Supporting Stateful Tasks in a Dataflow Graph

Vladimir Gajinov[*†]1 Srdjan Stipić[*†]1 Osman S. Unsal[†]1
Tim Harris[†]2 Eduard Ayguadé[*†]1 Adrián Cristal[§‡]1

* Universitat Politecnica de Catalunya, †Barcelona Supercomputing Center, † Microsoft Research Cambridge,
§ Artificial Intelligence Research Institute, ‡ Spanish National Research Council
1 {name}.{surname}@bsc.es, 2 tharris@microsoft.com

ABSTRACT

This paper introduces Atomic Dataflow Model (ADF) - a programming model for shared-memory systems that combines aspects of dataflow programming with the use of explicitly mutable state. The model provides language constructs that allow a programmer to delineate a program into a set of tasks and to explicitly define input data for each task. This information is conveyed to the ADF runtime system which constructs the task dependency graph and builds the necessary infrastructure for dataflow execution. However, the key aspect of the proposed model is that it does not require the programmer to specify all of the task's dependencies explicitly, but only those that imply logical ordering between tasks. The ADF model manages the remainder of inter-task dependencies automatically, by executing the body of the task within an implicit memory transaction. This provides an easy-to-program optimistic concurrency substrate and enables a task to safely share data with other concurrent tasks. In this paper, we describe the ADF model and show how it can increase the programmability of shared memory systems.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming.

Keywords

Dataflow, Transactional Memory, Parallelization.

1. INTRODUCTION

Current multicore processors are for the most part built on top of the shared memory architecture. However, shared memory programming can be quite difficult. The programmer has to reason about not one, but many threads of execution that concurrently access potentially shared data. Often, the costs of data access synchronization prevent a shared-memory program performance from scaling to high core counts. Moreover, solutions to this problem require significant programmer effort which defeats the objective to exert parallel programming to the general programming community.

Lately, there has been a resurgence of interest into the dataflow model (Perez, Badia, & Labarta, 2008; Diavastos, Trancoso, Lujan, & Watson, 2011) due to its ability to efficiently exploit parallelism. The main characteristic of the dataflow model is that the execution of an operation is constrained only by the availability of its input data. Following a single assignment rule the dataflow model is able to extract all the parallelism inherent in the program.

We argue that dataflow and shared memory programming can be integrated into a more versatile programming model which can provide a user with more expressive concurrency abstractions that are sufficiently easy to reason about. Accordingly, this paper introduces Atomic Dataflow Model (ADF) - a programming model for shared-memory systems that combines aspects of dataflow programming with the use of explicitly mutable state. The key idea is that computation is triggered by the flow of data between ADF components (tasks) but that, within a component, execution occurs by making atomic updates to the common mutable state. Following the algorithm, the programmer delineates a program into a set of tasks and defines explicit dependencies between them. The ADF model manages implicit inter-task dependencies automatically, by executing each task transactionally, which guarantees atomic shared memory updates from within the task. This leads to a flexible form of parallelism in which computation can be carried out in a pure dataflow fashion, in a common task-based style of shared memory programming, or in a combination of these two methods. Therefore, compared to each of the aforementioned techniques alone, the ADF model can represent a wider range of problems naturally.

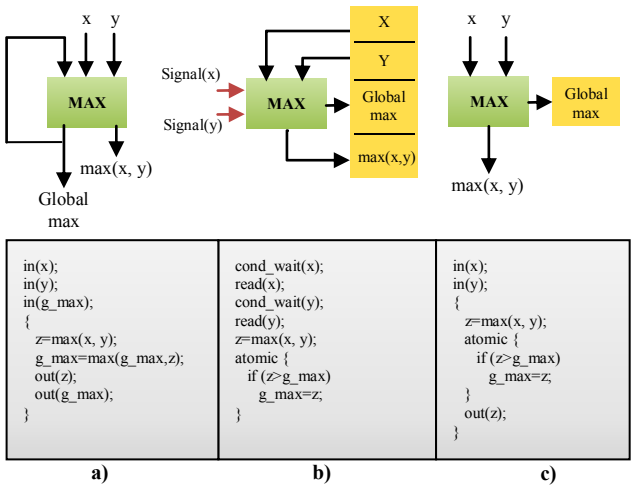

Figure 1: Max function with the global maximum: a) dataflow, b) shared memory and c) ADF implementation.

To illustrate the advantages of the ADF model let us consider a simple example from Figure 1. Given two input streams of integers, *x* and *y*, the goal is to discover which of the current values of these two streams is larger, but we would also like to find the global maximum of all values received on both streams. Dataflow model handles the input streams naturally, but in order to maintain the global maximum the state of the previous execution has to be provided as an input for the next execution - Figure 1a. This means that the state has to be copied after each invocation of the MAX node. On the contrary, shared memory model treats the global maximum as the mutable state which can be updated in-place - Figure 1b. Still, we need to synchronize access to this data in order to allow safe updates by concurrent threads. The issue with the shared memory implementation is how to deal with input data streams. A typical solution is to rely on some form of conditional waiting, i.e. conditional variables, transactional memory retry mechanism or message channels through which we implement coordination between different threads of execution. Ultimately, the ADF model, being the combination of the previous two models, can handle both of the above

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.
ACM 978-1-4503-1182-3/12/09.

problems naturally. Figure 1c illustrates the idea: we handle the streams and find the current maximum in a dataflow fashion, while the global maximum is treated as a mutable state.

2. THE ADF MODEL DESCRIPTION

Atomic dataflow model is based on execution of tasks which are scheduled according to the dataflow principles, when their input dependencies are satisfied. As such, the ADF task operates as a macro dataflow actor which processes one set of data, produces output data and then waits for the new matching set of input data.

Every ADF program starts with the initialization of the ADF runtime system. This also creates a pool of worker threads whose function is to execute enabled dataflow tasks. Then, the main thread creates all ADF tasks defined by a programmer and generates initial data values which are necessary to start the new dataflow execution, after which it starts the dataflow execution using the following ADF pragma directive:

```
#pragma adf_start
```

The main thread waits for the end of the dataflow execution, which is entirely self-scheduled, using the second ADF directive:

```
#pragma adf_taskwait
```

The basic building blocks in the ADF model are ADF tasks. The ADF task is characterized partly with explicitly defined input data dependency and partly with a mutable state that it can access. The ADF runtime system uses data dependency information to construct task graph that governs the execution of program tasks. In addition, the ADF task can safely access and update the mutable common state since the task body is enclosed in an implicit memory transaction which guarantees the atomicity of the task execution.

A task is defined using the *adf_task* pragma directive with the following syntax:

```
#pragma adf_task  [ trigger_set (<set>) ]
                  [ until (exit_condition) ]
                  [ relaxed ]
                  {  < task_body >   }
```

Basically, a programmer uses *adf_task* directive to transform a sequential program into the parallel form. For example, the following code fragment shows the implementation of the function from Figure 1 which uses this directive.

```
#pragma adf_task  trigger_set(x,y) until(done)
{
    z = max(x,y);
    if (z > g_max)
        g_max = z;
}
```

The *trigger_set* clause is used to define data dependencies for the task. Each data from the trigger set is associated with an implicit buffer which stores different values of the data item and enables asynchronous execution of producer and consumer tasks.

The ADF runtime system schedules tasks based on the availability of the trigger set data. A consumer task blocks if its input dependencies are not satisfied and waits for the producer task to commit new data values. When the input is ready, a consumer ADF task atomically processes the data, commits the changes and returns to the beginning of the task to wait for new data. During commit, the consumer task may produce the data needed by some other ADF task, which then processes the data further. Since each

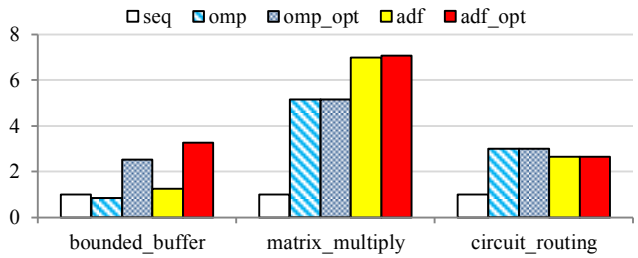

Figure 2: Speedup of parallel implementations.

ADF task is triggered only when a new data value is produced, a program is thus executed in a dataflow fashion. An important aspect of the ADF model is that it separates the transactional execution from the dataflow coordination. In case of abort as a result of conflicting changes to the shared mutable state, the task restarts a transaction using the same set of input values.

Usually, dataflow execution stops once it achieves the goal of the calculation; e.g. when the product of two matrices has been found. However, there are cases when the execution should proceed until some external signal is received. For example, a game server runs indefinitely processing clients' requests and terminates only when it is explicitly shutdown. In such case, the exit condition for a task can be defined using the *until* clause.

Finally, using the *relaxed* option it is possible to switch off the implicit TM synchronization of the task body. The implicit transaction guarantees the atomicity of the task execution in regard to the shared memory operations, but in some cases a programmer can provide more optimal synchronization by inserting transactions manually within the body of the task.

3. CONCLUSION

In Figure 2, we compare the performance of benchmark applications parallelized using the ADF model against the performance of the corresponding sequential and task-based OpenMP implementations. We implement two versions of parallel applications: one with implicit task transactions and the other with optimally placed transactions. The results show that added expressiveness of the ADF model also results in improved performance. Our ongoing experiments suggest that the model is suitable for parallelization of the Producer-Consumer type of applications as well as for applications that exhibit irregular parallelism. In the future, we plan to evaluate the ADF model using a wide range of applications and experiment with different implementations of the ADF runtime system.

4. ACKNOWLEDGMENTS

We thankfully acknowledge the support of Microsoft Research though the BSC-Microsoft Research Centre, the European Commission through the HiPEAC-3 Network of Excellence, the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980).

5. REFERENCES

Diavastos, A., Trancoso, P., Lujan, M., & Watson, I. (2011). Integrating Transactions into the Data-Driven Multi-threading Model using the TFlux Platform. Galveston Island, TX: *The first workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM).*

Perez, J. M., Badia, R. M., & Labarta, J. (2008). A dependency-aware task-based programming environment for multi-core architectures. *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, (pp. 142-151).