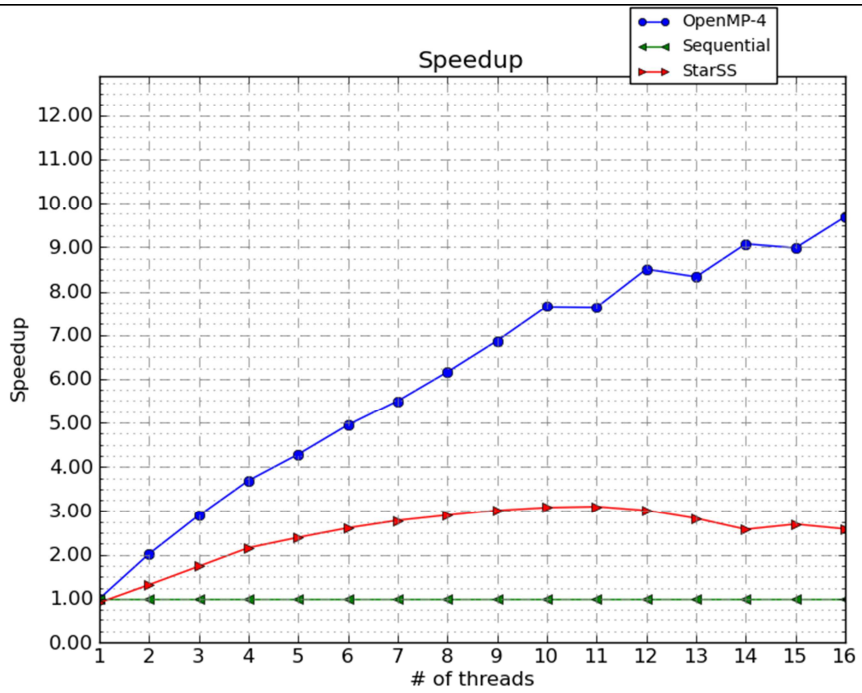
Three different versions of the STAP algorithm are used in this section:

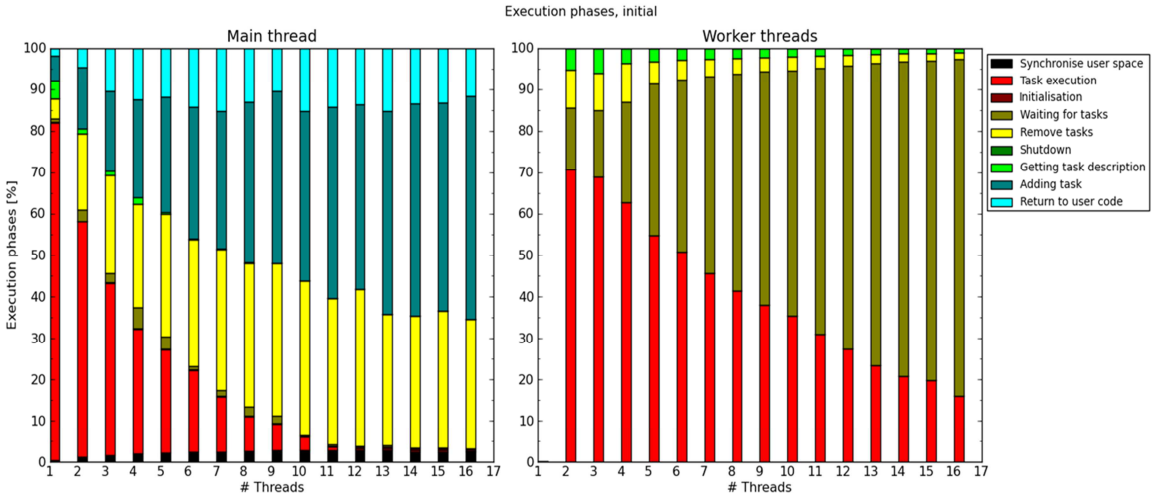
- An OpenMP-based implementation with parallel loops
- A StarSs-based implementation. It encapsulates functions that operate on data representing signals into tasks.
- A sequential implementation.

4.2.1 Initial Results

The initial measurements show that the OpenMP implementation of STAP outperforms StarSs implementation. Already the StarSs using a single thread is slower than the sequential implementation. The reason for this is that StarSS implementation suffers from the overhead introduced by the StarSs runtime. Such phenomenon is not visible for OpenMP implementation.

The largest difference between OpenMP and StarSS versions is visible in case of execution for 16 threads. The StarSs has an speedup of 2.0 against the sequential case, while the OpenMP implementation has an speedup larger than 9.5.





In the plot above showing the execution phases of the StarSs version it can be observed that the time the application spends executing the tasks is inversely proportional to the number of threads. This is visible for both main thread and worker threads. For the main thread, the task execution drops to more than 80% for a single-threaded execution to 0% for execution with 12 threads. That means that for a number of threads larger than 11, the main thread is completely busy managing and synchronizing tasks.

For worker threads this ratio drops from 70% for two thread execution to 15% for an execution with 16 threads. In this case the threads keep idling waiting for tasks being scheduled by the main thread.

4.2.2 Optimization

The optimization we applied to the StarSs implementation of STAP increases the granularity of the tasks with the objective of decreasing the relative overhead in the runtime.

- For example, tasks are generated inside loops that iterate over data that is further passed to generated tasks. Let's look at the call site of task `X_3_1:`

```
1. for (i=0; i<dim1; i++) {
2.     for (j=0; j<dim2; j++) {
3.         X_3_1(dim3, a[i][j], b[j][i]);
4.     }
5. }
```

  We can see that the body of the nested loops in line 3 is executed dim1*dim2 times. It means that if we change value loop variable is altered by, we change the number of tasks generated during loop execution: if loop variable i is altered by n, number of generated tasks is dim1*dim2/n. Aforementioned piece of pseudocode would look as follows:

```
1. for (i=0; i<dim1; i+=n) {
2.     for (j=0; j<dim2; j++) {
3.         X_3_1(n, dim2, dim3, &a[i][j], &b[j][i]);
4.     }
5. }
```

  This change groups data that is processed by the task into tiles; size of data increases n times.

- Tasks perform computation by looping over data structured into multidimensional arrays. Now let's look at the body of task `X_3_1`:

```
1. #pragma css task input(dim3, a) output(b)
2. void X_3_1(int dim3, Cplfloat a[dim3], Cplfloat b[dim3]) {
3.     int k;
4.     for (k=0; k<dim3; k++) {
5.         b[k].re = a[k].re;
6.         b[k].im = a[k].im;
7.     }
8. }
```

  We see here that task's execution time depends on number of iterations of the loop from line 4. So, if we alter size of iteration space we are able to change the duration of task's instance. This is achieved by increasing size of data task processes. This in turn can be achieved by applying changes from previous bullet. This group of changes applied to code above results in the following pseudocode:

```
1. #pragma css task input(n, dim2, dim3, a) output(b)
2. void X_3_1(int n, int dim2, int dim3
3.            , Cplfloat a[1][dim3], Cplfloat b[1][dim3]) {
4.     int i, j;
5.     for (i=0; i<n; i++) {
6.         for (j=0; j<dim3; j++) {
7.             (*(b+i))[j].re = (*(a+i*dim2))[j].re;
8.             (*(b+i))[j].im = (*(a+i*dim2))[j].im;
9.         }
10.    }
11.}
```

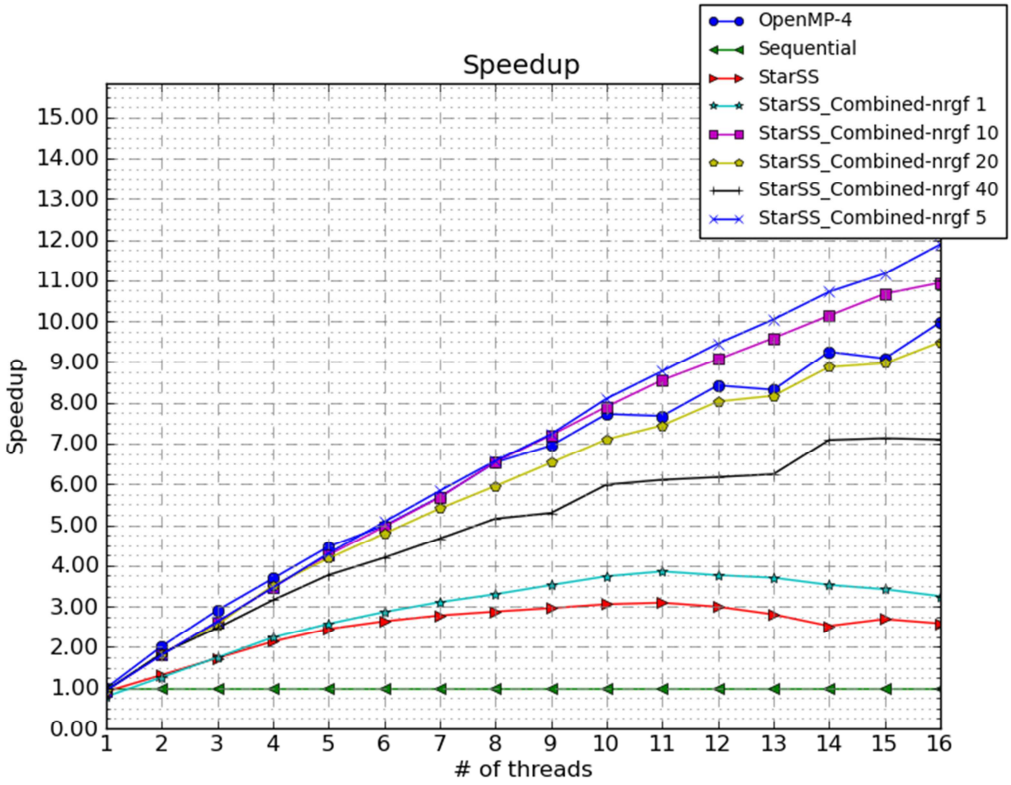
What happens in this piece of code is that body of the loops is now executed n*dim3 by changing size of data processed times n.

Optimization has been applied to all the tasks in StarSs-based implementation of STAP algorithm. In the implementation, the values that alter loop counters and change size of data processed by tasks are called `delta` and `delta_nrfg`.

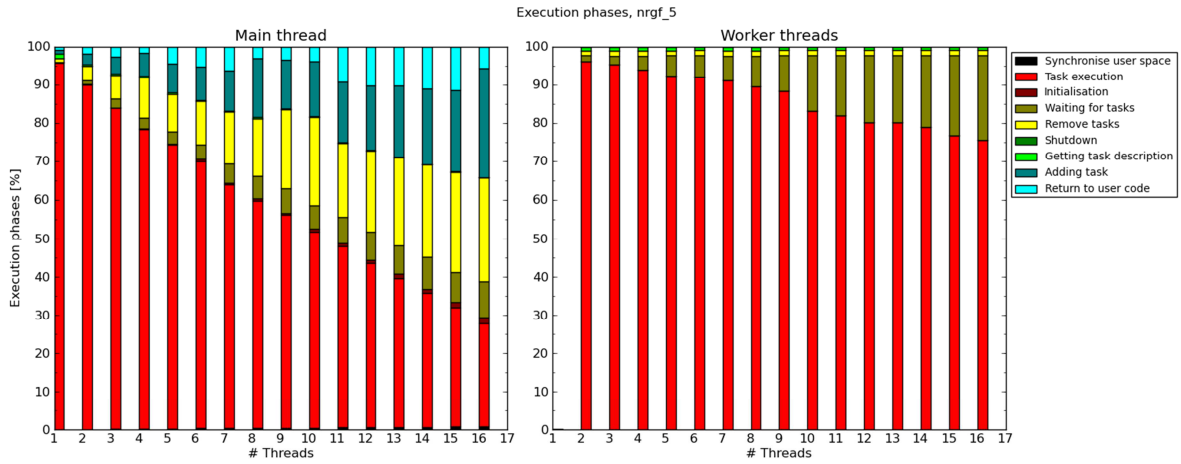
For each task, different values of `delta` and `delta_nrfg` were evaluated experimentally.

4.2.3 Final Results

After applying the optimization, the speedup improved visibly (see figure below). For the case of the optimized application the best speedup we achieved is almost 12 for 16 threads. This result was reached for StarSs implementation configured with `delta_nrgf` set to 5. It outperforms OpenMP implementation that reaches speedup of 10 for 16 threads.



Execution phases for the StarSS-based STAP implementation configured with `delta_nrgf` set to 5 also note improvement comparing to non-optimized implementation. This is shown on the following plot:



For 16 threads, the main thread contributes in tasks' execution, it is less busy with generating and scheduling them. The improvement introduced by the optimization also becomes visible if we compare single-threaded executions: in case of non-optimized implementation, the task execution was performed for 82% of lifetime of main thread; in case of the optimized implementation the task execution takes more than 95% of the execution time.

Improvement is also visible for worker threads. In the 16-thread execution, the workers spend 75% of time executing tasks; in case of non-optimized implementation, for the same number of threads, task execution reaches 15% and waiting for task to execute becomes threads' main activity. In optimized implementation waiting for tasks covers less than 30% of running time.

5 Conclusions

This document describes the results for WP2 of the TERAFLUX project in the second year. The activities related to the workpackage dealt with the selection of applications, characterization of the applications and porting of the applications to the different programming models.

Although the project has now a consolidated list of benchmarks, kernels and applications, the addition of new ones under request of partners or other stakeholders in the project are not discarded.

With regard to the characterization, relevant results have been obtained in terms of requirements for the characteristics of the underlying hardware. For example, aspects related to the requirements on memory bandwidth and latency, or to the requirements for performance of TM have been described.

Also, several efforts to port the applications to the project programming models are ongoing. Indeed, some groups are already working on the optimization of these applications based on the results of performance analysis.

Next year, the project partners will continue these porting activities with also performance analysis evaluation of the applications using the underlying architecture designed in the project.

References

[1] http://www.netlib.org/linpack/
[2] http://www.top500.org
[3] http://www.graph500.org
[4] Milan Pavlovic, Yoav Etsion, and Alex Ramirez. "On the Memory System Requirements of Future Scientific Applications: Four Case Studies", In IEEE Intl. Symp. on Workload Characterization (IISWC), Nov 2011.