University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

TIDeFlow: The Time Iterated Dependency Flow Execution Model

Daniel Orozco[1]
Elkin Garcia[1]
Robert Pavel[1]
Rishi Khan[2]
Guang Gao[1]

CAPSL Technical Memo 107
August 15th, 2011

[1] University of Delaware and
[2] ET International
{orozco, egarcia, rspavel, ggao}@capsl.udel.edu, rishi@etinternational.com

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

The many-core revolution brought forward by recent advances in computer architecture has created immense challenges in the writing of parallel programs for High Performance Computing (HPC). Development of parallel HPC programs remains an art, and a universal doctrine for synchronization, scheduling and execution in general has not been found for many-core/multi-core architectures. These issues are exacerbated by the popularity of traditional execution models derived from the serial-programming school of thought. Previous solutions for parallel programming, such as OpenMP, MPI and similar models, require significant effort from the programmer to achieve high performance.

This paper provides an introduction to the Time Iterated Dependency Flow (TIDeFlow) model, a parallel execution model inspired by dataflow, and a description of its associated runtime system. TIDeFlow was designed for efficient development of high performance parallel programs for many-core architectures.

The TIDeFlow execution model was designed to efficiently express (1) parallel loops, (2) dependencies (data, control or other) between parallel loops and (3) to allow composability of programs.

TIDeFlow is a work in progress. This paper presents an introduction to the TIDeFlow execution model and shows examples and preliminary results to illustrate the qualities of TIDeFlow.

The main contributions of this paper are:

1. A brief description of the TIDeFlow execution model, and its programming model,
2. A description of the implementation of the TIDeFlow runtime system and its capabilities and
3. Preliminary results showing the suitability of TIDeFlow to express parallel programs in many-core architectures.

1 Introduction

The computational power of processors continues to increase every day due, in part, to the increased number of parallel execution units brought forward by the many-core revolution. Unfortunately, the increase in computational power often comes at the cost of programmers and scientists expending significant effort exploiting the increased parallelism of many-core processors.

We argue that the difficulties in exploiting the computational power of many-core processor chips are, in part, due to the inertia of popular programming models such as OpenMP [4], MPI [12] or even serial languages. Expressing parallelism in an efficient way under those programming models is difficult. This is because the programmer is forced to intermix the application code with runtime-related decisions such as: when should synchronization be done, when should threads or tasks be created, when should signals be sent or received, what are the

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.

IDs of processors, and so on. These difficulties in programming are offset by the development of special purpose constructs, (*e.g.* OpenMP pragmas for stencil computations), but the problem remains unsolved.

The dataflow models of computation [14] provide an alternative environment that is intrinsically parallel. By writing programs as graphs, the programmer can be abstracted from the duties of synchronization, scheduling and task creation, leaving the runtime system responsible to meet them.

Each one of the many parallel execution models existing were designed to target particular necessities: Static Dataflow [6] is excellent for programs without recursion, Dynamic Dataflow [1] handles recursion and has been shown to exhibit excellent parallelism, and Macro Dataflow [20] leverages on the efficiency of serial architectures.

TIDeFlow is a parallel execution model that has been designed to efficiently express and execute HPC programs. TIDeFlow leverages the following characteristics of HPC programs: (1) the abundance of parallel loops, (2) expressiveness of dependencies (data, control or other) between parallel loops and (3) composability of programs. The resulting design, explained in Section 3 introduces weighted actors to address parallel loops and weighted arcs to address concurrent execution of tasks (e.g. overlapping of communication and computation, double buffering and pipelining).

The resulting TIDeFlow parallel execution model borrows many concepts from dataflow but does not aim to be strictly dataflow. TIDeFlow provides a pact between the programmer and the system in which the programmer expresses dependencies between tasks and the runtime system enforces them. The main difference between TIDeFlow and more traditional dataflow models is that dependencies can be data, control, or other.

A runtime system that implements the TIDeFlow execution model, hereby called the TIDeFlow Runtime System, has been developed. The TIDeFlow runtime system addresses the specific features of HPC programs in many-core architectures: The presence of shared memory within one computer node, the large number of compute cores, and the intrinsic nature of programs (loop structure, overlapping of communication and computation and so on).

The TIDeFlow Runtime system is fully distributed: There is no centralized process responsible for scheduling, synchronization or task creation or termination. Instead, each processor can, in a lock-free, concurrent way, schedule its own work, perform enforcement of local synchronization signals and participate in distributed constructs such as join operations between groups of tasks.

The effectiveness of TIDeFlow for the development of HPC programs was tested with several common HPC kernels such as Matrix Multiply, simulations of electromagnetic waves propagating in 1 and 2 dimensions and Reverse Time Migration (RTM). In all cases we see that TIDeFlow allows reasonable expressibility of the programs studied, while maintaining excellent scalability.

The rest of the paper is organized as follows: Section 2 provides relevant background and a summary of related work. Section 3 formally presents the TIDeFlow execution model. Section

4 describes the implementation of TIDeFlow and its runtime system. Section 5 presents experiments showing the usability of TIDeFlow. Section 6 presents a summary of this paper along with conclusions and Section 7 presents possible future work directions.

2 Background

This section presents relevant background in dataflow execution models and their relationship to the TIDeFlow execution model. Space limitations constrain the amount of information that can be presented here to dataflow models that directly influenced the design of TIDeFlow. Further information on dataflow models can be found in the comprehensive survey on the classical dataflow execution models written by Najjar et al. [14].

Included in the survey by Najjar et al. is a description of Static Dataflow [6] and Dynamic Dataflow [1]. Static Dataflow was studied very thoroughly in the development of TIDeFlow, and it was found to have limitations to express parallel loops and to execute recursion, Dynamic Dataflow was also studied. TIDeFlow borrows the idea of using colors to simultaneously distinguish multiple tokens in the same arc from Dynamic Dataflow.

Two works that build upon the concept of dataflow are Macro Dataflow [20] and the work of Gao [9]. Macro Dataflow contributed the idea of aggregating several operations into a single actor. TIDeFlow directly builds upon the work of Gao by facilitating the efficient pipelining of operations in a dataflow graph.

However, TIDeFlow differs from traditional dataflow implementations in many ways. One such difference is the use of weighted nodes, which are not present in Dynamic or Macro Dataflow. This was inspired by a study of Petri Nets [13]. We used the concept of places and transitions to help develop the concept of the weighted node. However, it is important to note that TIDeFlow relies on shared memory to transfer data between its actors which is not necessarily the case in Petri Nets.

Other approaches are similar to TIDeFlow in their objectives but not in their implementation or style. The Kernel for Adaptive, Asynchronous Parallel and Interactive Programming (KAAPI) [11], is an execution system inspired by Dataflow, that leverages on the C++ philosophy. Cilk [3], X10 [7], Habanero C, and Habanero Java [23] use asynchronous task creation to achieve parallelism. TIDeFlow is similar to those models in that its runtime system uses queues to distribute work among processors in a shared memory environment, but it differs in the way that parallel loops are represented and executed as well as in the way programs are expressed.

EARTH [15, 22] is a hybrid dataflow system where actors are classified as functions or threads. TIDeFlow borrows the technique for signal synchronization from EARTH, but it is different from EARTH in the way loops are represented and in the addition of weights in graphs.

3 TIDeFlow Execution Model

This section presents a description of the TIDeFlow execution model.

3.1 The Basics

TIDeFlow programs are expressed as graphs of weighted actors (nodes) and weighted arcs. Both actors and arcs have *state* and associated properties.

3.1.1 Actors

The TIDeFlow execution model is based in the observation that *HPC programs are composed mostly of parallel loops.*

Actors represent parallel loops: The nature of parallel loops are expressed through actor *properties* which are constant: the number of iterations in the loop (N) and a function f that contains the code to be executed by each iteration of the loop. As in Macro Dataflow [20], actors execute a group of operations. Unlike macro dataflow, however, actors represent a parallel loop rather than a single computation.

Actors have state: The state held by actors has been designed to ease the execution and management of parallel HPC programs. The state of an actor is composed of:

- A time instance: An integer, initialized to zero that increases its value by one every time an actor successfully fires.
- An execution state: An actor can be either *not enabled*, *enabled*, *fired* (executing), or *dead.*

Figure 1 shows the state and the properties of an actor in TIDeFlow.

3.1.2 Tokens

When an actor finishes execution, it may signal other actors that depend on it by creating tokens. Tokens are tagged by the time instance of the actor that produced them. This is similar to dynamic dataflow execution models such as the MIT Tagged Token model [2] where tokens are tagged with a color that in this case is the time instance at the time the tokens were produced. Unlike the MIT Tagged Token model, tokens do not carry data, they only convey the meaning of a dependence met for a particular time instance. Data is passed between actors through shared memory. This is similar to the EARTH model of computation [22].

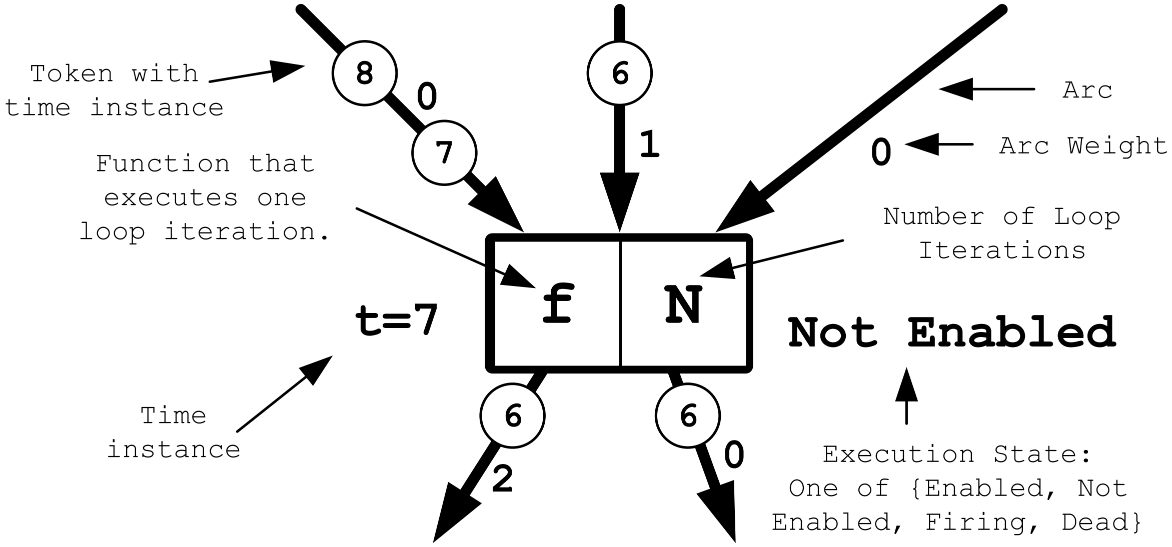

Figure 1: A TIDeFlow actor

3.1.3 Arcs

Arcs represent data, control, or other kinds of dependencies between actors.

Arcs carry tagged tokens: The time instance of each token can be used to distinguish between tokens in an arc. No two tokens contain the same time instance in the same arc because, at most, one token per time instance per arc can be produced. Arcs are allowed to carry an unbounded number of tokens, although particular implementations can restrict the number of tokens to a certain maximum.

The weight of an arc represents dependence distance: A weight of k in an arc indicates that at time instance t an actor must wait for a token tagged as $t - k$. By definition, if $t - k < 0$ the token is assumed to exist and the dependence is met. The weights on the arc allow a simple way to represent loop-carried dependencies between two actors.

3.2 Operational Semantics

This section provides a formal definition of the execution of a TIDeFlow program. The explanation is conducted by modeling the behavior of actors as Finite State Machines (FSM) (Figures 2 and 3). As it will be explained, the two main components of the actor's state, the *execution state* and the *time instance*, are independent, so a state diagram is presented for each.

3.2.1 Execution of an Actor

An actor fires (executes) when it enters the *firing* state. The effect of an actor firing is that each one of the N loop iterations represented by the actor are executed concurrently using the function f of the actor. In other words, parallel, concurrent invocations of $f(i)$, $i = 0, ..., N-1$ are called. The actor finishes execution when *all* of the parallel invocations of the loop iterations end. When the actor finishes execution it generates a termination signal and it may or may not generate output tokens.

When the actor finishes execution, the actor increases its time instance.

3.2.2 Signals Generated by Actors

An actor only generates signals when it finishes firing.

Termination Signals: When an actor finishes firing, exactly one termination signal is generated, and tokens in the output arcs may or may not be generated. The termination signals can be `CONTINUE`, `DISCONTINUE`, and `END`. `CONTINUE` indicates that the execution of the program should continue as normal, `DISCONTINUE` indicates that the actor should not be executed again, but the rest of the program should continue and `END` indicates that the actor and all of its consumers should end. The kind of signals generated is decided by the user through the return value of the function f associated with the actor. It is a requirement of the present specification that all parallel invocations of function f within the same actor and time instance return the same signal value.

Generated Tokens: The generation of tokens is subordinated to the termination signal. When `CONTINUE` is generated, exactly one token per output arc is generated and the token is tagged with the time instance with which the actor executed. `DISCONTINUE` is a termination signal that removes an actor from a graph along with its input and output dependency arcs. An actor that generates a `DISCONTINUE` signal is never fired again and it does not need to produce any tokens because the actor and all its associated arcs have been removed from the graph. At implementation, a simple flag can be used to indicate this. Finally an `END` signal does not generate any output tokens in any output arcs.

3.2.3 Actors With No Input Arcs

Conceptually, an actor becomes enabled when all its dependencies have been met. TIDeFlow takes the stance that an actor with no input dependencies has all its dependencies automatically met because it has zero pending dependencies. The result of this policy is that actors with no input arcs may fire again if they produce a `CONTINUE` signal after they fire.

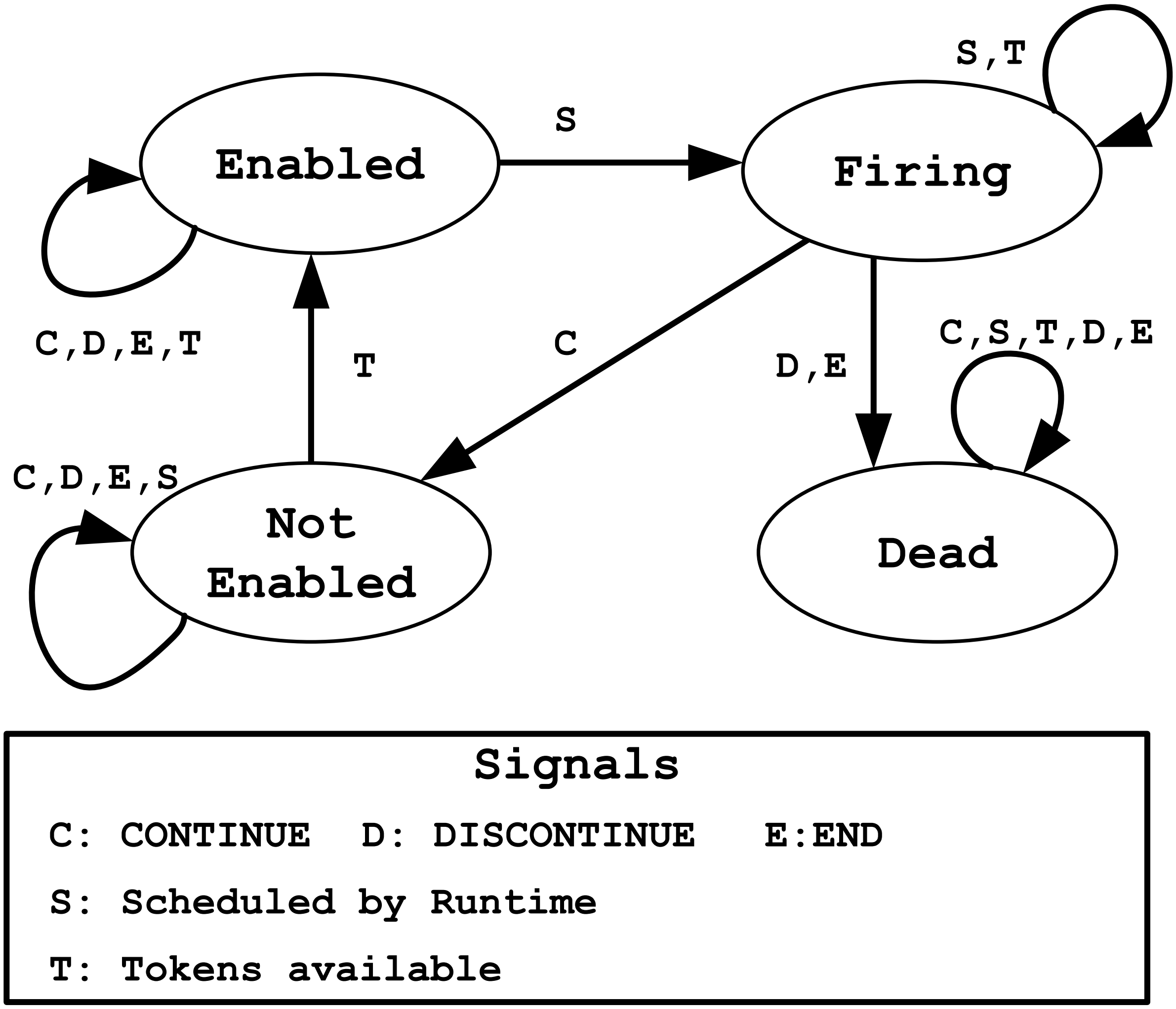

Figure 2: State Transitions for Actors

3.2.4 Token Matching

An actor can only enter the *enabled* state when (1) it is in the *not enabled* state and (2) there is one token tagged with the appropriate time instance in each arc. As explained in Section 3.1.3, the weight of the arc offsets the tag of the required token.

3.2.5 Actor Finite State Machine

The operational semantics can be summarized with the state diagram of Figures 2 and 3. The execution state of the actors is independent of the time instance of the actor. For that reason, separate FSM diagrams are given.

3.2.6 Termination of a Program

A program is defined to end when there are no actors that are firing or that are enabled.

3.3 Composability

Smaller programs can be used to build larger programs. This powerful property allows modularity and it gives the programmer useful abstractions that are not tied to implementation details such as the number of processors available.

Composability is achieved by allowing small programs to be seen as actors in the larger program that use them. In the current specification, to follow the conventions, the actor that represents the small program is defined to have only $N = 1$ iterations, because the small program is only executed once when it becomes enabled. Additionaly, by definition, small programs return a `CONTINUE` signal when they complete.

By the current specification a TIDeFlow program can only be constructed from existing programs. At this time, this design choice precludes the existence of recursive calls because a program can not be used as part of itself.

Each use of a program does not interfere in any way with uses of other programs: When a program is used as part of a larger program, it generates its own local state that exists for the duration of the program only. Other uses of the same program have their local state.

3.4 Example: FDTD1D Kernel

Figure 4 shows the kernel for a simulation of an electromagnetic wave propagating in 1 dimension using the Finite Difference Time Domain (FDTD) algorithm.

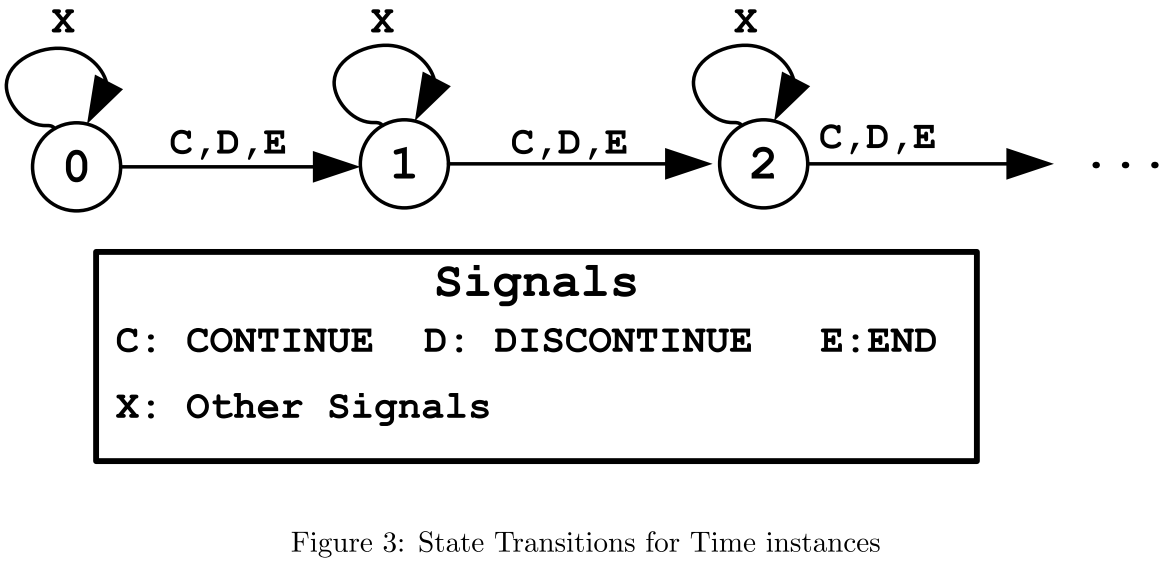

Figure 3: State Transitions for Time instances

```
int Ef( int i, int t ) {

  /* Single loop iteration */
  E(i,t+1) = k1 * E(i+1,t) + k2 * ( H(i+1,t) - H(i,t) );

  /* Termination Condition */
  if ( t == Timesteps ) { return END; }

  return CONTINUE;
}

int Hf( int i, int t ) {

  /* Single loop iteration */
  H(i,t+1) = H(i,t) + E(i,t+1) - E(i+1,t+1);

  /* Termination Condition */
  if ( t == Timesteps ) { return END; }

  return CONTINUE;
}

void EH_Kernel( void ) {
t = 0;

start:

  /* E Parallel Loop */
  parallel for i in 0 to N-1
    Signal = Ef( i, t );

  if ( Signal == END ) { return };

  /* H Parallel Loop */
  parallel for i in 0 to N-1
    Signal = Hf( i, t );

  if ( Signal == END ) { return };

  /* Time instance */
  t++;

  /* Loop to execute the kernels again */
  goto start;
}
```

Figure 4: FDTD1D Code

The serial version of the FDTD1D kernel shown in Figure 4 was written with the intention to demonstrate how to map an HPC program to a TIDeFlow program. A standard, equivalent, way to write the FDTD1D kernel can be found in [16, 21].

The equivalent FDTD1D TIDeFlow program is shown in Figure 5. The program is a direct map of the program in Figure 4.

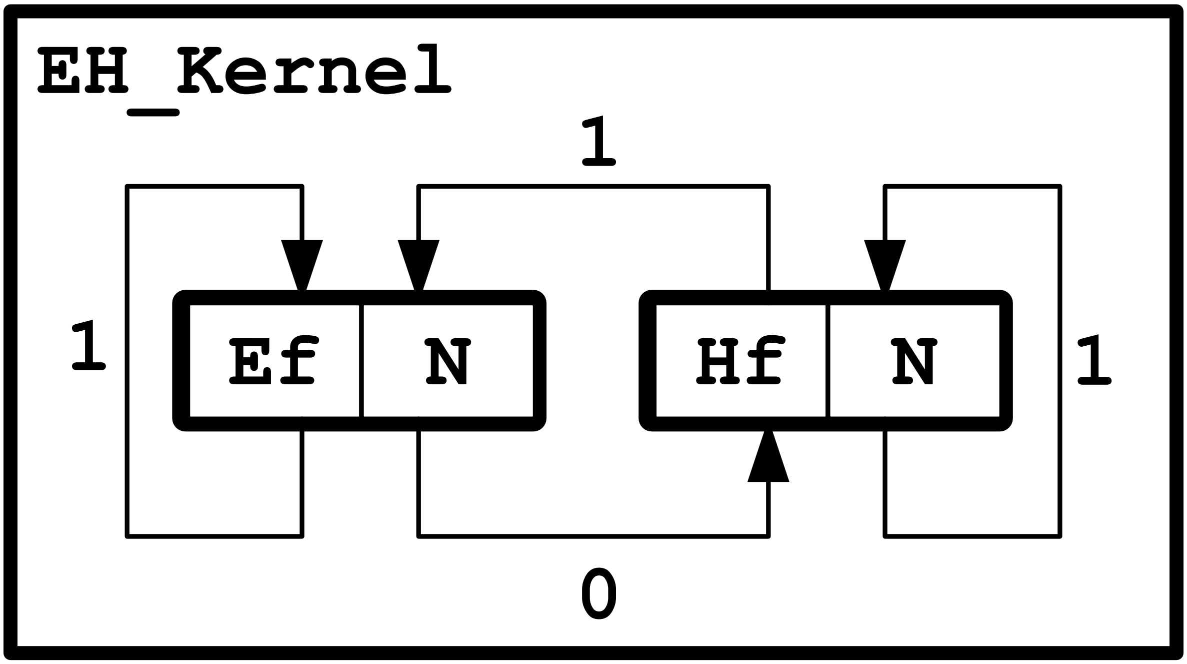

Figure 5: FDTD1D expressed in TIDeFlow

The parallel loops have become actors (`Ef` and `Hf`). The dependencies between the loops have been specified as arcs following the convention for weights in arcs: A weight of k exists between two actors if the execution of instance $t+k$ depends on an execution at time t. The `Hf` at time t needs the `Ef` at time t to have completed so the arc between them has a weight of zero. The `Ef` at time $t+1$ needs the `Hf` at time t to have completed so the arc between them has a weight of 1. In a similar fashion, `Ef` requires `E(i,t)` to compute `E(i,t+1)`, so it has a self-dependency with weight of $k=1$. A similar argument shows that there is a self dependency on the `Hf` with a weight of $k=1$.

3.5 Memory Model

This section strives to present an intuitive explanation of the TIDeFlow memory model. A formal, mathematical definition of the TIDeFlow memory model will be the subject of another publication.

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 way of actors to communicate data. This decision facilitates communication between actors at the expense of the necessity of additional rules to avoid race conditions.

The following statements represent the intuition behind the TIDeFlow memory model.

1. Execution of each one of the loop iterations that compose an actor appears serial.
2. No provisions are specified as to data sharing between loop iterations in the same actor, in the same time iteration: The model assumes that actors represent parallel loops and not other kinds of loops.
3. A dependency from an actor A to an actor B ensures that B can use data produced by A.
4. In order to use programs as part of larger programs, all memory operations in a program must have completed when the program ends.

4 Implementation of TIDeFlow

The implementation of TIDeFlow 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 [17] and new ways to reason about performance models [18].

This section presents an overview of those challenges and the products that resulted from their solution.

4.1 Programming Model

TIDeFlow programs are composed of (1) a graph, such as the graph of Figure 5 that represents the interactions between parallel loops and (2) functions that describe what computations must be done in the parallel loops, in the style of the code presented in Figure 4.

It is envisioned that a source to source compiler can obtain a graph description expressed in a simple graph format –such as the DOT language used by Graphviz [8] – and converts it to use a C interface to declare the existence of the actors and their relationship. At this point in the development of TIDeFlow, the programmer must use the C interface to declare the actors and the dependencies between them.

4.2 Intermediate Representation

Several desirable features were identified during the design cycle of the TIDeFlow runtime system, including the ability to support future compiler optimizations, or the possibility to change the program at various stages of compilation. As a result, an intermediate representation was designed to represent the program during the early stages of compilation.

The intermediate representation of TIDeFlow programs uses a small data structure to represent each actor. A program is represented by an array-of-structures that contains the actors.

The design of the intermediate representation aims to express the graph as a collection of integers. For example, whenever a reference to an actor is used, its offset in the program's array-of-structures is used rather than a pointer. This same representation is used to describe arcs: Two integers are used to represent an arc, representing the starting and ending actor of the arc.

Representing TIDeFlow programs as a collection of integers has the great advantage of allowing simple duplication of a program (for composability of programs) and portability to other architectures.

Currently, the intermediate representation is computed automatically by the TIDeFlow toolchain.

4.3 Compiling and executing a program

The final stage of compilation of a TIDeFlow program takes into account the architecture in which it runs and leaves the program resident in memory. At this point, saving a compiled program to non-volatile storage is not supported. For that reason, the compiler and the launcher of a TIDeFlow program are integrated into the same tool.

Compilation of a TIDeFlow program consists of translating the intermediate representation into an executable data structure: Offsets in the IR structures result in pointers in the final program and memory is allocated and initialized for the actors. The resulting executable data structure contains actors with their properties and their states, linked through pointers that follow the dependencies specified in the original program.

A TIDeFlow program is executed when a pointer to the program is passed to the runtime system. The runtime system scans the program and schedules all actors that have no dependencies for time instance zero. The execution continues until the runtime system detects that no more actors will be scheduled.

4.4 Runtime System

The TIDeFlow runtime system supports the execution of programs by providing scheduling, synchronization, initialization and termination of programs.

The basic unit of execution for scheduling and execution in the runtime system is the *task*. A task represents one unit of computation that must be done. For example, each one of the parallel iterations on an actor are 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.

Perhaps the most important feature of the runtime system is that it is *fully distributed*. There is no one process or 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 no one processor, thread, or task 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 our work in concurrent algorithms for runtime systems [18] and in task representation and management [19].

In our first study [18], we found that using a global queue in a decentralized system is possible if the queue is designed to sustain a high throughput of operations in a concurrent environment. Our study has resulted in a very efficient queue algorithm that can be used by processors to concurrently schedule their own work.

In our second study [17], we found that there is a high similarity in tasks from the same actor: They have the same properties and they execute the same function. Such similarity can be exploited to decrease the number of operations required to write those tasks in the centralized queue, greatly reducing the overhead of the runtime system.

Our runtime system, with the improvements developed through our work [18, 17], has resulted in a very high performance decentralized system, with overheads that are much lower than the average duration of tasks, even for fine grained programs. Section 5 shows that our runtime system is an excellent choice to support the execution of parallel programs on many-core architectures.

5 Experiments

This section presents experimental results regarding the usability of TIDeFlow. We show results on the scalability and performance of the TIDeFlow model itself.

It should be noted that presenting experimental evidence about every aspect of TIDeFlow is outside of the scope of this paper. Instead, the objective of this section is to show that TIDeFlow is *usable*, that it is reasonably efficient and that it is a reasonable choice to develop and execute HPC programs.

We believe that TIDeFlow is competitive, at least, when compared to other HPC parallel execution models. However, such a study will be the subject of another publication. The main objective of this publication is to present the TIDeFlow model and to present it as a tool for parallelism in many-core architectures.

5.1 Experimental Testbed

5.1.1 Many-Core Architecture Used

All experiments were run on Cyclops-64, a many-core architecture by IBM that features non-preemptive execution, no cache, and 160 execution units per chip, of which 156 are available to the user. Cyclops-64 has been described extensively in previous publications [10, 16, 5]. Cyclops-64 was chosen for our experiments because its large number of execution units allow excellent studies in scalability and parallelism for HPC programs.

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

Cyclops-64 processor chips are, at the moment this paper was written, available only to the US Government. For that reason, the results were produced with FAST [5], a very accurate simulator that has been shown to produce results that are within 10% of those produced by the real hardware.

5.1.2 Test Programs

The effectiveness of the TIDeFlow approach was tested using a simulation of an electromagnetic wave propagating using the FDTD algorithm in 1 and 2 dimensions (FDTD1D and FDTD2D), a 13-point Reverse Time Migration (RTM) kernel and matrix multiplication (MM).

FDTD1D computes a problem of size 499200 with 3 timesteps and tiles of size 800. FDTD2D computes a problem of size 750 by 750 with 3 timesteps and tiles of size 25 by 25. Matrix multiplication multiplies matrices of size 384 by 384, using tiles of size 6 by 6 using the tiling described in [10]. RTM was run with input size 132 by 132 by 132 with tiles of size 6 by 6 by 6 with 8 timesteps.

5.2 Scalability

Each test program was run with a varying number of active processing units to investigate the scalability of the TIDeFlow model as a whole.

Figure 6 presents a summary of the resulting experiments. The figure reports the speedup resulting from using multiple processors when compared to an execution that uses a single execution unit.

As can be seen from the figure, TIDeFlow provides good scalability for scientific programs.

5.3 Performance of the Runtime System

An analysis of the performance of a runtime system itself and not of a particular application using the runtime system is difficult because the activities of the runtime system ultimately depend on what the application requires.

A reasonable way to look into the performance of a runtime system is to look at the time it takes to execute operations. In that sense, a number of related publications by the authors have advanced techniques to optimize particular operations by the runtime.

Tasks in TIDeFlow are managed through a centralized queue that uses the CB-Queue algorithm (presented and discussed in detail in [18]). Previous research shows that the CB-Queue algorithm is an excellent choice for concurrent systems, and extensive experimental evidence of its usability and its impact as part of a runtime system were presented in one of our previous publications [18].

Task management was further improved with our polytask technique [17] that took advantage of the similarity between tasks in HPC programs. The study presented in [17] shows that polytasks allow a higher efficiency when executing fine grained programs.

An argument in [17] also shows that the efficiency of a runtime system is intrinsically tied to the program it runs. In particular, it provides an explanation saying that (1) when the granularity of a program is very fine, the intrinsic performance of the runtime has a high impact on efficiency and (2) when the granularity of a program is very coarse, the intrinsic performance of the runtime does not play a significant role in the overall execution of the program.

To finish the discussion on the performance of the runtime system we can analyze the results of one of our experiments designed to measure the overhead of individual runtime operations. In the experiment, we ran the RTM kernel again, with 156 processors and with profiling enabled

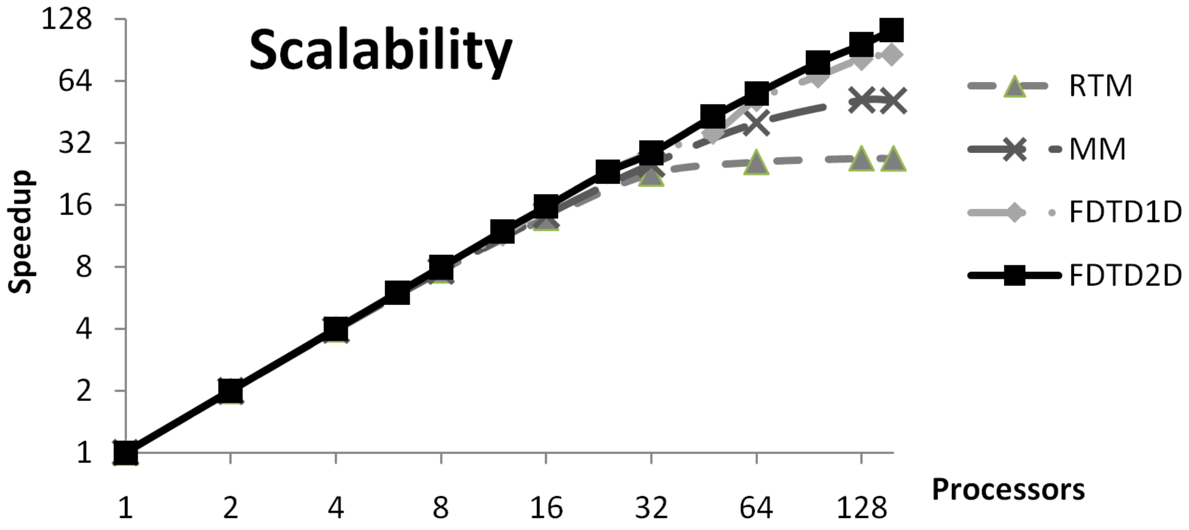

Figure 6: Scalability of several kernels using TIDeFlow

to see the amount of time taken by each runtime operation. We saw a total of 2.26×10^5 runtime operations made. Of those operations, 97% took less than 900 cycles to complete and 99.65% took less than 3000 cycles to complete. The remaining 0.25% operations corresponded to processors waiting for work to become enabled. To give a reference in the overhead of the runtime system, an average, assembly-optimized task in a program takes 30000 cycles, and a full program is in the order of 10^{10} cycles. As a conclusion, we observed that the overhead of the runtime is low when compared to the duration of individual tasks and programs.

6 Summary and Conclusions

This paper presented TIDeFlow, a parallel execution model designed to express and execute HPC programs.

The TIDeFlow model has roots in the dataflow model of execution. However, two main contributions are made to dataflow: (1) parallel loops are natively represented as a single actor using weighted nodes and (2) loop carried dependencies are represented as weighted arcs between actors.

In this paper, the TIDeFlow model has been formally defined: the operational semantics of execution have been described as Finite State Machines, a brief discussion of the memory model was presented, and the method by which to use weighted arcs to express loop carried dependencies has been described.

The experience of implementing the runtime system and executing the experiments shows that TIDeFlow is an effective way to develop and execute parallel programs. The graph programming model expresses parallelism more easily than traditional synchronization primitives, and the distributed runtime system provides very good scalability and low overhead.

In our experience developing the experiments presented in Section 5, we have found several good characteristics offered by TIDeFlow: (1) We found that it was easy to express double buffering by using the composability property to duplicate code and by using weighted arcs to indicate a time dependency between the loader stages, (2) the dependencies that we expressed through weighted arcs resulted in good task pipelining during execution, (3) we found that it was easier to express the dependencies through a graph rather than through other direct means such as conditional variables, or MPI processes and (4) load balancing was done automatically by the runtime system. For those reasons, we have concluded that TIDeFlow is a good choice for execution and development of HPC programs in many-core architectures.

7 Future Work

The TIDeFlow execution model is still a work in progress. Many aspects of TIDeFlow merit further investigation. This section presents a list of the research that will be conducted in the future.

The TIDeFlow memory model needs to be formalized and studied. This memory model then needs to be compared to existing memory models and an argument for its usability, either for execution or for development of processors, has to be made.

The properties of TIDeFlow programs must be explored. One such property is a comparison of the memory footprint of TIDeFlow programs in comparison to the memory footprint of an equivalent serial program. Other properties include, but are not limited to, well behavedness.

Additional extensions to the TIDeFlow model are being contemplated as well. Priorities of execution of actors may prove to be an interesting topic. The usability of mutual exclusion constructs could be studied. The behavior of programs with respect to composability could also be modified, for example, to allow recursion or the producing of arbitrary signals.

8 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] Arvind and D. E. Culler. *Dataflow architectures*, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[2] K. Arvind and R. S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39:300–318, March 1990.

[3] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356 –368, nov 1994.

[4] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[5] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Toward a software infrastructure for the cyclops-64 cellular architecture. In *High-Performance Computing in an Advanced Collaborative Environment, 2006.*, page 9, May 2006.

[6] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[7] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004.

[8] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz open source graph drawing tools. 2265:594–597, 2002.

[9] G. R. Gao. *A pipelined code mapping scheme for static data flow computers.* PhD thesis, Massachusetts Institute of Technology, 1986.

[10] E. Garcia, I. E. Venetis, R. Khan, and G. Gao. Optimized Dense Matrix Multiplication on a Many-Core Architecture. In *Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010), Part II*, volume 6272 of *Lecture Notes in Computer Science*, pages 316–327, Ischia, Italy, 2010. Springer.

[11] T. Gautier, X. Besseron, and L. Pigeon. 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, pages 15–23, New York, NY, USA, 2007. ACM.

[12] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, Cambridge, MA, USA, 1999.

[13] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, apr 1989.

[14] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the dataflow computational model. *Parallel Comput.*, 25:1907–1929, December 1999.

[15] S. Nemawarkar and G. Gao. Measurement and modeling of earth-manna multithreaded architecture. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1996. MASCOTS '96., Proceedings of the Fourth International Workshop on*, pages 109 –114, feb 1996.

[16] D. Orozco and G. Gao. Mapping the fdtd application to many-core chip architectures. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 309 –316, sept. 2009.

[17] D. Orozco, E. Garcia, and G. Gao. 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, pages 77–91, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] D. Orozco, E. Garcia, R. Khan, K. Livingston, and G. Gao. High throughput queue algorithms. *CAPSL Technical Memo 103*, January 2011.

[19] D. Orozco, E. Garcia, R. Pavel, R. Khan, and G. Gao. Polytasks: A compressed task representation for hpc runtimes. In *Proceedings of the 24th international conference on Languages and compilers for parallel computing*, LCPC'11.

[20] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 202–211, New York, NY, USA, 1986. ACM.

[21] A. Taflove and S. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, third edition, 2005.

[22] K. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, 1999.

[23] Y. Yan, S. Chatterjee, D. Orozco, E. Garcia, Z. Budimlic, J. Shirako, R. Pavel, V. Sarkar, and G. Gao. Hardware and Software Tradeoffs for Task Synchronization on Manycore Architectures. In *Proceedings of the Seventeenth International Conference on Parallel Computing (Euro-Par 2011). To appear.*, Lecture Notes in Computer Science, Bordeaux, France, 2011.