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

D3.4 – Initial report comparing and contrasting the developed models

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:

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#	Author	Organization	Change History
0.1	Mikel Lujan	UNIMAN	Initial template
0.2	Mikel Lujan	UNIMAN	Document structure
0.3	Mikel Lujan	UNIMAN	Added contribution from UNIMAN
0.4	Mikel Lujan	UNIMAN	Added contribution from UCY
0.5	Mikel Lujan	UNIMAN	Added contribution from CAPS
0.6	Mikel Lujan	UNIMAN	Added contribution from BSC
0.7	Mikel Lujan	UNIMAN	Added contribution from INRIA
0.8	Mikel Lujan	UNIMAN	Improved contribution from UCY
0.9	Mikel Lujan	UNIMAN	Improved contribution from BSC
1.0	Mikel Lujan	UNIMAN	Improved contribution from INRIA
1.1	Mikel Lujan	UNIMAN	Added executive summary & improved text

			for cohesion.
1.2	Mikel Lujan	UNIMAN	Improvements suggested
1.3	Mikel Lujan	UNIMAN	Author names
1.4	Mikel Lujan	UNIMAN	Internal feedback and improved executive summary.

Release Approval

Name	**Role**	**Date**
Mikel Lujan	Originator	**05-12-2012**
Mikel Lujan	WP leader	**12-12-2012**
Roberto Giorgi	Project Coordinator for formal deliverable	**14-12-2012**

TABLE OF CONTENT

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

Daniel Goodman, Chris Seaton, Salman Khan, Behram Khan, Mikel Luján, Ian Watson
University of Manchester

Albert Cohen, Léonard Gérard, Antoniu Pop
INRIA

Lefteris Eleftheriades, Natalie Masrujeh, George Michael, Lambros Petrou, Andreas Diavastos, Pedro Trancoso, Skevos Evripidou
University of Cyprus

Rahul Gayatri, Rosa M. Badia, Eduard Ayguadé
BSC

Laurent Morin, Stéphane Bihan, François Bodin
CAPS Enterprise

Glossary

TM	Transactional Memory
Dataflow computation	A dataflow computation is defined by a graph where the nodes are side-effect-free computations (functional computation) and the arcs represent dependencies. A node is activated and executed when its input dependencies have been satisfied, generating seamlessly parallel execution.
Transaction	A set of individual operations that need to be executed atomically, with guarantees of consistency and isolation
Atomicity	Transactions must appear to other transactions as if they occur in a single operation, or do not occur at all.
Consistency	One transaction must take the program from one consistent state to another.
Isolation	Transactions must act on isolation of each other.
TM mechanisms	The implementation of a TM system normally requires a means for detecting conflicts among executing transactions, and a means for versioning data used within a transaction to allow restoring the system state back to its origin should one or more transactions conflict.
Conflict	Two transactions conflict when the two transactions cannot be executed in parallel preserving the atomicity, consistency and isolation properties. There are data dependencies across the transactions (e.g. read-after-write or write-after-write) which would invalidate the parallel execution of those two transactions
Eager conflict detection	The TM system has a choice about when to check whether a number of transactions have a conflict. Eager attempts to detect the conflict during the execution of the transaction.
Lazy conflict detection	Lazy attempts to detect conflicts among the executing transactions when one of these attempts to commit.
Eager versioning	Eager versioning modifies directly memory and requires an undo log to restore the original state.
Lazy versioning	Lazy versioning buffers memory modifications done by a transactions and only once such transaction is allowed to commit, these modifications are propagated to memory visible by other threads.
Nested transaction	A transaction is nested when its execution is contained within the context of another transaction. Flattening treats the nested transactions as a merged single transaction. Open nesting has been proposed as a means to reducing unnecessary conflicts by allowing nested transactions to commit before their parent transaction has been done so.
Strong vs weak isolation	Strong isolation is where nothing can see the state within a transaction while it is executing. Weak isolation is where only other transactions are unable to see intermediate state, but other threads will not be prevented by the programming model from viewing the intermediate state.

Executive Summary

This report contains descriptions of the current state of work within the programming model development, and is split into three parts covering the high productivity model, the synchronous dataflow model and the high performance models.

The specific achievements and discussions are:
High Productivity Model – Scala (Section 3)

- Manchester University Transactions for Scala (MUTS) has been published in JPDC 2012.
- Dataflow Scala library (DFScala) improved and made available as open-source; published in DFM2012.
- Work analyzing how to combine dataflow and transactional memory for Lee's algorithm is recognized as best paper award in MULTIPROG 2012.

Synchronous Dataflow (Section 4)

- INRIA extended the data-flow synchronous Heptagon language with futures.
- The language comes with a formal proof of semantics preservation (elision of the asynchronous/parallelization annotations), and a compilation method extending the classical «compilation of parallelism» of synchronous languages. Best paper award at EMSOFT 2012.

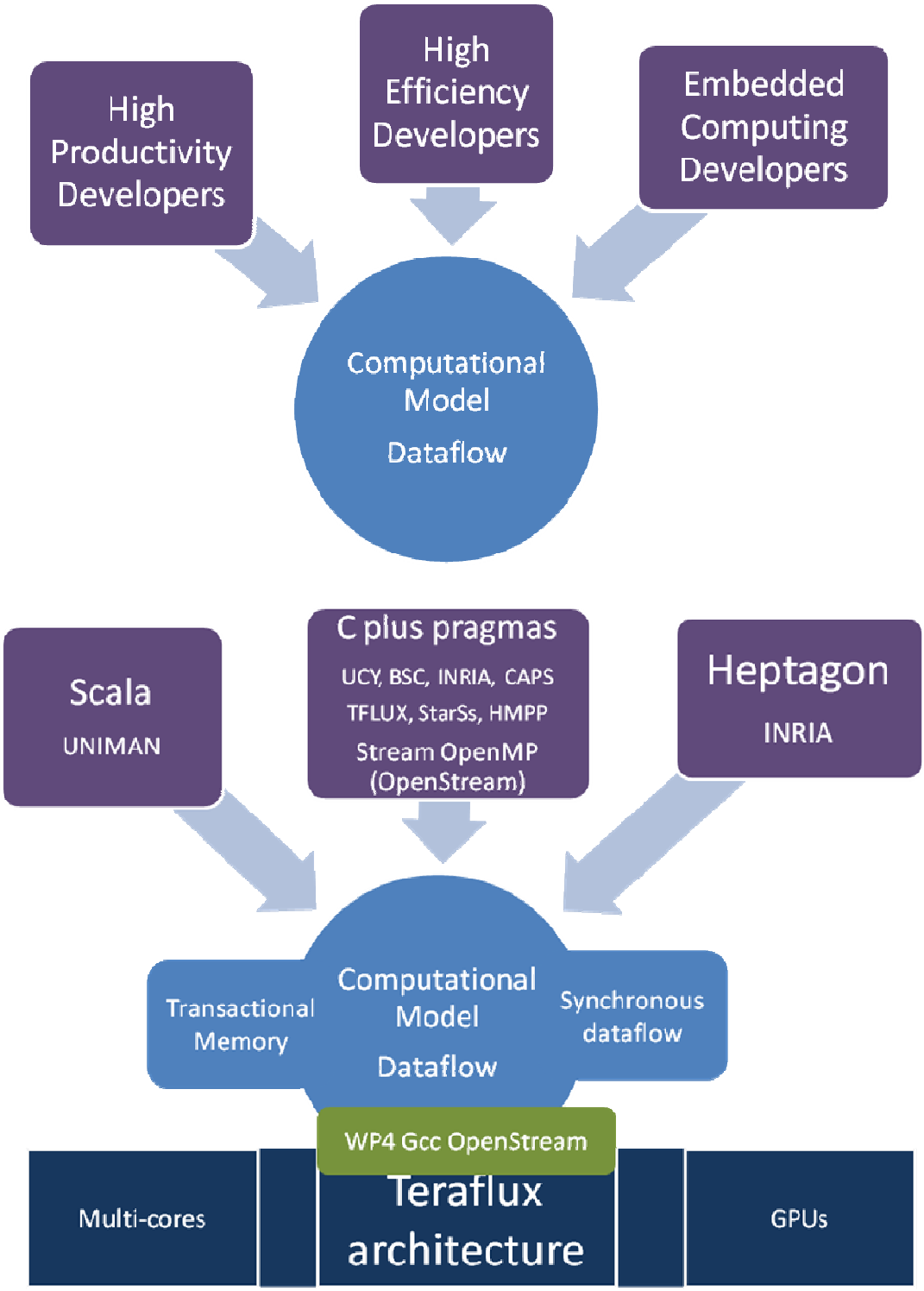
High Performance Model – C directives (Section 5)

- StarSs (from BSC) has extended their compiler and runtime system to support speculation within while-loops, generating promising initial results for the Graph500 benchmark.
- TFLUX (from UCY) has contrasted TFlux DDM with the PLASMA based programming model. This led to a publication at the Data-Flow Programming Models for Extreme Computing (DFM-2012).
- HMPP (from CAPS) has a proposal for their pragma directives to support dataflow programming on GPUs.
- INRIA has extended the streaming data-flow extensions of OpenMP, now called OpenStream, with first class streams, dynamic task creation, and modular compilation.

Overall, the programming models, formal description and supporting tools are maturing and this deliverable provides an intermediate checkpoint of progress towards the final deliverable for this workpackage due next year. A large amount of research has been carried out in these three years in WP3 and we start to observe common aspects among the different programming models. The creation of the task graph is supported with different syntax but the core functionality of describing a side effect free computation as a node in the graph is prevalent. The inputs and outputs are specifically annotated and permit the generation of the dataflow graph. We can observe a divergence on how rich the set of in-built dependencies each programming model provides specific support for. This divergence is not related with whether the dataflow graph generated is general, but is associated with covering well certain patterns of dependencies and the level of sophistication expected from the compiler when a pragma is encountered. The work with HMPP provides an industrial perspective of what features/functionalities are well understood.

1 Introduction

This document is an update on the work carried out in WP3. It is split into three distinct sections covering the work carried out on the high productivity programming model (Scala), on the synchronous concurrency (Heptagon) and on high performance models. Within the latter models, we cover progress with C-directive-based dataflow models (StarSs, TFLUX, HMPP, OpenStream).



1.1 Relation to other deliverables

This deliverable describes the existing work carried out to extend and implement dataflow and transactional models and it is a continuation of D3.1 and D3.2

1.2 Activity referred by this deliverable

This deliverable covers the work being carried out under WP3 in year 3 (i.e. T3.4).

1.3 Summary of previous work

The previous deliverable reported the progress with defining the programming models and the outcome of the initial experiments completed successfully. We had developed working software prototypes able to execute on standard multi-core platforms.

2 Summary of Dataflow and Transactional Memory

We present an executive summary of the decisions taken to combine dataflow and transactional memory. Full details can be found in D3.1, D3.2 & D3.3 and some key terms are defined in the Glossary of this deliverable. Appendix A summarizes the need for shared data in dataflow which motivates combining Transactional Memory and dataflow.

Dataflow Threads

The architecture and semantics is simplified when a transaction executes only within a single thread. Once a good understanding of Transactional Memory and Dataflow has been achieved, we intend to look into weakening these constraints.

Versioning and Conflict Detection

Because the project is fundamentally interested in an extensible system, it is felt that the communication required to provide the global observation needed to implement eager conflict detection coupled with the complexity it adds in order to provide correct execution and progress guarantees mean that it is better to opt for lazy conflict detection. This lazy detection can always be strengthened by checks at specified points within the transaction.

Nesting

Although true closed nested transactions are preferred, due to finite hardware resources and after a given depth, it will be reverted to flattened transactions. The first TM prototypes will implement flattening. Because of its non-intuitive semantics open nested transactions are not an option.

Syntax

Because of its clarity at a programmer level it is intended that TM syntax in the form of atomic blocks will be provided complete with supporting extensions.

Synchronization

In addition to providing atomic blocks it is intended that all forms of non-transactional synchronization construct are excluded as they break the atomicity of transactions.

As an update to these decisions, we note that in WP6 we are investigating how to optimize the detection mechanism by taking advantage of the structure within a node (a set of cores) by having conflict detection options more frequent than lazy.

3 High Productivity Programming Model: Scala

In this section we will describe the work carried out on the development of a high productivity programming model based on extensions to the Scala programming language. Currently this work consists of two main libraries which provide transactional memory and dataflow execution.

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

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

We provide updates on the new developments for these libraries and in particular how they can be applied to Lee's routing algorithm.

3.1 Manchester University Transactions for Scala (MUTS)

In D3.3 we provided a first description of the implementation of software transactional memory in Scala without making modifications to the Scalac compiler. This was possible thanks to a novel mechanism reliant on closures for marking the transactional areas of the code. This removes the need for programmers using this model to use a special version of the Scala compiler, so making our work more widely applicable.

The syntax provided by the closures is very simple and an example can be seen below.

```
// Program code before a transaction
...
// The transaction
val id = atomic {
   threadId += 1
   threadId
}
// More none transactional code
```

We have conducted a survey considering all the main techniques to bring software transactional memory into Scala as well as fully explored the capabilities of our closure-based approach. This has been published as a journal publication [5]. Where possible in the survey we provided references to implementations that instantiate each technique. As part of this survey we documented for the first time several novel techniques developed in the implementation of MUTS for Scala. We ordered the implementation techniques on a scale moving from the least to the most invasive in terms of modifications to the compilation and run-time environment. This showed that, while the less invasive options are easier to implement and more common, they are more verbose and invasive in the codes using them, often requiring changes to the syntax and program structure throughout the code.

There are a wide range of techniques for implementing the frontends of software TMs in Scala. This work through a combination of one or more of the following elements: Library Calls, Annotations, Byte-Code Rewriting and Compiler Modifications. Also Scala's ability to define anonymous functions and to pass them into other functions as arguments (Closures) is important in many Scala software TMs. Table 1 maps the different available techniques to the software TMs that implement these techniques.

Before we look at the different ways of implementing these frontends, we will briefly consider some of the points that these should be rated against. Table 2 provides a summary for each implementation

strategy of the properties that take a discrete value. We can observe that MUTS provides the most complete coverage offering a high productivity means of using TM in Scala.

Java Compatible: Scala is Java compatible; however this does not mean that all frontends are automatically compatible with Java. Some require features of Scala that cannot be described in Java code without a high level of understanding of the workings of the Scala compiler. For example using frontends that require the passing of compiler inserted implicit parameters or the passing of functions as arguments to other functions is problematic in Java code. For techniques that are not Java compatible we note if the technique can be combined with techniques that are Java compatible to allow transactions accessing the same shared state to be constructed in both Scala and Java.

Supports Strong Isolation: If a TM supports strong isolation no action of the programmer or user can result in transactional data being accessed from outside of a transaction. If instead only weak isolation is supported, it is possible to access data being manipulated in a transaction from outside of a transaction. This can then allow the non-committed state of a transaction to be observed and modified, potentially undermining the correctness of the code. Strong isolation can also be broken by the presence of non-transactional mutable data within a transaction; such data may exist as an optimisation.

Supports Legacy Code: The use of code compiled by others is a routine part of writing programs, however not all STMs are able to instrument this code. This means that it is not possible to include within transactions compiled methods that contain side effects. Examples of such methods are those contained within the Scala mutable collections and the Java Util libraries.

No Runtime Modifications: Some STMs form a simple library that can just be added to the classpath, others require the addition of more advanced JVM arguments, or the use of special compilers.

Clarity of Code: Different methods of implementing transactional frontends require different syntaxes. These different syntaxes offer differing levels of complexity and verbosity. Excessive verboseness or complicated syntax can act as a barrier to the use of the transactional frontend, or result in difficulty reading and writing code that uses it.

No Alternative Syntax: A specific point relating to clarity of code is that while some STM's may be less verbose than others, some of these require that the code inside a transaction uses a different syntax to code outside a transaction. For example, an alternative assignment statement is required for some techniques. As the standard assignment operator can still be used, but may not be valid, such changes not only increase the application programmer workload, but also allow for the creation of subtle bugs.

No Duplicate Methods: To call functions from both inside and outside a transaction some STMs required that the user constructs two copies of methods, one that is transactional and one that is not. Others are able to automatically construct a transactional and a non-transactional method from a single piece of user code. The ability to construct both methods from a single piece of code removes the opportunity for discrepancies between the two functions to be added when writing or maintaining the code, as well as reducing the workload for the programmer.

Guarantees Correct Transactions: Some STMs guarantee that all variables that need to be instrumented as part of a transaction will be, while others leave it to the programmer to ensure this. If the programmer fails to add all the required instrumentation then the transaction may no longer be correct and a race condition will have been introduced into the program.

Table 1: Survey of techniques and available systems to add Software Transactional Memory to Scala

Implementation Style		**Implementations**
Pure Libraries	Explicit library calls	—
	Reference cells	Multi-Verse CCSTM ScalaSTM RadonSTM
	Closures and reference cells	CCSTM ScalaSTM
Byte-code rewriting	Class annotations	Multi-Verse
	Method annotations	MUTS
	Closures	MUTS
	Parser Modifications	MUTS
Compiler Modifications		—

Table 2: Summary of approaches and their properties.

	Pure Libraries			Byte-code Rewriting				
	Explicit Library Calls	Reference Cells	Closures and Reference Cells	Class Annotations	Method Annotations	Closures	Parser Modifications	Compiler Modifications
Java Compatible	•	•		•	•			
Combinable with Java Compatible Techniques	-	-	•	-	-	•	•	
Supports Strong Isolation				•				•
Supports Legacy Code				•	•	•	•	
No Runtime Modifications	•	•	•					•
Guarentees Correct Transactions				•	•	•	•	•
No Alternative Syntax	•			•	•	•	•	•
No Duplicate Methods				•	•	•	•	•

3.2 Scala Dataflow Library (DFScala)

To compliment MUTS and to enable the development of dataflow code for a number of the applications selected in WP2 (see D2.3 for performance results and current status of developed applications) we have constructed a library to support the creation and execution of dataflow threads. The design of DFScala has been published in [7] and we have made DFScala available as open source code at

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

In particular, this year we have made improvements for performance, simplified the implementation to use less complex parts of Scala and improve tooling to help with the development of parallel applications with DFScala.

One distinguishing feature of DFScala is the static checking of the dynamically constructed dataflow graph. This static checking ensures that at runtime there will be no mismatch of the arguments to functions. DFScala does not require the usage of special types and thus a node can be generated from any existing Scala function without complex refactoring of code. Each node in the dataflow graph is a function which cannot be subdivided; a function is sequential. To support nested parallelism within a function, subgraphs can be created which are wholly contained within a function, returning a value to the node upon completion.

To improve friendliness, we have made changes that allow simplifying the creation of nodes in the graphs from:

```
val bThread = DFManager.createThread(createGraph _)
bThread.arg1 = b
bThread.arg2 = divisions / 2
bThread.arg4 = routes
bThread.arg5 = solutionsReduceThread.token1
bThread.arg6 = remainingReduceThread.token1
```

to a much shorter version:

```
val bThread = thread(createGraph _)
bThread(b, divisions / 2, Blank, routes,
solutionsReduceThread.token1, remainingReduceThread.token1)
```

The work on tools has focused on providing a simple visualization of the runtime schedule for dataflow graphs (Figure 1).

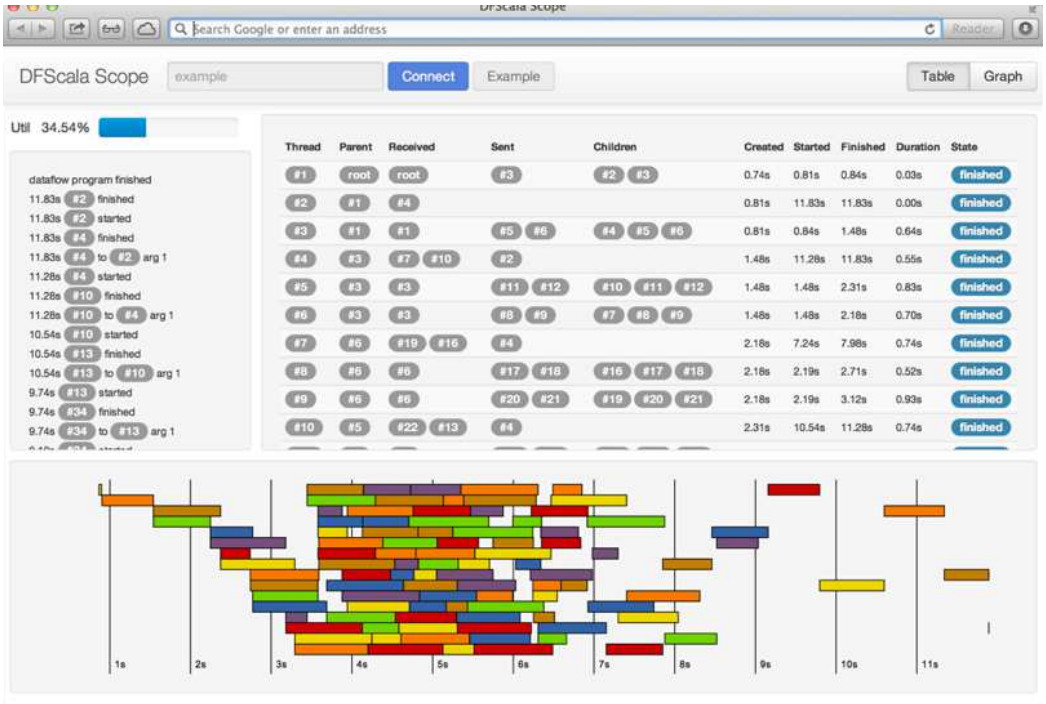


Figure 1: Simple visualization of the runtime schedule for dataflow graphs

3.3 Experiments with Lee's algorithm

To understand how to combine dataflow and transactional memory, we have used an established benchmark, Lee's algorithm for routing printed circuit boards, to make an assessment of their utility for creating efficient, simply written and correct parallel programs. Our experience was published in [6] and was awarded the best paper award.

We have shown initial evidence of the combined model of dataflow and TM making the parallel implementation of our program simpler, at the same time as achieving a real world performance increase compared to coarse locks, even when all overhead is included (MUTS – Software Transactional Memory).

Lee's algorithm solves the problem of finding independent routes between a set of pairs of points on a discrete grid. The applications originally proposed included drawing diagrams, wiring and optimal route finding, but the algorithm is now best known as a method for routing paths on a printed circuit board. There are later algorithms for route finding with less computational complexity, but Lee produces a shortest solution for any given route and board state, as opposed to using heuristics to arrive at a good-enough solution in less time. Figure 2 shows the output of one application of Lee's algorithm - routing paths on a printed circuit board - as generated by our implementation.

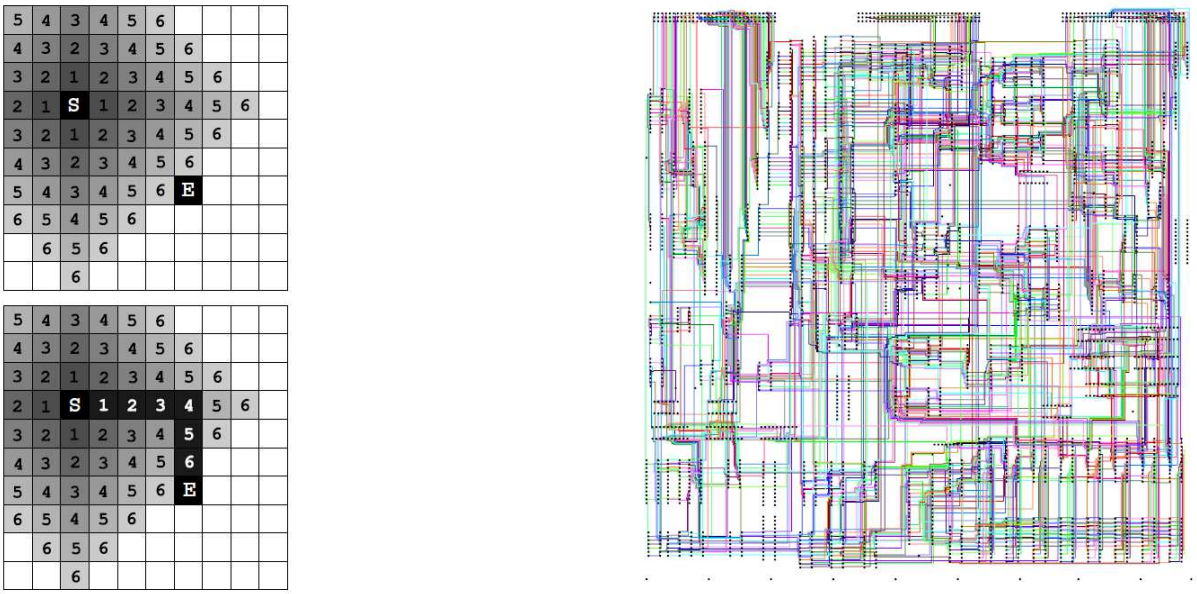


Figure 2: Lee's algorithm

A simple overview of the Lee algorithm is that from the start point of the route it looks at all adjacent points and marks them as having cost one. From each point P so marked, it marks each adjacent point P_{adj} as having cost $cost(P_{adj}) = cost(P) + 1$, as long they were not already marked with a lower cost. If an adjacent point is already part of another route then using that point would cost more, by an arbitrary constant factor, as a bridge of one route over another needs to be built. This expansion, as it is called, continues until the end point is a member of the set of adjacent points. Typically, this expansion forms a circle around the start point, enveloping obstacles such as existing routes, and finishing at the end point. A trace is then run from the end point to the start point, always moving to a

point of lower cost until the start point is reached. This produces a route which can be marked on the grid.

The key to the problem is that any set of subproblems still need to share access to a single resource; the grid. It is not simple for each thread to have an independent copy of the grid, as two threads could need use the same point for multiple routes and would then have to synchronise between themselves. This would add logic to the program that is unrelated to the algorithm that we are implementing. It is also not simple for each thread to have one part of the larger grid, as you cannot guarantee which parts a route will use before the expansion has been calculated and such a scheme would vastly increase the complexity of the program. However, given all routes on a circuit board it is unlikely that any two being routed at a particular time will conflict. The parallelism is there; it is just that it is hard to determine statically and is more apparent as the program is running. Work has already been done to use Lee to evaluate the runtime characteristics of a transactional program [8] and Lee is also known as Labyrinth in the STAMP benchmark suite.

We used the MUTS library to create a parallel implementation by modifying a coarse-lock (coarselock) version. Where the coarselock acquires a resource with exclusion of all other threads, we can instead perform the action inside a transaction that will only allow other threads to read the same values as long as they do not write to them, and will automatically retry if such a conflict is found.

```
// Atomically copy the shared data structure
val privateBoardState = atomic { boardState.freeze }

val expansion = expandRoute(board, route, privateBoardState)
val solution = traceRoute(board, route, expansion)

// Atomically write to the shared data structure
atomic {
     if (verifyRoute(route, solution, boardState))
          layRoute(route, solution, boardState)
     else
     scheduleForRetry(route)
}
```

This transactional version can be transformed into using dataflow (DFScala). The dataflow constructs provided allow us to express this creation of parallelism in a different way. Each route is a DFThread that will have its inputs ready at the start of the program's execution and so will all be runnable. DFScala will schedule them for us so that only a sensible number are running at any time. A final DFThread, the `collector thread' will be then created that has each route's DFThread as one of its arguments. This will therefore be run when the solutions are complete. This is a convenience construct provided by DFScala, as it is expected to be a common pattern, and replaces the synchronisation needed to create the list of solutions and the join operation to wait for all threads to finish that we used in coarselock.

```
// Accepts solutions as arguments and build a list from them
val solutionCollector = DFManager.
createCollectorThread[Solution](routes.length)
```

```
for (route <- routes) {
     // Create a thread to solve a route
     val routeSolver = DFManager.createThread(solveRoute _)
     routeSolver.arg1 = board
     routeSolver.arg2 = route
     routeSolver.arg3 = boardState
     // It will send the solutions to this function
     routeSolver.arg4 = solutionCollector.token1
}
```

The performance results published in [6] and summarized in Figure 3 bellow cover the scenario on desktop multi-cores. For next year, we will expand this analysis of the application and run on the Teraflux architecture and larger many-core systems.

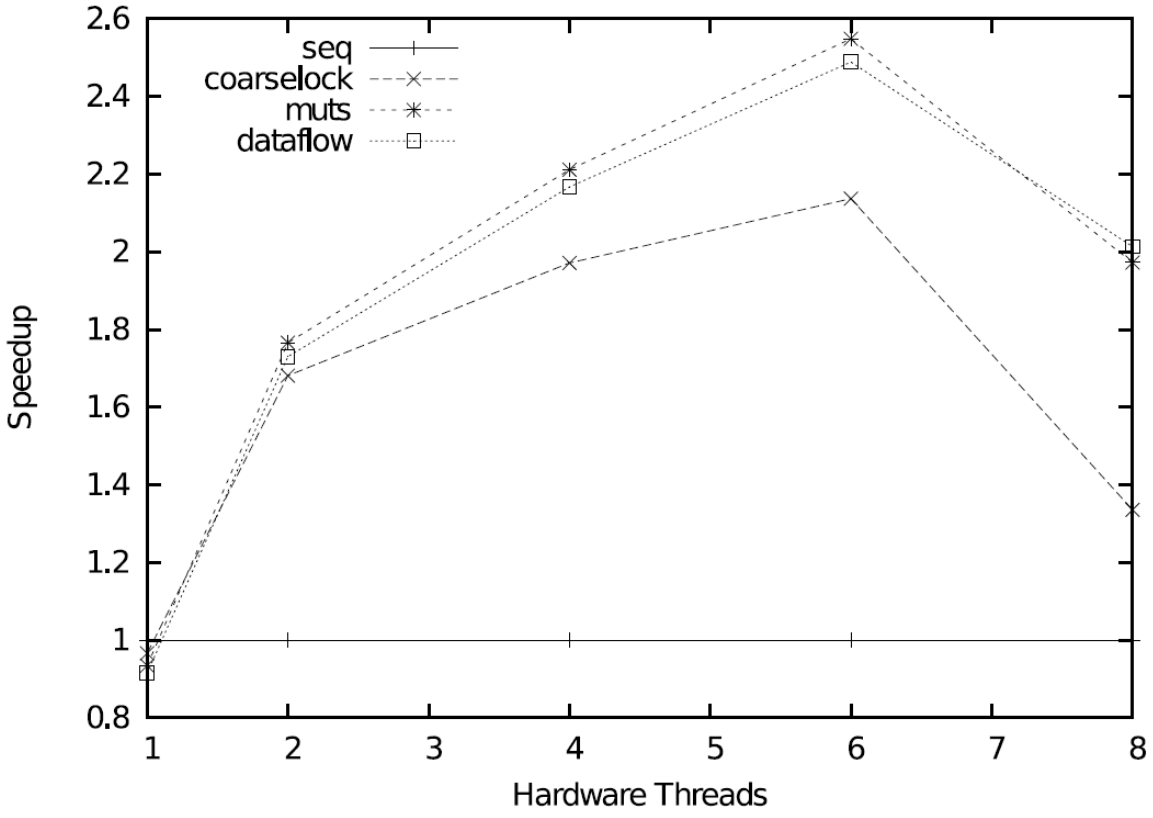


Figure 3: Speedup results for Lee's algorithm. "seq" represents the sequential execution; "coarselock" represents the results obtained with a coarse-grain locking implementation; "muts" represents the results obtained with TM; "dataflow" represents the results obtained when combining dataflow and TM.

4 Synchronous concurrency

Synchronous languages are devoted to the design and implementation of embedded software. They are particularly successful for safety-critical real-time systems. They facilitate the parallel modular specification and formal verification of systems to the generation of target embedded code. The synchronous model is based on the hypothesis of a logical global time scale shared by all processes which compute and communicate with each other instantaneously. This ideal model is then validated by computing the worst case execution time (WCET) of a single reaction. Nonetheless, global logical time may be difficult to preserve when the implementation is done on a parallel machine or performance is an issue. For example, when running a rare but long duration task concurrently with a frequent and faster task, the logical time step could naively be forced to be big enough for the longest task to fit in and short enough to keep up with the frequency of the small task. The classical solution is to decouple these tasks, running the long one across several steps. This is usually stated as the problem of « long duration tasks » in the literature. Several approaches have been considered in the past, always using distribution as a means to decouple the tasks, be it explicit language constructs to call external distributed functions or automatic/guided repartition techniques. The current practice of distribution is mostly manual with no warranty that it preserves the functional behaviour of the model. We believe that decoupling should be explicitly controlled by the programmer, within the synchronous language itself as a programming construct. The distribution will then be done according to this decoupling. The natural expression of decoupling is given by the notion of *future* introduced in Act1 and MultiLisp and present in modern languages like C++11, Java, F#.

A future is the promise of the result of a computation. Whereas a call to f(x) couples the computation of f(x) and the return of the result y, the asynchronous call async f(x) returns instantaneously a future a. Possibly latter on, when the actual result is needed, !a will block until f(x) has finished and return the result y. With the help of futures, we claim that synchronous languages are fit, not only to design the control and computations, but also to program the decoupling and distribution.

Our contributions can be summarized as follows. We consider a Lustre-like language extended with futures and explicit asynchronous function calls. This extension is modular and conservative w.r.t. the base language, in the following sense: a sequence of input/output values of the annotated (asynchronous) program is equal to the one of the not annotated one. In other words, the annotations preserve the original synchronous semantics. The implementation handles futures as a support library. They are treated like any value of an abstract type, the get operation !y is translated to the library one, and an asynchronous call async f(x) is a matter of wrapping it inside a concurrent task, managing inputs, and dealing with the filling of futures. The crucial memory boundedness of synchronous programs is preserved, as well as the ability to generate efficient sequential code for each separate process resulting from the distribution. The distributed program also stays free of deadlocks, livelocks and races.

To our knowledge, the use of futures in a synchronous language is unprecedented. We show via numerous examples how to desynchronize long-running computations, how to express pipelining, fork-join, and data-parallelism patterns. This way, we achieve much higher expressiveness than coordination languages with comparable static properties. In particular, the language captures arbitrary data-dependent control flow and feedback. It also highlights the important reset operator, leveraging rarely exploited sources of data-parallelism in stateful functions.

We demonstrated that such desynchronization can be achieved within bounded memory, and without the need for a garbage collector. This is the first language with futures with such guarantees, making our parallel extension of Heptagon particularly interesting for embedded and real-time applications.

This work was awarded the best paper award at EMSOFT 2012. The detailed semantics, a multitude of examples, and proofs can be found in [3].

A C backend with low-level futures and pthreads is available, complementing our earlier (less efficient) Java backend. For higher performance and integration with the TERAFLUX tool flow, an OpenStream backend is currently being finalized. The precise compilation method and experimental evaluation will be reported in the 4th year of the project.

5 High Performance Developers: C pragmas

5.1 StarSs

BSC is investigating how speculation can be brought into StarSs using Transactional Memory. StarSs, is a task based programming model for widely used Multi-core architectures. The programming model is based on data flow analysis and dynamic data dependency tracking by the runtime. Sometimes in order to extract more parallelism multiple tasks are allowed to simultaneously update memory locations. In such cases lock-based synchronization is used to maintain the correctness of the application. But locks suffer from the drawbacks of deadlock, livelock and priority inversion.
We introduced Software Transactional Memory (STM) based concurrency control mechanism to manage parallel updates. The comparison of results between lock-based approach and STM-based approach shows that applications with high lock contention have better performance with STM based approach [9].

5.1.1 Speculation in StarSs

StarSs provides synchronization constructs such as *"wait-on"*, to wait for a particular memory location to be updated before continuing execution and *"barrier"*, to block execution of all threads till each of them reaches a certain point of execution. Such constructs hamper the parallelism by leading to problems such as blocking of work generation and load balancing. The most common situations where these constructs are used are during if-condition and while-loops. Hence we speculate on the conditions of these loops.

In case of an if-condition such as:

```
T1(a);
//#pragma css wait on(a)
#pragma css speculate wait(a) values(b,c)
if(a)
{
      T2(b);
      T3(c);
}
```

We speculate that the if-condition will be evaluated to true and generate the tasks T2 and T3 inside a transaction instead of waiting for task T1 to finish. Latter when the values of b and c are required we check for the validity of if-condition and either commit the results of b and c or abort transaction.

5.1.2 Speculative execution of loops

The iteration space of while-loops is unknown. The termination condition of *i+1th* iteration depends on values generated in *i-th* iteration. In such cases running multiple iterations in parallel becomes difficult. The parallelism available in such cases boils down to concurrency present in a single loop iteration. We are trying to speculate on the termination condition of the while-loop and spawn multiple iterations speculatively. There are 2 major issues to tackle in this case, 1. If there are cross iteration dependencies, then at best their execution can only be pipelined. Iterations which have been speculatively spawned, but would not appear in the original sequential loop, need to be undone. We use STM, to tackle this issue by executing every iteration of the while-loop inside a transaction. Later

the termination condition is verified and depending on its results, either the values updated in the loop are committed or the transaction is aborted.

Syntax for speculation of loops in StarSs :

```
#pragma css speculate wait(a) values(b,c)
while(a)
{
      T2(b);
      T3(c);
      T1(a);
//#pragma css wait on(a)
}
```

As shown above, just before starting of the loop, we annotate it with the *speculate* pragma, implying that the iterations of this loop can be executed speculatively. The wait clause contains the variables based on which the termination condition of the while loop is evaluated. The *values* clause contains variables whose results have to be protected.

Compiler changes required: StarSs has a source to source compiler, which has been modified to interpret the loop following the speculate pragma as a speculative loop. The compiler creates a guard function which evaluates the loop condition.

```
int css_guard__cssgenerated(void*speculate_params__cssgenerated[1])
{
      return (*((int *) speculate_params__cssgenerated[0]));
}
```

A pointer to this function is passed as a parameter to the task. The variables used in the loop condition are packed into a void array and passed to this function as input. The function returns a value indicating whether the loop condition has evaluated to true or false. These tasks are marked as speculative tasks.

Runtime changes required: In the runtime, an input dependency is added between speculative tasks and the variables involved in evaluating the guard-function. In this way, we ensure that even if the variables involved in guard-function are evaluated by different tasks, their value remains consistent while evaluating the validity of the speculative task. Later inside each thread which executes the speculative task, a transaction is started and a transactional copy is made of the variables passed to the *values* clause. The updates made by the task are made on these transactional copies. After the execution of the task, a check is made using the function pointer to the guard function to evaluate the validity of the task. Depending on this check, the task variables from the *values* clause are either committed of aborted.

In this way, we speculatively execute multiple iterations of the loop, but abort the iterations which do not appear while running the loop sequentially. In order to control the amount of speculation, an environment variable (CSS_SPECULATION_TASKS) variable was added in the StarSs runtime. In the following set experiments that we report it is 10 by default, implying that the 10 iterations of the loop will be executed speculatively and then a *wait* is performed before continuing the execution for another 10 iterations.

5.1.3 First Experimental Results

The proposal has been evaluated with the BFS algorithm of Graph500. The original sequential algorithm iterates over an entire graph using 64 unique search keys. The termination BFS algorithm depends on complete traversal of the graph. The number of iterations needed to achieve this is unknown at compile time, hence a while-loop is used until every node/vertex in the graph has been visited. We speculate on this while-loop and the results obtained are shown in Figure 4.

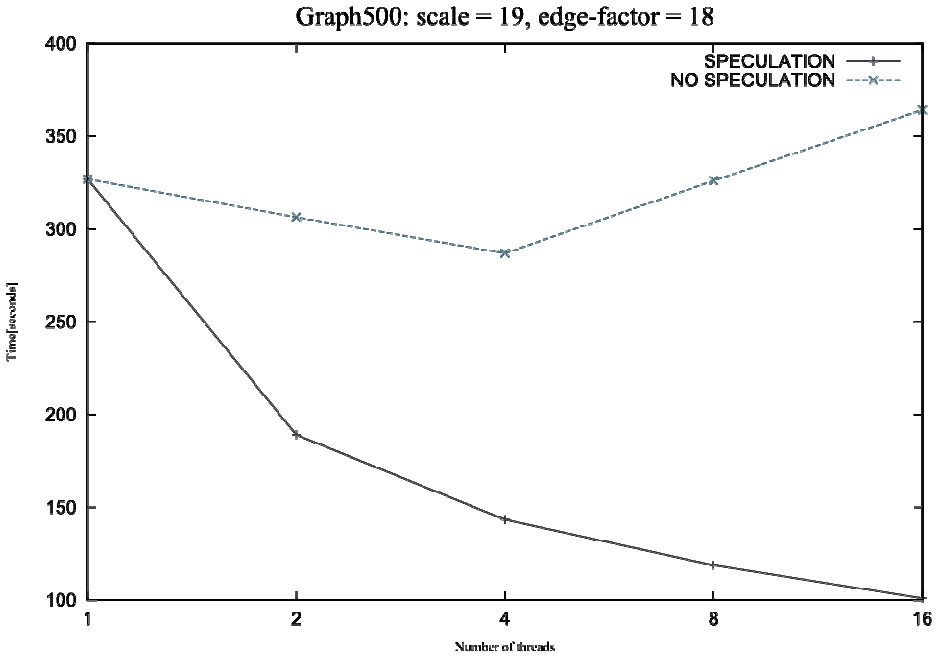


Figure 4 Performance of the Graph500 benchmark with StarSs+TM against regular StarSs. The chart show the benefits of dding TM to StarSs in this application

5.1.4 Summary

Since iteration space of while loops are unknown, we can only speculate on the execution of the loop over multiple iterations. Many irregular applications such as graph traversals and convergence algorithms suffer from this lack of knowledge of iteration space. But we have shown that by speculatively executing iterations of a loop we can extract good levels of parallelism.

5.2 HPC Applications with TFLUX DDM (UCY)

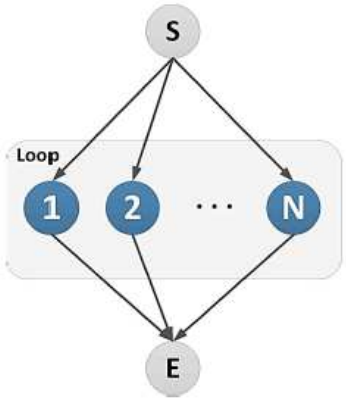
High-Performance Computing (HPC) applications have traditionally been developed using simple parallel models such as OpenMP and/or MPI, or through the use of custom tuned linear algebra libraries. With the revisiting of dataflow models as a way to overcome the limitations of the von Neumann model, we propose to study how to exploit the Tflux Data-Driven Multithreading (DDM) model for the implementation of efficient HPC applications. In this section we wish to contrast [11] the DDM model with the PLASMA based model [10].

As a preliminary evaluation of whether DDM can be a viable candidate for a model to implement HPC applications, we focused on the implementation of three kernel applications: Matrix-Multiply, LU, and Cholesky decomposition. The novelty in implementation for these applications was that we combined DDM code with optimized numerical analysis code. Thus we created D-threads for the operation on different blocks of data and then for the real operations on the data we used the highly optimized numerical analysis kernel libraries. With this approach we want to show that it is possible to get very high performance while using previously tuned code in combination with an efficient scheduling of the threads that operate on data blocks offered by the DDM model.

In Figure 5, Figure 6 and Figure 7 we show the pseudocode and also the thread dependency graph for implementations of each application in this study.

```
for i = 1 to B
  for j = 1 to B
    {load block C(i,j) into cache}
    for k = 1 to B
      {load block A(i,k) into cache}
      {load block B(k,j) into cache}
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
      {do a matrix multiply on blocks}
    {write block C(i,j) back to memory}
```

Consider A,B and C to be N by N matrices of b by b blocks where $b = N/B$



Pseudocode Dependency graph

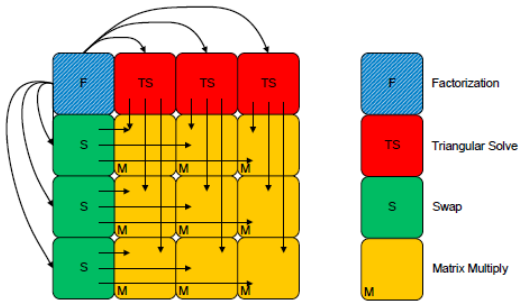
Figure 5 Pseudocode for the blocked matrix-multiply implementation and the thread dependency graph for the same applications

Details for the analysis and implementation of the applications can be found in [1].

We evaluated the applications by executing them on a native multi-core machine using out DDM runtime with a software implementation of the TSU. For more details about TSU, see D6.1 and D6.2. We compare the results obtained with this setup with the execution of the same applications using the optimized PLASMA library. With this work, our objective was to study the scalability of the performance as the number of cores in the system is increased. On Figure 8, Figure 9 and Figure 10 we show the charts with the performance for the three applications for executions with up to 48 cores.

```
for k = 1 to B
  {factorize block A(k,k)}
  for i = k+1 to B
    {triangular solve block A(k,i)}
  for i = k+1 to B
      {apply swap on block A(k,i)}
  for i = k+1 to B
    for j = k+1 to B
      A(i,j) = A(i,j) + A(i,k) * A(k,j)
      {do a matrix multiply on blocks}
```

Consider A to be a N by N matrix of b by b blocks

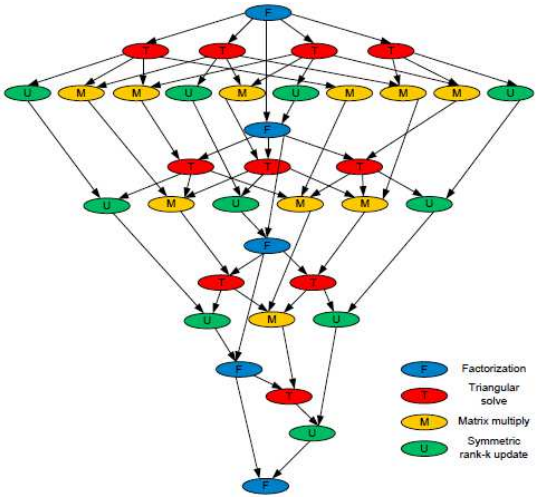


Pseudocode(a) Dependency graph

Figure 6 Pseudocode for the LU decomposition implementation and the thread dependency graph for the same applications

```
for k = 1 to B
  {factorize block A(k,k)}
  for i = k+1 to B
    {triangular solve A(i,k)<--(k,k)}
  for i = k+1 to B
    {symmetric rank-k upadate A(i,i)<--A(i,k
    for j = i+1 to B
        A(j,i) = A(j,i) + A(j,k) * A(i,k)
        {do a matrix multiply on blocks}
```

Consider A to be a N by N matrix of b by b blocks



Pseudocode Dependency graph

Figure 7 Pseudo-code for the Cholesky decomposition implementation and the thread dependency graph for the same applications

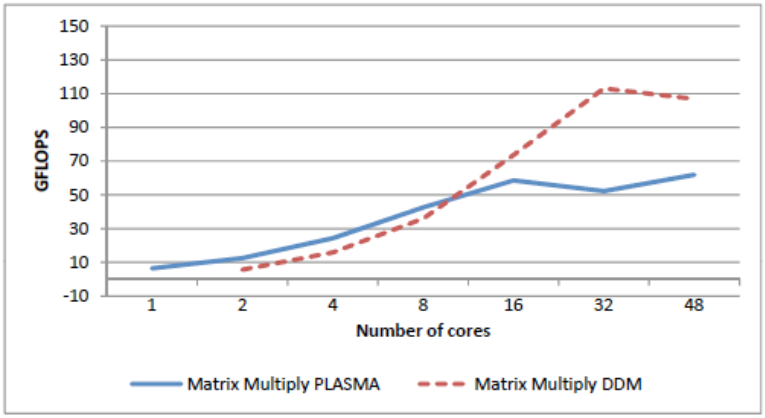


Figure 8 Performance of Matrix Multiply implemented with DDM

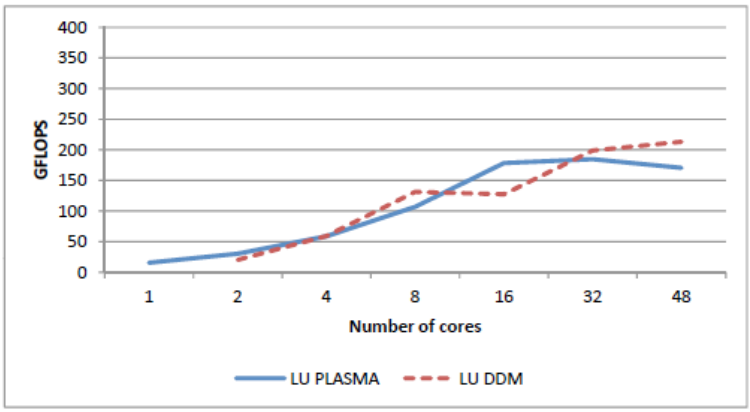


Figure 9 Performance of LU Decomposition implemented with DDM

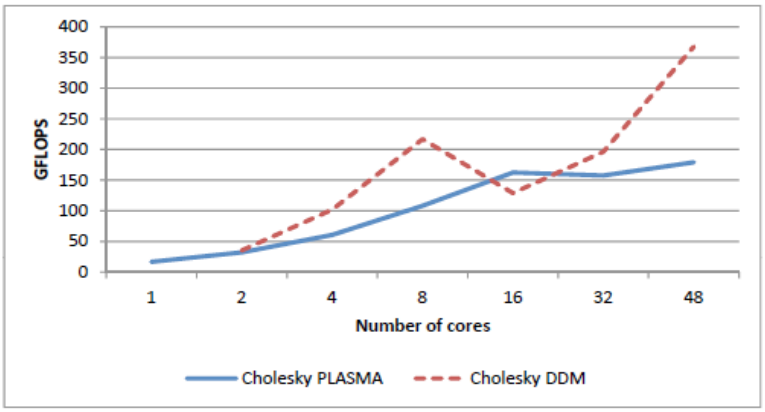


Figure 10 Performance of Cholesky Decomposition implemented with DDM

It is possible to observe that the DDM implementation scales well as the number of cores increases. Some results do not follow the trend perfectly and we estimate that it is due to the fact that as we increase the number of cores, there are caching effects that change the performance of the applications. For example the increase of cores that surpass the number of cores in a die of the chip and thus result in the use of a separate partition of the L3 cache and also when we surpass the number of cores in a processor and thus some cores will be sharing the data across the caches in two different

chips. Also relevant to notice from the results is the fact that the results show that DDM surpasses the performance obtained by PLASMA in most cases and this difference is quite significant for the setups with large number of cores.

The results observed in this study are very encouraging and show that DDM can handle the parallelization required for linear algebra applications for present and future multi and many-core systems. As such we will investigate the use of DDM as a viable candidate for HPC and look at ways to improve programmability such as the development of DDM libraries with relevant numerical analysis kernels, similarly to BLAS and PLASMA.

5.3 HMPP: a Directive-based Programming Model

The OpenHMPP [4] (and OpenACC) programming model proposes a data parallel programming model based on the codelet concept. In TERAFLUX, CAPS has been investigating the extension of the current CAPS products with the data flow model designed by the Workpackage 3 in a manner that is compatible with the existing OpenHMPP implementation. The goal is to make available two different directives based models to the users; each implementing a different parallel programming approach: the first using the regular OpenHMPP concept, the second using an advanced data flow concept. Depending on the structure of a computation phase in the application, the user will use the most adequate parallel approach. The design of this extension has been performed according to the following consideration:

- Minimize the number of changes to OpenHMPP;
- Execution with current OpenHMPP model is correct.

This extension consists of two main constructs:

1) A new directive to define data flow regions;
2) A variant of the callsite directives denoted “dfcallsite”.

The data flow region is delimited using a DFRegion pragma on a statement block (denoted DFR in the remainder of this document) as shown below:

```
#pragma hmpp DFRegion in(…), out(…)
{
     // set of statements
}//end of data flow region
```

Data flow regions can contain the following statements:

- Procedure calls with a dfcallsite directive;
- Statements that are not affected by the tasks computations. Output arguments of tasks cannot be used in non-dfcallsite statements.

The statements in a DFR aim at creating the task graph. These statements can be arbitrarily complex but a task creation cannot depend on the result of one of the tasks. These statements are executed on the host system.

Data flow regions shall have the same semantic as the sequential execution of the region (which represents a particular schedule of the tasks). In and out region arguments are contiguous memory blocks. Other memory blocks can be used as internal storage for the region. They are dead variables at the entry and the exit of the DFR.

The proposed extension to OpenHMPP is based on the current concept of Codelets. They are pure functions that can be remotely executed in a given address space.
In the context of this work, OpenHMPP codelets have a set of restrictions:

- Codelets arguments are limited to scalar and mirrored data
- Codelets code generation must not lead to data exchange or synchronization with the master program

Codelets falling in this category are denoted DFCodelets. From the data flow model point of view, a DFCodelet can be seen as a data flow threads at execution. The DFCodelets must have the clauses:

- `args[*].mirror`
- `args[*].transfer=manual`

to ensure the proper declaration for the argument mode management. A typical codelet declaration pattern is shown below:

```
#pragma hmpp c1 codelet, args[A].io=in, args[C].io=out, &
#pragma hmpp & args[*].mirror, args[*].transfer=manual
void compute1(float *A, float *C);
```

DFCodelet granularity can encompass from a few statements to a large set of statements. This later is targeted with this work since it is expected that, in general, the synchronization operations may be expensive. However, when considering the TERAFLUX system, this constraint may be alleviated thanks to the hardware based thread management. There are no restrictions made on the statements except that the code generation should lead to a unique accelerator kernel. This constraint is necessary to ensure that no synchronization between the device and the host is needed to execute a task. DFCodelets are expected to exhibit parallelism in their computation. This parallelism can then be used to exploit SIMD/SIMT parallelism available in many devices. This is taken care of by HMPP code generation. An example is given below:

```
#pragma hmpp c1 codelet, args[A].io=in, args[C].io=out, &
#pragma hmpp & args[*].mirror, args[*].transfer=manual
void compute1(int j, float *A, float *C);
#pragma hmpp c2 codelet, args[C].io=in, args[B].io=out, &
#pragma hmpp & args[*].mirror, args[*].transfer=manual
void compute2(float *B, float *C);
#pragma hmpp c3 codelet, args[C].io=in, args[B].io=out, &
#pragma hmpp & args[*].mirror, args[*].transfer=manual
void compute3(float *B, float *C);

#pragma hmpp DFRegion in(A), out(B)
{
   for (i=0 ;i< n ;i++){
   #pragma hmpp c1 dfcallsite, device="1”
   compute1(i, A, C[i]) ;
   }

   for (i=0 ; i<n ;i++){
     if (i=0)
       #pragma hmpp c2 dfcallsite, device="2”
       compute2(C[i], B) ;
          else
       #pragma hmpp c3 dfcallsite, device="3”
       compute3(C[i], B) ;
   }
}//end of data flow region
```

5.4 OpenStream

INRIA is working on streaming dataflow using a pragma-based approach. This deliverable extends our proposal of a streaming data-flow extension presented in earlier deliverables. We coined the name OpenStream for these data-flow streaming extensions of OpenMP:

http://www.di.ens.fr/StreamingOpenMP

OpenStream is an expressive programming model to allow the composition of tasks communicating through first-class data-flow streams, as well as separate compilation. We provide more general dynamic constructs to support complex data structures and unbounded fan-in and fan-out communications. In contrast with our previous work, we introduce strongly typed, first-class streams that may be freely combined with recursive computations and dynamic data structures, while preserving modular (separate) compilation. We also add variadic stream clauses to construct arbitrarily complex, dynamic, possibly nested task graphs, and we provide syntactic support for broadcast operations and for synchronization with futures.

Higher expressiveness may improve productivity, but it often comes with performance overheads, impacting the compiler optimizations and increasing the complexity of the necessary runtime support. However, it is also an important asset: general point-to-point synchronization alleviates scheduling constraints of simpler programming models like Cilk. Detailed experiments to study the performance benefits and caveats of this extra expressiveness have been conducted, and published in [1].

OpenStream relies on programmer annotations to specify the data flow between OpenMP tasks and to build the program task graph. Task graphs need be neither regular nor static, unlike the majority of the streaming languages. OpenStream programs allow dynamic connections between tasks, multiple tasks interleaving their communications in the same streams, and arbitrary and variable fan-in, fan-out and communication rates in a dynamically constructed task graph. The language also supports modular composition, separate compilation, and first-class streams (streams as arguments and return values). Despite this expressiveness, the model preserves functional determinism of Kahn networks by enforcing a precise interleaving of data in streams derived from the sequential control flow of the main program.

The syntactic extension to the OpenMP 3.0 language specification consists in two additional clauses for `task` constructs, the `input` and `output` clauses presented in Figure 11. The baseline syntax is the same as the one presented in D3.1.

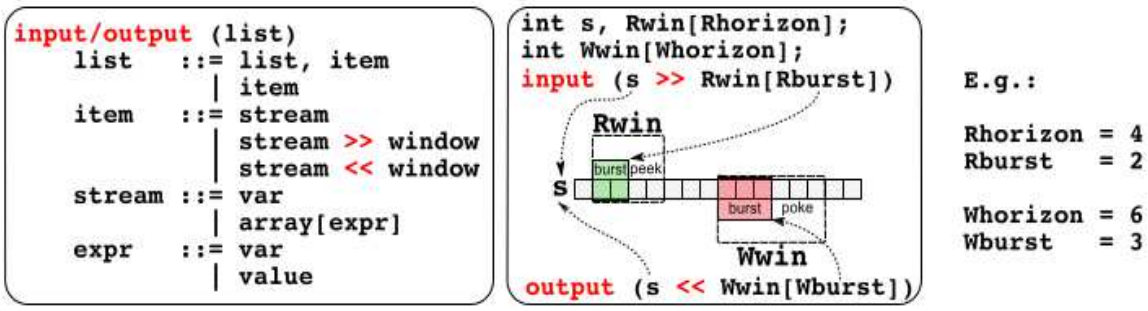


Figure 11 Usage of the input and output clauses in OpenStream

Both clauses take a list of items, each describing a stream and its behaviour with regard to the task to

which the clause applies. If the item notation is in the abbreviated form `stream`, then the stream can only be accessed one element at a time through the same variable stream. In the second form, `stream >> window`, the programmer uses the C++-flavoured `<< >>` stream operators to connect a sliding window to a stream, gaining access to multiple stream elements, within the body of the task.

Tasks compute on streams of values and not on individual values. To the programmer, streams are simple C scalars, transparently expanded into streams by the compiler. An array declaration (in plain C) defines the sliding window accessible within the task and its size, the *horizon*. The connection of a sliding window to a stream in an `input` or `output` clause allows to specify the *burst*, which is the number of elements by which the sliding window is shifted after each activation. In the previous figure, the input window `Rwin` would be shifted by two elements, while the output window `Wwin` would be shifted by three elements. The data-flow case corresponds to *horizon=burst*. In the more general case where *horizon>burst*, the window elements beyond the burst are accessible to the task; for an output window, the burst and horizon must be equal. Task activation is enabled by the availability, on each input stream, of all horizon elements on the input window, and is driven by the control flow of the main OpenMP program.

```
#pragma omp task output (x)                          // Task T1
  x = ...;
for (i = 0; i < N; ++i) {
  int window_a[2], window_b[3];

  #pragma omp task output (x << window_a[2])         // Task T2
    window_a[0] = ...; window_a[1] = ...;
  if (i % 2) {
    #pragma omp task input (x >> window_b[2])        // Task T3
      use (window_b[0], window_b[1]);
  }
  #pragma omp task input (x)                         // Task T4
    use (x);
}
```

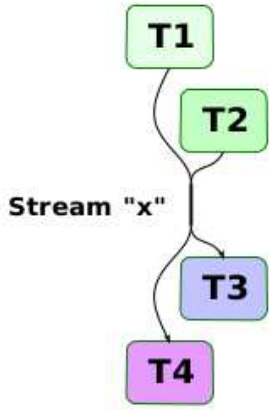


Figure 12 Syntax of the input and output clauses

The example in Figure 12 illustrates the syntax of the `input` and `output` clauses. Task T1 uses the abbreviated syntax to produce one data element for stream `x`. The semantics of stream operations is to interleave accesses, as illustrated in Figure 13, in task creation order.

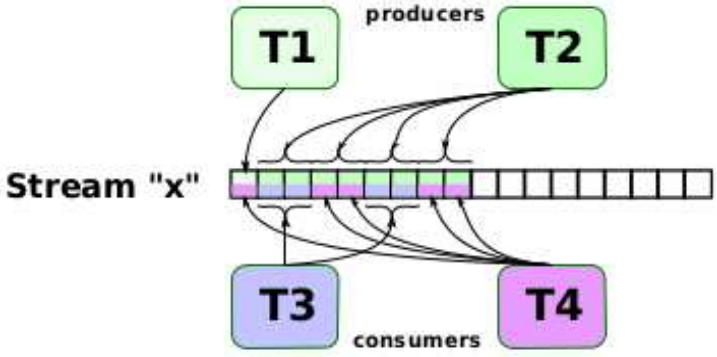


Figure 13 Interleaved accesses to the stream "x". Tasks T1 and T2 are producers, T3 and T4 are consumers.

This order is determined by the flow of control spawning tasks, called *control program*. In our

example, T1 introduces a delay in stream `x`. Task T2 is also a producer, adding two elements to stream x at each activation. Tasks can be guarded by arbitrary control flow, as is the case for T3, which reads three elements at a time and discards two elements. T4 also reads from x, interleaving its accesses to the stream with the accesses from T3. This interleaving is entirely determined by the schedule of the control program, in this case it is a sequence (T4, T3, T4, T4, T3, ...).

In addition to `input` and `output` clauses, we provide a convenience clause for performing pure peek operations (i.e., when a task reads on a stream without advancing the stream, through a 0-burst access window). The `peek` clause does not introduce any new semantics; it makes broadcast operations explicit.

In a streaming context, broadcasting can be performed without copying, by allowing multiple tasks to read the same data in a stream without advancing the reading index. The `peek` clause uses the same syntax as the `input` clause, except that the `burst` is disregarded, implicitly understood to be 0. To simplify the complementary operation of discarding elements from a stream, whether already read or not, we add a new `tick` directive. It has similar semantics to a code-less task using an `input` clause on a given stream: it advances the read index in streams, playing a similar role to advancing the logical clock represented by stream access indexes. The Figure 14 presents the syntax of `peek` and `tick`, as well as the semantically equivalent code.

```
int win[horizon];
// implicit 0-burst
#pragma omp task peek (x >> win)

// advance clock by ''burst''
#pragma omp tick (x >> burst)
```

≡

```
int win[horizon];

#pragma omp task input (x >> win[0])

#pragma omp task input (x >> win[burst])
  { ; }
```

Figure 14 Syntax of peek and tick clauses, and the semantically equivalent code

One of the main roles of our streaming annotations is to describe, in a compact way, how the dynamic task graph of an application is built. To generate arbitrary task graphs, it is necessary to allow connecting tasks to dynamically variable numbers of streams. However, this poses a challenge due to the static nature of compiler directives: the number of streaming clauses present on a task's pragma directive is inherently static. To specify a variable number of connections, we allow to simultaneously access multiple streams of an array through an array of windows.

```
int stream_array[N] __attribute__((stream));
int window_array[num_streams][num_elements];

#pragma omp task input (stream_array >> window_array[num_streams][num_elements])
  ... = window_array[0..num_streams-1][0..num_elements-1];
```

Figure 15 Example of how to allow multiple streams of an array, exploiting an array of windows

The Figure 15 shows an example of an array of stream access windows connected, in a *variadic clause*, to multiple streams from an array of streams. The window `window_array` gives simultaneous access to the first `num_streams` streams in `stream_array`. The number of streams connected must be at most the size of the array.

So far, we presented the declaration of stream variables as a plain C variable declaration. However, this poses problems for compiling streaming programs where streaming tasks occur in function calls, let alone programs divided in multiple translation units, and it makes type checking very difficult. To

enable modular compilation, we need an interface to pass streams as parameters to functions and to store stream references in data structures, making streams first class entities which can be manipulated like any C variable. Consistently with our compiler directive approach to streaming, we add variable and parameter declaration attributes to type streams.

Streams are implicitly separated between a stack-allocated stream reference, which can be freely manipulated by the programmer, and the heap-allocated data structure used and managed by the runtime. In general, the user needs not know about the latter and can simply consider streams to behave like any stack-allocated variable.

```
// Declare a typed stream
int scalar_stream __attribute__((stream));

// Declare a typed array of streams (allocates runtime data on the heap)
int stream_array[size] __attribute__((stream));

// Declare a typed array of stream references (no allocation of runtime data)
int stream_ref_array[size] __attribute__((stream_ref));

// Function taking an array of streams as parameter
void foo (int x[] __attribute__((stream)));

// Call-site for a function taking an array of streams as parameter
foo (stream_array);
```

Figure 16 Different types of stream declarations

Figure 16 shows the different types of stream declarations, as scalar variables, arrays of streams and arrays of stream references, and parameters to functions. Both streams and stream references can be manipulated in the same way, but stream references are not initialized and must be set by the programmer with an assignment. They are used for managing collections of streams, in particular for variadic streaming clauses.

The management of stream data structures is generally done entirely by the runtime, which transparently updates a reference counter to ensure timely deallocation. However, some advanced uses of stream references require programmer intervention in the form of runtime calls to increment and decrement the reference counter; specifically when they escape the current scope of the stream variable because the stream is returned by a function or it is stored in a heap allocated data structure. We purposefully chose this explicit approach to resource management to avoid relying on a general-purpose garbage collector. Indeed, a garbage collector would have to rule the whole heap memory. Since first-class streams allow handling most situations automatically, programmer intervention is seldom necessary; so far, we only used explicit reference counting in complex, compiler generated codes that would be written in simpler ways by a programmer, without requiring explicit reference counting.

Finally, to allow recursion with concurrent tasks, and more importantly to enable the parallel execution of the control program, we add support for task nesting, valid for any arbitrary nesting of streaming and non-streaming tasks.

As we target more than just structured nesting graphs, we need to be able to communicate streams to nested tasks, which allows them to further generate tasks accessing these streams. This is possible by passing streams by value to nested tasks, using the `firstprivate` clause. This clause copies the stream reference alone and issues the proper runtime calls to ensure proper management of the stream

data structure (reference counting), without any further programmer involvement. Let us illustrate this on the recursive implementation of Fibonacci.

```
void stream_fibo (int n, int cutoff, int sout __attribute__((stream))) {
  int x;
  if (n <= cutoff) {
#pragma omp task output (sout << x)
      x = sequential_fibo (n);
    } else {
      int s1 __attribute__ ((stream));
      int s2 __attribute__ ((stream));

#pragma omp task firstprivate (s1)
      stream_fibo (n - 1, cutoff, s1);
#pragma omp task firstprivate (s2)
      stream_fibo (n - 2, cutoff, s2);
#pragma omp task input (s1, s2) \
                 output (sout << x)
      x = s1 + s2;
}}
```

```
// Main:
int stream __attribute__ ((stream));
int numiters = ...;
int cutoff = ...;
int result;

#pragma omp task firstprivate (stream)
  stream_fibo (n, cutoff, stream);

#pragma omp task input (stream >> result)
   printf ("Fibo␣result:␣%d", result);
```

Figure 17 Implementation of the recoursive Fibonacci alogirithm (portion of the main, and actual function)

The right side of Figure 17 shows the main function of the program, which declares a `stream`, passed by copy to a first task that initiates the recursion and on which a second task will read the final `result`. The left side of the figure shows the recursive function taking as parameter a stream on which it writes its result. It further spawns two tasks to generate the remainder of the recursion.

We mentioned above that a restriction on this type of nesting is necessary to preserve determinism. Indeed, the problem comes from the fact that we rely on the total order on read (or, independently, write) accesses to each stream, which derives from the order of generation of tasks performing such accesses, to guarantee the determinism of the schedule of data in streams. If the *control program*, which is the thread of control that reaches a task construct, is not sequential, then concurrency between the generation of tasks performing the same type of access to the same stream will lead to non-determinism. We must therefore ensure that the order of creation (not execution) of tasks producing to or consuming from each stream is preserved. This can either be achieved by ensuring that all tasks producing (or *independently* consuming) data in a stream are created by a single task, as is the case in the Fibonacci example, or that the order of creation is enforced through dependences in the task graph.

To complete this feature list, we also demonstrated the embedding of dynamic futures and the translation of StarSs into OpenStream. This experiment is discussed in [1], with a summary of the translation from StarSs in D4.6.

Finally, the presentation above used an informal semantics. We also defined a formal semantics for a subset of the OpenStream language. This semantics has been the occasion to coin a purposely designed formal model, called Control-Driven Data Flow (CDDF), generalizing a variety of imperative models of parallel computation. This formal model has been submitted to the ACM TOPLAS journal and is available as a research report [2]. We are extending this model to handle parallel task creation and refine its formal properties. This work will be reported in the fourth year.

6 Summary

This document has described the research carried out in the WP3 of the Teraflux project during the third year. It is split into three distinct sections covering the work carried out on the high productivity programming model, on the synchronous concurrency and on high performance models. Within the latter models, we cover progress with C-directive-based dataflow models (StarSs, TFLUX, HMPP, OpenStream). The executive summary has presented the main achievements obtained during this year; including two best paper awards.

In addition, it does contain the rationale for needing a means for handling shared mutable state in dataflow models. This deliverable has covered the work being carried out in T3.4.

Overall, the programming models have past the initial definition stage and we start to observe maturity with respect to the tools available (see for example the open-source tools). These advances provide a solid foundation for the next year. Nonetheless, we start to observe common aspects among the different programming models. The creation of the dataflow task graph is supported with different syntax but the core functionality of describing a side effect free computation as a node in the graph is prevalent. The inputs and outputs are specifically annotated and permit the generation of the dataflow graph. We can observe a divergence on how rich a set of dependencies each programming model provides specific support for. We can also observe a divergence with respect to the extra information that can optimize the runtime scheduling of the dataflow graph. These divergences are not to do with whether the dataflow graph generated is general, but is associated with covering well certain patterns of dependencies and the level of sophistication expected from the compiler when a pragma is encountered. The work with HMPP provides an industrial perspective of what features/functionalities are well understood.

Next year, the projects partners will provide the final dataflow computational models (i.e. T3.4 Consolidated Dataflow Models) and learn from the interactions with WP2 (applications), WP4 (compilation tools) and WP5 (execution on the Teraflux architecture). In particular, the formal definition and implementation of the memory types will occupy a lot of our time.

References

[1] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. ACM Transactions on Architecture and Code Optimization (TACO), selected for presentation at the HiPEAC 2013 Conf., January 2013.
[2] Antoniu Pop and Albert Cohen. Control-Driven Data Flow. Research Report RR-8015, INRIA, July 2012.
[3] Albert Cohen, Léonard Gérard, and Marc Pouzet. Programming parallelism with futures in Lustre. In ACM Conf. on Embedded Software (EMSOFT), Tampere, Finland, October 2012. Best paper award.
[4] HMPP User's Manual. CAPS enterprise, 2012.

[5] Daniel Goodman and Behram Khan and Salman Khan and Mikel Luján and Ian Watson . Software transactional memories for Scala. Journal of Parallel and Distributed Computing, 2012. http://dx.doi.org/10.1016/j.jpdc.2012.09.015

[6] C. Seaton, D. Goodman, M. Luján, and I. Watson. Applying dataflow and transactions to Lee routing. In Proceedings of the 7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG), 2012. Best Paper Award.

[7] D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Luján, and I. Watson. DFScala: High level dataflow support for Scala. In Proceedings of the 2nd International Workshop on Data-Flow Models For Extreme Scale Computing (DFM), 2012.

[8] Ian Watson, Chris Kirkham and Mikel Luján. A Study of a Transactional Parallel Routing Algorithm. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques - PACT, pp. 388-398, 2007.

[9] Rahulkumar Gayatri, Rosa M. Badia, Eduard Ayguadé, Mikel Luján, Ian Watson. Transactional Access to Shared Memory in StarSs, a Task Based Programming Model. Euro-Par 2012: 514-525.

[10] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, and H. Ltaief. PLASMA Users Guide. Technical report, ICL, UTK, 2009.

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

Appendix A

The Need for Shared Mutable State in Dataflow Models

It is widely accepted that a functional programming approach together with dataflow execution can lead to the efficient exploitation of implicit parallelism. This computational model is usually associated with styles of programming based on pure functions as these map readily on to the execution model. However there are many computations which cannot be easily or efficiently expressed in a pure functional style as they have a structure where the parallel manipulation of shared mutable state is fundamental to the problem being solved. In Teraflux we argue for a new computational model which combines Transactional Memory (TM) and Dataflow to overcome this limitation. Using a simple example, we illustrate why the parallel manipulation of mutable state is necessary and how the addition of TM to Dataflow can provide an elegant and efficient parallel computational model.

Background on Dataflow

In the 1970s and 1980s, there was a strong belief, amongst some, that dataflow approaches would provide the solution to general purpose parallel systems which were easy to program. However, there were three main impediments:

1. Integrated circuit technology developed at a remarkable rate ensuring that serial instruction execution speeds were able to satisfy most performance requirements.

2. Dataflow and associated models have a requirement for high communication rates. The level of integration available in the 70s and 80s did not permit efficient implementations of the necessary functionality.

3. The absence of side effects in dataflow models (e.g. functional programming) permits easy parallelization. Unfortunately, there are many cases in real programs where the use of state is either necessary for efficiency or is a fundamental part of the problem being solved. In these circumstances, functional approaches are unsuitable.

Nowadays only the third issue still needs to be addressed. Existing functional languages that have tried to address this need for shared state either its introduction can rapidly destroy both the mathematical cleanliness of the language with implications to the ability of exploiting parallelism with dataflow (SML, F#, Haskell), or introduces an implicit lock to protect against concurrent accesses (M-structures) which quickly become unfit for purpose due to the lack of composition of locks. M-structures quickly result in deadlocks caused by different access patterns within different threads. To write correct code with M-Structures is difficult and M-Structures are widely noted as a failure including by many of those behind their original development.

An Example with Shared Mutable

The Dataflow plus Transactions model can best be understood by studying an example. Such a study is necessarily simplified but should serve to illustrate the essence of the approach. Assume a graph where a value at each node represents some changing local parameter. This parameter might represent a physical load on a particular local resource. We want to maintain a continuous histogram of load

values in order to inform a load balancing system. In practice, it is likely that the structure would be a general graph but here we will assume a binary tree as this greatly simplifies the explanation.

The natural way to express this is a recursive function which visits a node, updates a global histogram and then spawns new threads which visit the children. In addition to updating the histogram we want to know when an exploration of the tree has terminated. The basis of the computation might be a node definition and function of the following form:

```
typedef struct node {
   int old_value, new_value;
   struct node *left, *right;
} node;

int visit (node* t) {
   if (t == NULL) return 1;
   histogram[t->old_value]--;
   histogram[t->new_value]++;
   t->old_value = t->new_value;
   return visit(t->left) && visit(t->right);
}
while (visit (root));
```

We have used an imperative style using C to express the program but the overall computation has two distinct forms. The first is functional in style and can lead to a highly parallel execution. The second is an imperative update of histogram values.

We will assume a dynamic dataflow model where the dataflow nodes are threads executing the `visit' function at the level of a function body. The execution proceeds by constructing, for each function body, a new dataflow node for each new function call within the body together with a continuation node to receive the results of those newly created nodes. The continuation is subject to normal dataflow execution rules. This is essentially packet based graph reduction [8]. This will result in a parallel execution graph of the form shown in Figure 18

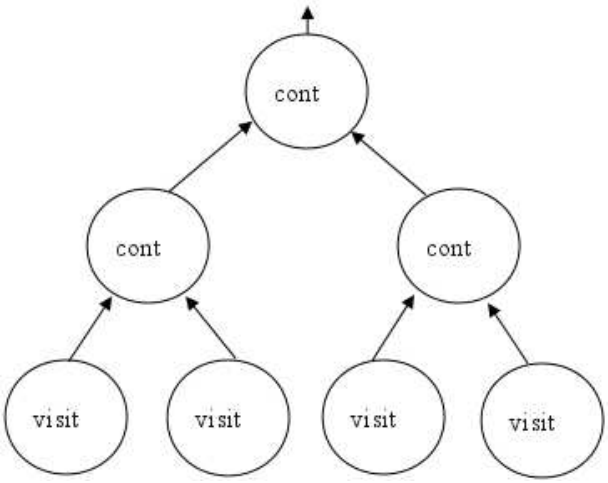


Figure 18 Parallel Dataflow graph: nodes represents threads

The nodes at the bottom level are in the process of execution, this may either result in the generation of further nodes to explore more of the graph or, if the leaves have been reached, the return of values to a waiting continuation which is synchronized by dataflow firing rules.

However, we have so far ignored the updates to the state of the global array histogram, these can clearly occur from any of the parallel executing `visit' nodes. Unless the update action (add and subtract) is performed atomically, the resulting values may be incorrect. In general, the updates will consist of a memory read, an arithmetic operation and a memory write which cannot be separated. In addition, we would like to keep the overall state of the histogram consistent by performing the operation on two entries as a single action. We could achieve this by explicit global locking of the histogram array. This can be done by either using a single coarse grain lock or a set of finer grain locks to improve performance. In either case the addition of this explicit locking would both add significant complication to the code and potentially have an impact on the runtime parallelism.

Our proposal is to introduce transactions into the pure dataflow model. This involves a small modification to the visit function to indicate which parts of the code need to be executed atomically. Assuming the support of a TM system, our execution is going to proceed in one of two ways:

a) In the absence of conflict, i.e. there are no other threads attempting to perform an operation on the same elements of the histogram array, the parallel execution will proceed uninterrupted.

b) If conflict occurs, one of the conflicting threads will proceed uninterrupted but the others will serialize (e.g. by abandoning their execution and retrying)

An important property of transactions is their *isolation*. The detection of conflict and possible retrying is transparent to the application (other than a possible increase in execution time). This ensures that, in a model which is otherwise functional, it is possible to generate implicit parallelism via dataflow execution, but provide the ability to manipulate shared state where necessary.

Deficiencies of Alternative Formulations without Transactions
As already discussed, it is widely accepted that a functional programming approach together with dataflow execution can lead to the efficient exploitation of implicit parallelism. The drawback of the approach is that it cannot easily handle state based computation. In order to appreciate the issues, we will consider expressing the described histogram example using current functional approaches.

Infinite Histories − The classic way to deal with state in a functional program is to regard the updates to a variable as a sequence of state changes forming an infinite history. Any function wishing to change the state of a variable is passed it as a parameter and returns a new version which is the next in the sequence. This is passed in turn as a parameter to the next function which wants to update the state.

The fundamental problem with this approach is that it is essentially serial. It is clearly not possible to have branches in a history which would result if such a variable were passed to multiple functions in parallel. It is therefore not possible to use this technique to produce a parallel version of the above program.

Serialised State Manipulation − The infinite histories approach requires a function to take a parameter and produce an updated version as a result. In a purely functional world this requires the production of a new updated copy. If the variable is a monolithic structure, such as the histogram array in our

example, it may be necessary to copy a large structure whilst updating only a small part of it. This can be highly inefficient.

However, as long as the serialisation of updates can be ensured, this is not strictly necessary and an updateable variable can be used. This is essentially the Monad approach where the type system is used to ensure the serial usage. Although this approach is powerful in achieving efficiency whilst maintaining clean properties of the language, it is again fundamentally serial.

Partial Histograms − Rather than construct a histogram directly, we could write our `visit' function to return a partial histogram which represented the contribution of itself and all its descendent nodes. Our program would then construct a tree-like dataflow graph where the partial results were combined by the continuation functions and eventually returned from the root. This would have the following disadvantages:

a) The complexity of the continuation would be increased significantly.
b) The copying of data between parent and child computations could add significant overhead.
c) The overall histogram would be updated only when the computation returned to the root. In a continuous system, such delays may well be unacceptable.

This formulation becomes even more unattractive if we were operating on a more general graph where the issue of when and where to combine partial results would be more complex.