Unified Data-Flow Platform for General Purpose Many-core Systems

Andreas Diavastos
Pedro Trancoso

TR-17-2 **September 2017**

University of Cyprus

Technical Report

Unified Data-Flow Platform for General Purpose Many-core Systems

Andreas Diavastos and Pedro Trancoso

UCY-CS-TR-17-2

September 2017

Table of Contents

Abstract

The current trend in high performance processor design is to increase the number of cores as to achieve desired performance. While having a large number of cores on a chip seems to be feasible in terms of the hardware, the development of the software that is able to exploit that parallelism is one of the biggest challenges. In this work we propose a Data-Flow based system that can be used to exploit the parallelism in large-scale many-core processors in an efficient way. We propose to use a Data-Flow system that can be adapted to the two major many-core implementations: clustered and shared memory. In particular we test the original TFlux Data-Driven Multithreading platform on the 61-core Intel Xeon Phi processor and extend the platform for execution on the 48-core Intel Single-chip Cloud Computing (SCC) processor. Given the modularity of the TFlux platform, changes to support the new architecture are done at the back-end level and thus the application does not need to be re-written in order to execute on different systems. The experiments are performed on real off-the-shelf systems executing a Data-Flow software runtime. While different approaches and optimizations are chosen to exploit the different characteristics of each architecture, the evaluation shows good scalability for a common set of applications for both architectures.

Index terms— dataflow; many-cores; programming model; Data-Driven Multithreading; Intel SCC; Intel Xeon Phi

1 Introduction

Scaling the performance of an application can be achieved by either improving the hardware, or developing more efficient software to solve the particular problem. To retain the power-performance efficiency to an acceptable level we are currently exploring parallel processing as the way to scale performance. Consequently, the trend today is to include more and more cores into the processor resulting in what is known as a multi- and many-core processor. From the hardware design perspective, the more parallel units offered for execution, the higher the performance that can be achieved. From the software design perspective though, this new trend creates new challenges. To achieve scalable performance in these new systems, programmers are required to think parallel. This essentially means learning new programming models and developing new algorithms that exploit parallelism.

In addition, there are different implementations for many-core processors. In particular, we observe two major trends: cache-coherent shared memory and clustered distributed memory architectures. Each architecture has different characteristics along with their advantages and disadvantages. In order to fully exploit the performance, the programmers need to be aware of the underlying architecture and apply specific optimizations. This is an additional factor in the difficulty of exploiting the performance of future large-scale systems. Therefore, the solution to the scalability of the performance depends on scalable hardware with increasing number of computational units along with efficient programming and parallel execution models that hide the hardware complexity from the programmer.

Data-Flow is a natural paradigm for describing parallelism using directed graphs based on the path of the data [1]. Nevertheless, the original Data-Flow implementations suffered from serious limitations as they required specialized hardware [2]. New hardware designs though, make the Data-Flow model a possible solution for scaling the performance of applications. Consequently, researchers have proposed new alternatives based on this model that are able to exploit the parallelism effectively. One such alternative is the Data-Driven Multithreading (DDM) [3]. Following the footsteps of the Data-Flow model it can exploit the maximum available parallelism of an application by exploiting data dependencies, while at the same time overcomes overheads by increasing the granularity of the execution blocks. In DDM the synchronization part of the program is separated from the communication part allowing it to hide the synchronization and communication delays [4]. Different implementations of the DDM model exist. Some implement the scheduling unit in hardware as to minimize the effects of the overheads and others in software as to guarantee portability of the platform to any new hardware design.

The focus of this paper is to show how a software implementation of the DDM model of execution can offer performance scalability on different many-core architectures by efficiently exploiting the parallelism and at the same time relieve the programmer from the hardware details, such as architecture and data communication. The complete software platform we present in this work offers an environment for parallel execution on many-core architectures that is able to scale without major hardware requirements or programming effort. Our implementation is based on the TFlux platform [5]. To test our implementation we use one representative for each of the major categories of many-core processors: the 61-core Intel Xeon Phi processor [6], as a representative of a cache-coherent many-core processor; and the 48-core Intel Single-chip Cloud Computer (SCC) [7], as a representative of clustered many-core processors. For the performance evaluation of our implementation we used the same benchmark suite that was used for the evaluation of the original TFlux implementation that includes applications with different characteristics. Our proposed system includes a source-to-source preprocessor that takes as input programs in C, augmented with directives that specify the threads and their dependencies, as well as, a runtime system to handle the scheduling of the threads in a Data-Flow manner. Given the modularity of the platform, we just need to write one application, from which we can generate the code for execution on either many-core system. Our evaluation is performed on real Intel Xeon Phi and Intel SCC systems and our execution model is implemented in software as a library that is linked to the application.

The contributions of this paper are as follows:

- The extension of the TFlux platform to execute on both cache-coherent and clustered many-core systems with the same application sources; The former is the original TFluxSoft implementation [5] while the latter is the TFluxSCC previously presented in [8];
- The evaluation of the extended platform on a real 61-core Intel Xeon Phi and a 48-core Intel SCC systems, highlighting the relevant characteristics used to enable scalable performance for each architecture.

This paper is organized as follows. In Section 2 we discuss the related work around the Data-Flow model and the two many-core systems used in this work. In Section 3 we present the DDM model that is based on the Data-flow model of execution. In Section 4 we explain in detail the TFlux platform as an implementation of the DDM model. In Section 5 we analyse the characteristics of the many-core systems used for the evaluation of this work, namely the Intel Xeon Phi and the Intel SCC. In Section 6 we present our implementations and details on how we modified the TFlux platform to support different many-core systems. In Sections 7 and 8 we show the setup and present the results of our evaluation, respectively. Finally, in Section 9 we outline our conclusions.

2 Related Work

There are several projects currently, targeting the exploitation of parallelism for many-core architectures. The industry is moving closer to the development of many-core processors with Intel proposing recently the Many Integrated Cores (MIC) Architecture [9] and introducing the 48-core SCC processor [7] as an experimental processor and launching the Xeon Phi as a coprocessor unit. The latest released Xeon Phi [6] has 61 cores and further exploits parallelism using wide vector units, while the former adopts the clustered architecture with simple hardware design as to deliver efficient scaling.

The Intel Xeon Phi offers a fully cache-coherent environment across all cores. For this reason it supports all standard programming models like OpenMP* [10], POSIX threads [11] or MPI [12] without the immediate need to rewrite an application. It provides high performance in a low power envelope, by utilizing four-way simultaneous multi-threading cores with it's 512-wide vector units. It targets high performance computing centers with some already starting to develop systems with multiple Xeon Phi units [13].

Totoni *et al.* in [14] use CHARM++ and MPI message passing paradigms to implement parallel applications with different characteristics in order to evaluate the Intel SCC platform in matters of performance. They get speedup results of up to 32.7x for 48 cores and they propose more sophisticated cores for the future many-cores in order to increase the performance, mainly for the sequential execution. RCKMPI [15] by Intel and SCC-MPICH [16] by RWTH Aachen University implement customized MPI libraries aiming to improve the message passing model with respect to the SCC many-core architecture. These two implementations use an efficient mix of Message Passing Buffer (MPB) and DDR3 shared memory for low-level communication in order to achieve higher bandwidth and lower latency. In [17] the authors examine various performance aspects of the SCC using a stream benchmark and the NAS Parallel Benchmarks [18] *bt* and *lu*. Their findings show that for these benchmarks the data exchange based on message passing is faster than shared memory data exchange and in order to improve the memory access behaviour you must increase both the clock frequency of the mesh network and the memory controllers.

Intel SCC avoids the hardware-based cache coherency and introduces a software-oriented message passing based architecture instead. A software cache-coherency implementation for the SCC system can act as another potential solution for creating simpler many-core architectures, free of complex hardware. As X. Zhou *et al.* propose in [19], Software Managed Cache Coherence (SMCC) shows a comparable performance to hardware coherency while offering the possibility of having dynamically reconfigurable coherence domains on the chip. The unnecessary complex hardware support for applications with little sharing and the inability to support heterogeneous platforms make the SMCC achieve better use of silicon with significant reduction of hardware budget.

In addition, we also have other architectures like the GPUs' [20] that have been around for some time and support parallel execution with hundreds of cores. GPU's offer large computing power with low cost but the programming of such engines is still trivial and the programmer must have all the information of the underlying hardware in order to achieve good performance. GPUs are special purpose hardware units and the range of applications that offer significant performance increase on a GPU system is limited to data parallel applications.

Programmability in many-core architectures is another challenge the community has to address. OpenCL represents a parallel programming standard especially for heterogeneous computing systems. SnuCL [21] is an OpenCL framework for heterogeneous CPU/GPU clusters that provides ease of programming for such systems. This framework achieved high performance on a cluster architecture with a designated, single host node and many compute nodes equipped with multi-core CPUs and multiple GPUs. The scalability though refers only to medium-scale clusters, since large-scale clusters may lead to performance degradation due to the centralized task scheduling model followed. Lee *et al.* in [22] presents a new OpenCL framework, this time for homogeneous many-cores with no hardware cache coherency, such as the Intel SCC. The framework includes a compiler and an OpenCL runtime which together with the dynamic memory mapping mechanism preserve coherency and consistency between CPU cores on the SCC architecture with a small overhead.

OmpSs [23] is a task-based programming model based on the OpenMP [10] and the StarSs [24] models. It's target is heterogeneous multi-core architectures, thus it incorporates the use of OpenCL and CUDA kernels. OmpSs outperforms OpenCL or OpenMP in some applications for the same platforms while it offers a more flexible programming environment to exploit multiple accelerators. The DEEP project [25] uses OmpSs as its' primary model in an effort to implement a novel architecture for high-performance computing (HPC) consisting of two components - a standard HPC cluster with traditional multi-core systems and a cluster of many-core processors called *Booster* equipped with Intel Xeon Phi coprocessors.

TERAFLUX [26] was a large-scale project funded by the European Union aiming to solve the challenges of programmability, manageable architecture design and reliability of a 1000+ core chips by using the Data-Flow principles. The idea was to develop new programming models, compiler analysis and optimization technologies in order to build a scalable architecture based mostly on off-the-shelf components while simplifying the design of such Tera-device systems.

Our work differs from other projects as we propose a complete programming and execution platform, for a many-core system using global address space, based on Data-Flow principles. Our

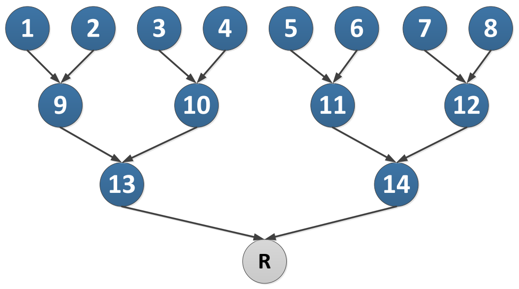

Figure 1: A Data-Flow graph for calculating a reduction operation. The numbered nodes represent the DThreads with their ID and the arcs represent the dependencies between the DThreads.

implementations target a conventional hardware approach. Using a software approach our implementations are easily ported to new architectures without the need of specialized hardware. To the best of our knowledge, what we present in this work is the only implementations of the DDM model on many-core architectures.

3 Data-Driven Multithreading

Scalability is becoming more relevant for a larger range of applications as the number of computation units increase within a single processor. Recently we have seen an increasing interest in the Data-Flow model as a way to efficiently exploit the maximum available parallelism of applications. DDM is a Data-Flow model where the granularity of the Data-Flow code is a thread [3] [4]. Each thread in DDM is scheduled in a Data-Flow way at runtime, but the instructions in a thread are executed in a control-flow way.

DDM programs are composed of Data-Driven Threads (DThreads) that contain an arbitrary number of instructions. Within a DThread the instruction execution follows the classic control-flow model, thus allowing any other runtime or compile-time optimizations to be performed. The DThreads are distinguished by the producer-consumer relationship that connects them. This producer-consumer relationship, also called a *dependency*, between two or more DThreads can be expressed by declaring the input and output data of a DThread in a DDM program. Combining the DThreads and their dependencies, a *Synchronization Graph* (SG) is created where the nodes represent the DThreads and the arcs the data dependencies between them as shown in Figure 1. The total number of producers a DThread has is called *ReadyCount*.

A DThread can be scheduled for execution as soon as its' producer threads have completed their execution. The scheduling for execution of the DThreads is handled by a Thread Scheduling Unit (TSU) at runtime in a Data-Flow manner. The TSU loads the SG described earlier, prior to the execution and handles the execution of all DThreads in a DDM program. More specifically, a TSU is responsible for the updates of the structures of the SG, such as the *ReadyCount* values. When a producer completes its' execution the TSU will update the *ReadyCount* values of all the consumer DThreads of that specific producer. As soon as the *ReadyCount* of a consumer reaches zero, the TSU releases the specific DThread and when a processor core becomes available it is scheduled for execution.

Although the dependency analysis of a program may be quite a difficult task, the Data-Flow execution model is designed to expose the maximum available parallelism and the DDM model can achieve this from scheduling the threads. Depending on the runtime implementation this scheduling could result in minimum overhead.

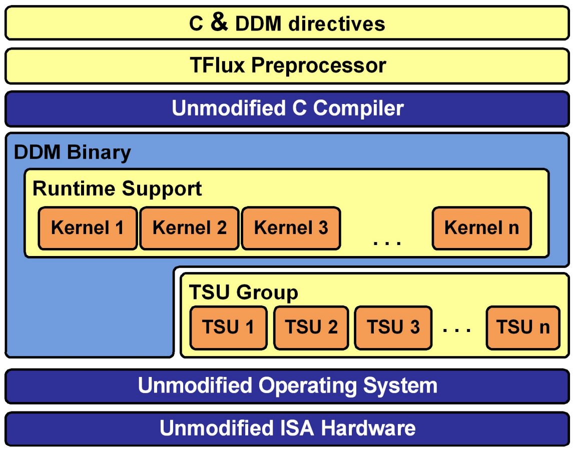

Figure 2: The layered design of the TFlux system [5]

4 The TFlux Platform

This work is based on the TFlux implementation [5] of the DDM model. We chose TFlux as it is a complete platform that includes a programming environment for DDM applications using compiler directives, a source-to-source preprocessor that translates the application augmented with the directives into DDM parallel code and a Thread Scheduling Unit (TSU) that handles the Data-Flow scheduling of the DThreads at runtime. An important advantage of TFlux is that it is not built for a specific machine but rather works as a virtualization platform for DDM program execution on a variety of computing systems. Another reason we chose TFlux is that it is platform independent, as the produced parallel code is in ANSI-C which complies with the supported programming languages for most systems and in this case the Intel SCC and the Intel Xeon Phi.

Using the TFlux directives the programmer can define the DThreads that form a DDM program along with their dependencies. TFlux also requires a TSU to enforce the Data-Flow execution of the DThreads. Figure 2 illustrates the layered design of the TFlux system. At the top layer of the design we have the source code of a DDM application that are developed using ANSI-C with DDM directives [27]. This abstracts the details of the underlying hardware. A DDM source-to-source preprocessor is used to parse the C+DDM directives code and produce a C program that can be compiled with any commodity C compiler. The executable produced invokes the *TFlux Runtime Support* operations that allow the application to execute as a DDM program. The DDM Binary runs on top of an unmodified Unix-based Operating System (OS) and its' primary responsibility is to dynamically load the DThreads into the TSU and invoke all TSU operations required for the DDM execution. At the lowest layer of Figure 2 the OS and ISA Hardware are kept unmodified.

Some of the original TFlux platform implementations are [5]:

- *TFluxHard* for a shared memory chip multi-processor augmented with a hardware implementation of the TSU;
- *TFluxSoft* that targets commodity multi-core processors with single shared address space and a TSU implementation in software.

TFluxHard and *TFluxSoft* implement the DDM model for conventional shared memory multi-processors. For the *TFluxHard* implementation the TSU is implemented in hardware and integrated in the processor for fast exchange of update messages, whereas for the *TFluxSoft* implementation the TSU is implemented in software and executed on a separate core and thus can be executed on any commodity multi-core processor.

Table 1: TFlux DDM pragma directives

`#pragma ddm startprogram`	Define the boundaries of a DDM program
`#pragma ddm endprogram`	
`#pragma ddm block` *ID*	Define the boundaries of a block of threads with identifier *ID*
`#pragma ddm endblock`	
`#pragma ddm thread` *ID* `kernel` *NUMBER*	Define the boundaries of a DDM thread with identifier *ID* and the kernel *NUMBER*
`#pragma ddm endthread`	
`#pragma ddm for thread` *ID*	Define the boundaries of a loop thread with identifier *ID*
`#pragma ddm endfor`	
`#pragma ddm kernel` *NUMBER*	Declare the number of kernels
`#pragma ddm var` *TYPE NAME*	Declare a *shared* variable
`#pragma ddm private var` *TYPE NAME*	Declare a *private* variable

```
#pragma ddm startprogram
 #pragma ddm block 1
   ...
  #pragma ddm for thread 4 unroll 1
     for (j = 0; j < 512; j++)
     {
        for (i = 0; i < 512; i++)
           yp[j][i]+= c[i][j]*(y[j]+k3[j]);
     }
  #pragma ddm endfor

  #pragma ddm for thread 5 reduction sum + double total depends(4)
     for (i = 0; i < 512; i++)
     {
        if(!initFlag02){
           initFlag02=1;
        for (j = 0; j < 512; j++)
           k4P[j]=h*(pow[j]-(yp[0][j]+yp[1][j]));
        }
     yout[i] = y[i] + (2*k3[i]+k4P[i])/6.0;
     sum+=yout[i];
     }
  #pragma ddm endfor
 #pragma ddm endblock
#pragma ddm endprogram
```

Figure 3: Code example of how to use TFlux directives in a program.

4.1 Thread Scheduling Unit

The TSU's task is to load the SG of a DDM application and according to the dependencies described by the SG, will initiate and schedule the execution of all DThreads in a Data-Flow manner. In the first implementation of DDM, the D^2NOW [3], each processor needed to have its own private TSU since the execution nodes were independent machines. In TFlux these private TSUs were unified in a single unit named *TSU Group* (shown in Figure 2). This unit is logically split in *n+1* parts. One part per core (totalling *n*) for the core's own TSU operations and one common part which is located on a dedicated core and manages the common TSU operations.

4.2 Programming and Compilation Toolchain

As mentioned earlier DDM applications are developed using ANSI-C with DDM directives [27] as the ones shown in Table 1. The directives are used to define the start and the end of the code of the DThreads and to express the dependencies between them. These dependencies will then be used to create the SG needed by the TSU in order to synchronize the execution. In Figure 3 we show an example of how to program a sequential application using TFlux directives. The code in the figure is part of an actual DDM program that was used for the evaluation in this work. A DDM program is declared by using the directives

```
#pragma ddm startprogram
```

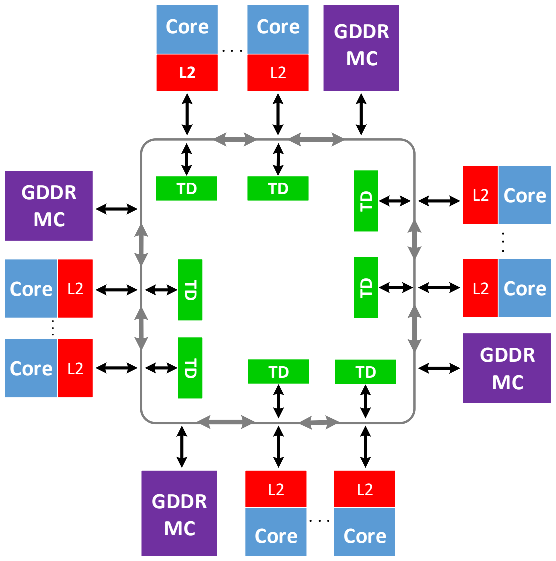

Figure 4: The Intel MIC top level architecture.

```
#pragma ddm endprogram
```

and each DDM program is separated into DDM blocks with the directives

```
#pragma ddm block ID
#pragma ddm endblock
```

The DDM blocks contain the DThreads of the application. DThreads that belong to the same DDM block will be executed in parallel based on their dependencies and DDM blocks will be executed between them in a sequential order. In TFlux we can identify two kinds of DThreads, simple DThreads using the directives

```
#pragma ddm thread ID kernel NUMBER
#pragma ddm endthread
```

where we declare the ID of each DThread and the kernel that will execute on (when using thread pinning the kernel is the same as the core id). We can also identify loop DThreads using the *for* loop directives

```
#pragma ddm for thread ID
#pragma ddm endfor
```

where each loop iteration is scheduled as a separate thread thus allowing to efficiently handle cross-iteration dependencies. In the example of Figure 3, we show that *for* loop directives can also support unrolling and the execution of parallel functions such as a reduction operation.

The DDM C Preprocessor [27] is used to parse the C+DDM directives code and produce a C program that can be compiled with any commodity C compiler. The executable produced invokes the *TFlux Runtime Support* operations that allow the application to execute as a DDM program.

5 Many-core Architectures

The need to optimize performance per watt has resulted in the increase of the number of cores in processor chips [28]. Many light-weight cores will be replacing the few sophisticated cores we have currently in multi-core chips, creating the many-core chips with hundreds or more cores. In our work

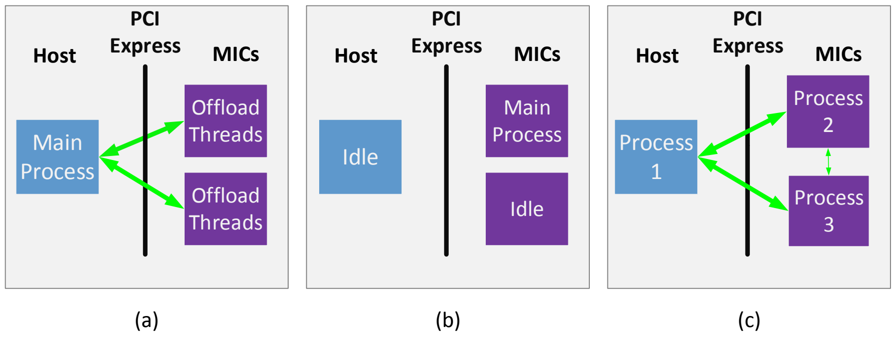

Figure 5: Intel Xeon Phi execution modes: (a) *Offload mode*, (b) *Coprocessor-only mode* and (c) *Symmetric mode*. The arrows show the communication between processes of the same program

we examine two processor chips that belong to the Intel Many Integrated Core (MIC) Architecture family. The Intel Xeon Phi coprocessor [6] and the Intel SCC experimental processor [7]. The Xeon Phi is a released product that works as a coprocessor unit alongside an Intel Xeon processor unit. The SCC is an experimental processor that works as a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research.

5.1 Intel Xeon Phi

The Xeon Phi coprocessor [6] (codename Knights Corner) is part of the Many Integrated Core (MIC) Architecture family from Intel. Each coprocessor is equipped with up to 61 processor cores connected by a high-performance on-die bidirectional ring interconnect as shown in Figure 4. Each coprocessor includes 8 memory controllers supporting up to 16 GDDR5 channels (2 per memory controller) with a theoretical aggregate bandwidth of 352 GB/s. Each core is a fully functional, in-order core, that supports 4 hardware threads. To reduce hot-spot contention for data among the cores, a distributed tag directory is implemented such that every physical address is uniquely mapped through a reversible one-to-one address hashing function. This also provides for a framework for more elaborate coherence protocol mechanisms than the individual cores could provide.

The working frequency of the Xeon Phi cores is 1GHz (up to 1.3GHz) and their architecture is based on the x86 Instruction Set Architecture (ISA), extended with 64-bit addressing and 512-bit wide vector instructions and registers that allow for significant increase in parallel computation capabilities. With a 512-bit vector set, a throughput of 8 double-precision or 16 single-precision floating point operations can be executed on each core per cycle. Assuming a throughput of one fused multiply-add operation per cycle, this brings the theoretical peak double precision floating point performance to almost 1 Tflop/s. Each core has a 32KB L1 data cache, a 32KB L1 instruction cache and a 512KB L2 cache. The L2 caches of all cores are interconnected with each other and the memory controllers via a bidirectional ring bus, effectively creating a shared, cache-coherent last-level cache of up to 32MB.

Although coprocessors in general are used as accelerators for offloading computation from host CPU processes, the Xeon Phi is more flexible and offers three different modes of execution [29]. The *Offload mode*, where the Xeon Phi is used as an accelerator for offloading computation in the form of parallel threads from the host process that is executed on the Host CPU as shown in Figure 5(a). The offloading of parallel threads to the Xeon Phi is done using directives in the source code to denote the parallel section to be executed on the coprocessor. The *Coprocessor-only mode*, where the Xeon Phi works as a stand-alone compute unit for the program (Figure 5(b)). And finally the *Symmetric mode*, where multiple processes of the same program can be launched on both the host CPU and the Xeon Phi and work as any other message passing model as shown in Figure 5(c).

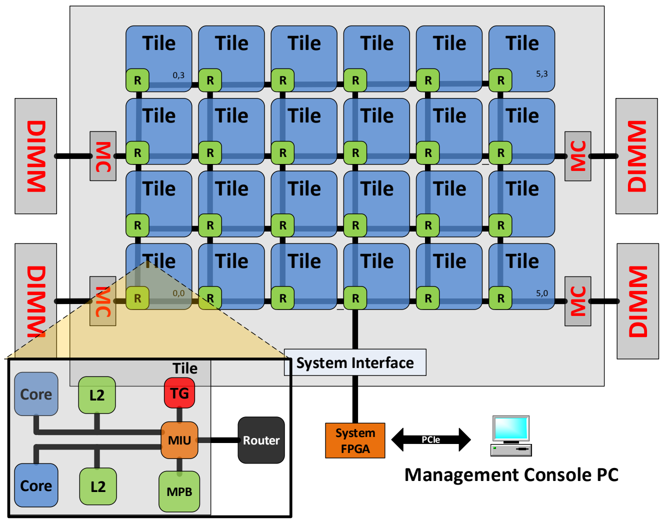

Figure 6: Intel SCC top-level and tile top-level architecture

5.2 Intel SCC

The Intel SCC processor consists of 24 dual-core tiles interconnected by a 2D-grid network as illustrated in Figure 6. These tiles are organized in a 6x4 mesh with each one containing two cores with dedicated L1 and L2 caches of 16KB and 256KB respectively, 16KB Message Passing Buffer (MPB) for message storing travelling through the network, a Traffic Generator (TG) for testing the performance of the on-chip network, a Mesh Interface Unit (MIU) connecting the tile to its' network router and two test-and-set registers.

The maximum main memory the current system can support is 64GB and the 32-bit memory addresses of the cores are translated into system addresses by the MIU through a lookup table (LUT). The main memory of the system is located outside the chip and the access to it is achieved through four on-chip DDR3 Memory Controllers (MC). The SCC supports both distributed and shared memory models. The system memory is composed of four regions. The private memory of each core (*Private off-chip memory*), the global address space of the system (*Shared off-chip memory*), the Message Passing Buffer (MPB) used to store messages to be sent through the network (*Shared on-chip memory*) and lastly the *L2 cache* of each core.

The Intel SCC is equipped with a large number of light-weight processing units with low power consumption and with multiple memory controllers serving them. Based on the characteristics of the Intel SCC we understand that future many-core processors will focus on energy-efficient designs. Expensive, in matters of design- and operating-cost, hardware support units may be avoided and other solutions must be exploited. One example on the Intel SCC is the cache coherency mechanism [30], where caching is only supported for data allocated on the private address space. Techniques like this though, have an impact on the performance and the programming of new architectures.

Being an experimental processor, the SCC comes with a concept Programming API called RCCE that supports the Single Program Multiple Data (SPMD) model of execution. It supports C/C++ programming languages with RCCE API extensions to implement the message passing communication of the system. More information on the RCCE communication environment and all the operations supported can be found in [28].

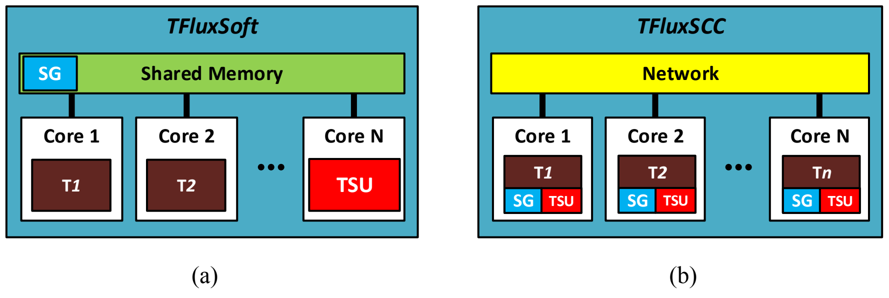

Figure 7: The two TFlux implementations: (a) the original TFluxSoft and (b) the TFluxSCC. The SG in blue box shows where the Synchronization Graph resides, the TSU in red box represents the Thread Scheduling Unit and the T* in brown boxes represent the application threads.

6 TFlux Implementations for Many-cores

In this work we study the scalability of the DDM model using the TFlux platform on two many-core systems, the Intel Xeon Phi and the Intel SCC. We show that the DDM model of execution can scale for certain applications on non-coherent and coherent many-core architectures by only changing the back-end of the runtime system. TFlux uses compiler directives as to define DThreads and the dependencies amongst them as described earlier. Given that TFlux is an existing platform, in our implementations of TFlux for many-core processors we keep the programming style and syntax the same.

In this paper we examine two different scenarios of DDM execution on many-core processors using the TFlux platform. First we test the original *TFluxSoft* [5] implementation on the coherent Intel Xeon Phi coprocessor and then we present a new implementation of TFlux for the non-coherent Intel SCC. We call the Intel SCC implementation *TFluxSCC* [8]. The differences between the two implementations lie on the memory model used for storing the synchronization graph and also the hierarchical structure of the TSU that is based on the architectural differences of the two hardware systems. The common parameter of the two implementations is that the data of the applications are stored in the shared memory of each system, thus allowing us to retain the programming style of TFlux the same across the two different platforms.

For the execution of *TFluxSoft* on the Xeon Phi an instance of the SG is stored in shared memory and is used by all cores in the execution while the TSU follows a centralized approach, occupying a dedicated core as shown in Figure 7(a). In *TFluxSCC* (Figure 7(b)) we create a private copy of the synchronization graph for each core in the execution and follow a decentralized TSU approach, where the scheduling tasks are distributed to all cores.

6.1 TFluxSoft

The scheduling of parallel threads in the original *TFluxSoft* is a semi-centralized operation. This means that except from the TSU thread, part of the scheduling operations are executed by the application threads. This technique was used as an optimization that reduces the overheads of the TSU functionalities on a multi-core system [5]. Since there is only one instance of the SG in TFluxSoft, all the threads in the execution will share the same SG structures. This needs monitoring for mutual access and locking of shared structures that form the SG in the source code. In TFluxSoft the TSU that handles the update messages involving all application threads is a centralized operation (Figure 7(a)). This means that the update messages that notify a thread that is ready to execute is sent explicitly by the TSU thread. The TSU thread executes on a dedicated core and implements a busy wait loop in the background until an update message is ready to be send. More implementation details on this can be found in [5].

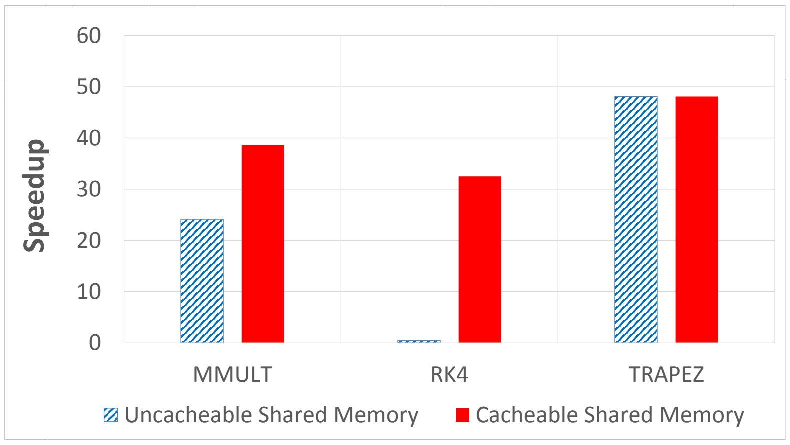

Figure 8: Comparing speedup results of *uncacheable* against *cacheable* shared memory on the Intel SCC using the TFluxSCC implementation

6.2 TFluxSCC

In the previous scenario, the TFluxSoft implementation takes advantage of the coherent shared memory environment of the Xeon Phi to manage correct simultaneous access on the shared application data. In the case of the Intel SCC, although it supports global address space, there is no cache-coherency protocol. In order to avoid data sharing conflicts between cores and to ensure correctness of a parallel execution, the SCC does not allow the use of the local caches for storing data coming from the global address space. This means that shared memory parallel models running on the SCC will not be able to cache data, and will consequently suffer major performance losses.

In Figure 8, we show the impact on the performance of three applications when not caching data coming from the global address space of the SCC (*Uncacheable Shared Memory*), compared to a scenario where the caching is enabled (*Cacheable Shared Memory*). The TRAPEZ application is the only one that shows no performance degradation and that is because it only uses a small number of data for the computation that are also private to each core. The number of data exchanges between cores and the number of updates to the main memory during computation are small in TRAPEZ so the non-caching technique will not affect its performance. In the case of MMULT and RK4 though, where a large number of data are used by all cores for the execution, the impact on the performance is significant. One would argue that despite the performance benefits shown in the *Cacheable Shared Memory* scenarios, the application results would be incorrect, due to the lack of a coherency mechanism. But, the DDM model and consequently our TFluxSCC implementation [8], doesn't require cache coherency as it doesn't allow simultaneous access on shared data. In a DDM program, data is shared through the input and output of the threads as they are defined by the dependencies declared by using the *pragma* directives. Consequently, at each moment only one thread can access a specific data structure on the global address space. Thus, caching global address space data is possible when using TFluxSCC. To ensure correctness, we flush the cache after a thread completes its execution to guarantee that the shared data is saved back to main memory as the SCC uses a write-back policy that makes the time of updating data in main memory unpredictable.

To achieve this we modified the SCC configuration (using a modified Linux image, provided by the SCC platform) as to allow the caching of data coming from the global address space. We also modified the TFluxSCC runtime as to flush the cores caches after writing data to the global address space. Although this may induce an overhead it can be removed in future many-core systems by using a write-through cache policy that will add no performance overheads.

To impose data-flow scheduling, *TFluxSoft* uses a single instance for the SG that is stored in the shared memory of the system as explained earlier. The SG is protected from mutual access

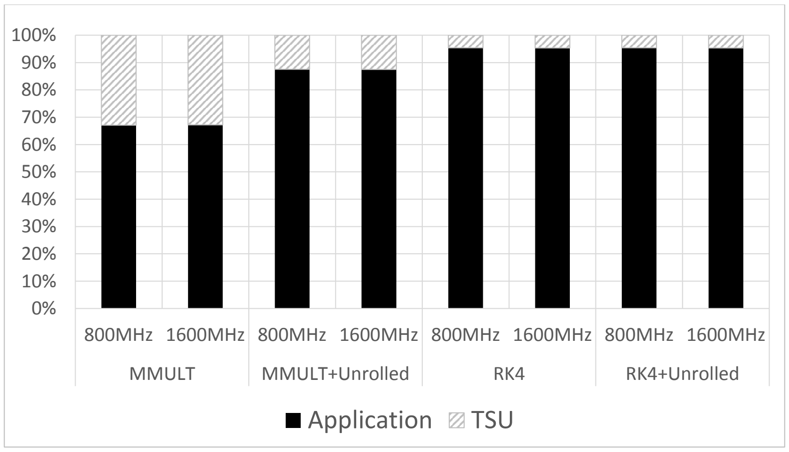

Figure 9: Execution time breakdown for different network frequencies for TFluxSCC on the Intel SCC. The black part of the bars represents the percentage of the total execution time that is used by the application threads, while the grey part represents the percentage for the TSU.

with locking provided by the Xeon Phi coherency mechanisms. As the SCC doesn't support cache coherency for data in the shared memory, there is no mechanism for monitoring and locking data either. What we propose with TFluxSCC, is a completely decentralized runtime system that is able to scale and consume as few resources as possible, thus reducing the overheads to a minimum. We create multiple copies of the SG and distribute a copy to each core in the execution as a private SG for exclusive use, thus avoiding any unsafe mutual accesses. The hierarchical structure of the TSU in *TFluxSCC* shows a runtime system where the TSU functionalities are distributed to each core in order to achieve scalability regardless of the number of cores used. In Figure 7(b) we show the schematic diagram of this implementation while in [8] we explain in more detail the reasons we choose this specific implementation for the Intel SCC.

In *TFluxSoft* the TSU thread occupied a single core for scheduling operations to avoid interference with application threads. Using a distributed runtime in *TFluxSCC* as shown in Figure 7(b), and because the SCC doesn't support multithreading, the application and the TSU threads will share the same computation unit, that could create a potential bottleneck in the performance. To study the overhead of the runtime system (Figure 9) we measure the execution time for the application *i.e.* the time spent to execute pure application code, and the execution time of the TSU, *i.e.* the time spent to execute the different scheduling procedures, such as the update of the local TSU structures after a thread execution and the communication between the TSU of each core, that is the exchange of update messages with consumer threads on different cores. We also test two different network frequencies to find out whether the message exchange process of the TSU can be improved by the current hardware. In our baseline scenario, the SCC network was operating at 800MHz. We then increased the network frequency to the maximum, which is 1600MHz and observe that the TSU time remains constant. This means that the time spent by the TSU for message exchanging in the *TFluxSCC* implementation is small and doesn't depend on the network frequency. When we apply unrolling though to the applications we observe that the overall TSU overhead reduces. This happens because in DDM every loop iteration is considered a separate thread, thus the more iterations we have, the more threads we have to schedule. Using unrolling, we reduce the total number of threads we have to schedule, thus reducing the total time consumed by the TSU procedures. Overall, the results of Figure 9 show that the overhead imposed by the TSU is small and it is only affected by the application and the number of threads used in the execution. Thus, making the platform independent of the hardware specifics.

Conventional shared memory parallel programming models require more sophisticated hardware components, that in systems of 100s -1000s of cores would be both power and performance inefficient

Table 2: Experimental workload description and problem sizes

Benchmark	Source	Description	Problem Size			
			Small	Medium	Large	X-Large
MMULT	kernel	Matrix multiply	64x64	128x128	256x256	512x512
RK4	kernel	Differential equations	1024	2048	4096	8192
Trapez	kernel	Trapezoidal rule for integration	30	31	32	33
Blackscholes	PARSEC	Black-Scholes Partial Differential Equation	4096	16384	65536	131072

due to the extra hardware and the overheads imposed by the coherency protocol [31]. Using TFluxSCC we manage to guarantee correct execution without major impact on the performance while at the same time maintain hardware simple and scalable.

6.3 Compilation Toolchain

As described earlier we made changes only to the back-end of the platform in order to support the new implementation and maintain the programming style and syntax of TFlux the same. To achieve this, we updated the DDM C source-to-source Preprocessor [27] to be able to generate source code for the new implementation by using the command line argument *-scc* during the preprocessing phase. For the *TFluxSoft* implementation we use the *-soft* flag. In the case of *TFluxSCC* we integrated in the preprocessor the SCCs programming API called RCCE (as explained in Section 5.2). The basic operations added on top of the previous implementations of TFlux are the initialization of the platforms' API, the support for memory allocation on the global address space and most importantly, the flushing of a cores' cache after writing data in the global address space. This latter operation is necessary for the model to explicitly synchronize the cache contents to memory as to ensure correctness, given that no hardware cache-coherency is available and the current cache policy used by the SCC is *write-back*. We also updated the preprocessor as to produce code that integrates the new TSU hierarchical rules as in this case each DThread executes its own TSU tasks separately.

By updating the preprocessor we created a single tool, the DDM C Preprocessor, that produces parallel source code for two different implementations (*TFluxSoft* and *TFluxSCC*) from the same C+DDM code. This means that the interface of TFlux (*i.e.* the directives in Table 1) is kept the same to both implementations. This way, we provide backward compatibility with the previous implementations of the TFlux platform. Thus, no change is needed to the previously implemented applications.

7 Experimental Setup

We evaluated our implementations using four different benchmarks, that were also used in the evaluation of the original TFlux Platform in [5]. Three of them are kernels that represent common scientific operations [32] and one from the *PARSEC* [33] suite. All the benchmarks are briefly described along with their different problem sizes in Table 2. For the experiments on both systems we used three different input problem sizes: *Small*, *Medium* and *Large*. For the Intel Xeon Phi scenario we chose to use an *X-Large* input size as well in order to show that scalability of the performance increases while increasing the problem size. We didn't use this last problem size for the Intel SCC as the size of the SCC global address space was limited, thus the input didn't fit the shared memory of the system.

For the TFluxSoft scenario our hardware setup was an Intel Xeon Phi 7120P with 61 cores and 4 threads per core, totaling 244 threads. The coprocessor we used has a total of 16GB of main memory and runs at 1.238GHz. To cross-compile our benchmarks for the Xeon Phi we used the Intel *icc* v.14.0.1 compiler with the *-mmic* flag indicating the MIC architecture and the -O3 optimization flag that includes auto-vectorization as well. We executed our applications in *Coprocessor-only mode* as described in Section 5.1.

The TFluxSCC was evaluated on an Intel SCC experimental processor, RockyLake version. The system has a total of 32GB of main memory and we used a balanced frequency setting for our experiments of 800MHz for the tile, the mesh interconnection network and the DDR3 memory and Memory Controllers. The operating system used for the Intel SCC cores was the Linux_dcm kernel provided by Intel SCC Communities repository that supports caching the data coming from the off-chip shared memory to L2 cache. To cross compile our benchmarks for the SCC we used the GCC v.3.4.5 compiler with the optimization flag -O3. For porting and executing the applications on the SCC we used the RCCE v1.4.0 tool-chain [28].

We executed the applications using up to the maximum number of cores for each processor - 60 for the Xeon Phi and 48 for the SCC. For the Xeon Phi the maximum number of cores available is 61 but we reserved a core for the OS as to avoid the interaction of the OS operations that could harm our applications performance. This is not done on the Intel SCC as each core on the SCC executes it's own instance of the OS. For the evaluation we measure execution time and in order to minimize the statistical error, we executed each experiment ten times. The results shown are produced by the arithmetic average of the measurements after excluding the outliers, *i.e.* the smallest and largest execution times. In this work we only study the scalability of the performance on each system. Thus the baseline execution for every scenario is the execution of single-threaded implementations of the benchmarks on a single core for each system. Finally, since our study emphasizes on the scalability of the DDM model, all results are reported as normalized *Speedup* compared to the baseline execution, so that we can compare them across different platforms.

8 Experimental Results

With this work we perform a scalability study of the performance on two many-core processors for applications with different characteristics. In our workload we included applications that are compute-bound like Trapez that executes multiple iterations of the Trapezoidal rule on the same data. Other applications that have a combination of memory- and compute-bound nature such as MMULT and Blackscholes and finally RK4 has a complex dependency graph. As we described in Section 7, we used three different problem sizes for the Intel SCC (*TFluxSCC*) and four for the Intel Xeon Phi (*TFluxSoft*). In Figure 10 we only show results for the large size for the SCC and the large and x-large sizes for the Xeon Phi as these are the most representative as they account for the scenarios with the least runtime relative overhead. The *SCC-L* represents the *TFluxSCC* implementation on the Intel SCC with the large input size and the *Phi-L* and *Phi-XL* represent the *TFluxSoft* implementation on the Intel Xeon Phi with large and x-large input sizes respectively.

The Blackscholes application seems to perform poorly compared to the other three applications. But in fact we achieve the same speedup for 16 threads as in the evaluation of PARSEC [33]. Increasing the number of threads results in little performance increase and this is because the problem size of each separate thread is too small. Even when we increase the application problem size to x-large in the *Phi-XL* we don't get enough performance increase as the problem size per thread remains low. Although, the thread dependencies within the application are few the runtime system in this case dominates the execution, thus the *SCC-L* performance is very low as well.

In MMULT, which is also a combination of compute- and memory-bound application, we observe the opposite behaviour as the application performance is increasing. The difference compared to Blackscholes is that the problem size per thread is large in this case. Even if it gets smaller when increasing the number of threads, the problem size per thread is still large enough to overcome the runtime overheads of the scheduler. Increasing the application problem size to x-large increases the ratio between application work and scheduling work thus the performance keeps increasing.

Except for a small (insignificant, in matters of execution time) reduction part, at the end of it's execution Trapez has no data sharing as the work of each thread is performed on private data. Based on this, we would expect to see a linear performance scalability of Trapez for both systems. Instead, the Intel Xeon Phi implementation under-performs compared to the Intel SCC implementation, which achieves linear performance scalability. The difference we see in the performance of the two implementations is due to the runtime system and more specifically the shared TSU in the *TFluxSoft* implementation. As explained in Section 6, in order to maintain a coherent TSU graph in *TFluxSoft*, locking is used on accessing the TSU structures by the threads. This comes with

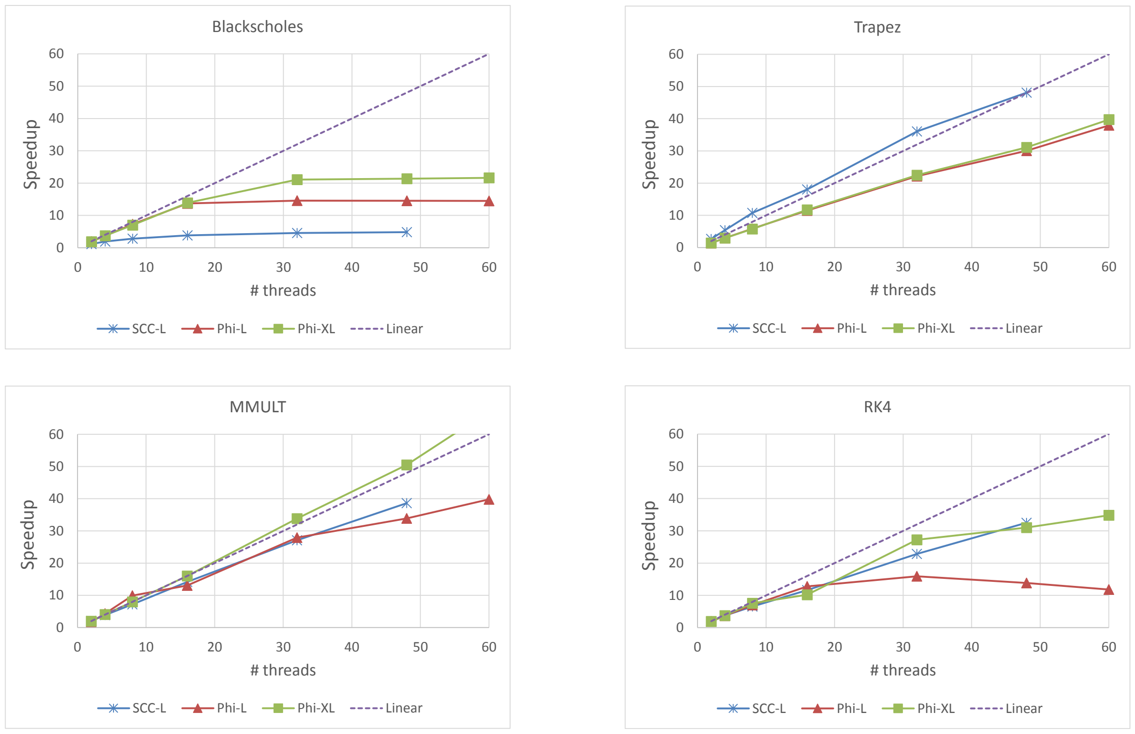

Figure 10: Comparing the speedup results of the two implementations of TFlux for the Intel SCC with large problem size (*SSC-L*) and the Intel Xeon Phi with large problem size (*Phi-L*) and x-large problem size (*Phi-XL*).

an overhead on the execution that can't be hidden even if we increase the problem size of the application (as in MMULT) because the work of each individual thread doesn't change (just like in Blackscholes). In the SCC and the *TFluxSCC* implementation we don't have this issue as each core has it's own private TSU graph and there is no simultaneous access on the same TSU structures, thus no locking is needed.

The final application we used in our evaluation is RK4, which is a combination of MMULT and Trapez. The *TFluxSCC* implementation performs better for large size as there are no conflicts or locking on the shared TSU structures compared to the *TFluxSoft* implementation (as in Trapez). As explained earlier, RK4 is an application with complex dependencies. RK4 executes multiple loops of code with multiple iterations. The dependencies in the application are described as cross-loop dependencies which means that each loop depends on its previous loop. As we increase the problem size we essentially increase the number of the iterations for each loop but the cross-loop dependencies remain the same. This means the problem size of each individual thread increases (just like in MMULT) and the TSU overheads in the TFluxSoft implementation are overlapped by the memory latencies of the application. Eventually this increases the scalability of the performance as we increase the application problem size. RK4 though presents another issue that is a result of the weak scaling technique we use for increasing the problem size. After a particular number of threads the performance of RK4 drops on the Xeon Phi implementation. As we are using weak scaling, the size of data to be processed increases, thus we overcome the overheads of the scheduling but as we increase the number of threads in the execution, we reduce the portion of work for each thread and increase the scheduling overheads as well since more threads will now share common TSU structures. This happens for all application but for different threads for each one. We can overcome this by using strong scaling, where we increase the number of threads and the input size such that the portion of work for each thread is the same and large enough to overcome the scheduling overheads.

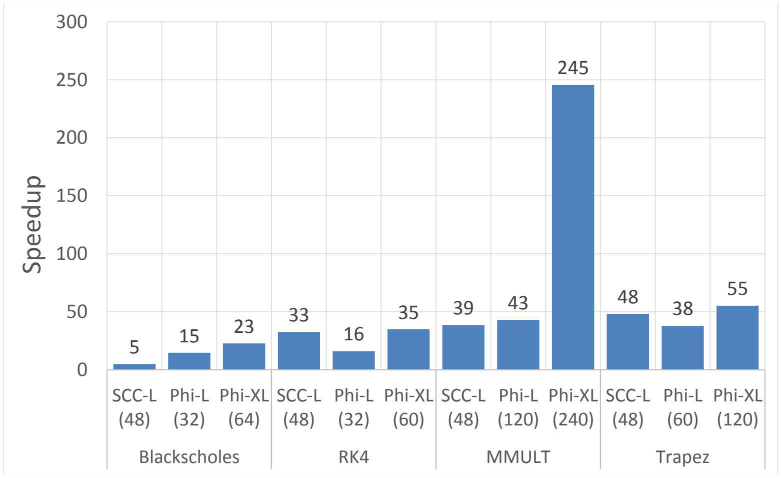

Figure 11: Maximum speedup results for each application on each system with the *Large* and *X-Large* problem sizes. Each bar represents the maximum speedup achieved for each one and in parenthesis we show with how many threads this speedup was achieved.

In another test we used the multithreading capabilities of the Xeon Phi and in Figure 11 we show the maximum speedup achieved for each application on each system with large and x-large input sizes. Maximum performance is achieved when using the x-large input size on the Intel Xeon Phi. Multithreading helps applications that have a combination of compute- and memory-intensive nature, like MMULT as it gets an overlap of the memory latency with the computation of the extra threads. For MMULT we observe a large speedup for the x-large input size on the Xeon Phi as the data can fit the L2 cache of the system and the particular size can hide the overheads produced by the runtime system. Our analysis shows that memory latencies on accessing data in MMULT create overheads on the application threads as they wait for memory loads. When we increase the application problem size the work of each thread is also increased. This means that the memory overheads will eventually hide the scheduling overheads imposed by the TSU, while the extra computation from the extra threads (four per core) will overlap the memory overheads resulting in a linear performance scalability as it happens for *Phi-XL*. In general we observe that the SCC system and it's architectural structure (clustered hierarchy) can benefit applications with complex dependencies (RK4) as it can avoid simultaneous access on shared TSU structures and costly mechanisms like monitoring and locking of shared data by using a private copy of the TSU graph on each processing core. It can also produce linear performance on applications with little application data sharing among the execution threads (Trapez) as it avoids the costly flushing of L2 caches as explained in Section 6.

9 Conclusions

In this work we exploit the Data-Flow model on two many-core systems with different characteristics. We extended the TFlux platform and create a unified Data-Flow platform to support the programming and execution of DDM applications on a shared memory many-core (Intel Xeon Phi) and a clustered many-core (Intel SCC). Using a set of applications with different characteristics we were able to show that the performance scales well and good speedup is observed for most applications. It is relevant to notice that these are executions of real applications on real many-core processors and that this is the first attempt to scale a software implementation of the DDM model on a many-core processor.

We believe that our implementation is suitable for future generations of many-core processors as it is a software implementation that can be easily configured to any many-core architecture. It has limited demands on hardware support that allows for a simpler design with a larger number of execution units and thus increases the parallelism offered by the hardware. The TFluxSCC implementation only requires a global address space for storing application data and a selective data-cache flush policy for data that were written and came from the global address space. Cache-coherence is not a requirement and this leads to using simpler, more scalable hardware. The TFluxSoft implementation can be ported as is on any cache-coherent shared memory many-core processor. The implementation itself allows for any hardware optimizations to be performed without any interference to or from the runtime system.

References

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[3] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, 2006.

[4] K. Stavrou, P. Evripidou, and P. Trancoso, "DDM-CMP: Data-Driven Multithreading on a Chip Multiprocessor," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3553, pp. 364–373.

[5] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "TFlux: A portable platform for data-driven multithreading on commodity multicore systems," in *Proceedings of the 37th ICPP '08.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 25–34.

[6] J. Jeffers, "Intel® xeon phi coprocessors," in *Modern Accelerator Technologies for Geographic Information Science.* Springer, 2013, pp. 25–39.

[7] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *IEEE International ISSCC 2010.* IEEE, 2010, pp. 108–109.

[8] A. Diavastos, G. Stylianou, and P. Trancoso, "Tfluxscc: Exploiting performance on future many-core systems through data-flow," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, March 2015, pp. 190–198.

[9] A. Duran and M. Klemm, "The intel® many integrated core architecture," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on.* IEEE, 2012, pp. 365–366.

[10] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, Jan 1998.

[11] U. Drepper and I. Molnar, "The native posix thread library for linux," *White Paper, Red Hat Inc*, 2003.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the {MPI} message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789 – 828, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167819196000245

[13] S. Saini, H. Jin, D. Jespersen, H. Feng, J. Djomehri, W. Arasin, R. Hood, P. Mehrotra, and R. Biswas, "An early performance evaluation of many integrated core architecture based sgi rackable computing system," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–12.

[14] E. Totoni, B. Behzad, S. Ghike, and J. Torrellas, "Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs," in *Proceedings of the ISPASS '12*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 78–87.

[15] I. Comprs Urea, M. Riepen, and M. Konow, "RCKMPI Lightweight MPI Implementation for Intels Single-chip Cloud Computer (SCC)," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2011, vol. 6960, pp. 208–217.

[16] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor," in *HPCS International Conference*. IEEE, 2011, pp. 525–532.

[17] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the Intel SCC," in *Proceedings of the IEEE CLUSTER '11*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 139–149.

[18] D. Baliley, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, *et al.*, "The NAS parallel benchmarks-summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, vol. 91. New York, NY, USA: ACM, 1991, pp. 158–165.

[19] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha, "A Case for Software Managed Coherence in Manycore Processors," in *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10*, 2010.

[20] Q. Huang, Z. Huang, P. Werstein, and M. Purvis, "GPU as a general purpose computing resource," in *Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008.* IEEE, 2008, pp. 151–158.

[21] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ICS '12*. New York, NY, USA: ACM, 2012, pp. 341–352.

[22] J. Lee, J. Kim, J. Kim, S. Seo, and J. Lee, "An OpenCL framework for homogeneous manycores with no hardware cache coherence," in *PACT'11*. IEEE, 2011, pp. 56–67.

[23] A. Duran, A. Eduard, M. B. Rosa, L. Jess, M. Luis, M. Xavier, and P. Judit, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626411000151

[24] J. Planas, R. M. Badia, E. Ayguad, and J. Labarta, "Hierarchical task-based programming with starss," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009. [Online]. Available: http://hpc.sagepub.com/content/23/3/284.abstract

[25] N. Eicker, T. Lippert, T. Moschny, and E. Suarez, "The deep project - pursuing cluster-computing in the many-core era," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 885–892.

[26] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri, *et al.*, "TERAFLUX: Harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, 2014.

[27] P. Trancoso, K. Stavrou, and P. Evripidou, “DDMCPP: The data-driven multithreading c pre-processor,” in *The 11th Workshop on Interaction between Compilers and Computer Architectures*, 2007, p. 32.

[28] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, *et al.*, “The 48-core SCC processor: the programmer’s view,” in *Proceedings of the ACM/IEEE SC ’10*, 2010, pp. 1–11.

[29] R. Rahman, *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers.* Apress, 2013.

[30] M. Loghi, M. Poncino, and L. Benini, “Cache Coherence Tradeoffs in Shared-memory MPSoCs,” *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 383–407, May 2006.

[31] D. Hackenberg, D. Molka, and W. E. Nagel, “Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems,” in *Proceedings of the MICRO 42.* New York, NY, USA: ACM, 2009, pp. 413–422.

[32] M. C. Seiler and F. A. Seiler, “Numerical recipes in C: the art of scientific computing,” *Risk Analysis*, vol. 9, no. 3, pp. 415–416, 1989.

[33] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’08. New York, NY, USA: ACM, 2008, pp. 72–81.