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

D3.5 – Overall Computational Model Final Report

Due date of deliverable: 31 March 2014
Actual Submission: 19th May 2014

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

Lead contractor for the deliverable: UNIMAN

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**
1	Mikel Lujan	UNIMAN	
2	Mikel Lujan	UNIMAN	Improvements based on internal feedback
3	Mikel Lujan	UNIMAN	Improvements based on internal review
4	Roberto Giorgi	UNISI	Coordinator's review

Release Approval

Name	**Role**	**Date**
Mikel Lujan	Originator	31/Mar/2014
Mikel Lujan	WP leader	30/Apr/2014
Roberto, Giorgi	Project Coordinator for formal deliverable	11/May/1014

TABLE OF CONTENT

List of contributors to the writing of the document.

Daniel Goodman, Chris Seaton, Salman Khan, Behram Khan, Pareskevas Yiapanis, Christos Kotselidis, Mikel Luján, Ian Watson
University of Manchester

Rahul Gayatri, Rosa M. Badia, Eduard Ayguadé
BSC

Albert Cohen, Léonard Gérard, Feng Li, Antoniu Pop
INRIA

Laurent Morin
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.
DFR	Dataflow region
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 transaction 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 work within the programming model development, which has normally being split into three parts covering the high productivity model, the synchronous dataflow model and the high performance models.

The specific achievements and discussions for Year 4 are:
High Productivity Model – Scala (Section 2)

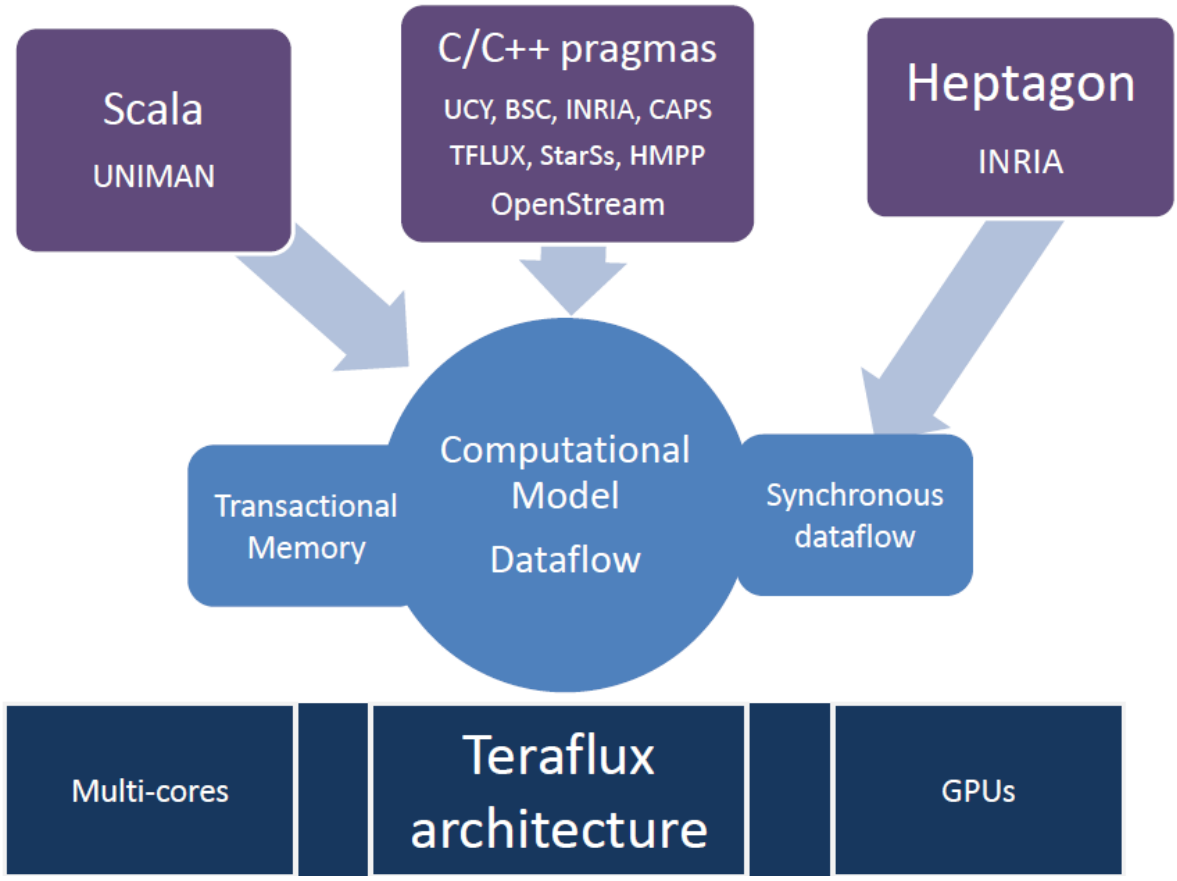
- Manchester University Transactions for Scala (MUTS) and Dataflow Scala library (DFScala) have been combined to develop complex parallel frameworks such as Pregel and MapReduce.
- Work understanding how to combine dataflow and transactional memory has been extended to Pregel
- Development of a Scala compiler plugin to help identify which variables should be protected with transactions.
- Analysis of whether of scheduling based on software developers knowledge, presented in DFM 2013.

High Performance Model – C directives (Section 3)

- StarSs (from BSC) has improved their compiler and runtime system to support speculation and developer more complex applications including and analysis of the overheads incurred by using Software Transactional Memory.
- CAPS has a proposal for their pragma directives to support dataflow programming on GPUs.
- INRIA has extended the streaming data-flow extensions of OpenMP, called OpenStream, with support for Owner Writable Memory and Transactional Memory.

1 Introduction

This document is the final deliverable of the work carried out in WP3. Work within WP3 has been 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 have covered progress with C-directive-based dataflow models (StarSs, HMPP, OpenStream). This final year the work has focused on Scala, StarSs, OpenStream and HMPP and this is reflected in the contents of this deliverable. For completeness we also summarize the work of UCY with TFLUX in this workpackage, although no new development has occurred in Year 4.



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, D3.2, D3.3 and D3.4 and WP2 contains some of the performance results for applications implemented using programming tools developed in this workpackage.

1.2 Activity referred by this deliverable

This deliverable covers the work being carried out under WP3 in year 4 (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. In particular we present below an executive summary, for convenience of reading, of decisions taken to combine Dataflow and Transactional Memory (for more details we refer to deliverable D3.1, D3.2, D3.3 and D3.4).
In particular, we recall that Appendix-A of D3.4 summarizes the need for shared data in dataflow, which motivates combining Transactional Memory and Dataflow.
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.

2 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.

In the previous reporting period we realized two main libraries which provide transactional memory and dataflow execution.

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

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

We provided updates on the new developments for these libraries and in particular how they can be applied to Lee's routing algorithm in D3.4. This deliverable focuses on how we have used these open source tools to develop on top more complex frameworks; e.g. Google's Pregel & MapReduce.

2.1 Manchester University Transactions for Scala (MUTS)

We briefly recall that in D3.4 we provided a description of the implementation of software transactional memory in Scala without making modifications to the Scala 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 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].

2.2 Scala Dataflow Library (DFScala)

We also briefly recall that in D3.4, to compliment MUTS and to enable the development of dataflow code for a number of the applications selected in we have constructed a library to support the creation and execution of dataflow threads.

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.

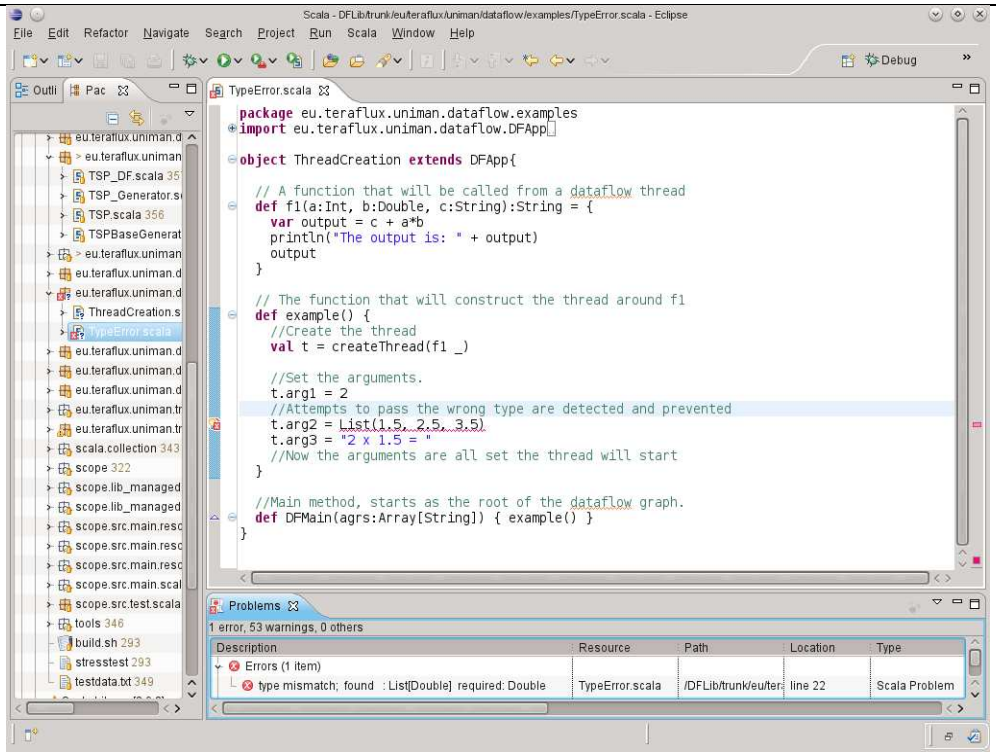


Figure 1 DFScala static type checking - error found by Eclipse IDE.

The performance results published in [6] and summarized below 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.

2.3 Building a Parallel Framework: Pregel

Google has put into production several frameworks (e.g. MapReduce, Pregel and Percolator) to facilitate the software development of parallel applications running on their datacenters. MapReduce and Pregel follow dataflow principles and we have developed on top of DFScala and MUTS equivalent frameworks which run on many-core architectures rather than at datacenter level. We will focus on Pregel in this deliverable as MapReduce has received much more attention and it is well understood by now.

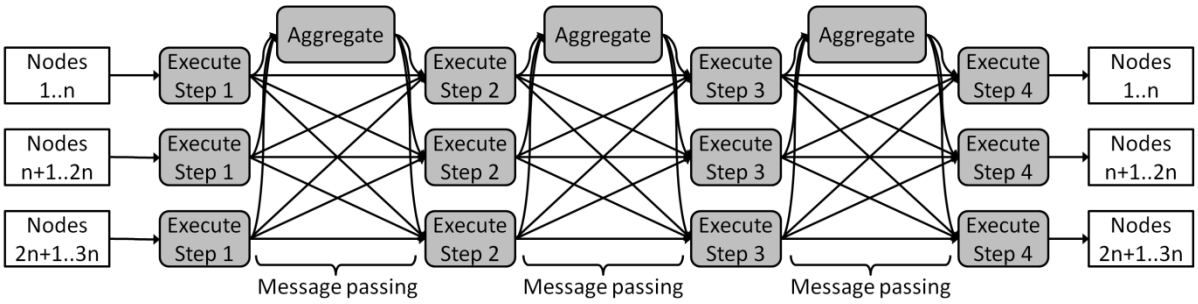
Pregel targets at stepwise graph based computations. With this framework a graph structure is constructed where each node in the graph has some private state, a function that it can execute at each step, and a possibly empty set of vertices to other nodes. On each step each node will:

- Receive messages from other nodes.
- Execute the nodes function taking any received messages and the private state of the node as input.
- Send messages to other nodes.
- Initiate changes to the graph.

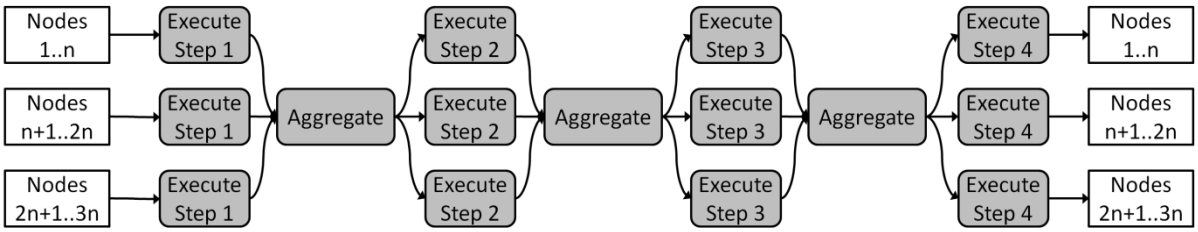
All actions are completed on all nodes before the next step starts. At the end of any step a node can go into a sleep state until either the computation ends or the nodes receives a message. The computation ends when all the nodes have gone to sleep. To use Pregel a user is only required to provide: the function that the nodes will execute; a function to construct the initial state of the graph; and a

function to return the final state of the graph. There is the option of providing aggregator functions that will combine outputs from the nodes, making them available to all other nodes at the next step of the computation.

Similar to MapReduce, Pregel is a dataflow pattern encapsulated in a library that takes the required functions and input data as arguments. Individual nodes or sets of nodes will have their computation at each cycle computed by a dataflow thread. This thread will take as arguments the current state of the node/nodes it is calculating for and any messages to these nodes from the dataflow threads in the preceding step. It will then call the supplied function for each node in its care. These function calls will generate lists of messages (possibly empty) which are passed to the threads in the next step. Each thread includes a flag with the messages marking if it is ready for the graph to terminate. When all threads are ready for the graph to terminate it will. This creates the following 4 step Pregel pattern computing on 3n nodes:



While transactions are not required for this framework the use of shared state to manage message passing becomes an essential mechanism for passing messages as the number of threads increases either because of an increasing problem size or because fewer nodes are evaluated per thread. Without the use of shared state threads that have no messages to communicate to other threads will have to send out ever larger numbers of message lists containing no messages. The effect of this can be clearly seen in the next graph:



We have compared the scaling of the two versions of the Pregel framework implemented in Scala, the first built just using dataflow and the second built using dataflow and transactions. The next graph shows that without shared state the problem fails to scale as we increase the number of threads.

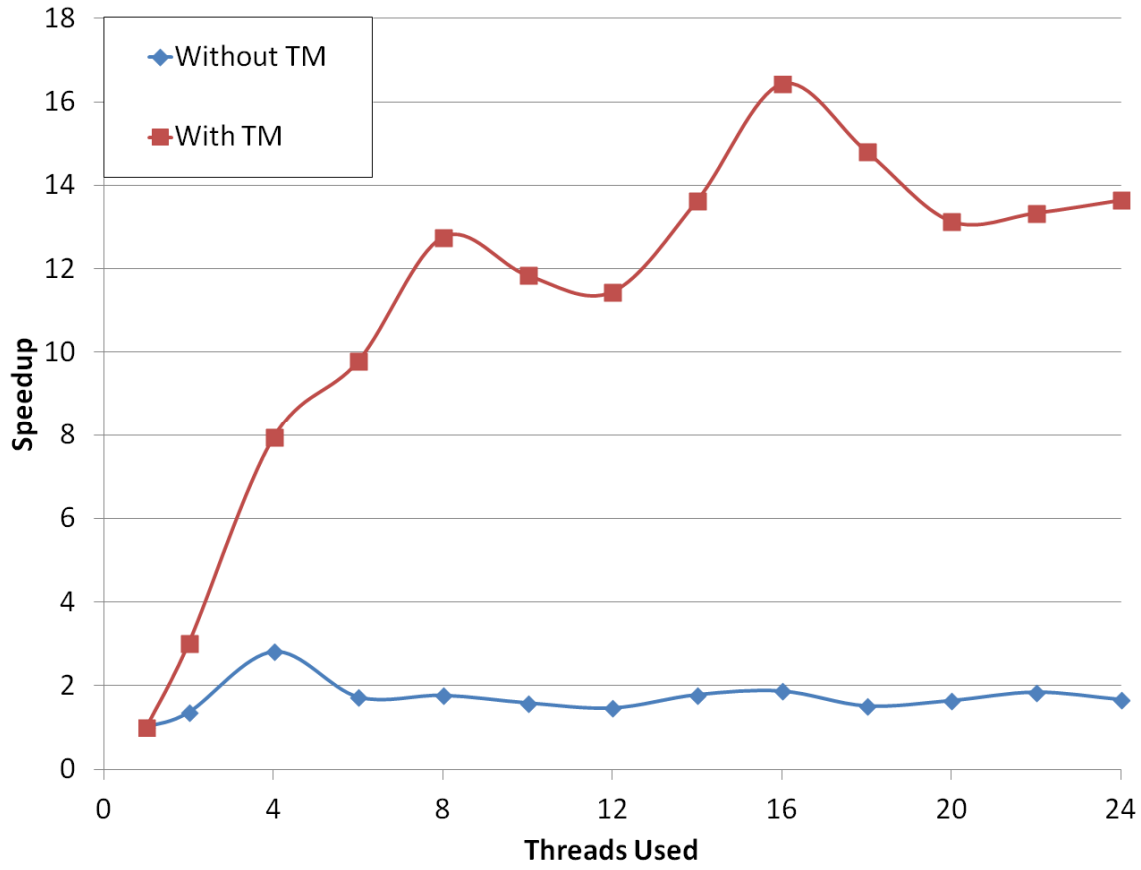


Figure 2 A demonstration of the effect of adding transactional memory to our implementations in Scala of the Pregel framework to reduce the number of dataflow tokens being passed. In this instance the framework is computing the single source shortest path (SSSP) problem. This calculates for each node in the graph the shortest path from a specified node. In this case, a random graph comprising of 100,000 nodes.

Finally, Google's Percolator is their most recent framework. This framework uses transactions to address the absence of interactivity in a MapReduce invocation. Specifically it was designed to address the inability to insert new data into an executing MapReduce computation. This limit on MapReduce interactivity meant that Google could only start the MapReduce to construct a web index once they had finished a complete crawl of the web. We have not implemented Percolator in Scala, but it serves as an example of another framework outside of TERAFLUX that is following the principles investigated in this workpackage.

2.4 User Assisted Scheduling of Dataflow Programs

The determinism and race condition free properties of pure dataflow programs make them very appealing as a means of constructing programs for multi-core processors. However, pure dataflow programs are limited by their determinism which prevents the construction of programs that would traditionally require shared state for either efficiency or to support unstructured interactions.

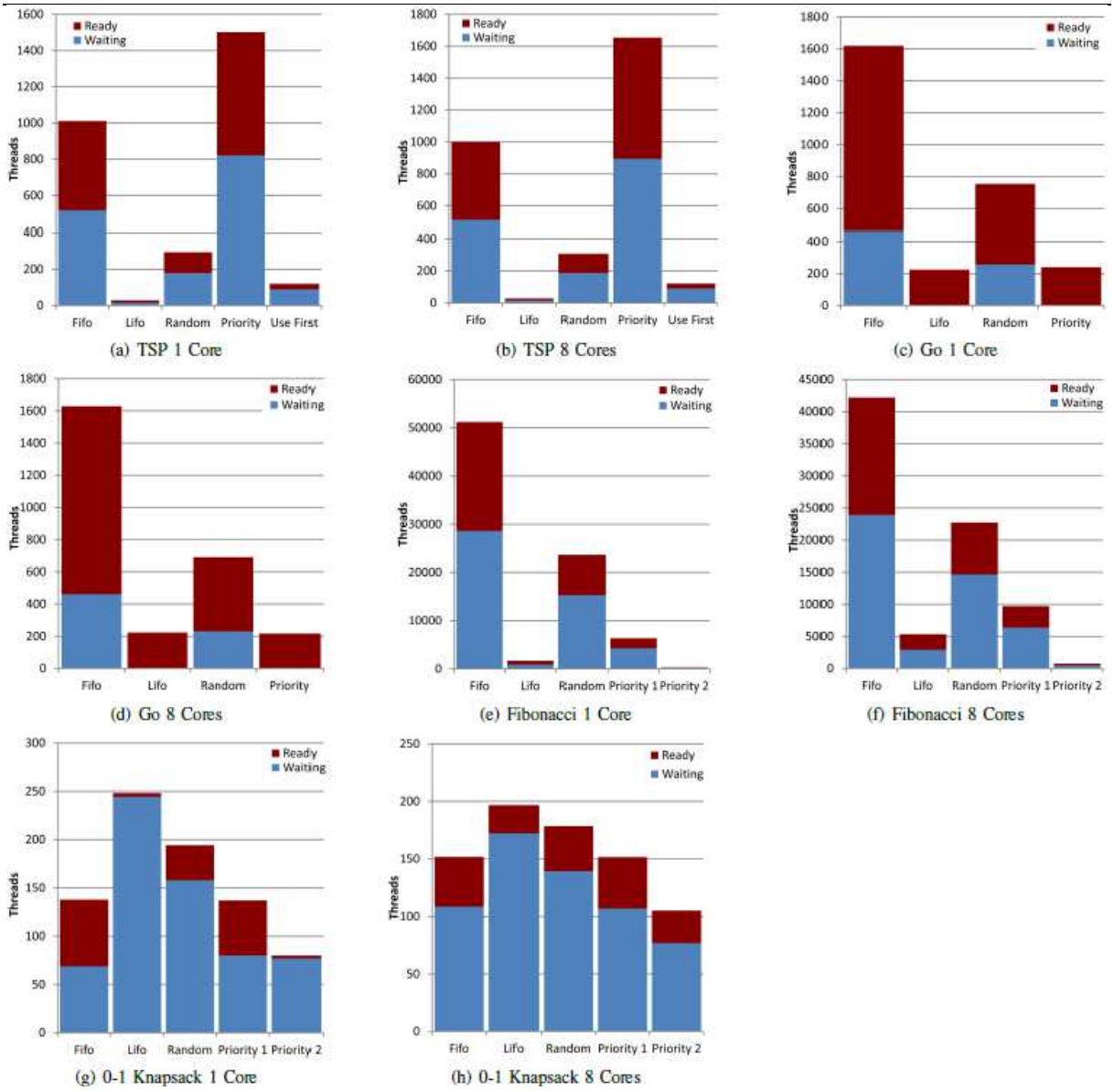
An example of a problem that requires shared state to be solved efficiently is the travelling salesman problem. This takes a connected graph as input, in which the nodes represent cities and the arcs represent roads with weights recording the distances between these cities. It returns a tour where

every city in the graph is visited and the distance travelled is the shortest possible. Accurate solutions to this problem are used on a daily basis by logistics companies.

A brute force approach to this problem is not practical as there are n! possible tours for n cities. Instead efficient techniques for solving this require a shared updatable lower bound which is updated as better solutions are found. The presence of this lower bound allows these techniques to discard any solution that will exceed this lower bound before any further time is spent on it. As a result the most efficient versions are those which can quickly reduce the lower bound to represent the length of the shortest tour, and can efficiently calculate the lower bound for the partially constructed.

In pure dataflow applications scheduling can have a significant effect on the memory footprint and number of active tasks for a given program. However, in impure programs (dataflow with shared state), scheduling not only affects the system resources, but can also affect the overall time complexity and accuracy of the program.

To address both of these aspects we describe and analyse effective extensions to a dataflow scheduler (prototyped in DFScala) to allow programmers to provide priority information describing the preferred execution order of a dataflow graph. We show that even very crude task priority metrics can be extremely effective, providing an average saving of 91% over the worst case scenario and 60% over the best case naive scenario.



2.5 Towards Detecting Automatically the Memory Type of Program Variables

In TERAFLUX, we defined different kind of memory types (cf. D7.1, program variables belonging to one memory type will only accept a subset of operations). Having to remember and identify correctly all these memory usages is not ideal for a software developer. We have investigated whether following certain design patterns coupled with static compiler analysis can be used to automatically detect for example whether a given variable would be read and written at runtime thus requiring protection using transactions.

Thread Local Storage (or Thread Local Memory) - this memory has no visibility to other threads and the values contained within this memory cannot be passed directly to other threads. The data may still be passed to other threads via either an implicit or explicit copy to a different style of memory.

Frame Memory (or Constant Memory) - the value of this memory may be modified by the thread that allocated it. This memory will become constant once it becomes visible outside the allocating thread. To become visible outside the thread a reference must be passed out of the allocating thread. This can currently occur either through the passing of a reference into the frame of another thread or the setting of the reference into a piece of transactional memory and the encompassing transaction then successfully committing.

Transactional Memory - Transactional memory is mutable at all times, but can only be read from or written to from within a transaction. The only possible exception to this is when transactional memory is in the thread where transactional memory is first allocated.

Owner Writable Memory (OWM) - OWM memory is used as an optimisation on single assignment memory, and code operating on OWM must be race condition free. In the event of a race condition involving OWM the program is incorrect and the behaviour is undefined. If protection against race conditions is required, transactions should be used.

The recommended design patterns have as major purpose to ensure that we can, by relatively simple static analysis, detect those variables which are writable by multiple threads and hence need to be transactional. An initial assumption is that thread level parallelism will be exploited at relatively coarse grain and therefore an individual Scala function will be executed within a single thread. It will probably be necessary to relax this if and when we introduce data parallelism. The restrictions all relate to multiple update of variables. They are best expressed by positive statements of updates, which are allowed together with a negative statement which relates to variables being passed as function parameters. The following assumes that a whole program view is available, further thought needs to be given to separately compiled classes and libraries.

Rule 1: A variable may be the object of multiple updates if it is a static variable accessible by the scope rules. This is allowed whether or not the access occurs from multiple threads. This allows the use of static global variables defined in singleton objects which are required to be transactional if accessed by multiple threads.

Rule 2: A variable may be the object of multiple updates if it is declared locally within a function and the updates occur either directly in that function body or from within a nested function definition. The variable cannot be referenced from any separate threads which may be generated within the function or any nesting. Such variables are always thread local.

Rule 3: A variable may be the object of multiple updates if it is an instance variable of a class and the updates occur within a function defined in the class.

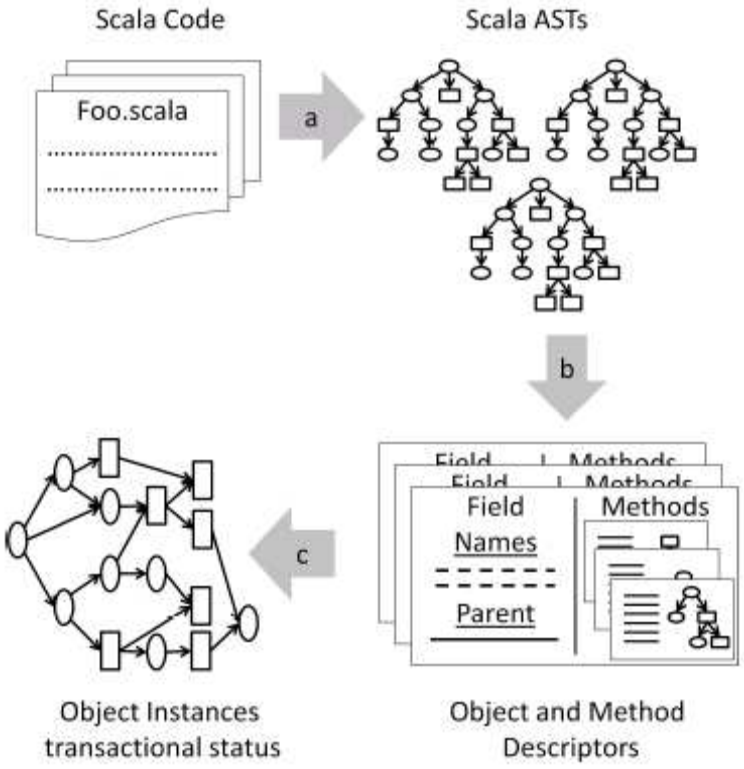
Rule 4: A variable may not be the object of multiple updates if it is referenced via a parameter passed to a function. The purpose of these rules is to prohibit the arbitrary distribution of updateable variables (or strictly references to them) via parameters. However, Rule 3 does permit the update of fields of objects by calls to functions defines in the objects class (i.e. via the “this” pointer).

Based on the above rules and our aims are:
1. To ensure that all reads and modifications to transactional state only occurs within a transaction.
2. To ensure that all state that is modified after becoming visible outside of the dataflow thread is marked as transactional.

3. To ensure that all state that does not become visible outside of the dataflow thread is marked as local state.

We have developed a prototype plugin for the Scala compiler which implements the static analysis required and show the potential. However, we are not making claims of completeness of all the corner cases, as the plugin is in early experimental stages.

The plugin follows the following phases:



3 High Performance Developers: C pragmas

3.1 StarSs

For the sake of easier reading, we shortly recall that BSC has investigated how speculation can be brought into StarSs [9], specifically SMPs, 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].

3.1.1 Speculation in StarSs

As described in D3.4, (section 5.1) 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);
}
```

For example 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. Compiler and runtime changes were required: StarSs and applications/evolutions reported.

3.1.2 Overhead of STM in the context of task speculation

In the fourth year of the project we have analyzed the overheads of where speculation is used to extract more parallelism in SMPSs, an implementation of StarSs [8]. SMPSs is a task-based programming model for Symmetric Multiprocessors (SMPs). Speculation is used to overcome the synchronization pragmas in SMPSs, which block the generation of work and lead to underutilization

of the available resources. TinySTM, a Software Transactional Memory (STM) library is used for an STM based implementation to achieve speculation. The `speculate` pragma can be used with if-condition and while-loop programming constructs to speculatively generate tasks blocked until the loop-predicate is evaluated. Speculatively generated tasks are executed as transactions in order to maintain correctness in case the speculation fails. TinySTM library calls are used to execute transactional SMPSs tasks. If the speculation fails a rollback is performed; the updates performed by the tasks are undone as the associated transaction is aborted.

We measured the overhead incurred due to the use of TinySTM library in SMPSs and we analyzed the acceptable overhead with the TinySTM-based implementation to achieve speculation. The speculative tasks, apart from being control dependent on the loop predicate, may also be data dependent on the earlier tasks. Hence, the use of the `speculate` pragma will add one of the following types of tasks to the SMPSs Task Dependency Graph (TDG) of an application:

- tasks which are control-dependent on the earlier tasks
- tasks which are data-dependent tasks on the earlier tasks

Tasks which are only control-dependent on the earlier tasks allows speculative and non-speculative tasks to execute in parallel, but with tasks, which are data-dependent on the earlier tasks, the only parallelism available is the overlap of task generation with task execution. We concentrate our analysis on applications where speculatively generated tasks are data-dependent on the earlier tasks since this is the minimum performance gained by the idea of task-speculation in SMPSs. The applications analyzed were Jacobi, Gauss-Seidel and Lee-routing. In case of Jacobi and Gauss-Seidel speculative tasks are data-dependent on the earlier tasks. In Lee-routing, `speculate` pragma was added to overcome a synchronization pragma that was used to enforce control dependence. The performance timings presented in the case of Lee-routing cover only the phase where the `speculate` pragma was added.

Figure 3 shows the performance of Lee-routing application with and without speculation. The timings were taken for a phase of the application where the `speculate` pragma was used and the benefits achieved due to the simultaneous execution of speculative tasks with earlier tasks is evident.

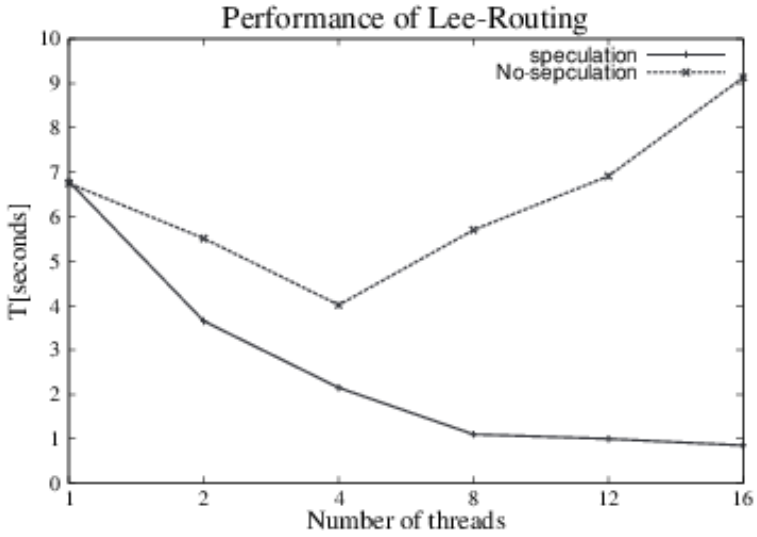


Figure 3 Performance of speculative and non-speculative Lee-routing application

The problem size for the Jacobi and Gauss-Seidel applications shown of the Figure 4 is a system of linear equations with 4096 unknowns. Each task processes 512KB block of data. The chosen task granularity gives optimal performance in the non-speculative version of the application. The figure shows that even though the speculative versions scale, the overhead incurred does not allow any performance improvement compared to the non-speculative versions. But with increase in the number of threads the absolute difference in the performance between the speculative and non-speculative versions reduces. With higher number of threads more resources are available to avail the parallelism extracted from the `speculate` pragma. This shows us that the idea can be successfully applied to

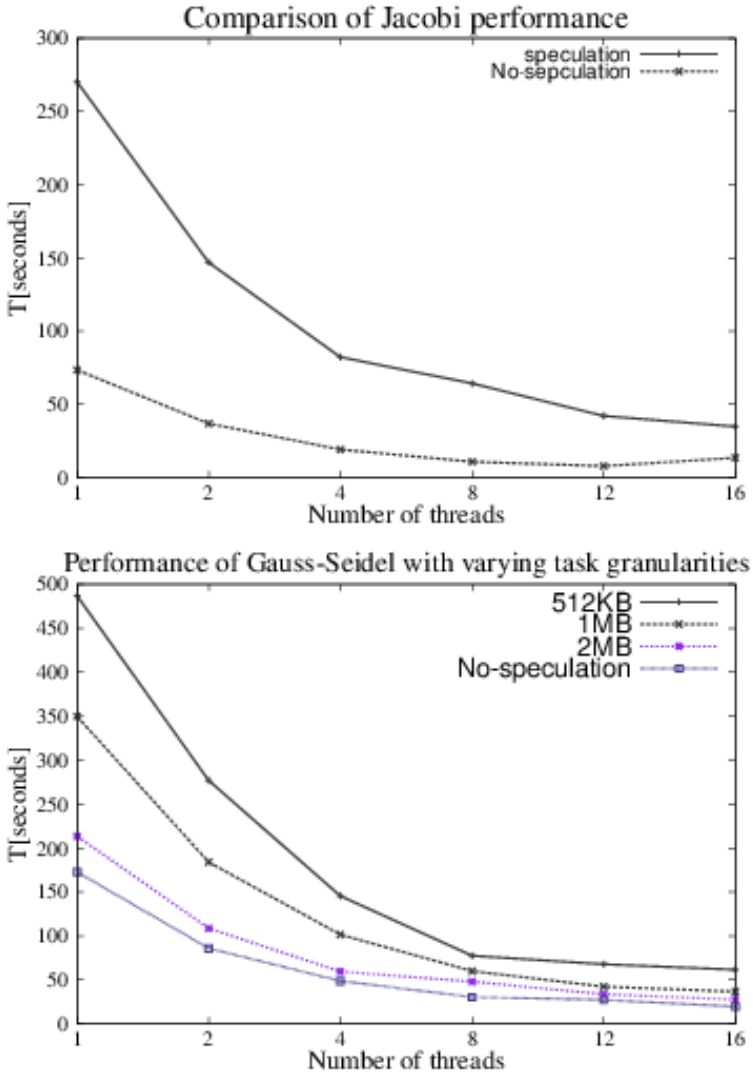


Figure 4 Performance of Jacobi while varying the number of threads and Gauss-Seidel applications while varying both number of threads (horizontal axis) and varying tasks granularities (different curves at 512KB, 1MB, 2MB granularities

obtain some scalability.

One of the major reasons of overhead with TinySTM is the conflict detection performed by the library. This is an unnecessary and unavoidable overhead. Unavoidable since it is a part of the library and unnecessary because of the presence of a task dependency graph. We also observed that with increase in the task granularity the speculative versions of the applications perform better as shown in Figure 4. The legend in the figure represents task-granularities.

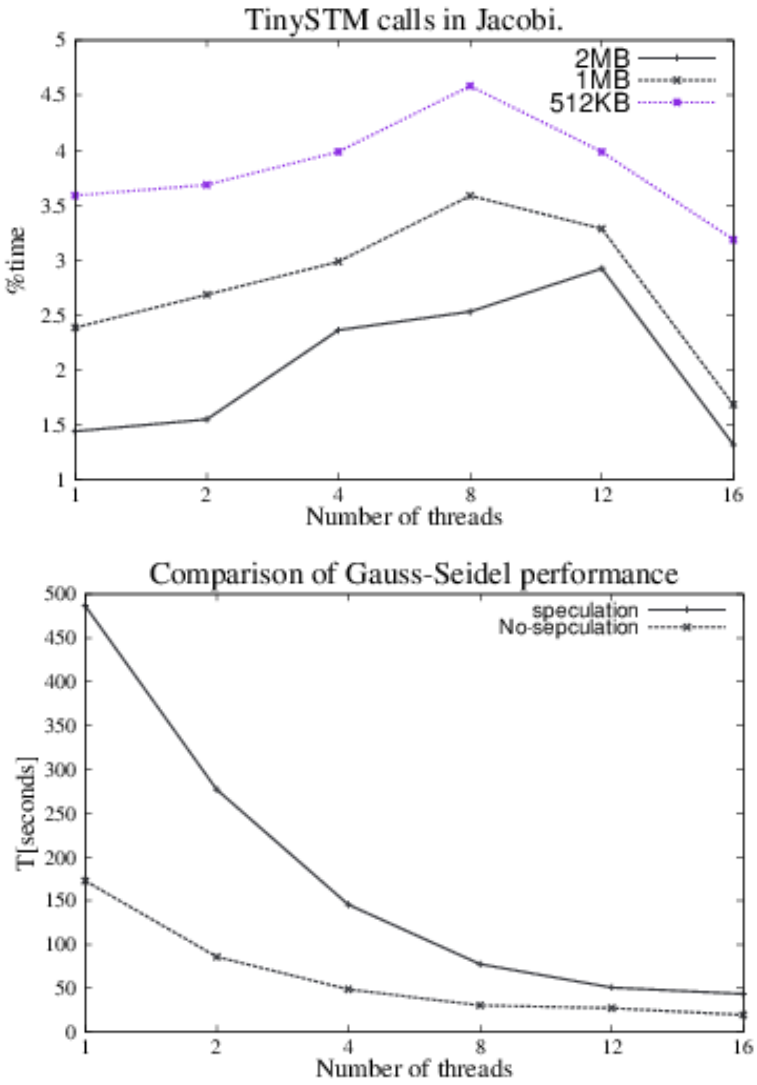


Figure 5 Relative time spent in the TinySTM library for speculative Jacobi and Gauss-Seidel applications while varying both number of threads (horizontal axis) and varying tasks granularities (different curves at 512KB, 1MB, 2MB granularities

We also evaluated the relative time spent by speculative versions of the applications in the TinySTM library and conclude that the overhead of the library should be less that 1% of the total execution time to gain any performance benefits.

3.2 - Integrating Dataflow in CAPS compiler (TF-OpenACC)

The OpenHMPP [14] and OpenACC propose 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 dataflow model investigated in the Workpackage 3 in a manner that is compatible with the existing OpenHMPP implementation and OpenACC. This deliverable focuses on the extensions made to OpenACC following the dataflow approach investigated in TERAFLUX: TF-OpenACC. Typically, OpenACC can be used on GPU and CPU. The CAPS many-core compiler is able to generate OpenCL as well as CUDA code when dealing with GPUs.

OpenACC [11] proposes a set of directives to describe kernels to remotely execute on an accelerator in parallel, and a set of data management techniques. This proposal is based on the OpenACC data transfer management mechanisms (See [12] for details) and on the integration of kernels inside codelets for the task computation description.

The constraints on the design of this extension that have been taken into account in the following way:

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

In the remainder of this document we describe TF-OpenACC for the C language. In the future, this extension will be proposed for FORTRAN. A variant will also be envisioned for C++.

The following sections describe the new directives to be added to OpenACC. We also present the constraint to the tasks codes (i.e. kernels) and give an overview of the data flow code region runtime behavior. We explain how the data are managed and describe a first implementation. This first implementation does not aim at being efficient but at demonstrating the concept.

3.2.1 New Directives Overview

TF-OpenACC is based on a new pair of directives describing the limits of a data flow region. In a region a set of asynchronous tasks are created. A task is an execution instance of a kernel section. The kernel has to be encapsulated inside a pure function called a DFCodelet, as defined in the OpenHMPP standard [13]. The synchronization between the tasks is performed according to the data dependencies between the tasks arguments.

Contrary to the OpenACC specification, the tasks are not necessarily and statically assigned to a particular device according to the owner compute rule of the arguments. However, this first region will restrict data flow region to accelerators having one single device or devices with a shared memory address space. The data are allocated using the "mirror" approach used in the CAPS compiler i.e. a data blocks on the Host has a mirrored version on the accelerator device updated according the OpenACC semantic and directives.

Data Flow Region (DFR)

The data flow region is delimited using an "acc dataflow" pragma on a statement block (denoted DFR hereafter). This is illustrated in **Figure 6**.

```
#pragma acc dataflow copyin(a,c), copyout(e)
{
     // set of statements
}//end of data flow region
```

Figure 6: TF-OpenACC Data Flow Region.

Data flow regions can contain the following statements:

- DFCodelet calls with a kernel directive;
- Statements which behavior is not affected by the tasks computations. Output arguments of tasks cannot be used in the region except for another kernel statement.

Data flow regions must have the same semantic as the sequential execution of the region. 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 exit of the DFR.

Figure 7gives an example of a region containing two DFCodelets. Variable A is an input to the region and variable C is the output. Variable B is an intermediate variable, which in and out values are ignored before and after. Task corresponding to compute2 is synchronized on the completion of compute1, assuming B is an input to compute2 and output of compute1.

The device clause on the kernel directive is an extension of OpenACC. It allows kernels to be executed on a specified device even if the arguments are located on a different device. Note that the first version will not support devices that do not share a common memory address space. By default, the same device is used for the same dataflow region.

```
void compute1(const float *a, float *b, const int n);

void compute2(const float *b, float *c, const int n);

void figure2(const int n, const float a[n], float b[n], float c[n])
{
#pragma acc dataflow copyin(a), create(b), copyout(c)
  {
#pragma acc kernels, pcopyin(a), pcopyout(b), device(1)
    compute1(a, b, n);

#pragma acc kernels, pcopyin(b), pcopyout(c), device(1)
    compute2(b, c, n);
  } // end of FD region
}
```

Figure 7: Example of a data flow region.

Figure 8 shows an example where k input tasks are connected to k output tasks. The dataflow region is launched multiple times asynchronously using a dataflow OpenACC parallel region. Inside each region two kernels are executed, and depending on the region a different kernel variant is chosen.

```
void compute1(float alpha, float *b, const int n);
void compute2(float beta, float *b, float *d, const int n);
void compute3(float beta, float *b, float *d, const int n);

void figure3(const int k, const int n, const float a[k], float b[n], float c[k], float
d[n])
{
  int index;
  for (index=0; index<k; index++) {
    float a_index = a[index], c_index = c[index];
#pragma acc dataflow copyin(n, a_index,c_index), copyout(d), async(index)
    {
#pragma acc kernels, pcopyin(a_index), pcopyout(b)
      compute1(a_index, b, n);

      if ((index == 0) || (index == k-1)) {
#pragma acc kernels, pcopyin(c_index,b), pcopyout(d)
        compute2(c_index, b, d, n);
      }
      else {
#pragma acc kernels, pcopyin(c_index,b), pcopyout(d)
        compute3(c_index, b, d, n);
      }
    } // end of FD region "index"
  } // end of loop

  for (index=0; index<k; index++) {
#pragma acc wait(index)
    ;
  }
 // Wait for all DFR
}
```

Figure 8: Example of a data flow region with the creation of multiple tasks.

Data Flow Region Characteristics

A data flow region describes a parameterized data flow graph with the following characteristics:

- The data dependencies between the tasks follows the sequential semantic of the C language. The execution of the DFR in parallel or sequentially leads to the same results (if no I/O status errors are in the code);
- The creation of tasks is driven by the statements in DFR block; The creation of tasks is independent of the tasks execution themselves;
- The task allocation on device is either allocated according to the owner compute rule of the mirrored data (default OpenACC behavior) or according to the device clause. Note that in future version, this later one may induce mirrors reallocations;
- All kernels inside a dataflow region are asynchronous;
- The internal data flow graph is limited to direct acyclic graphs (DAG).

Devices and Resources

All devices and mirrors are allocated prior entering the DFR.

3.2.2 Kernels in dataflow regions and DFCodelets

The proposed extension to OpenACC is based on the current concept of OpenHMPP Codelets. They are pure functions that can be remotely executed in a given address space.

In the context of this work, Codelets have a set of restrictions. Codelets arguments are limited to scalar and mirrored data

- the first version can be limited to mirrored data, scalar data would be supported in a second version,
- Codelets code generation must not lead to data exchange or synchronization with the master program
- For CUDA or OpenCL codes, it is composed of a sequence of kernel launches,
- It does not contain any implicit transfers,
- Reductions are not supported ;

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 pattern is given in **Figure 9**. The DFCodelet calls must be declared with an explicit description of data I/O status to ensure the proper declaration for the argument mode management:

```
#pragma acc kernels, pcopyin(A), pcopyout(C)
compute1(A, C);
```

Figure 9: DFCodelet pattern.

DFCodelet Granularity

DFCodelet granularity can encompass 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 (cf. D7.1, D6.1, D6.2, D6,3, D6.4).

DFCodelet Body Statements

There are no restrictions for the statements except that code generation must lead to one unique accelerator kernel. This constraint is necessary to ensure that no synchronization between the device and the host is needed to execute a task.

DFCodelet Inner Parallelism

DFCodelets are expected to exhibit parallelism in their computation. This parallelism can then be used to exploit SIMD/SIMT parallelism available in many computing cores. This is taken in charge by the CAPS compiler code generation.

3.2.3 Data Flow Region and Data

A data flow region (DFR) describes a parameterized data flow task graph. This introduces many limitations to the content of the region. A data flow region is executed as a slave of a master program.

Data Flow Region

A data flow region is described using a directive denoted "acc dataflow". This directive has two clauses:

- pcopyin or copyin(list of variables): list of variables (or addresses) that are not scalar variables and input to the region.
- pcopyout or copyout(list of variables): list of variables (or addresses) that are not scalar variables and output to the region.

Note that these clauses are identical to the clauses defined in the OpenACC standard. See [11] for a detailed description of the semantic.

```
#pragma acc dataflow copyin(n, a_index,c_index), copyout(d), async(index)
{
    . . .
}//end of data flow region
```

Figure 10: DFR directive.

Data Flow Region Statements

The DFR statements 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.

Figure 11 shows an example of incorrect statement in a DFR. The creation of the compute2 task depends on the value produced by the compute1 task that is not part of the considered model.

```
#pragma acc dataflow copyin(A, C), copyout(B)
{
  #pragma acc kernels, copyin(A), copyout(C)
  compute1(A,C) ;

   // !!! Forbidden dependency on C !!!
  if (C[i])
    #pragma acc kernels, copyin(C), copyout(B)
    compute2(C,B) ;
}//end of data flow region
```

Figure 11: Incorrect statement of a data flow region.

Data Flow Region Execution Model

There are two main parts in the execution model:

- Data flow execution inside the regions;
- The region inside the host program.

Master-Slave Execution

The execution of a DFR region is executed as a whole in synchronous mode with the host program, or asynchronously when the appropriate OpenACC "async"/"wait" clauses are used.

Dynamic Data Flow

The execution of the statement inside the DFR creates the data flow graph according to the DFCodelets and the data dependencies between the tasks argument. This model is very similar to the StarSs model.

Data Management

This section describes how data structures are handled in a DFR. There are three cases to consider:

- Data structures that are input to the DFR
- Data structures that are output to the DFR
- Data structures that are temporary structures to send and receive data between tasks

The allocation of data for a DFR follows the usual allocation mirroring mechanism of the CAPS compiler. In short, the DFR tasks compute on mirror data. This has multiple advantages:

The DFR can be executed in different address spaces than the host program. Mirrors can also be used by other compute phases that exploit the data parallel model of OpenACC. The tasks can themselves exploit the data parallel code generation of OpenACC. Tasks can also be executed on the host. A rollback mechanism can be implemented (for conformant codes).

Block Data Allocation

The allocation of the data structures have to follow these rules:

- All data structures are contiguous memory block.
- All data/mirror are allocated prior to entering a region including DFCodelet arguments,
- Rollback mechanism is performed by restoring region in-out data.

Dealing with Multiple Address Spaces

A mirror can only belong to one address space. As a consequence if DFCodelets are exchanging data from different address spaces, mirrors would need to migrate from one device to another one. In the current version, no data exchanges are supported. The DFR can support multiple devices if they share the same memory address space.

DFCodelet Arguments

DFCodelet arguments of two kinds:

- Scalar variables
- Allocated structures are contiguous block of memories that are managed as mirrored data. In this case, argument can be sub-block of a main block

Data Related Synchronizations

DFCodelet arguments are seen as tokens:

- No synchronizations based on scalar variables are allowed
- Cannot synchronize on mirror that are on different devices

Synchronizations are performed at the level of the allocated mirrored, not at the level of sub-blocks that may be used by DFCodelets.

Dataflow management: an instance at runtime of each thread argument is associated to one synchronization token (very much like in [7]).

3.2.4 Implementation, features and restrictions

This section presents implementation options. As a first step, the proof of concept is based on current CAPS manycore compiler version 3. A data flow task library may be added to trigger the task execution. This library makes the interface with the CAPS compiler runtime that provides support for allocation memory and resources.

Implementation restrictions

Multiple Files Limitations **-** DFCodelets can be declared in multiple files but the data flow regions is defined in a unique file.

Source Language **-** This work is limited to the C code.

Target Language **-** CUDA, OpenCL Accelerator, and CPU.

A task management library

This library implements a data flow manager on top of the CAPS compiler public runtime API. This library main function is to track data dependencies to trigger DFCodelets execution. It can be for example based on the light weighted thread (QLib - Sandia).

Debugging

The CAPS compiler describes clearly the dataflow computed at compile time using a text report and a graphic representation of the dependences using the "graphviz" library for instance. Then, the work is left to usual debugger that understands HMPP (e.g. Allinea DDT).

Prototype and evaluation

The TF-OpenACC extension specification has led to the implementation of an operational prototype based on the CAPS many-core compiler suite version 3.3.1. This prototype is a fork of the CAPS compiler and has not been integrated inside the commercial version of the compiler currently in version 3.4.3.

The prototype supports all the features described in this document excepted the support of multiple devices (with the keyword extension "device(n)"). It has been validated on a various set of tests among the following:

- The simple example provided by the specification in figure 2, Appendix A1,
- The task distribution example provided by the specification in figure 3, see Appendix A2,
- A scatter/gather example, see Appendix A3,

In all these examples, the data dependences computed at compile time are provided. The OpenACC target used is CUDA on an NVidia GPU.

The validation machine has the following specification:

- Dual Socket Intel(R) Xeon(R) CPU X5560 @ 2.80GHz (total of 8 physical cores),
- 24 Go of RAM,
- x86_64 GNU/Linux version 3.11.0-18-generic
- NVidia GPU, GeForce GTS 450
- CUDA SDK version 5.0.23
- GNU C Compiler version 4.4.7

3.3 OpenStream and Owner Writeable Memory

We recall, for easy of reading, that in the previous period INRIA has worked on streaming dataflow using a directive-based approach. We coined the name OpenStream for these dataflow streaming extensions of OpenMP 3.0:

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

OpenStream is an expressive programming model to allow the composition of tasks communicating through first-class dataflow 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.

Additionally, in the fourth period of the project, further extended the functional nature of pure dataflow programs implies that all operations are side-effect free. The absence of side effect means that if tokens are allowed to carry vectors, arrays, or other complex data structures, an operation on a data structure results in a new data structure. The problem of efficiently representing and manipulating complex data structures in a dataflow execution model has remained a fundamental and practical challenge. Owner Writable Memory (OWM) has been proposed in TERAFLUX to manage complex data structures in dataflow programs. The name and idea origins from our collaborator Prof. Ian Watson from University of Manchester (cf. D7.1). OWM implements a globally addressable memory (in software or hardware, depending on the instantiation). Before a thread could write to a portion of memory, it has to claim ownership beforehand. At any time point only the thread who has the ownership of the memory could write to it. When write ownership is successfully acquired, any read from another thread is not guaranteed to see consistent data. When write ownership is released, a consistent view of data must be visible to any other thread. Note the release operation could be performed explicitly by the thread or implicitly by the model. The latter is achieved when the OWM is used by a thread to write its results, which are made available to the consumer thread upon the completion of the execution of the thread. This memory can serve the requirements of the single assignment semantics required for functional objects. However, the ability for other threads to subsequently reclaim write ownership adds to flexibility of usage. Please note that unlike classical acquire/release", OWM is not a synchronization algorithm. It relies on external synchronization and dependence enforcement mechanisms (dataflow) to implement race-free in-place communication. It also defines a global address space.

OWM is integrated into the OpenStream compiler as a language extension.

The OWM extension of OpenStream takes the form of a simple "cache" clause in the task pragma:

```
#pragma omp task cache (ACCESS_MODE: MEM[OFF:SIZE])
```

The cache clause subscribes the task with the OWM subregion described by `MEM[off:size]` and "ACCESS_MODE" can be read (R),write (W) or read-write (RW). The current clause syntax supports only one dimensional arrays, but it may easily be extended to multiple dimension arrays.

```
int sync __attribute__ ((stream));

DATA *A = tstar_owm_alloc (N * N * sizeof (DATA));
/* task 1. */
#pragma omp task cache (W: A[:N*N]) output (sync)
{
  for (i = 0; i < N; i++)
    A[i][i] = i;
}
/* task 2. */
#pragma omp task cache (R: A[:N*N]) input (sync)
{
  for (i = 0; i < N; i++)
    ... = A[i][i];
}
```

OWM extension to OpenStream

The simple usage of the pragma is described above. `tstar_owm_alloc` allocates the OWM memory with size `N*N*sizeof(DATA)`. Task 1 writes to this OWM memory region and task 2 reads from this OWM region. Note that two tasks are synchronized by stream sync, task 2 will only be executed when task 1 finishes. Use cases of OWM in OpenStream are presented in the WP2 and WP7 deliverables.

4 Summary

This document has described the research carried out in the WP3 of the TERAFLUX project during the fourth year. We have covered progress with C-directive-based dataflow models (StarSs, CAPS, OpenStream). CAPS (our commercial partner is WP3) has shown how they have included dataflow directives to OpenACC. BSC has continued with the work on using speculation as part of dataflow to increase parallelism available. BSC has reported the analysis of overhead that TinySTM brings and offer some light of when it would be profitable to use speculation given those overheads. With Scala, UNIMAN has provided more evidence of the advantages of bringing together dataflow and transactional memory by looking a Pregel (a distributed framework for Graphs published by Google. UNIMAN has also reported their progress on allowing developers to express preferences for task scheduling as well as facilitating the correct usage of the different types of TERAFLUX memory. With OpenStream, INRIA has reported how Owner Writable Memory can now be express in the language and in WP2 and WP7 further information can be found about the performance improvements derived. This deliverable has covered the work being carried out in T3.4.

Overall, the programming models have matured with significant number of applications being ported (see WP2 deliverable) and most of the tools are available to be downloaded as open-source tools to increase dissemination and impact.

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 have 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 by CAPS provides an industrial perspective of what features/functionalities are well understood.

References

[1] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and dataflow 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. EuroPAR 2012: 514-525.

[11] OpenACC Consortium, "The OpenACC Application Programming Interface Version 2.0," 17 06 2013. [Online]. Available: http://www.openacc-standard.org/node/297.

[12] CAPS entreprise, HMPP Directives Reference Manual, Version 3.2.0, 2012.

[13] OpenHMPP Consortium Association, "OpenHMPP New Standard for Many-Core," 10 06 2011. [Online]. Available: http://www.openhmpp.org/. [Accessed 10 12 2011].

[14] NVidia, "NVIDIA, Cray, PGI, CAPS Unveil 'OpenACC' Programming Standard for Parallel Computing," 11 14 2011. [Online]. Available: http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09 &version=live&prid=821214 &releasejsp=release_157.. [Accessed 01 09 2012].

[15] The OpenCL Specification v1.1 r36, "The OpenCL Specification," 30 9 2010. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[16] HPCwire, "CAPS Entreprise Now Supports OpenACC Standard," 02 05 2012. [Online]. Available: http://www.hpcwire.com/hpcwire/2012-05-02/caps_entreprise_now_supports_openacc_standard.html.

[17] Marco Solinas, Rosa M Badia, François Bodin, Albert Cohen, Paraskevas Evripidou, Paolo Faraboschi, Bernhard Fechner, Guang R Gao, Arne Garbade, Sylvain Girbal, Daniel Goodman, Behran Khan, Souad Koliai, Feng Li, Mikel Luján, Laurent Morin, Avi Mendelson, Nacho Navarro, Antoniu Pop, Pedro Trancoso, Theo Ungerer, Mateo Valero, Sebastian Weis, Ian Watson, Stéphane Zuckermann, Roberto Giorgi. The TERAFLUX Project: Exploiting the DataFlow Paradigm in Next Generation Teradevices. In Proceedings of the 2013 Euromicro Conference on Digital System Design (DSD), 272-279.

[18] Roberto Giorgi, Rosa M Badia, François Bodin, Albert Cohen, Paraskevas Evripidou, Paolo Faraboschi, Bernhard Fechner, Guang R Gao, Arne Garbade, Rahul Gayatri, Sylvain Girbal, Daniel Goodman, Behran Khan, Souad Koliaï, Joshua Landwehr, Nhat Minh Lê, Feng Li, Mikel Lujàn, Avi Mendelson, Laurent Morin, Nacho Navarro, Tomasz Patejko, Antoniu Pop, Pedro Trancoso, Theo Ungerer, Ian Watson, Sebastian Weis, Stéphane Zuckerman, Mateo Valero. TERAFLUX: Harnessing Dataflow in Next Generation Teradevices. Journal Microprocessors and Microsystems, 2014. http://www.sciencedirect.com/science/article/pii/S0141933114000490

[19] A. Diavastos, P. Trancoso, M. Lujan and I. Watson, “Integrating Transactions into the Data-Driven Multi-threading Model using the TFlux Platform” in Proc. of the Data-Flow Execution Models for Extreme Scale Computing (DFM) Workshop, Galveston, Texas, U.S.A., October 2011

Appendix A – CAPS Compiler: Reference codes annexes

Appendix A1: Basic synchronization example

- Source code

```
void compute1(const float *a, float *b, const int n);

void compute2(const float *b, float *c, const int n);

void figure2(const int n, const float a[n], float b[n], float c[n])
{
#pragma acc dataflow copyin(a), create(b), copyout(c)
  {
#pragma acc kernels, pcopyin(a), pcopyout(b)
    compute1(a, b, n);

#pragma acc kernels, pcopyin(b), pcopyout(c)
    compute2(b, c, n);
  } // end of FD region
}

void compute1(const float *a, float *b, const int n)
{
  int i;
/* #pragma omp parallel for */
  for (i=0; i<n; ++i)
    {
      b[i] = a[i] / 3.14f;
    }
}

void compute2(const float *b, float *c, const int n)
{
  int i;
/* #pragma omp parallel for */
  for (i=0; i<n; ++i)
    {
      c[i] = b[i] * b[i];
    }
}

extern void fill(const int n, const float value, float t[n]);

#define N 300000
static const int n = N;
float a[N];
float b[N];
float c[N];
void example(void)
{
  fill(n, 2, a);
  fill(n, 1.578, b);
  fill(n, 1.04, c);
  figure2(n, a, b, c);
  return 0;
}
```

- Compilation output

```
hmpp -k gcc -c -Wall -I/home/laorans/travail/DataFlow/HMPP-
DataFLow/build/hmpp/x86_64/debug//include figure2.c -o figure2.o
Parse acc dataflow copyin(a), create(b), copyout(c)
Create region figure2.c:8
```

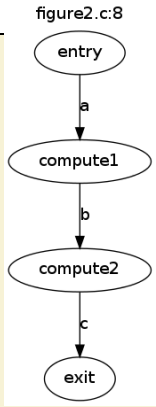


Figure 12, figure2.c, Data Dependencies computed at compile time

```
Parse acc kernels, pcopyin(a), pcopyout(b)
Add call compute1(a, b, n) to region
Parse acc kernels, pcopyin(b), pcopyout(c)
Add call compute2(b, c, n) to region
1 regions found
Create CFG for: figure2.c:8
0: a
1: b
2: c
3: n
Found node compute1(a, b, n)
Found node compute2(b, c, n)
Dataflow CFG is:
CFG for region figure2.c:8
0: a
1: b
2: c
3: n
4 nodes:
        a       b       c       n
0: <entry>:
        write   none    none    none
2: compute1:
        read    write   none    none
3: compute2:
        none    read    write   none
1: <exit>:
        none    none    read    none
3 edges:
<entry> --> compute1
compute1 --> compute2
compute2 --> <exit>
Processing figure2.c:8
Build data dependencies for region figure2.c:8
Process node <exit>
Process node compute2
addDEdge (compute2,<exit>,'c')
Process node compute1
addDEdge (compute1,compute2,'b')
Process node <entry>
addDEdge (<entry>,compute1,'a')
q 0 : (<entry>-exe, compute1-exe, compute2-exe)
q 1 : (<exit>-wait for q0, <exit>-exe)
Found 2 queues
Written figure2_4ryhwv.halt.i
hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] figure2.c:21: Loop 'i' was
shared among gangs(192) and workers(256)
(last message repeated 1 more time)
hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__figure2_16__0q7vrdyf_cuda.hmf.cu".
hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] figure2.c:31: Loop 'i' was
shared among gangs(192) and workers(256)
(last message repeated 1 more time)
hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__figure2_20__pscahkc2_cuda.hmf.cu".
hmpp: [Warning HP0391] <preprocessor>:12: Variable 'n' in Data clause has no effect if
read-only in the enclosed Kernel/Parallel regions
<preprocessor>: In function 'figure2':
<preprocessor>:36: warning: implicit declaration of function 'openacci_set_device_hint'
<preprocessor>:12: warning: implicit declaration of function 'openacci_enter_region'
<preprocessor>:12: warning: implicit declaration of function 'openacci_push_data'
<stdin>:1: warning: implicit declaration of function 'openacci_call'
<stdin>:1: warning: implicit declaration of function 'openacci_fallback'
<stdin>:1: warning: implicit declaration of function 'openacci_leave_region'

<preprocessor>:24: warning: implicit declaration of function 'openacci_wait'
figure2.c: In function 'example':
figure2.c:50: warning: 'return' with a value, in function returning void
figure2.c: In function 'hmppsi_lookup':
figure2.c:56: warning: implicit declaration of function 'hmpprti_lookup_grouplet'
figure2.c:56: warning: return makes pointer from integer without a cast
figure2.c: At top level:
figure2.c:54: warning: 'hmppsi_lookup' defined but not used
```

```
hmpp -k gcc -Wall main.o figure2.o -o test.exe
```

- Execution output

```
./test.exe
[     0.207778] ( 0) INFO : Enter    data (queue=none, location=<preprocessor>:12)
[     0.208117] ( 0) INFO : Acquire  (target=cuda)
[     0.208357] ( 0) INFO : Acquired (device='cuda#0 [GeForce GTS 450]')
[     0.208496] ( 0) INFO : Allocate a[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:12)
[     0.233900] ( 0) INFO : Upload   a[0:300000] (element_size=4, queue=none,
location=<preprocessor>:12)
[     0.234515] ( 0) INFO : Allocate b[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:12)
[     0.234719] ( 0) INFO : Allocate c[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:12)
[     0.234919] ( 0) INFO : Allocate n[0:1] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:12)
[     0.235240] ( 0) INFO : Enter    kernels (queue=0, location=<preprocessor>:16)
[     0.236452] ( 0) INFO : Allocate __hmpp_vla_sizes__a[0:1] (element_size=8,
memory_space=host, queue=0, location=<preprocessor>:16)
[     0.236572] ( 0) INFO : Upload   __hmpp_vla_sizes__a[0:1] (element_size=8, queue=0,
location=<preprocessor>:16)
[     0.236673] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=0, location=<preprocessor>:16)
[     0.236759] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=0,
location=<preprocessor>:16)
[     0.236853] ( 0) INFO : Call     __hmpp_acc_region__figure2_16__0q7vrdyf (queue=0,
location=<preprocessor>:16)
[     0.237048] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=0,
location=<preprocessor>:16)
[     0.237192] ( 0) INFO : Free     __hmpp_vla_sizes__a[0:1] (element_size=8, queue=0,
location=<preprocessor>:16)
[     0.237281] ( 0) INFO : Leave    kernels (queue=0, location=<preprocessor>:16)
[     0.237378] ( 0) INFO : Enter    kernels (queue=0, location=<preprocessor>:20)
[     0.237785] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=0, location=<preprocessor>:20)
[     0.237876] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=0,
location=<preprocessor>:20)
[     0.237950] ( 0) INFO : Allocate __hmpp_vla_sizes__c[0:1] (element_size=8,
memory_space=host, queue=0, location=<preprocessor>:20)
[     0.238033] ( 0) INFO : Upload   __hmpp_vla_sizes__c[0:1] (element_size=8, queue=0,
location=<preprocessor>:20)
[     0.238124] ( 0) INFO : Call     __hmpp_acc_region__figure2_20__pscahkc2 (queue=0,
location=<preprocessor>:20)
[     0.238255] ( 0) INFO : Free     __hmpp_vla_sizes__c[0:1] (element_size=8, queue=0,
location=<preprocessor>:20)
[     0.238346] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=0,
location=<preprocessor>:20)
[     0.238434] ( 0) INFO : Leave    kernels (queue=0, location=<preprocessor>:20)
[     0.238506] ( 0) INFO : Wait     (queue=none, awaited=0, location=<preprocessor>:24)
[     0.238776] ( 0) INFO : Free     n[0:1] (element_size=4, queue=none,
location=<preprocessor>:12)
[     0.239049] ( 0) INFO : Download c[0:300000] (element_size=4, queue=none,
location=<preprocessor>:12)
[     0.239751] ( 0) INFO : Free     c[0:300000] (element_size=4, queue=none,
location=<preprocessor>:12)
[     0.239916] ( 0) INFO : Free     b[0:300000] (element_size=4, queue=none,
location=<preprocessor>:12)
[     0.240081] ( 0) INFO : Free     a[0:300000] (element_size=4, queue=none,
location=<preprocessor>:12)
[     0.240243] ( 0) INFO : Leave    data (queue=none, location=<preprocessor>:12)
start
done
```

Appendix A2: Control flow synchronization example

- Source Code

```
void compute1(float alpha, float *b, const int n);
void compute2(float beta, float *b, float *d, const int n);
void compute3(float beta, float *b, float *d, const int n);

void figure3(const int k, const int n, const float a[k], float b[n], float c[k], float
d[n])
{
  int index;
  for (index=0; index<k; index++) {
    float a_index = a[index], c_index = c[index];
#pragma acc dataflow copyin(n, a_index,c_index), copyout(d), async(index)
    {
#pragma acc kernels, pcopyin(a_index), pcopyout(b)
      compute1(a_index, b, n);

      if ((index == 0) || (index == k-1)) {
#pragma acc kernels, pcopyin(c_index,b), pcopyout(d)
        compute2(c_index, b, d, n);
      }
      else {
#pragma acc kernels, pcopyin(c_index,b), pcopyout(d)
        compute3(c_index, b, d, n);
      }
    } // end of FD region
  } // end of loop

  for (index=0; index<k; index++) {
#pragma acc wait(index)
    ;
  }
}
void compute1(const float alpha, float *b, const int n)
{
  int i;
/* #pragma omp parallel for */
  for (i=0; i<n; ++i)
    {
      b[i] = alpha * alpha * i;
    }
}

void compute2(float beta, float *b, float *d, const int n)
{
  int i;
/* #pragma omp parallel for */
  for (i=0; i<n; ++i)
    {
      d[i] = b[i] / beta;
    }
}

void compute3(float beta, float *b, float *d, const int n)
{
  int i;
/* #pragma omp parallel for */
  for (i=0; i<n; ++i)
    {
      d[i] = b[i] + beta;
    }
}
```

```
extern void fill(const int n, const float value, float t[n]);
#define K 16
#define N 300000
static const int k = K;
static const int n = N;
float a[K];
float b[N];
float c[K];
float d[n];
void example(void)
{
  fill(k, 2, a);
  fill(n, 1.578, b);
  fill(k, 1.04, c);
  fill(n, 1.699, d);
  figure3(k, n, a, b, c, d);
  return;
}
```

- Compilation output

```
hmpp -k gcc -c -Wall -I/home/laorans/travail/DataFlow/HMPP-
DataFLow/build/hmpp/x86_64/debug//include main.c -o main.o
0 regions found
Written main_7Gt6tX.halt.i
main.c: In function 'main':
main.c:16: warning: implicit declaration of function 'printf'
main.c:16: warning: incompatible implicit declaration of built-in function 'printf'
hmpp -k gcc -c -Wall -I/home/laorans/travail/DataFlow/HMPP-
DataFLow/build/hmpp/x86_64/debug//include figure3.c -o figure3.o
Parse acc dataflow copyin(n, a_index,c_index), copyout(d), async(index)
Create region figure3.c:11
Parse acc kernels, pcopyin(a_index), pcopyout(b)
Add call compute1(a_index, b, n) to region
Parse acc kernels, pcopyin(c_index,b), pcopyout(d)
Add call compute2(c_index, b, d, n) to region
Parse acc kernels, pcopyin(c_index,b), pcopyout(d)
Add call compute3(c_index, b, d, n) to region
Parse acc wait(index)
1 regions found
Create CFG for: figure3.c:11
0: a_index
1: b
2: c_index
3: d
4: n
Found node compute1(a_index, b, n)
Found node compute2(c_index, b, d, n)
Found node compute3(c_index, b, d, n)
Node trueBlock-0 has 1||0 pred
Link compute1 to compute2
Node falseBlock-0 has 1||0 pred
Link compute1 to compute3
Node after-0 has 1||0 succ
Link compute2 to <exit>
Link compute3 to <exit>
Dataflow CFG is:
CFG for region figure3.c:11
0: a_index
1: b
2: c_index
3: d
4: n
```

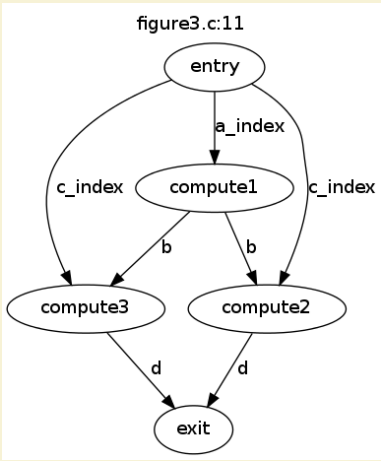


Figure 13: figure3.c, Data Dependencies computed at compile time

```
5 nodes:
        a_index b       c_index d       n
0: <entry>:
        write   none    write   none    write
2: compute1:
        read    write   none    none    none
3: compute2:
        none    read    read    write   none
4: compute3:
        none    read    read    write   none
1: <exit>:
        none    none    none    read    none
5 edges:
<entry> --> compute1
compute1 --> compute2
compute1 --> compute3
compute2 --> <exit>
compute3 --> <exit>
Processing figure3.c:11
Build data dependencies for region figure3.c:11
Process node <exit>
Process node compute3
addDEdge (compute3,<exit>,'d')
Process node compute2
addDEdge (compute2,<exit>,'d')
Process node compute1
addDEdge (compute1,compute3,'b')
addDEdge (compute1,compute2,'b')
Process node <entry>
addDEdge (<entry>,compute1,'a_index')
addDEdge (<entry>,compute3,'c_index')
addDEdge (<entry>,compute2,'c_index')
q 0 : (<entry>-exe)
q 1 : (compute1-exe)
q 2 : ()
q 3 : (compute3-wait for q1, compute3-exe)
q 4 : (<exit>-wait for q3, <exit>-wait for q6, <exit>-exe)
q 5 : ()
q 6 : (compute2-wait for q1, compute2-exe)
Found 7 queues
Written figure3_vzaHXt.halt.i
hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] figure3.c:37: Loop 'i' was
shared among gangs(192) and workers(256)
(last message repeated 1 more time)
hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__figure3_19__3l2rxy37_cuda.hmf.cu".
hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] figure3.c:47: Loop 'i' was
shared among gangs(192) and workers(256)
(last message repeated 1 more time)
hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__figure3_26__v3352zdi_cuda.hmf.cu".
hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] figure3.c:57: Loop 'i' was
shared among gangs(192) and workers(256)
(last message repeated 1 more time)
hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__figure3_33__n5wy5sgy_cuda.hmf.cu".
hmpp: [Warning HP0391] <preprocessor>:15: Variable 'a_index' in Data clause has no effect
if read-only in the enclosed Kernel/Parallel regions
hmpp: [Warning HP0391] <preprocessor>:15: Variable 'c_index' in Data clause has no effect
if read-only in the enclosed Kernel/Parallel regions
hmpp: [Warning HP0391] <preprocessor>:15: Variable 'n' in Data clause has no effect if
read-only in the enclosed Kernel/Parallel regions
<preprocessor>: In function 'figure3':
<preprocessor>:45: warning: implicit declaration of function 'openacci_set_device_hint'
<preprocessor>:15: warning: implicit declaration of function 'openacci_enter_region'
<preprocessor>:15: warning: implicit declaration of function 'openacci_push_data'
<stdin>:1: warning: implicit declaration of function 'openacci_call'
<stdin>:1: warning: implicit declaration of function 'openacci_fallback'
<stdin>:1: warning: implicit declaration of function 'openacci_leave_region'
<preprocessor>:25: warning: implicit declaration of function 'openacci_wait'
figure3.c: In function 'hmppsi_lookup':
figure3.c:85: warning: implicit declaration of function 'hmpprti_lookup_grouplet'
figure3.c:85: warning: return makes pointer from integer without a cast
```

```
figure3.c: At top level:
figure3.c:83: warning: 'hmppsi_lookup' defined but not used
hmpp -k gcc -Wall main.o figure3.o -o test.exe
```

- Execution output

```
./test.exe
[     0.201299] ( 0) INFO : Enter    data (queue=none, location=<preprocessor>:15)
[     0.201616] ( 0) INFO : Acquire  (target=cuda)
[     0.201844] ( 0) INFO : Acquired (device='cuda#0 [GeForce GTS 450]')
[     0.201979] ( 0) INFO : Allocate a_index[0:1] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.227291] ( 0) INFO : Upload   a_index[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.227532] ( 0) INFO : Allocate b[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.227738] ( 0) INFO : Allocate c_index[0:1] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.227843] ( 0) INFO : Upload   c_index[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.227936] ( 0) INFO : Allocate d[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.228125] ( 0) INFO : Allocate n[0:1] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.228227] ( 0) INFO : Upload   n[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.228433] ( 0) INFO : Enter    kernels (queue=1, location=<preprocessor>:19)
[     0.229598] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=1, location=<preprocessor>:19)
[     0.229723] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=1,
location=<preprocessor>:19)
[     0.229838] ( 0) INFO : Call     __hmpp_acc_region__figure3_19__3l2rxy37 (queue=1,
location=<preprocessor>:19)
[     0.230031] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=1,
location=<preprocessor>:19)
[     0.230194] ( 0) INFO : Leave    kernels (queue=1, location=<preprocessor>:19)
[     0.230308] ( 0) INFO : Wait     (queue=6, awaited=1, location=<preprocessor>:25)
[     0.230391] ( 0) INFO : Enter    kernels (queue=6, location=<preprocessor>:26)
[     0.230816] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=6, location=<preprocessor>:26)
[     0.230911] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=6,
location=<preprocessor>:26)
[     0.230985] ( 0) INFO : Allocate __hmpp_vla_sizes__d[0:1] (element_size=8,
memory_space=host, queue=6, location=<preprocessor>:26)
[     0.231066] ( 0) INFO : Upload   __hmpp_vla_sizes__d[0:1] (element_size=8, queue=6,
location=<preprocessor>:26)
[     0.231142] ( 0) INFO : Call     __hmpp_acc_region__figure3_26__v3352zdi (queue=6,
location=<preprocessor>:26)
[     0.231254] ( 0) INFO : Free     __hmpp_vla_sizes__d[0:1] (element_size=8, queue=6,
location=<preprocessor>:26)
[     0.231344] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=6,
location=<preprocessor>:26)
[     0.231424] ( 0) INFO : Leave    kernels (queue=6, location=<preprocessor>:26)
[     0.231491] ( 0) INFO : Wait     (queue=none, awaited=3, location=<preprocessor>:38)
[     0.231562] ( 0) INFO : Wait     (queue=none, awaited=6, location=<preprocessor>:39)
[     0.231629] ( 0) INFO : Free     n[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.231729] ( 0) INFO : Download d[0:300000] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.232365] ( 0) INFO : Free     d[0:300000] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.232558] ( 0) INFO : Free     c_index[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.232658] ( 0) INFO : Free     b[0:300000] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.232807] ( 0) INFO : Free     a_index[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.232962] ( 0) INFO : Leave    data (queue=none, location=<preprocessor>:15)

(...)
```

```
[     0.295439] ( 0) INFO : Enter    data (queue=none, location=<preprocessor>:15)
[     0.295502] ( 0) INFO : Allocate a_index[0:1] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.295715] ( 0) INFO : Upload   a_index[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.295814] ( 0) INFO : Allocate b[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.295999] ( 0) INFO : Allocate c_index[0:1] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.296101] ( 0) INFO : Upload   c_index[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.296192] ( 0) INFO : Allocate d[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.296376] ( 0) INFO : Allocate n[0:1] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:15)
[     0.296479] ( 0) INFO : Upload   n[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.296573] ( 0) INFO : Enter    kernels (queue=1, location=<preprocessor>:19)
[     0.296647] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=1, location=<preprocessor>:19)
[     0.296729] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=1,
location=<preprocessor>:19)
[     0.296803] ( 0) INFO : Call     __hmpp_acc_region__figure3_19__3l2rxy37 (queue=1,
location=<preprocessor>:19)
[     0.296872] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=1,
location=<preprocessor>:19)
[     0.296955] ( 0) INFO : Leave    kernels (queue=1, location=<preprocessor>:19)
[     0.297020] ( 0) INFO : Wait     (queue=6, awaited=1, location=<preprocessor>:25)
[     0.297092] ( 0) INFO : Enter    kernels (queue=6, location=<preprocessor>:26)
[     0.297163] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=6, location=<preprocessor>:26)
[     0.297245] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=6,
location=<preprocessor>:26)
[     0.297317] ( 0) INFO : Allocate __hmpp_vla_sizes__d[0:1] (element_size=8,
memory_space=host, queue=6, location=<preprocessor>:26)
[     0.297395] ( 0) INFO : Upload   __hmpp_vla_sizes__d[0:1] (element_size=8, queue=6,
location=<preprocessor>:26)
[     0.297468] ( 0) INFO : Call     __hmpp_acc_region__figure3_26__v3352zdi (queue=6,
location=<preprocessor>:26)
[     0.297535] ( 0) INFO : Free     __hmpp_vla_sizes__d[0:1] (element_size=8, queue=6,
location=<preprocessor>:26)
[     0.297616] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=6,
location=<preprocessor>:26)
[     0.297695] ( 0) INFO : Leave    kernels (queue=6, location=<preprocessor>:26)
[     0.297758] ( 0) INFO : Wait     (queue=none, awaited=3, location=<preprocessor>:38)
[     0.297827] ( 0) INFO : Wait     (queue=none, awaited=6, location=<preprocessor>:39)
[     0.297892] ( 0) INFO : Free     n[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.297984] ( 0) INFO : Download d[0:300000] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.298548] ( 0) INFO : Free     d[0:300000] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.298704] ( 0) INFO : Free     c_index[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.298803] ( 0) INFO : Free     b[0:300000] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.298950] ( 0) INFO : Free     a_index[0:1] (element_size=4, queue=none,
location=<preprocessor>:15)
[     0.299103] ( 0) INFO : Leave    data (queue=none, location=<preprocessor>:15)
[     0.299177] ( 0) INFO : Wait     (queue=none, awaited=0, location=figure3.c:27)
[     0.299246] ( 0) INFO : Wait     (queue=none, awaited=1, location=figure3.c:27)
[     0.299309] ( 0) INFO : Wait     (queue=none, awaited=2, location=figure3.c:27)
[     0.299370] ( 0) INFO : Wait     (queue=none, awaited=3, location=figure3.c:27)
[     0.299432] ( 0) INFO : Wait     (queue=none, awaited=4, location=figure3.c:27)
[     0.299495] ( 0) INFO : Wait     (queue=none, awaited=5, location=figure3.c:27)
[     0.299555] ( 0) INFO : Wait     (queue=none, awaited=6, location=figure3.c:27)
[     0.299618] ( 0) INFO : Wait     (queue=none, awaited=7, location=figure3.c:27)
[     0.299681] ( 0) INFO : Wait     (queue=none, awaited=8, location=figure3.c:27)
[     0.299743] ( 0) INFO : Wait     (queue=none, awaited=9, location=figure3.c:27)
[     0.299805] ( 0) INFO : Wait     (queue=none, awaited=10, location=figure3.c:27)
[     0.299867] ( 0) INFO : Wait     (queue=none, awaited=11, location=figure3.c:27)
[     0.299929] ( 0) INFO : Wait     (queue=none, awaited=12, location=figure3.c:27)
[     0.299992] ( 0) INFO : Wait     (queue=none, awaited=13, location=figure3.c:27)
```

```
[     0.300054] ( 0) INFO : Wait     (queue=none, awaited=14, location=figure3.c:27)
[     0.300116] ( 0) INFO : Wait     (queue=none, awaited=15, location=figure3.c:27)
start
done
```

Appendix A3: Basic synchronization example

- Source code

```
/* # */
/* #             -- k1 --                 -- k1b -- */
/* #            /        \               /         \ */
/* # scatter --<          >-- gather --<            >-- gatherb */
/* #            \        /               \         / */
/* #             -- k2 --                 -- k2b --  */
/* # */

void scatter(const float *a, float *b, int n);
void k1(float *b, float *c, int n);
void k2(float *b, float *d, int n);
void gather(float *c, float *d, float *e, int n);
void k1b(float *e, float *f, int n);
void k2b(float *e, float *g, int n);
void gatherb(float *f, float *g, float *h, int n);

void scatter_gather(const int n, const float a[n],
                    float b[n], float c[n], float d[n],
                    float e[n], float f[n], float g[n], float h[n])
{
#pragma acc dataflow copyin(a), copyout(h)
  {
#pragma acc kernels, pcopyin(a), pcopyout(b)
    scatter(a, b, n);

#pragma acc kernels, pcopyin(b), pcopyout(c)
    k1(b, c, n);

#pragma acc kernels, pcopyin(b), pcopyout(d)
    k2(b, d, n);

#pragma acc kernels, pcopyin(c,d), pcopyout(e)
    gather(c, d, e, n);

#pragma acc kernels, pcopyin(e), pcopyout(f)
    k1b(e, f, n);

#pragma acc kernels, pcopyin(e), pcopyout(g)
    k2b(e, g, n);

#pragma acc kernels, pcopyin(f,g), pcopyout(h)
    gatherb(f, g, h, n);
  }
}

void scatter(const float *a, float *b, int n)
{
  int i;
  for (i=0; i<n; ++i)
    {
      b[i] = a[i] * 5 / (i%2);
    }
}
```

```
void k1(float *b, float *c, int n)
{
  int i;
  for (i=0; i<n; ++i)
    {
      c[i] = b[i] * 5 / (i%3);
    }
}
void k2(float *b, float *d, int n)
{
  int i;
  for (i=0; i<n; ++i)
    {
      d[i] = b[i] - i + n / 3;
    }
}

void gather(float *c, float *d, float *e, int n)
{
  int i;
  for (i=0; i<n; ++i)
    {
      e[i] = c[i] + d[i] - n / 3;
    }
}

void k1b(float *e, float *f, int n)
{
  int i;
  for (i=0; i<n; ++i)
    {
      f[i] = e[i] * 5 / (i%3);
    }
}

void k2b(float *e, float *g, int n)
{
  int i;
  for (i=0; i<n; ++i)
    {
      g[i] = e[i] - i + n / 3;
    }
}

void gatherb(float *f, float *g, float *h, int n)
{
  int i;
  for (i=0; i<n; ++i)
    {
      h[i] = f[i] - g[i] + n / 3;
    }
}

extern void fill(const int n, const float value, float t[n]);
#define N 300000
static const int n = N;
float a[N],b[N],c[N],d[N];
float e[N],f[N],g[N],h[N];
void example(void)
{
  fill(n, 2, a);
  fill(n, 0, b);
  fill(n, 0, c);
  fill(n, 0, d);
  fill(n, 0, e);
  fill(n, 0, f);
  fill(n, 0, g);
  fill(n, 0, h);
  scatter_gather(n, a, b, c, d, e, f, g, h);
  return;
}
```

- Compilation output

```
hmpp -k gcc -c -Wall -I/home/laorans/travail/DataFlow/HMPP-
DataFLow/build/hmpp/x86_64/debug//include main.c -o main.o
0 regions found
Written main_cN2GUe.halt.i
main.c: In function 'main':
main.c:16: warning: implicit declaration of function 'printf'
main.c:16: warning: incompatible implicit declaration of built-in function 'printf'

hmpp -k gcc -c -Wall -I/home/laorans/travail/DataFlow/HMPP-
DataFLow/build/hmpp/x86_64/debug//include scatter_gather3.c -o scatter_gather3.o
Parse acc dataflow copyin(a), copyout(h)
Create region scatter_gather3.c:22
Parse acc kernels, pcopyin(a), pcopyout(b)
Add call scatter(a, b, n) to region
Parse acc kernels, pcopyin(b), pcopyout(c)
Add call k1(b, c, n) to region
Parse acc kernels, pcopyin(b), pcopyout(d)
Add call k2(b, d, n) to region
Parse acc kernels, pcopyin(c,d), pcopyout(e)
Add call gather(c, d, e, n) to region
Parse acc kernels, pcopyin(e), pcopyout(f)
Add call k1b(e, f, n) to region
Parse acc kernels, pcopyin(e), pcopyout(g)
Add call k2b(e, g, n) to region
Parse acc kernels, pcopyin(f,g), pcopyout(h)
Add call gatherb(f, g, h, n) to region
1 regions found
Create CFG for: scatter_gather3.c:22
0: a
1: b
2: c
3: d
4: e
5: f
6: g
7: h
8: n
Found node scatter(a, b, n)
Found node k1(b, c, n)
Found node k2(b, d, n)
Found node gather(c, d, e, n)
Found node k1b(e, f, n)
Found node k2b(e, g, n)
Found node gatherb(f, g, h, n)
Dataflow CFG is:
CFG for region scatter_gather3.c:22
0: a
1: b
2: c
3: d
4: e
5: f
6: g
7: h
8: n
```

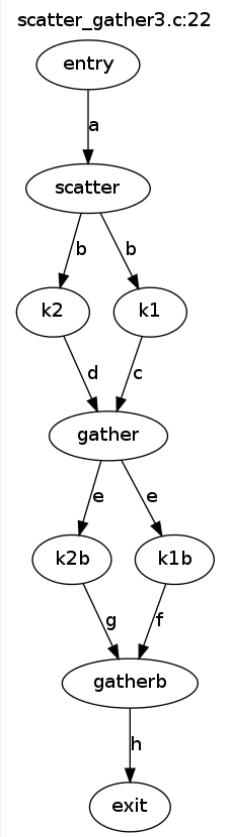


Figure 14: scatter_gather3.C, DATA DEPENDENCIES COMPUTED AT COMPILE TIME

```
9 nodes:
        a       b       c       d       e       f       g       h       n
0: <entry>:
        write   none    none    none    none    none    none    none    none
2: scatter:
        read    write   none    none    none    none    none    none    none
3: k1:
        none    read    write   none    none    none    none    none    none
4: k2:
        none    read    none    write   none    none    none    none    none
5: gather:
        none    none    read    read    write   none    none    none    none
6: k1b:
        none    none    none    none    read    write   none    none    none
7: k2b:
        none    none    none    none    read    none    write   none    none
8: gatherb:
        none    none    none    none    none    read    read    write   none
1: <exit>:
        none    none    none    none    none    none    none    read    none
8 edges:
<entry> --> scatter
scatter --> k1
k1 --> k2
k2 --> gather
gather --> k1b
k1b --> k2b
k2b --> gatherb
gatherb --> <exit>

Processing scatter_gather3.c:22
Build data dependencies for region scatter_gather3.c:22
Process node <exit>
Process node gatherb
addDEdge (gatherb,<exit>,'h')
Process node k2b
addDEdge (k2b,gatherb,'g')
Process node k1b
addDEdge (k1b,gatherb,'f')
Process node gather
addDEdge (gather,k2b,'e')
addDEdge (gather,k1b,'e')
Process node k2
addDEdge (k2,gather,'d')
Process node k1
addDEdge (k1,gather,'c')
Process node scatter
addDEdge (scatter,k2,'b')
addDEdge (scatter,k1,'b')
Process node <entry>
addDEdge (<entry>,scatter,'a')
q 0 : (<entry>-exe, scatter-exe)
q 1 : (k2-wait for q0, k2-exe)
q 2 : (gather-wait for q1, gather-wait for q7, gather-exe)
q 3 : (k2b-wait for q2, k2b-exe)
q 4 : (gatherb-wait for q3, gatherb-wait for q6, gatherb-exe)
q 5 : (<exit>-wait for q4, <exit>-exe)
q 6 : (k1b-wait for q2, k1b-exe)
q 7 : (k1-wait for q0, k1-exe)
Found 8 queues
Written scatter_gather3_QCu3VO.halt.i
hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] scatter_gather3.c:49: Loop
'i' was shared among gangs(192) and workers(256)
(last message repeated 1 more time)
hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__scatter_gather_22__rxkufdsq_cuda.hmf.cu".
hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] scatter_gather3.c:58: Loop
'i' was shared among gangs(192) and workers(256)
(last message repeated 1 more time)
hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__scatter_gather_27__we7v929a_cuda.hmf.cu".
hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] scatter_gather3.c:67: Loop
'i' was shared among gangs(192) and workers(256)
```

```
   (last message repeated 1 more time)
   hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__scatter_gather_32__m7t6hy1i_cuda.hmf.cu".
   hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] scatter_gather3.c:76: Loop
'i' was shared among gangs(192) and workers(256)
   (last message repeated 1 more time)
   hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__scatter_gather_38__sqzf3jti_cuda.hmf.cu".
   hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] scatter_gather3.c:85: Loop
'i' was shared among gangs(192) and workers(256)
   (last message repeated 1 more time)
   hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__scatter_gather_43__ib5eb500_cuda.hmf.cu".
   hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] scatter_gather3.c:94: Loop
'i' was shared among gangs(192) and workers(256)
   (last message repeated 1 more time)
   hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__scatter_gather_48__nfnpel8g_cuda.hmf.cu".
   hmppcg: src/dpil/acc/WorkSharingPass.cc:388: [Message DPL0099] scatter_gather3.c:103: Loop
'i' was shared among gangs(192) and workers(256)
   (last message repeated 1 more time)
   hmpp: [Info] Generated codelet filename is
"__hmpp_acc_region__scatter_gather_54__ejhi1ko8_cuda.hmf.cu".
   hmpp: [Warning HP0391] <preprocessor>:18: Variable 'n' in Data clause has no effect if
read-only in the enclosed Kernel/Parallel regions
   <preprocessor>: In function 'scatter_gather':
   <preprocessor>:92: warning: implicit declaration of function 'openacci_set_device_hint'
   <preprocessor>:18: warning: implicit declaration of function 'openacci_enter_region'
   <preprocessor>:18: warning: implicit declaration of function 'openacci_push_data'
   <stdin>:1: warning: implicit declaration of function 'openacci_call'
   <stdin>:1: warning: implicit declaration of function 'openacci_fallback'
   <stdin>:1: warning: implicit declaration of function 'openacci_leave_region'
   <preprocessor>:26: warning: implicit declaration of function 'openacci_wait'
   scatter_gather3.c: In function 'hmppsi_lookup':
   scatter_gather3.c:131: warning: implicit declaration of function 'hmpprti_lookup_grouplet'
   scatter_gather3.c:131: warning: return makes pointer from integer without a cast
   scatter_gather3.c: At top level:
   scatter_gather3.c:129: warning: 'hmppsi_lookup' defined but not used
   hmpp -k gcc -Wall main.o scatter_gather3.o -o test.exe
```

Execution output

```
./test.exe
[     0.209093] ( 0) INFO : Enter    data (queue=none, location=<preprocessor>:18)
[     0.209416] ( 0) INFO : Acquire  (target=cuda)
[     0.209650] ( 0) INFO : Acquired (device='cuda#0 [GeForce GTS 450]')
[     0.209789] ( 0) INFO : Allocate a[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.235402] ( 0) INFO : Upload   a[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.236023] ( 0) INFO : Allocate b[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.236230] ( 0) INFO : Allocate c[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.236420] ( 0) INFO : Allocate d[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.236607] ( 0) INFO : Allocate e[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.236793] ( 0) INFO : Allocate f[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.236976] ( 0) INFO : Allocate g[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.237161] ( 0) INFO : Allocate h[0:300000] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.237346] ( 0) INFO : Allocate n[0:1] (element_size=4, memory_space=cudaglob,
queue=none, location=<preprocessor>:18)
[     0.237643] ( 0) INFO : Enter    kernels (queue=0, location=<preprocessor>:22)
[     0.238837] ( 0) INFO : Allocate __hmpp_vla_sizes__a[0:1] (element_size=8,
memory_space=host, queue=0, location=<preprocessor>:22)
[     0.238960] ( 0) INFO : Upload   __hmpp_vla_sizes__a[0:1] (element_size=8, queue=0,
location=<preprocessor>:22)
[     0.239055] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=0, location=<preprocessor>:22)
[     0.239135] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=0,
location=<preprocessor>:22)
[     0.239224] ( 0) INFO : Call     __hmpp_acc_region__scatter_gather_22__rxkufdsq
(queue=0, location=<preprocessor>:22)
[     0.239413] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=0,
location=<preprocessor>:22)
[     0.239546] ( 0) INFO : Free     __hmpp_vla_sizes__a[0:1] (element_size=8, queue=0,
location=<preprocessor>:22)
[     0.239629] ( 0) INFO : Leave    kernels (queue=0, location=<preprocessor>:22)
[     0.239729] ( 0) INFO : Wait     (queue=7, awaited=0, location=<preprocessor>:26)
[     0.239807] ( 0) INFO : Enter    kernels (queue=7, location=<preprocessor>:27)
[     0.240207] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=7, location=<preprocessor>:27)
[     0.240298] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=7,
location=<preprocessor>:27)
[     0.240370] ( 0) INFO : Allocate __hmpp_vla_sizes__c[0:1] (element_size=8,
memory_space=host, queue=7, location=<preprocessor>:27)
[     0.240450] ( 0) INFO : Upload   __hmpp_vla_sizes__c[0:1] (element_size=8, queue=7,
location=<preprocessor>:27)
[     0.240522] ( 0) INFO : Call     __hmpp_acc_region__scatter_gather_27__we7v929a
(queue=7, location=<preprocessor>:27)
[     0.240655] ( 0) INFO : Free     __hmpp_vla_sizes__c[0:1] (element_size=8, queue=7,
location=<preprocessor>:27)
[     0.240740] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=7,
location=<preprocessor>:27)
[     0.240818] ( 0) INFO : Leave    kernels (queue=7, location=<preprocessor>:27)
[     0.240883] ( 0) INFO : Wait     (queue=1, awaited=0, location=<preprocessor>:31)
[     0.240955] ( 0) INFO : Enter    kernels (queue=1, location=<preprocessor>:32)
[     0.241323] ( 0) INFO : Allocate __hmpp_vla_sizes__b[0:1] (element_size=8,
memory_space=host, queue=1, location=<preprocessor>:32)
[     0.241410] ( 0) INFO : Upload   __hmpp_vla_sizes__b[0:1] (element_size=8, queue=1,
location=<preprocessor>:32)
[     0.241481] ( 0) INFO : Allocate __hmpp_vla_sizes__d[0:1] (element_size=8,
memory_space=host, queue=1, location=<preprocessor>:32)
[     0.241559] ( 0) INFO : Upload   __hmpp_vla_sizes__d[0:1] (element_size=8, queue=1,
location=<preprocessor>:32)
[     0.241634] ( 0) INFO : Call     __hmpp_acc_region__scatter_gather_32__m7t6hy1i
(queue=1, location=<preprocessor>:32)

[     0.241728] ( 0) INFO : Free     __hmpp_vla_sizes__d[0:1] (element_size=8, queue=1,
```

```
location=<preprocessor>:32)
[     0.241812] ( 0) INFO : Free     __hmpp_vla_sizes__b[0:1] (element_size=8, queue=1,
location=<preprocessor>:32)
[     0.241896] ( 0) INFO : Leave    kernels (queue=1, location=<preprocessor>:32)
[     0.241963] ( 0) INFO : Wait     (queue=2, awaited=1, location=<preprocessor>:36)
[     0.242035] ( 0) INFO : Wait     (queue=2, awaited=7, location=<preprocessor>:37)
[     0.242110] ( 0) INFO : Enter    kernels (queue=2, location=<preprocessor>:38)
[     0.242540] ( 0) INFO : Allocate __hmpp_vla_sizes__c[0:1] (element_size=8,
memory_space=host, queue=2, location=<preprocessor>:38)
[     0.242628] ( 0) INFO : Upload   __hmpp_vla_sizes__c[0:1] (element_size=8, queue=2,
location=<preprocessor>:38)
[     0.242707] ( 0) INFO : Allocate __hmpp_vla_sizes__d[0:1] (element_size=8,
memory_space=host, queue=2, location=<preprocessor>:38)
[     0.242790] ( 0) INFO : Upload   __hmpp_vla_sizes__d[0:1] (element_size=8, queue=2,
location=<preprocessor>:38)
[     0.242863] ( 0) INFO : Allocate __hmpp_vla_sizes__e[0:1] (element_size=8,
memory_space=host, queue=2, location=<preprocessor>:38)
[     0.242946] ( 0) INFO : Upload   __hmpp_vla_sizes__e[0:1] (element_size=8, queue=2,
location=<preprocessor>:38)
[     0.243031] ( 0) INFO : Call     __hmpp_acc_region__scatter_gather_38__sqzf3jti
(queue=2, location=<preprocessor>:38)
[     0.243150] ( 0) INFO : Free     __hmpp_vla_sizes__e[0:1] (element_size=8, queue=2,
location=<preprocessor>:38)
[     0.243247] ( 0) INFO : Free     __hmpp_vla_sizes__d[0:1] (element_size=8, queue=2,
location=<preprocessor>:38)
[     0.243334] ( 0) INFO : Free     __hmpp_vla_sizes__c[0:1] (element_size=8, queue=2,
location=<preprocessor>:38)
[     0.243415] ( 0) INFO : Leave    kernels (queue=2, location=<preprocessor>:38)
[     0.243483] ( 0) INFO : Wait     (queue=6, awaited=2, location=<preprocessor>:42)
[     0.243558] ( 0) INFO : Enter    kernels (queue=6, location=<preprocessor>:43)
[     0.243936] ( 0) INFO : Allocate __hmpp_vla_sizes__e[0:1] (element_size=8,
memory_space=host, queue=6, location=<preprocessor>:43)
[     0.244024] ( 0) INFO : Upload   __hmpp_vla_sizes__e[0:1] (element_size=8, queue=6,
location=<preprocessor>:43)
[     0.244100] ( 0) INFO : Allocate __hmpp_vla_sizes__f[0:1] (element_size=8,
memory_space=host, queue=6, location=<preprocessor>:43)
[     0.244181] ( 0) INFO : Upload   __hmpp_vla_sizes__f[0:1] (element_size=8, queue=6,
location=<preprocessor>:43)
[     0.244254] ( 0) INFO : Call     __hmpp_acc_region__scatter_gather_43__ib5eb500
(queue=6, location=<preprocessor>:43)
[     0.244350] ( 0) INFO : Free     __hmpp_vla_sizes__f[0:1] (element_size=8, queue=6,
location=<preprocessor>:43)
[     0.244433] ( 0) INFO : Free     __hmpp_vla_sizes__e[0:1] (element_size=8, queue=6,
location=<preprocessor>:43)
[     0.244507] ( 0) INFO : Leave    kernels (queue=6, location=<preprocessor>:43)
[     0.244570] ( 0) INFO : Wait     (queue=3, awaited=2, location=<preprocessor>:47)
[     0.244639] ( 0) INFO : Enter    kernels (queue=3, location=<preprocessor>:48)
[     0.244990] ( 0) INFO : Allocate __hmpp_vla_sizes__e[0:1] (element_size=8,
memory_space=host, queue=3, location=<preprocessor>:48)
[     0.245076] ( 0) INFO : Upload   __hmpp_vla_sizes__e[0:1] (element_size=8, queue=3,
location=<preprocessor>:48)
[     0.245148] ( 0) INFO : Allocate __hmpp_vla_sizes__g[0:1] (element_size=8,
memory_space=host, queue=3, location=<preprocessor>:48)
[     0.245223] ( 0) INFO : Upload   __hmpp_vla_sizes__g[0:1] (element_size=8, queue=3,
location=<preprocessor>:48)
[     0.245298] ( 0) INFO : Call     __hmpp_acc_region__scatter_gather_48__nfnpel8g
(queue=3, location=<preprocessor>:48)
[     0.245389] ( 0) INFO : Free     __hmpp_vla_sizes__g[0:1] (element_size=8, queue=3,
location=<preprocessor>:48)
[     0.245471] ( 0) INFO : Free     __hmpp_vla_sizes__e[0:1] (element_size=8, queue=3,
location=<preprocessor>:48)
[     0.245546] ( 0) INFO : Leave    kernels (queue=3, location=<preprocessor>:48)
[     0.245609] ( 0) INFO : Wait     (queue=4, awaited=3, location=<preprocessor>:52)
[     0.245676] ( 0) INFO : Wait     (queue=4, awaited=6, location=<preprocessor>:53)
[     0.245745] ( 0) INFO : Enter    kernels (queue=4, location=<preprocessor>:54)
[     0.246157] ( 0) INFO : Allocate __hmpp_vla_sizes__f[0:1] (element_size=8,
memory_space=host, queue=4, location=<preprocessor>:54)
[     0.246244] ( 0) INFO : Upload   __hmpp_vla_sizes__f[0:1] (element_size=8, queue=4,
location=<preprocessor>:54)
[     0.246314] ( 0) INFO : Allocate __hmpp_vla_sizes__g[0:1] (element_size=8,
memory_space=host, queue=4, location=<preprocessor>:54)
[     0.246393] ( 0) INFO : Upload   __hmpp_vla_sizes__g[0:1] (element_size=8, queue=4,
location=<preprocessor>:54)
```

```
[     0.246462] ( 0) INFO : Allocate __hmpp_vla_sizes__h[0:1] (element_size=8,
memory_space=host, queue=4, location=<preprocessor>:54)
[     0.246535] ( 0) INFO : Upload   __hmpp_vla_sizes__h[0:1] (element_size=8, queue=4,
location=<preprocessor>:54)
[     0.246610] ( 0) INFO : Call     __hmpp_acc_region__scatter_gather_54__ejhi1ko8
(queue=4, location=<preprocessor>:54)
[     0.246708] ( 0) INFO : Free     __hmpp_vla_sizes__h[0:1] (element_size=8, queue=4,
location=<preprocessor>:54)
[     0.246795] ( 0) INFO : Free     __hmpp_vla_sizes__g[0:1] (element_size=8, queue=4,
location=<preprocessor>:54)
[     0.246874] ( 0) INFO : Free     __hmpp_vla_sizes__f[0:1] (element_size=8, queue=4,
location=<preprocessor>:54)
[     0.246948] ( 0) INFO : Leave    kernels (queue=4, location=<preprocessor>:54)
[     0.247009] ( 0) INFO : Wait     (queue=none, awaited=4, location=<preprocessor>:58)
[     0.247072] ( 0) INFO : Free     n[0:1] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.247235] ( 0) INFO : Download h[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.247867] ( 0) INFO : Free     h[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.248045] ( 0) INFO : Free     g[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.248192] ( 0) INFO : Free     f[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.248338] ( 0) INFO : Free     e[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.248485] ( 0) INFO : Free     d[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.248630] ( 0) INFO : Free     c[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.248776] ( 0) INFO : Free     b[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.248921] ( 0) INFO : Free     a[0:300000] (element_size=4, queue=none,
location=<preprocessor>:18)
[     0.249067] ( 0) INFO : Leave    data (queue=none, location=<preprocessor>:18)
Start
done
```