Int J Parallel Prog (2016) 44:278–307
DOI 10.1007/s10766-015-0373-6

The Design and Implementation of TIDeFlow: A Dataflow-Inspired Execution Model for Parallel Loops and Task Pipelining

Daniel Orozco[1] · **Elkin Garcia**[1] · **Robert Pavel**[1] · **Jaime Arteaga**[1] · **Guang Gao**[1]

Received: 31 January 2013 / Accepted: 8 July 2015 / Published online: 21 July 2015

Abstract This paper provides an extended description of the design and implementation of the Time Iterated Dependency Flow (TIDeFlow) execution model. TIDeFlow is a dataflow-inspired model that simplifies the scheduling of shared resources on many-core processors. To accomplish this, programs are specified as directed graphs and the dataflow model is extended through the introduction of intrinsic constructs for parallel loops and the arbitrary pipelining of operations. The main contributions of this paper are: (1) a formal description of the TIDeFlow execution model and its programming model, (2) a description of the TIDeFlow implementation and its strengths over previous execution models, such as the ability to natively express parallel loops and task pipelining, (3) an analysis of experimental results showing the advantages of TIDeFlow with respect to expressing parallel programs on many-core architectures

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

✉ Daniel Orozco
orozco@udel.edu

Elkin Garcia
egarcia@udel.edu

Robert Pavel
rspavel@udel.edu

Jaime Arteaga
jaime@udel.edu

Guang Gao
ggao@capsl.udel.edu

[1] University of Delaware, Newark, DE, USA

and (4) a presentation of the implementation of a low overhead runtime system for TIDeFlow.

Keywords Dataflow · Task pipelining · Parallel execution models · TIDeFlow · Runtime system · Graph languages · Codelets · Iterated dataflow · Dependency graph

1 Introduction

The need for the careful allocation and scheduling of resources on modern processors has increased the difficulty of efficiently utilizing said resources. Poor allocation and scheduling of resources leads to contention and, ultimately, poor performance.

Abundant examples of the difficulties of resource orchestration can be found throughout the literature of parallel execution. In many cases, the time spent developing the application itself pales in comparison to the time and effort spent in adjusting the timing of operations to fully utilize the available resources. These problems have been documented, with great detail, by Garcia [16], in his work on matrix multiplication.

The progressive efforts made by Garcia can be observed in his sequence of publications [16–19], where he described each step of the development of a highly optimized version of matrix multiplication. One interesting point evidenced in the experience by Garcia, is the significant amount of effort that must be devoted to achieve proper overlapping and pipelining of operations. This effort required the development of his own primitives for synchronization and scheduling because the tools available to him, pThreads [4] and TNT [8], did not provide the functionality that he required.

The difficulties found by Garcia are the legacy of decades of execution in environments dominated by very few processors where resource contention was not an issue. This was because such contention either never occured or it was reasonably simple to hand-code a solution. However, that is no longer the case for complex programs in many-core architectures where hundreds of processing units compete for dozens of different resources.

In such a scenario, it is highly probable that resource contention will occur, and efforts to hand-code a solution are likely to be nontrivial. For example, Garcia encountered that correctly allocating memory bandwidth to processors at specific times was crucial to obtain high performance. In addition to being crucial, it was also hard to achieve, both in terms of achieving it during execution, and in terms of writing code that would do such management. This problem is typical of modern many-core architectures, such as Cyclops-64 [7], where explicit memory management is done by the programmer.

This paper describes an execution model where it is possible to express and execute highly parallel programs it is necessary to manage shared resource, such as memory bandwidth, between many threads.

The issue can be illustrated by the simple case of a parallel processor with some fast on-chip memory and some slow off-chip memory where every processor does memory movements back and forth from slow to fast memory and viceversa, interleaved with some computation.

How can this behavior, *and its constraints*, (bandwidth, available on-chip memory space and so on) be expressed using traditional models?

This problem has been successfully solved in the past, albeit at great expense to the programmer. The constructs to express resource constraints are cumbersome, and hard to write and debug. For example, although possible, it is hard to write a program where several tasks execute concurrently, and efficiently, sharing a single resource, without contention. Locks and critical sections, for example, which come to mind first, achieve concurrency, but they are prone to contention and wasted time.

Garcia's experience is an excellent example of why traditional programming models provide only awkward ways to address these issues. Although arbitrary functionality is possible to be obtained with OpenMP [5], MPI [21] or pThreads [4], the solutions found are far from simple. The programmer has to devise clever and complex constructs that are hard to maintain when the program is upgraded or when it must run alongside other programs.

To address the issues of traditional programming models, we propose a new model, based on the dataflow model of computation [24]. We have used the dataflow model of computation because it allows expression of arbitrary parallelism and synchronization, and it dissociates the efforts of synchronization and scheduling. These characteristics can be observed also in systems such as the Codelet model [39], and others.

Our proposed model, which we call the Time Iterated Dependency Flow (TIDeFlow) Model, provides intrinsic constructs for parallelism, resource management and pipelining in a program. Parallelism is achieved through a dataflow-style graph representation for programs and pipelining is achieved by allowing the programmer to specify constraints for the execution of parts of the program. These constraints can be in the form of ordering between parts of a program, availability of resources, or number of times some other part of the program has executed. Scheduling is completely decoupled from synchronization, greatly simplifying the development of programs.

Figure 1 illustrates the advantages TIDeFlow provides for the description of parallel programs. Several things can be noted in the Figure: (1) the graph programing model is an intuitive way to represent parallel programs, (2) parallel loops are easily represented by adding a weight next to each operation (node) in the graph (3) the execution flow is controlled by tokens, and (4) loops are easily represented with backedges.

An equivalent program written in pThreads, that would achieve the same functionality, would be more difficult to write. First, threads for the parallel loops have to be created and synchronized and second, there is not a single, simple construct that would

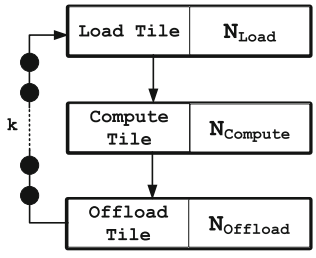

Fig. 1 A simple TIDeFlow program that achieves pipelining of a tiled computation

allow control of the execution of several instances of the operations in the way tokens do.

We argue that the complexity of a solution using pThreads is not due to poor implementation of the desired behavior, but rather, by the lack of adequate primitives in serial programming models to address arbitrary parallelism.

Our TIDeFlow model addresses all of these problems. These are some of the most relevant features of TIDeFlow:

- A graph programming model.
- Native constructs to represent parallel loops.
- Native constructs to express resource constraints.
- Automatic pipelining during execution.
- Dissociation between synchronization and scheduling.

We also show in Sect. 11 an implementation of a high performance runtime system for TIDeFlow, which has very low latency, and which can be easily used by the programmer. In our implementation of the Runtime System, we have introduced recently developed techniques to increase the performance of the basic operations. With our implementation, we have achieved an unprecedented runtime overhead of a few hundred clock cycles per task.

We will describe in detail each one of the elements in the TIDeFlow execution model and their usefulness. Then, we provide several real life examples, where we compare in detail the implementation of common algorithms and we show the advantages and disadvantages of using TIDeFlow over traditional programming models.

The rest of this paper is organized as follows: Sect. 2 provides relevant background. Sect. 3 provides a broad overview of the TIDeFlow execution model. Sections 4 and 5 go into greater detail on arcs, actors and tokens, the components of the TIDeFlow program model. In Sect. 6 we focus on the high degree of composability inherent in TIDeFlow and in Sect. 7 we show how the TIDeFlow model allows for the runtime system to efficiently pipeline tasks with minimal user input. Section 8 discusses the underlying memory model of TIDeFlow, while Sect. 9 presents the implementation of the TIDeFlow model using several examples. Section 10 introduces the concept of parallel program traces, followed by Sect. 11 where the usability and performance of TIDeFlow is evaluated through the execution of several benchmarks. Finally, Sect. 12 presents a brief summary of the paper and Sect. 13 describes our future work to improve the TIDeFlow execution model.

2 Background

This section presents a brief description of some of the dataflow models analyzed for the design of the TIDeFlow model and how their advantages and disadvantages were taken into consideration during the formal definition of the model (a more comprehensive study of dataflow models can be found in the work by Najjar et al. [24]). The first model studied was the Static Dataflow model proposed by Dennis [9]. This model is very good at describing parallel programs but it presents difficulties in the representation of parallel loops and in the execution of recursive functions. TIDeFlow overcomes the former by using a single actor to represent a parallel loop. TIDeFlow

also supports the combination of several operations in one actor, which was originally proposed in the Macro Dataflow model [35], and implements an efficient operation pipelining in the dataflow graph as proposed by Gao [13].

The Petri Net model [23] was also studied due to its advantages in the description and study of parallel systems and because it models resource sharing better than the Static Dataflow model. Specifically, the concepts of transitions and places of a Petri Net were used to build the definition of weighted nodes in the TIDeFlow model, which is one the features that makes TIDeFlow very distinctive from other dataflow models.

Like TIDeFlow, several models used the concept of queues to distribute the work between actors. Among these models are the Kernel for Adaptive, Asynchronous Parallel and Interactive Programming (KAAPI) [20], Cilk [3], X10 [11], Habanero C, and Habanero Java [38]. But unlike those models, TIDeFlow uses different rules for the expression of dataflow programs, namely the representation of parallel loops as a single actor.

EARTH or Efficient Architecture for Running Threads [25,37] is a model that implements a dataflow program using commodity hardware. This model features two levels of parallelism classifying the actors as threaded procedures and fibers. The techniques used on EARTH for the synchronization of fibers are also used on TIDeFlow, but both models differ in the rules used for the representation of parallel loops and in that the TIDeFlow model adds weights to actors.

TIDeFlow is a dataflow-based model and as such, a comparison with other non-dataflow models such as OpenMP [5], or models used for SIMD machines and GPUs (such as CUDA [26] or OpenCL [36]) would be more philosophical than practical. For example, TIDeFlow execution is controlled by dependencies, while OpenMP is controlled by the flow of the program (i.e. an OpenMP task does not start when its data is available, but rather when the parent thread reaches a particular point). A more general discussion is presented in the conclusions section.

3 An Overview of the TIDeFlow Model

The motivation behind the development of the TIDeFlow model is to support the execution and development of HPC programs. To achieve that goal, the necessities of High-Performance Computing (HPC) programs must be identified and understood.

We start with the observation that many scientific HPC programs are related to the simulation of physical phenomena, and that they are usually composed of repetitive patterns of regular computations such as Fast Fourier Transforms (FFT) [29] Matrix Multiplications (MM) [16] or Jacobi kernels [27].

Of particular interest is that some of the innermost loops in most of these computations are fully parallel. For example, each one of the dot products required to compute each element in matrix multiplication can be executed in parallel, as well as the butterflies in an FFT computation, and each one of the data elements in a Jacobi computation.

One of the things that became apparent in our work with FFT, MM and Jacobi is that successfully obtaining a highly optimized parallel program requires the ability to express arbitrary dependencies between parts of the program. We have also became

aware, that although it is possible to express arbitrary relationships using traditional techniques such as pThreads, it is a laborious and error prone task.

Those reasons have prompted us to use dataflow as a starting point for TIDeFlow. We use directed graphs to represent programs, which enables us to express arbitrary dependencies between parts of a program.

This ability to express arbitrary dependencies between programs is a powerful tool when specifying constructs such as task pipelining, overlapping of communication and computation, or dependencies between parallel loops.

With our model, we support arbitrary dependence relationships between parts of a program by expressing programs as graphs. Graphs in the TIDeFlow model are represented as a collection of *actors* that are connected by directed *arcs*. As in dataflow, actors represent the computations of the program while arcs represent the dependencies between those computations. Unlike dataflow, however, the arcs in a TIDeFlow program are not restricted to data dependencies only, and instead, they can represent other kinds of dependencies such as resource, control, or simply desired ordering between computations. These extensions allow for powerful constructs such as automatic pipelining of tasks or automatic load balancing.

The execution rules for TIDeFlow programs are similar to the rules that govern dataflow programs [9]. Actors are allowed to execute whenever their input dependencies are met. The arcs, which represent these dependencies, may carry tokens to indicate that a dependency has been met. Tokens do not carry data in the TIDeFlow model, instead, they indicate that a dependency has been met. This decision enables tokens to represent data dependencies as well as other kinds of dependencies.

The following is a more formal explanation of the TIDeFlow execution model.

4 Actors

Actors in TIDeFlow are similar to Macro Dataflow actors [35] in that they both execute a set of sequential operations. However, the fundamental unit of computation in TIDeFlow is an actor representing a parallel loop, since generally HPC applications are mostly composed of such control flow statements.

The execution of these actors is supported by a fast and decentralized runtime system, responsible for scheduling and assigning tasks to the available hardware.

4.1 Definition

The representation of a TIDeFlow actor is presented in Fig. 2, where N is the number of loop iterations, t is the number of *time instances* the actor has already been executed, and *f()* is the code to be executed by each loop iteration. An actor is executed a total of T time instances; T is provided by the user at the beginning of the program. The time instance is passed to the user code during execution, and is typically used to take decisions about the termination of the program through return signals, or to construct multidimensional loops.

The number of iterations in the loop (N) and the loop function ($f()$), along with a set of user-defined values that are available to the function, represent the actor's *properties*,

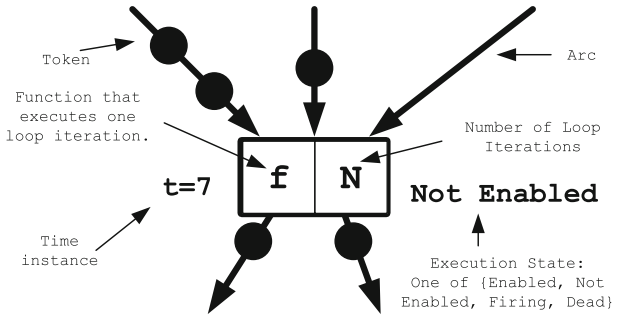

Fig. 2 Generic representation of a TIDeFlow actor

```
1 ...
2 for t in 0 to T-1
3
4   /* Parallel Loop */
5   for i in 0 to N-1
6     f(i,t);
7   end for
8
9 end for
10 ...
```

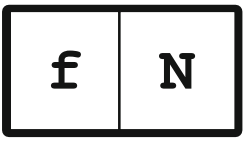

Fig. 3 A parallel loop and its TIDeFlow actor

which are constant during the execution of the program. In the most common case, these user-defined values are pointers to statically allocated memory that can be used to pass data between actors.

The ability to represent loops as actors is a very powerful tool that isolates users from the difficulties of managing and scheduling the tasks in parallel loops. This is shown in Fig. 3, where the code of a generic *for* loop is represented by a single TIDeFlow actor.

4.2 States

Each TIDeFlow actor holds a state that is used by the runtime system to handle its scheduling and execution. At any moment, an actor can be either at the *not enabled*, *enabled*, *fired* (executing), or *dead* state. The role of each state is:

- *Not enabled* The actor is waiting to have at least one token available per input arc.
- *Enabled* There is at least one token available in each input arc. The actor is eligible for scheduling and execution.
- *Fired* The actor is currently being executed.
- *Dead* The actor will not be executed again.

Figure 4 shows a formal representation of the possible transitions between actor states.

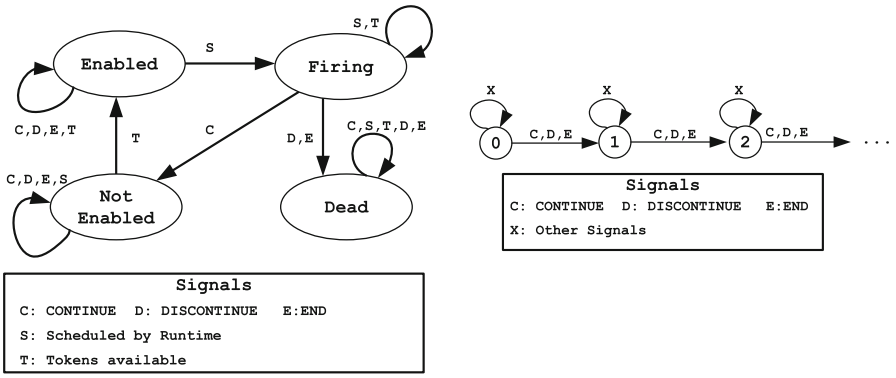

Fig. 4 State transitions for actors (*left*) and time instances (*right*)

4.3 Execution

As in dataflow, actors may be executed when there is a token available at each of the actor's input arcs. When firing, an actor consumes exactly one token from each input arc and, if it produces tokens, it will produce exactly one token in each output arc.

Also alike dataflow, TIDeFlow actors with no input arcs have their dependences met at all times, which means that they are always available to be executed. The result of this property is that actors with no input arcs are continuously being executed and producing output tokens until they enter the *dead* state.

When an actor is scheduled for execution, it may enter into the *firing* state, in which the runtime system assigns available hardware resources to it. When the actor fires, the runtime system creates N concurrent invocations of the loop function $f()$, one for each loop iteration. When all of these iterations have been computed, the actor's time instance t is incremented by one and a termination signal is generated. An actor is considered to have completed execution when all the invocations of $f(i, t)$ have been computed, with $i = [0, N)$ and $t = [0, T)$.

4.4 Termination Signals and Token Generation

The final step in the execution of an actor is the generation of a termination signal and, depending on it, the generation of exactly one token per output arc.

The termination signals are defined by the return value of the operation executed by the actor. In the current implementation, macros have been provided to the user so that the proper return signal is generated. Each one of those signals is used to control the way the program will continue its execution. The possible termination signals that can be generated are `CONTINUE`, `DISCONTINUE`, and `END`.

`CONTINUE` specifies that execution of the program should proceed, and that exactly one token must be generated into each output arc. `DISCONTINUE` specifies that the actor generating the signal should not be executed again, and that it should remove itself from the program, removing with it all of its input and output arcs. No tokens

```
int64_t Hello( void * parameters, int it, int t )
{
  /*
    This program will print:
    Hello World!
  */
  printf( "Hello World!\n" );
  return( END );
}
```

Fig. 5 TIDeFlow Hello World program

need to be generated because the actor removed itself as well as all of its output arcs from the program. `END` specifies that the actor generating the signal should not execute again, and neither should the actors that depend on it. To achieve this, an actor returning and `END` signal will mark itself as *dead* to prevent it from being executed again, and it will not generate any output tokens to prevent execution of the actors that depend on it. It is important to note that once an actor becomes dead through generation of an `END` signal, all of the actors downstream from it will eventually stop execution as well because they will not receive the tokens they require to execute, eventually terminating the execution of the program.

4.5 Basic Program Examples

4.5.1 Hello World

Following the tradition of many programming languages, the introduction of the basic TIDeFlow constructs is done here by presenting a "Hello World" program in Fig. 5.

Note that the `Hello` actor used in this figure has no input arcs and as such, it is always enabled. Also of interest is to note that all actors are parallel loops, and during execution, the user code receives as parameters the iteration instance to execute. The time instance and predefined, fixed user parameters, which are part of the actor state, are also passed to the user code.

4.5.2 Iterations and Time Instances

A more comprehensive example on time instances and iterations is shown in Fig. 6. In this example, the codelet returns `CONTINUE` and it will continue executing if its dependencies are met. Also, the runtime system takes charge of scheduling and passing the relevant parameters to each actor instance. Execution traces corresponding to the program in Fig. 6 are shown in Fig. 7. Note that all parallel iterations in a time instance must finish before the time instance is incremented.

4.5.3 Termination Signals

The use of termination signals is shown in the codelet of Fig. 8. The TIDeFlow graph used is shown the same of Fig. 6. In this case, the codelet will stop after iteration $t = 4$.

```
int64_t Example( void * parameters, int it, int t )
{
  /*
    This program will iterate from it=0
    to infinity, and
    for each value of "t",
    the program will execute
    the printf statement below, in parallel,
    for it=0,..2.
  */
  printf( "it=%d, t=%d\n" );
  return( CONTINUE );
}
```

Example	3

A possible output of this program is:

```
it=1, t=0
it=2, t=0
it=0, t=0
it=2, t=1
it=1, t=1
it=0, t=1
it=0, t=2
it=1, t=2
it=2, t=2
...
```

Fig. 6 TIDeFlow program example

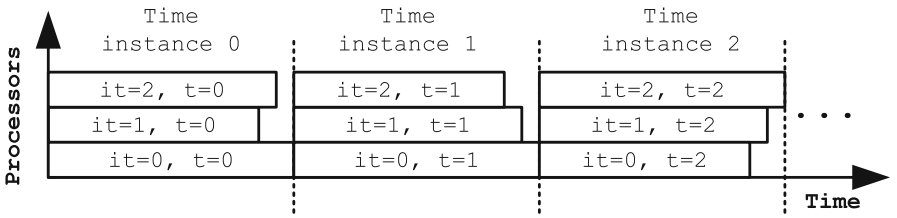

Fig. 7 Execution trace of the example program

```
int64_t Example( void * parameters, int it, int t )
{
  int t;

  printf( "Hello World at t = %d\n", t );

  if ( t == 4 )
    return( END );

  return( CONTINUE );
}
```

Fig. 8 Use of signals from the user code to control execution

5 Arcs and Tokens

When an actor finishes execution, it may signal other actors that depend on it by creating tokens. Tokens do not carry data, they only convey the meaning of a dependence met from one actor to another. Data is passed between actors through shared memory. This is similar to the EARTH model of computation [37].

The arcs in TIDeFlow graphs provide a simple way to express dependencies between actors. Dependencies in TIDeFlow typically represent data dependencies in producer-consumer scenarios but they may also represent resource or control dependencies. Arcs are allowed to carry an unbounded number of tokens, although particular implementations can restrict the number of tokens to a certain maximum.

5.1 Representing Outer Loop Carried Dependencies

Although many inner loops in HPC programs are embarrassingly parallel, the iterations of outer loops cannot, in general, be executed in parallel. The reason is that the outer loops of an HPC program usually express the high level relationships between computation stages in a program. In most cases, the outer loops in an HPC program capture the causality relationships between parts of a program. They can represent a time-ordering, or a sequence of communication and computation and so on.

Consider the case of an application where tiling has been employed to improve locality. When optimized, the tiling approach must be accompanied by matching memory movement and by successful memory management. In the simplest configuration, a buffer will be used to improve locality. Given the nature of on-chip memories in many-core processors, memory movement is explicit to and from the buffer. Figure 9 shows in detail the dependencies of such an approach. The computation section must finish before the results are offloaded to memory. Computation must wait for the results to be loaded to memory.

However, there is a *loop carried dependency* between offloading of results and loading of the next data block: The *next* loading operation must wait until the results of the *previous* tile computation have finished. These outer loop carried dependencies are similar to traditional loop carried dependencies in that they refer to dependencies between different iterations of a loop. However, the outer loop carried dependencies not only express dependencies between individual memory accesses by the loops but also the conceptual (control, data, resource) dependencies between them.

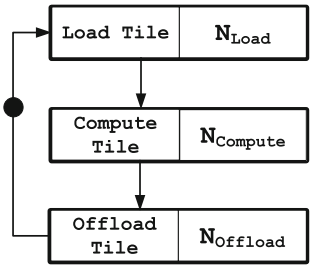

Fig. 9 Dependencies between memory movement and computation in a tiled application

The dependency between offloading the data in a buffer and using the buffer again in the next iteration is represented by a backwards arc in the program. To allow execution of the first actor, the backwards edge is initialized with one token.

The concept of dependencies can be extended to allow dependencies between different time instances. These dependencies are referred to as *outer loop carried dependencies*. The name comes from the fact that there exists a loop carried dependency present in the outer loop. It should be pointed out that innermost loop carried dependencies are not possible in TIDeFlow because inner loops are expected to be parallel and no data communication or synchronization is supported between loop iterations within the same parallel loop.

The dependence distance in outer loop carried dependencies is controlled by the number of tokens placed in the arc at the start of the program. In general, an outer loop carried dependency of distance k between an actor A and an actor B can be represented by placing k initial tokens in the arc that connects A and B. The additional number of tokens regulate the execution in such a manner that time instance $t + k$ of B will wait for a token produced by A at time instance t.

5.2 Examples

The examples in this section are helpful in understanding the meaning and the use of outer loop dependencies.

5.2.1 Overlapping Communication and Computation

Tiling [22], the common construct for memory locality, can be used to illustrate the use of dependencies. When tiling is used in a program, some memory is loaded from lower levels of the memory hierarchy into on-chip memory. Once the memory has been loaded, the processors can work on it.

In an application that uses tiling, two buffers are used to overlap communication and computation: Computation can be done on one buffer while memory movement is done on the other. Figure 10 shows a TIDeFlow program that will optimize the use of memory bandwidth and processor resources. The number of loop iterations in the loader codelets is 8 because, for this example, we assume that it takes exactly 8 memory-movement threads to saturate the memory bandwidth of the many-core processor used. The dependency between the loaders and their respective compute actors are data dependencies; they indicate that computation can only proceed when the load has completed. Of particular interest is the dependency between the loader

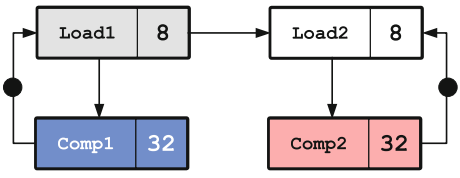

Fig. 10 TIDeFlow construct for overlapping of communication and computation

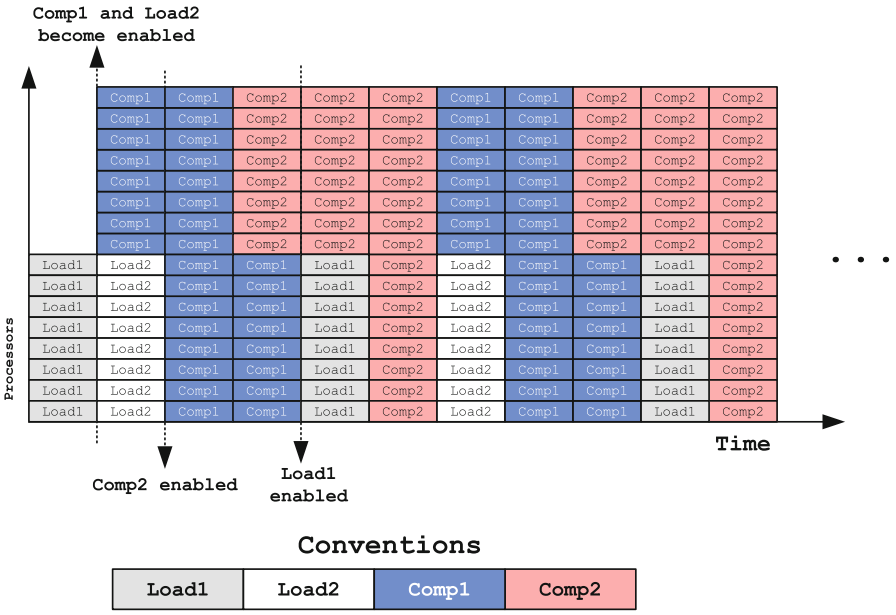

Fig. 11 A possible execution trace for the program of Fig. 10

actors, which indicates a resource dependency: One of the loader actors has to wait for the other to finish there is not enough available memory bandwidth.

Figure 11 shows a possible execution trace for the program of Fig. 10. To simplify the trace example, it has been assumed that only 16 processors participate in the computation.

5.2.2 Using Outer Loop Carried Dependencies

The use of loop carried dependencies can be illustrated through an application where several buffers are available. Each buffer is used to hold some data that needs to be computed as a tile [22].

Processors can asynchronously do memory movement to put memory into the buffers. In that way, more than one buffer can be ready for execution, and more than one memory transfer, either from main memory or to main memory, can be happening at the same time. This approach is useful when the computation of a tile takes an unpredictable amount of time, since it allows slow computation of some tiles to be amortized over several tiles.

Now, let us consider the dependencies between the operations. Three main operations are done: (1) Computation of a tile, (2) prefetching the data needed by a tile and (3) offloading the data computed by a tile.

The computation of a tile can proceed only after the data prefetching for that tile has completed. For that reason, there is a dependency between the memory movement (percolation) [17] required and the computation of the tile. In a similar way, offloading the data computed can only happen once the computation of the tile has finished, so

there is a dependence from the computation step to the offloading step. However, there is not an immediate dependency between offloading of a tile and loading of the next tile. In fact, if there is a total of k buffers, a particular buffer is only reused after other $k - 1$ buffers have been used.

For this reason, there is an *outer loop carried dependency* between offloading a buffer and prefetching the same buffer with *dependence distance* of k. Figure 1 shows the dependency graph for this situation.

5.2.3 Expressing Pipelining Through Backedges

The program of Fig. 12 presents an example of how backedges in programs cause pipelined execution.

The program represents a typical tiled computation where three buffers are available. First, buffers are loaded with data, then, computation is performed using the data loaded, followed by an unloading of the data.

As can be observed in the figure, the backedge between the `O` actor and the `L` actor restrains the speed at which `L` can execute, ultimately resulting in an optimal pipeline. The resulting pipelined execution was possible, in this case, because there were enough processors to execute the actors as they become available.

5.2.4 A Matrix Multiplication Kernel

Figure 13 shows one of the ways to implement the inner kernel of matrix multiply presented by Garcia et al. [16] and a high level pseudocode for the computation of the tiles.

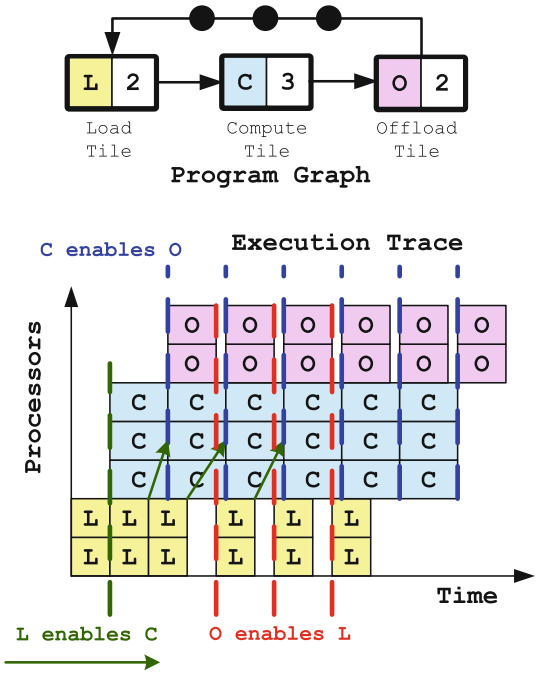

Fig. 12 Execution trace of a pipelined program

```
...
/* Globals */
int buffer;
...
for step in 1 to NumSteps

  /* Select which buffer to use */
  buffer = step % 2;

  /* Load buffer */
  for i in 0 to 7
   BufferLoader(i)
  end

  /* Innermost loop: Computes one tile */
  for i in 0 to 1023
     SubTile(i);
  end for

  /* Offload buffer */
  for i in 0 to 7
   BufferOffloader(i)
  end

end for
...
```

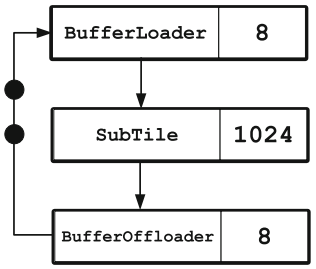

Fig. 13 Tiled matrix multiplication using TIDeFlow

The implementation uses two buffers in on-chip memory. A loop carried dependency of distance 2 between the `BufferLoader` actor and the `BufferOffloader` actor allows overlapping of communication and computation between the two buffers used in the program. Because all the innermost parallel loops are embarrassingly parallel, they have been expressed as single TIDeFlow actors.

5.2.5 A Program Where Actors Execute Only Once

The use of the `DISCONTINUE` signal can be illustrated with the example of Figs. 14 and 15.

The actors in the program of Fig. 15 produce a *DISCONTINUE* signal when they finish execution, effectively removing the actors from the program graph. The act of

Fig. 14 Example of a program with sequential statements

```
...
x = malloc( ... );

for i in 0 to N-1
  x[i] = 0;

for i in 0 to N-1
  x[i]++;

free( x );
```

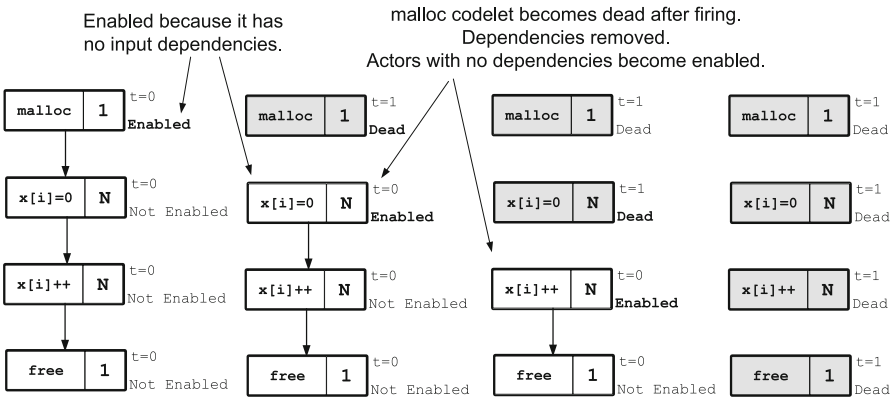

Fig. 15 TIDeFlow program graph for the the program of Fig. 14

removing some actors from a program may result in enabling other actors, as is the case in Fig. 15. The overall effect is a TIDeFlow program with an execution similar to that of a serial program.

6 Composability

To increase the programmability of a model, it is important to provide the programmer with a high degree of modularity and a layer of abstraction to avoid burdening the programmer with implementation details such as the number of available processors. One way to achieve this is through Composability that increases portability drastically (Fig. 16).

Composability allows large programs to be built using smaller programs, each one represented by an actor that is executed only once (i.e. $N = 1$) whenever it is enabled and that generates a `CONTINUE` signal after completion. The main advantage of this approach is that two actors, representing copies of the same small program as part of a larger one, do not interfere with each other during execution. This because each actor has its own local state for the lifetime of the program. However, the use of existing and small programs for the construction of larger programs excludes the possibility of having recursive calls in TIDeFlow since a program cannot be build using copies of itself.

As TIDeFlow is designed from the ground up to support a high degree of composability, the rules to execute TIDeFlow composable programs and actors are the same as those defined in Sect. 4. The rules for execution of an actor A that represents a small

Fig. 16 C interface to use TIDeFlow programs as part of larger programs

```
1 int = AddCodeletSet(
2    CodeletSet *ProgramContext,
3    CodeletSet *Program_To_Use,
4    char * Name
5    );
```

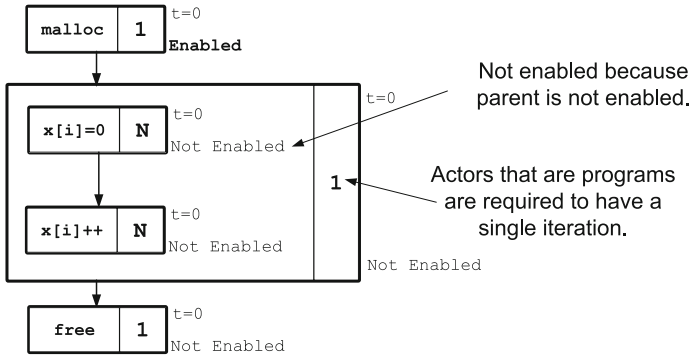

Fig. 17 Construction of a program using small TIDeFlow programs

program P are similar: The program P starts execution when the actor A fires. The actor A completes execution when the program P finishes.

The example of Fig. 17 demonstrates the composability of our model.

7 Task Pipelining

This ability to express task pipelining constructs at the program graph level is a powerful feature that can significantly simplify the optimization of HPC programs. In Garcia's optimization of the matrix multiplication program [16], it was found that a significant amount of time and effort was devoted to designing and implementing a good strategy for task pipelining. Garcia's efforts included the development of hand-made synchronization primitives and a synchronization scheme developed using previous program traces. Although effective, Garcia's approach is cumbersome, time consuming, and potentially error prone. The details of these efforts are further expanded upon in several publications [14,16–18].

In contrast, the development of a TIDeFlow program designed to compute the same matrix multiplication is drastically simplified through the use of TIDeFlow graphs. The weighted actors and the initial tokens in the backedges in TIDeFlow program graphs allow for natural pipelining of tasks during execution. Figure 12 demonstrates that, given an infinite number of processors, the computation is able to determine and utilize the optimal pipelined schedule as per Dynamic Dataflow [2]. Thus, the task of scheduling and pipelining is left to the TIDeFlow runtime system.

While the majority of task pipelining duties are left to the runtime, a programmer can still benefit from traditional techniques used in the optimization of programs. Aside from optimization of the underlying tasks of the programs, these techniques can also be used to assign priorities to tasks to further improve the effectiveness of the pipelining during execution.

Priorities are provided as a mechanism through which the programmer can identify the tasks that make up the critical path of execution. The runtime system can then schedule these tasks with a higher priority to prevent stalls wherever possible.

And, because dependencies are still enforced, the correctness of the results remains unchanged.

Figures 10 and 11 provide an example of a situation where priorities can be employed to minimize stalls during execution. The tasks that form the critical path of the program (Fig. 10) are the loader actors, which have been set to have high priority. During the execution (Fig. 11), the loaders are executed as soon as possible to enable the next set of computation tasks and to prevent stalls.

Experiments with the TIDeFlow system have shown that only two levels of priority (low and high) are enough to assist the scheduler during execution.

8 Memory Model

TIDeFlow will run correctly on any system with a Sequentially Consistent memory model. TIDeFlow will also run correctly on most modern architectures, such as x86, ARM and Cyclops-64, because their memory models allow the operations required by TIDeFlow. TIDeFlow can also run with other, weaker memory models. For a full description of what, exactly, is required from a memory model to support the TIDeFlow system, read the rest of this section.

The memory model of TIDeFlow has been designed to provide useful constructs for programmers while at the same time allowing simple practical implementations in many-core systems.

Seeking simplicity of implementation and design, the TIDeFlow model uses shared memory as the main mechanism for data communication. This decision facilitates communication between actors at the expense of the necessity of additional rules to avoid race conditions.

The following rules form the core of the TIDeFlow memory model.

Rule 0: A TIDeFlow system has shared memory. All processors have access to all the shared memory. Processors can allocate and deallocate memory for their use or for use by other processors. Global variables are allowed.

Rule 0 specifies that TIDeFlow is a *shared memory system*. And all communication is done through memory.

Rule 1: Memory operations made by a loop iteration appear to complete in program order to the processor that issued them. No ordering is guaranteed between memory operations issued by different processors.

Execution of each one of the loop iterations that compose an actor appears serial. Rule 1 supports serial execution of individual loop iterations. Rule 1 does not provide any limitations between memory accesses made by two different processors.

Rule 1 does not specify what happens when loop iterations that belong to the same actor try to access the same memory location. Rule 1 assumes that actors represent parallel loops without data races and not other kinds of loops.

Note that the TIDeFlow model does not allow data sharing between iterations that belong in the same parallel loop. Attempts to share data or to build synchronization constructs may result in undefined behavior.

Rule 2: If there is a dependency from an actor A to an actor B, and A produces one token h that is later consumed by B, then all of A's memory operations before the token h was produced will be seen as complete by B when B consumes the token h.

Rule 2 specifies that all memory operations from an actor will be seen as complete by all other actors that depend on it. Rule 2 is the main mechanism for orchestrating data sharing.

Rule 3: All memory operations in a program must have completed when the program ends.

Rule 3 supports composability. All memory operations of a program will complete once the program completes. This rule ensures that actors depending on data produced by a program (that was used as an actor) will have full access to all the memory produced by the program.

The requirements for the memory model of TIDeFlow seek to allow low-overhead runtimes because copying of data is not required between actors. However, this advantage come at the expense of having the user be responsible for the memory management.

9 TIDeFlow Implementation

The previous sections described the development of the theoretical foundations for the TIDeFlow execution model. This section focuses on the challenges to overcome in the implementation of our execution model as a runtime. In our implementation, we pursued several goals: a focus on simplicity, the capability to represent the model with fidelity, and the ability to provide high performance while remaining easy to use.

The remainder of this section describe the details of each part of the tools that form the TIDeFlow system.

9.1 TIDeFlow C Interface

A TIDeFlow program can be described as a combination of graphs and small functions (*codelets*) such as those of Fig. 6. A graph-based language [12] was considered as a solution to represent TIDeFlow graphs. However, we decided that a programming model based on the C programming language was sufficiently simple and could be developed in a feasible amount of time while still allowing enough flexibility to represent a large set of applications. In this C interface, the initialization of the runtime system and creation of TIDeFlow programs are the responsibility of the programmer. We will now discuss the initialization and execution of TIDeFlow programs.

9.1.1 Initializing the TIDeFlow Runtime System

The first step in running a TIDeFlow program is to initialize the runtime and allocate processors. The C API for this is as follows:

```
void InitRuntime( intNumProcessors );
```

The initialization process completes three key tasks. First, it allocates the specified number of processors. Second, it initializes a CB-Queue [32,33] that is modified to support and manage TIDeFlow actors. Finally, it signals all processors to begin polling the global queue to find work to execute.

9.1.2 Creation of TIDeFlow Programs

The creation of a TIDeFlow program consists of four key steps. First, a memory context must be created for the program. Next, all actors must be added to the program graph. Then, all dependencies between actors must be defined. Finally, static parameters are passed to the actors.

Creation of a Program Context A context must be created for a TIDeFlow program. A pointer to this context can be used to identify the program and also to include that program as part of larger ones. The following C interface is used:

```
CodeletSet * CreateCodeletSet( char * ProgramName );
```

Addition of Actors or Programs to a Context After the creation of a program context, actors must be added to it using the interface depicted in Fig. 18. This interface allows us to specify the function used for the actor and also the number of loop iterations the actor will execute the function. An integer is returned that serves as an identifier to the actor in conjunction with the pointer to the context.

In addition, the interface allows for the inclusion of priorities for the execution of the actors. The two levels of priorities supported for TIDeFlow (low and high) help the programmer to guide the scheduling and synchronization task based on his knowledge of the application. During runtime, high priority tasks will always be scheduled first over tasks of low priority. This allows actors in the critical path to be executed first.

Addition of Dependencies Between Actors After adding all actors in a TIDeFLow program, dependencies between them need to be specified. The integer returned in the creation of the actor is used to establish its dependencies. The interface to specify dependencies is given in Fig. 18.

Providing Static Parameters to Actors Communication between TIDeFlow actors is done through shared memory and not through tokens. For this reason, actors require

```
int AddCodelet(
   CodeletSet *ProgramContext,
   void (*function)( void *, int ),
   int LoopIterations,
   char * ActorName
   );

void SetPriority(
   CodeletSet *ProgramContext,
   int ActorID,
   int Priority /* 0: High, 1: Low */
   );
```

```
void SetDependency(
   CodeletSet *ProgramContext,
   int SourceActor,
   int DestinationActor,
   int TimeOffset,
   char *DependencyName
   );
```

Fig. 18 C interfaces to add actors and specify dependencies

```
void SetStaticData(
   CodeletSet *ProgramContext,
   int ActorID,
   uint64_t Data[8]
   );

uint64_t GetStaticData(
   void * parameters, /* Provided by the runtime */
   int DataIndex /* An integer from 0 to 7 */
   );
```

Fig. 19 C interface to set and get static data particular to an actor

pointers to memory where they can produce and consume data. Up to eight 64-bit constants can be provided to each actor using `SetStaticData` during the construction of the program. At runtime, actors can obtain any of these values through a call to `GetStaticData`. Their interfaces are provided in Fig. 19.

9.1.3 Running TIDeFlow Programs

After a program is properly set up and the runtime is initialized, a TIDeFlow program can be given to the runtime for execution. The interface is as follows:

```
void SignalSet( CodeletSet * ProgramContext );
```

9.2 Intermediate Representation

During compilation, the graph contained in a TIDeFlow program is represented automatically by the TIDeFlow toolchain as an array of structures. These structures are composed of a set of integers that make reference to actors and that describe the connections of the arcs in the program. Each actor has an associated offset in the array of structures, which is used every time the compiler needs to make a reference to it. Arcs between actors are expressed as a pair of integers that represent the actors it links.

This intermediate representation offers the possibility to change the program at different stages of compilation, to include additional compiler optimization features in the future, and to allow portability of the program to various architectures. Additionally, composability benefits from this intermediate representation since duplication of programs can be made by just copying the array of structures that represent a program.

9.3 Compilation and Execution of a TIDeFlow Program

In the last stage of compilation, the TIDeFlow toolchain translates the intermediate representation into an executable data structure capable of being executed in the target architecture. The translation uses pointers to represent the offsets of actors and their dependencies according to the array of structures in the intermediate representation. During this stage, memory is also allocated and initialized with the properties and states of the actors.

Once the executable data structure is ready, the runtime system receives a pointer to the program and starts its execution by scheduling all the actors with no dependencies at $t = 0$. During execution, the runtime system monitors constantly the number of actors to be scheduled and finishes the program as soon as this number reaches zero.

The TIDeFlow program is executed right after the translation of the intermediate representation into the executable data structure since saving this structure on a non-volatile memory is not supported at the moment. For this reason, the compiler and the launcher are merged in the same tool in the TIDeFlow toolchain.

9.4 TIDeFlow Runtime System

The TIDeFlow runtime system supports the execution of programs by providing scheduling, synchronization, initialization and termination of programs. It has been designed to support execution in an environment without virtualization. The runtime system is directly embedded in the application binary and it is able to perform all task management operations. How this is accomplished and the role of the runtime system and its relationship to the toolchain is shown in Fig. 20.

Figure 20 provides a comparison of TIDeFlow's approach to compilation and execution with that of a traditional approach. Note that TIDeFlow is able to run without the support of a traditional operating system. Instead, TIDeFlow's runtime system, embedded in the application binary, performs all task management operations.

The implementation of TIDeFlow's runtime system presented several challenges that ultimately resulted in interesting advances and tools: A fully distributed runtime system, a programming language to describe program graphs, concurrent algorithms [32], and new ways to reason about performance models [34].

9.4.1 Synchronization and Scheduling

The basic unit of execution for scheduling and execution in the runtime system is the *task*. As explained, each one of the parallel iterations of an actor is represented by a single task in the runtime system. To allow immediate visibility of available work, all tasks that become enabled are written to a queue that can be accessed concurrently by all processors.

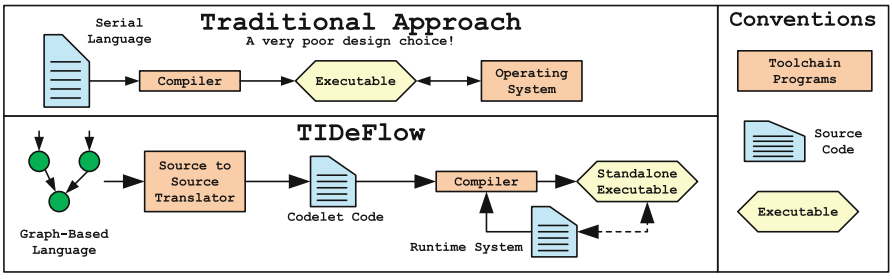

Fig. 20 TIDeFlow toolchain

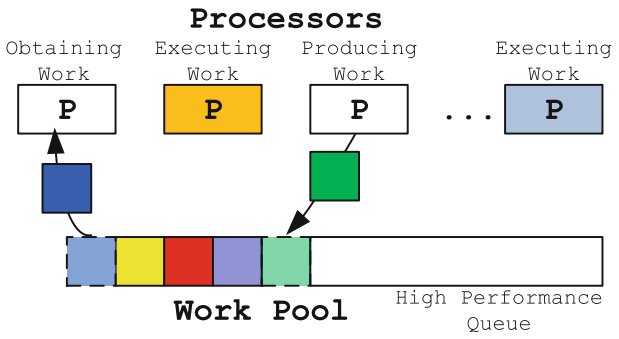

Fig. 21 TIDeFlow scheduling queue

Perhaps the most important feature of the runtime system is that its control is *fully distributed*. There is no one process, either thread or task, in charge of the runtime system duties. Instead *each* processor concurrently (1) performs its own scheduling and (2) handles localized signals related to the actor being executed by the processor, including enabling other actors and writing them to the global task queue. The TIDeFlow runtime system is fully distributed with regard to the processors, because none of them is responsible for scheduling, but it is still centralized from the point of view of the memory because the runtime system uses a single, global queue.

Development of a decentralized runtime system required advances in concurrent algorithms and in the internal representation of actors and tasks. These advances were achieved by work in concurrent algorithms for runtime systems [31] and in task representation and management [32,33]. The resulting high performance queue (Fig. 21, and described in detail by Orozco et al. [34]) was able to adequately support task management with very low overhead.

10 Parallel Program Traces

Program traces are a powerful way to provide insight into the behavior of a parallel program. Through profiling and traces, a programmer can take decisions about parallelism, priorities, resource allocation and other things.

A program trace describes the activity that each processor executed at any point of time. The information reported for each processor includes the task that the processor was executing, and in some cases, the dependence relation between tasks.

TIDeFlow provides native support to create program traces. The runtime system provides the option of logging all events and producing a report file afterwards. The events observed by each processor are assembled together to produce a program trace of the TIDeFlow program executed. The logs of events at each processor are placed into a global queue that uses the CB-Queue algorithm, where they are read at the end of the execution and dumped to a file.

Figure 22 shows an example of a program trace where the horizontal dimension represents time, the vertical dimension represents the processors, and the color rep-

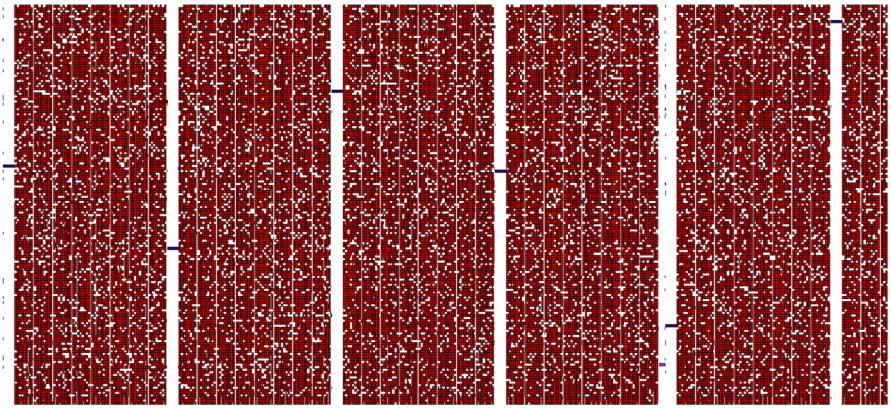

Fig. 22 Execution trace of matrix multiplication

resents tasks. This specific trace, and its associated profile, were obtained from the execution of an early version of matrix multiply.

Profiles, such as the ones in Fig. 22, are useful because they show where the time is spent in the program and allows the programmer to identify sources of overhead.

A separate visualization tool is also provided with TIDeFlow to allow easy interpretation of the profiler plots. The visualization tool can read profiler files and provides an interactive environment where the programmer can zoom into parts of the program to better analyze its behavior.

11 Experiments

In this section, we present the results of experiments designed to evaluate the usability, scalability, and performance of the TIDeFlow model.

While we believe that TIDeFlow is competitive with other parallel execution models designed with HPC in mind, such a study is beyond the subject of this paper and will be examined in a later publication. Instead, the objective of this section is to demonstrate the effectiveness of the TIDeFlow model and to evaluate its usability as a tool for the design and execution of parallel programs in many-core architectures.

11.1 Experimental Testbed

11.1.1 IBM Cyclops-64

All experiments were executed on the IBM Cyclops-64, a many core architecture with no cache and 160 non-preemptive execution units per chip, of which 156 are available to the user.

Cyclops-64 has been described extensively in previous publications [7,15,28]. We chose Cyclops-64 for our experiments because it possesses a large number of execution units that are useful to study the scalability and parallelism of HPC programs.

The TIDeFlow runtime system, associated tools, and the programs used in the experiments were all written in C and compiled with ET International's compiler with compilation flags `-O3 -g`.

As of the writing of this paper, physical Cyclops-64 chips are only available to the US government. For this reason, the experiments were executed on FAST [7], a very accurate Cyclops-64 simulator that has been demonstrated to produce results that are within 10 % of those produced by the real hardware.

11.1.2 Test Programs

We tested the TIDeFlow approach through three key benchmarks. First, we simulated the propagation of an electromagnetic wave in 1 and 2 dimensions using the FDTD algorithm (FDTD1D and FDTD2D). Next, we utilized a 13-point Reverse Time Migration (RTM) kernel. Finally, we studied an implementation of a matrix multiplication (MM) algorithm.

FDTD1D computes a problem of size 499,200 and tiles of size 800 for 3 timesteps. FDTD2D computes a problem of size 750×750 and tiles of size 25×25 for 3 timesteps. RTM was run for 8 timesteps with an input size of $132 \times 132 \times 132$ and tiles of size 6×6. The MM benchmark multiplies matrices of size 384×384, using tiles of size 6×6. For this benchmark, we utilize the tiling described in [15]. The parameters and sizes of the problems solved were chosen to be among the largest problem sizes that could be simulated.

11.2 Scalability of the Runtime System

To investigate the scalability of the TIDeFlow model, each test program was run with a wide range of active processing units. Figure 23 summarizes the experimental results. The figure reports the speedup resulting from using multiple processors when compared to an execution that used only a single execution unit. As can be seen, the FDTD benchmarks results behave almost as a straight line until around 156 processors, while the MM and RTM benchmarks are linear until 32 processors approx., becoming a flat line after that number without decrease in performance. Examination of the MM and RTM benchmarks reveal that the scalability becomes saturated around 32 processors due to the low amount of parallelism available, and not due to problems with the TIDeFlow implementation, which is able to support high levels of parallelism, as in the case of FDTD.

These results demonstrate that TIDeFlow is an excellent alternative for the computation of scientific programs using large numbers of processors.

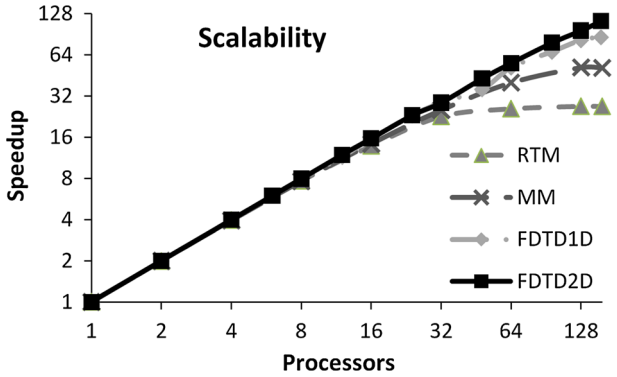

Fig. 23 Scalability of several kernels using TIDeFlow

11.3 Performance of the Runtime System

Evaluation of the performance of a runtime system must be done with care to avoid the common problem of evaluating the performance of the application and not that of the runtime system itself.

A possible approach to evaluate the performance of the runtime system will be to indirectly measure it by comparing many runs of several applications with varying execution parameters such as memory traffic, system load, task size and so on. Althouth this is a valid approach, its is indirect and extracting meaningful results from such measurements is hard.

In our case, having the source code of the runtime system, we enjoy the advantage of being able to fully instrument it, and together with access to a hardware counter in the processor that counts the number of cycles elapsed since the processor booted, we are able to exactly log the amount of time spent by the runtime system.

Our instrumentation consists of (1) one instruction to read the hardware cycle counter in the processor upon entry in the runtime system code, (2) one instruction to read the hardware cycle counter upon exit into the runtime system and (3) very few instructions to store the values into memory (in practice, with a multiple store instruction, and an atomic increment, it is possible to store both values and update a pointer with only 2 instructions). At the end of the program, the runtime system dumps the values read to be analyzed.

The data obtained from the profiler can be used to find how much time was spent inside the runtime system. Each data entry provided by the profiler is composed of the following fields: Start and end time for a task executed, a boolean value that is true when there are more tasks available than the total number of available threads, and the total time spent inside the runtime system,including idle time waiting for tasks to become available.

We can easily identify the overhead of the runtime system if we look at the profiler data for tasks that were executed when there where enough tasks available for all threads. The overhead obtained this way would be a worst-case overhead, because the scheduler will be working for the most number of threads. In all other situations, the overhead will be equal or better (lower) than the one we measured.

Inspection of the profiles in our applications has shown one case (MM) where 8580 tasks meet our criteria (the criteria is: tasks executing while there are enough tasks to keep all threads running). The average overhead introduced by the runtime system was 2 μs (1065 cycles), and, for 89 % of all tasks, the overhead was between 1.24 and 1.26 μs (620 and 630 cycles). This is a remarkably low overhead, considering that, for example, the overhead of the runtime in an OpenMP `parallel for` is greater than 100 μs (50,000 cycles) in an optimized implementation [6]. Other cases exhibit similar behavior. In RTM, our longest and most complex application, we have found that 97 % of the time, the runtime overhead was below 1.8 μs (900 cycles), and 99.65 % of the time, the overhead was below 6 μs (3000 cycles).

The speed of the runtime system can be put in perspective if it is compared to a typical task, which is 60 μs (30,000 cycles) for hightly optimized, tiled, small assembly tasks, or longer for more typical tasks writen in high programming languages. Even in the case of small, assembly-written tasks, the runtime system has a very small overhead of around 3 % or lower.

The runtime system is able to achieve a very high performance because it uses a very high performance queue algorithm [34] complemented with the polytasks technique [30] which allows compressed representation of tasks, increasing the efficiency of the runtime system.

The profiling of our runtime system shows that it has performance comparable to the hand-written assembly runtime system by Garcia [16]. Our profiling inspection also shows that, like Garcia, we were able to closely approach the peak performance of Cyclops-64.

12 Summary and Conclusions

This paper has introduced a novel parallel execution model named TIDeFlow for the design and execution of HPC programs targeting many-core architectures.

Its main features were presented: the representation of a parallel loop as a single actor using weighted nodes, the use of tokens in arcs to express loop-carried dependencies, and its fully-distributed runtime system, where each processor schedules its work.

The three major components of the TIDeFlow model have also been presented and defined: The behavior of actors, explained using Finite State Machines; its memory model; and the rules needed to describe loop-carried dependences as weighted arcs.

TIDeFlow exhibits several important characteristics for the design of efficient HPC programs: Its graph programming model allows expressing parallelism with a greater level of abstraction in comparison with traditional programming models based on synchronization primitives; its composability property is an effective way to build complex programs; its ability to express pipelining between tasks by adding tokens to arcs and the parallel nature of its runtime system that results in automatic load balancing.

The experiments, performed on a (simulated) 160-core architecture, showed that TIDeFlow presents very good scalability and very low overhead in the execution of HPC programs due to the distributed nature of its runtime system and the use of a high-throughput queue algorithms.

TIDeFlow is neither better not worse than other execution models not based in dataflow (such as OpenMP, CUDA or OpenCL). It is *different*: TIDeFlow allows controlling execution through dependencies rather than through control flow. In terms of overhead of implementation, we have shown in Sect. 11 that the overhead of most of TIDeFlow's runtime operations is lower than the overhead of OpenMP's runtime operations, however, this comparison is of limited usefulness since the intrinsic operations of TIDeFlow (such as actor scheduling and signaling) are different to the intrinsic operations of OpenMP (such as barriers).

We have searched the literature to find publications that would allow us to make a comparison between TIDeFlow and other systems, and we have found that a direct comparison is not possible because other studies do not use comparable hardware, they do not have the same intrinsic operations, or they just focus on other issues such as raw performance, memory latency, cache behavior, and other issues that do not apply go our experimental testbed. As an example, typical issues with GPU programming deal with memory bandwidth and processor utilization, not with scheduling or signaling of tasks. Typical OpenMP papers deal with absolute performance, but rarely with ease of programmability and so on. Some language extensions such as OmpSs[10] complement OpenMP to allow dependencies, but publications on it focus on overall runtime of some applications rather than on the individual cost of one runtime system operation and their experiments have been conducted on architectures that are different. As such a direct comparison will be of limited use.

13 Future Work

Future work will focus on improve resiliency to the failures that are inevitable in exascale systems. Current techniques based on global synchronization rely on global checkpoints to save the state of the processors. However, these techniques are inadequate for systems with hundreds or thousands of processors because failures in such a machine occur more often and propagate faster than global checkpoints can handle. A technique based on local checkpoints and redundant policies could be explored and introduced into TIDeFlow.

A study in energy efficiency must also be performed in order to make TIDeFlow energy-aware. The scheduling rules and the runtime system must be improved to support the design of HPC programs with a compromise between its energy consumption and its computational performance.

Additionally, we will perform a detailed comparison of TIDeFlow and more conventional GPU-based solutions. However, preliminary study suggests that the use of task queues in TIDeFlow will relieve the programmer of the burden of grouping the work in a manner designed to maximize occupancy on GPUs. Similarly, while we have primarily focused our experiments on regular applications, irregular applications make up a significant portion of scientific computing and are a highly researched problem in GPU computing. Because of the dataflow inspired nature of TIDeFlow, we are able to leverage existing work and techniques to better approach these problems [1].

Finally, future research will address the inclusion of new features in the TIDeFlow execution model such as the execution of actors with arbitrary priorities, the use of mutual exclusion constructs, and the support for recursion.

Acknowledgments This research was made possible by the generous support of the NSF through Grants CCF-0833122, CCF-0925863, CCF-0937907, CNS-0720531, and OCI-0904534.

References

1. Agrawal, G., Saltz, J.: Interprocedural data flow based optimizations for compilation of irregular problems. In: Huang, C.H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., Padua, D. (eds.) Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 1033, pp. 465–479. Springer, Berlin (1996). doi:10.1007/BFb0014218
2. Arvind, Culler, D.E.: Dataflow Architectures, pp. 225–253. Annual Reviews Inc., Palo Alto. http://portal.acm.org/citation.cfm?id=17814.17824 (1986)
3. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Foundations of Computer Science, 1994 Proceedings, 35th Annual Symposium on, pp. 356 –368 (1994). doi:10.1109/SFCS.1994.365680
4. Butenhof, D.: Programming with POSIX Threads. Addison-Wesley Professional, Boston (1997)
5. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). MIT Press, Cambridge (2007)
6. del Cuvillo, J., Zhu, W., Gao, G.: Landing openmp on cyclops-64: an efficient mapping of openmp to a many-core system-on-a-chip. In: CF '06: Proceedings of the 3rd Conference on Computing Frontiers, ACM, New York, NY, USA, pp. 41–50, (2006a). doi:10.1145/1128022.1128030
7. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Toward a software infrastructure for the cyclops-64 cellular architecture. In: High-Performance Computing in an Advanced Collaborative Environment, p. 9 (2006b). doi:10.1109/HPCS.2006.48
8. Del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.: Tiny threads: a thread virtual machine for the cyclops64 cellular architecture. In: Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, IEEE, p. 8 (2005)
9. Dennis, J.B.: First version of a data flow procedure language. In: Programming Symposium, Proceedings Colloque sur la Programmation. Springer, London, pp. 362–376 (1974). http://portal.acm.org/citation.cfm?id=647323.721501
10. Duran, A., Ayguad, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), pp. 173–193 (2011). doi:10.1142/S0129626411000151
11. Ebcioglu, K., Saraswat, V., Sarkar, V.: X10: programming for hierarchical parallelism and non-uniform data access. In: Proceedings of the International Workshop on Language Runtimes, OOPSLA (2004)
12. Ellson, J., Gansner, E., Koutsofios, L., North, S., Woodhull, G.: Graphviz—open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S.(eds.) Graph Drawing. Lecture Notes in Computer Science, vol. 2265, pp. 483–484. Springer, Berlin Heidelberg (2002). doi:10.1007/3-540-45848-4_57
13. Gao, G.R.: A pipelined code mapping scheme for static data flow computers. PhD thesis, Massachusetts Institute of Technology. http://hdl.handle.net/1721.1/37165 (1986)
14. Garcia, E., Orozco, D., Khan, R., Venetis, I., Livingston, K., Gao, G.: A dynamic schema to increase performance in many-core architectures through percolation operations. In: Proceedings of the 2013 IEEE International Conference on High Performance Computing (HiPC 2013), Bangalore. IEEE Computer Society (2013)
15. Garcia, E., Venetis, I.E., Khan, R., Gao, G.: Optimized dense matrix multiplication on a many-core architecture. In: Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010), Part II, Springer, Ischia, Italy, Lecture Notes in Computer Science, vol. 6272, pp. 316–327 (2010b)
16. Garcia, E., Venetis, I.E., Khan, R., Gao, G.R.: Optimized dense matrix multiplication on a many-core architecture. In: Euro-Par 2010-Parallel Processing, pp. 316–327 (2010c)
17. Garcia, E., Orozco, D., Khan, R., Venetis, I.E., Livingston, K., Gao, G.R.: Dynamic percolation: a case of study on the shortcomings of traditional optimization in many-core architectures. In: ACM International Conference on Computing Frontiers 2012 (CF'12) (2012a)

18. Garcia, E., Orozco, D., Pavel, R., Gao, G.: A discussion in favor of dynamic scheduling for regular applications in many-core architectures. In: Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), 2012 IEEE 26th International, IEEE, pp. 1591–1600 (2012b)
19. Garcia, E., Orozco, D., Pavel, R., Gao, G.R.: A discussion in favor of Dynamic Scheduling for regular applications in Many-core Architectures. In: Proceedings of 2012 Workshop on Multithreaded Architectures and Applications (MTAAP 2012); 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012), pp. 1591–1600. ACM, Shanghai (2012)
20. Gautier, T., Besseron, X., Pigeon, L.: Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, PASCO '07, pp. 15–23. ACM, New York, NY, USA (2007)
21. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge (1999)
22. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 319–329. ACM (1988)
23. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989). doi:10.1109/5.24143
24. Najjar, W.A., Lee, E.A., Gao, G.R.: Advances in the dataflow computational model. Parallel Comput. **25**, 1907–1929 (1999)
25. Nemawarkar, S., Gao, G.: Measurement and modeling of earth-manna multithreaded architecture. In: Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. MASCOTS '96, Proceedings of the Fourth International Workshop on, pp. 109–114 (1996). doi:10.1109/MASCOT.1996.501002
26. Gulati, K., Khatri, S.P.: GPU architecture and the CUDA programming model. In: Hardware acceleration of EDA algorithms, pp. 23–30. Springer US (2010). doi:10.1007/978-1-4419-0944-2_3
27. Orozco, D.: Tideflow: a parallel execution model for high performance computing programs. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, p. 211 (2011)
28. Orozco, D., Gao, G.: Mapping the FDTD application to many-core chip architectures. In: Parallel Processing. ICPP '09. International Conference on, pp. 309–316 (2009)
29. Orozco, D., Xue, L., Bolat, M., Li, X., Gao, G.R.: Experience of optimizing FFT on intel architectures. In: Parallel and Distributed Processing Symposium. IPDPS 2007. IEEE International, IEEE, pp. 1–8 (2007)
30. Orozco, D., Garcia, E., Gao, G.: Locality optimization of stencil applications using data dependency graphs. In: Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing, LCPC'10, pp. 77–91. Springer, Berlin (2011a)
31. Orozco, D., Garcia, E., Khan, R., Livingston, K., Gao, G.: High throughput queue algorithms. Tech. rep., CAPSL Technical Memo 103 (2011b)
32. Orozco, D., Garcia, E., Pavel, R., Khan, R., Gao, G.R.: Polytasks: a compressed task representation for hpc runtimes. In: Proceedings of the 24th International Conference on Languages and Compilers for Parallel Computing, LCPC 11 (2011c)
33. Orozco, D., Garcia, E., Pavel, R., Khan, R., Gao, G.R.: Polytasks: a compressed task representation for hpc runtimes. CAPSL Technical Memo 105 (2011d)
34. Orozco, D., Garcia, E., Khan, R., Livingston, K., Gao, G.R.: Toward high-throughput algorithms on many-core architectures. ACM Trans. Archit. Code Optim. **8**(4), 49 (2012)
35. Sarkar, V., Hennessy, J.: Partitioning parallel programs for macro-dataflow. In: Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP '86, pp. 202–211. ACM, New York, NY, USA (1986). doi:10.1145/319838.319863
36. Stone, J.E., Gohara, D., Shi, G.: Opencl: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66 (2010)
37. Theobald, K.: Earth: an efficient architecture for running threads. PhD thesis, University of Delaware (1999)
38. Yan, Y., Chatterjee, S., Orozco, D., Garcia, E., Budimlic, Z., Shirako, J., Pavel, R., Sarkar, V., Gao, G.: Hardware and software tradeoffs for task synchronization on manycore architectures. In: Proceedings of the Seventeenth International Conference on Parallel Computing (Euro-Par 2011), Bordeaux, France, Lecture Notes in Computer Science (2011)
39. Zuckerman, S., Suetterlein, J., Knauerhase, R.,Gao, G.: Using a codelet program execution model for exascale machines: position paper. In: Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, pp. 64–69. ACM (2011)