Dynamic Dataflow

Position Paper

Dominic Duggan
Stevens Institute of Technology
dduggan@stevens.edu

Jianhua Yao
Stevens Institute of Technology
jyao1@stevens.edu

Abstract

Dataflow networks have gained renewed attention in recent years, with application in various forms of stream processing. This position paper considers motivation for an approach to supporting the dynamic update of portions of a dataflow network, in the context of static scheduling of such networks. A semantics for dynamic dataflow graph update ensures that correctness of a network execution is preserved under such dynamic updates, where static scheduling places a bound on the amount of buffering required in the execution.

1. Introduction

Dataflow or stream processing is becoming increasingly important, with the growing prevalence of signal, video and audio processing, particularly on mobile devices. Dataflow processing is a good match with multicore and GPGPU parallel architectures that are now prevalent on desktop computers, and will shortly be available on consumer mobile devices. The data parallelism of such architectures is at least potentially a good match with the demands of stream processing applications.

A plausible scenario for long-lived parallel stream processing is the need to be able to update these applications "on the fly." Although much signal processing is now performed in hardware, the clear trend is towards more and more of this to be done in software. A key motivation is application flexibility, e.g., the need to be able to adapt to new signal processing algorithms and protocols. Another motivation provided by parallelization is that it may be useful to be able to reorganize dataflow nets, based on profiling data available at run-time that may reflect characteristics of the input data as well as contention for resources from other applications on the same device.

We have developed a semantics for dataflow computations with dynamic module update. Our starting point is a computational model similar to that originally proposed by Kahn [2]. This provides for a network of sequential *actors*, each implemented in a conventional sequential language such as C or Algol. Actors are connected by communication buffers on which they can send and receive data. A key point is that actors cannot nondeterministically select among inputs on several input channels, nor can they test input channels for available inputs (so polling cannot be implemented). This restricts each actor to a completely deterministic semantics. The combination of implicit parallelism and deterministic execution makes dataflow computation a good fit with some of the current thinking of how best to successfully exploit the parallelism available in modern multicore and GPGPU architectures, in those domains where the dataflow paradigm is applicable.

Kahn process networks have not seen widespread use in application development. One of the issues is that, because of the restriction to deterministic semantics, buffers may grow without bound. For example, if an actor is performing the merge of two input streams, and data is consistently arriving faster on one input channel than on another, then the input buffer for the faster producer will grow as data backs up there. It is tempting to relax the determinism restriction, but this must be done carefully, if the result is to have a well understood semantics that the application developer can use to reason about program execution. For example, early semantics of nondeterminism for Kahn networks were not continuous.

In the embedded systems and digital signal processing community, a very useful class of restricted Kahn networks has been identified, the so-called *synchronous dataflow (SDF)* [3] networks. SDF networks enable static scheduling for multi-rate applications. More recently, new domain-specific languages such as Streamit [4] have been defined, based on the principles of SDF, but also providing support for compiling programmer code to run on modern parallel architectures.

Fig. 1(a) considers a three-stage dataflow graph, where we assume that the underlying program is to executed on a quad-core processor. The first stage involves twelve computation steps that can be parallelized into four parallel processes, each executing three steps. There are two intermediate stages that can be executed in parallel, each involving two computation steps. Finally the third stage involves twenty computation steps, parallelized into five steps performed by four actors executing in parallel.

Fig. 1(b) considers a variation on this scenario, where the first stage is parallelized into three actors instead of four, and the two intermediate stage are executed on the fourth processor. Although the intermediate stages depend on the results from the first stage, they can be executed in parallel with that first stage by pipelining the dataflow execution, processing the second stage of a cycle of the execution of the dataflow graph in parallel with the first stage of the next cycle of the execution of the dataflow graph.

Fig. 1(c) considers another variation, this time where the graph is being executed on two processors instead of four. This may be necessary as a power-saving measure, for example, shutting down some of the processors to save battery life. An example motivation for dynamic update is to change the dataflow graph to accomodate these kinds of runtime changes, while ensuring that the interfaces between sections of the dataflow graph continue to be supported by their replacements.

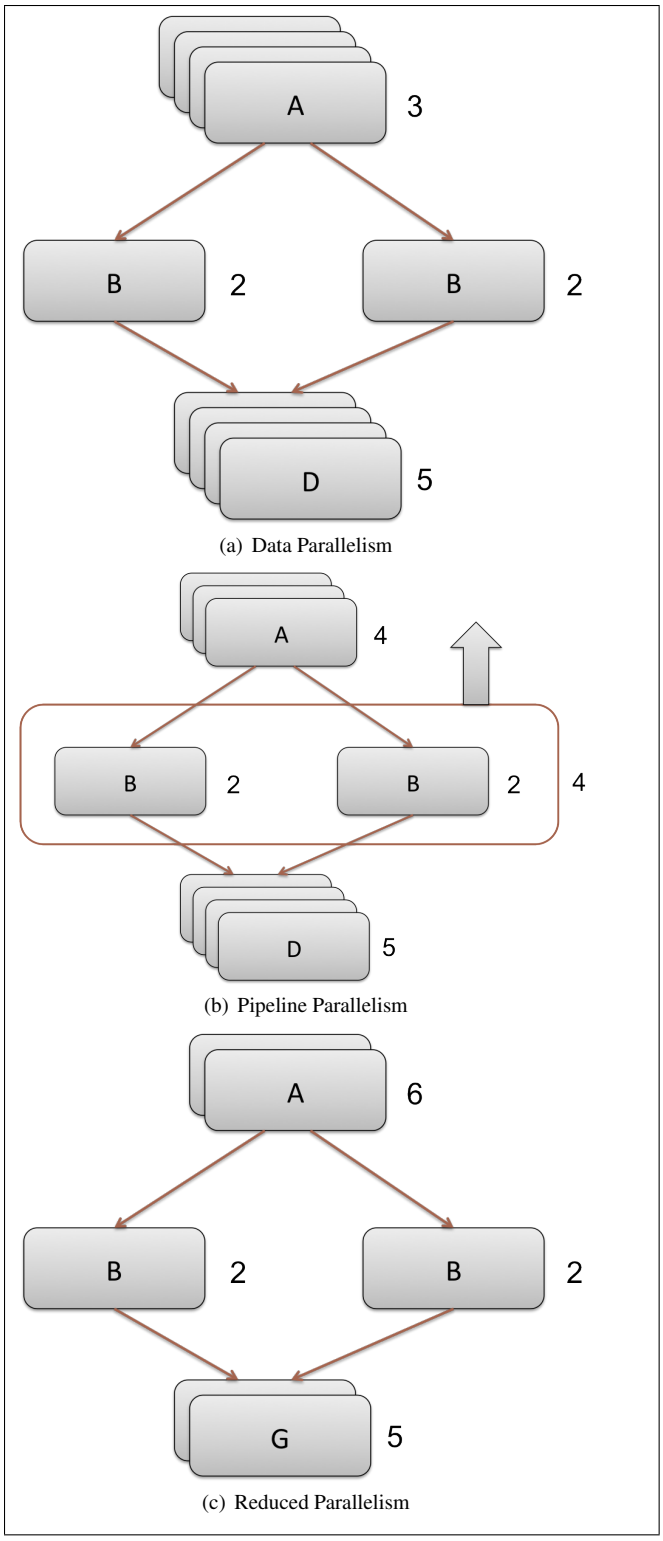

Figure 1. Pipeline and Data Parallelism

Our goal is to provide a type system that can describe points in the execution of a dataflow graph where parts of the graph can be executed. For example, we wish to be able to support the safe transition of a dataflow graph from that in Fig. 1(b) to that in Fig. 1(c). One strategy is simply wait for the termination of the execution of the dataflow graph, and start the next execution of the dataflow graph under the modified configuration. However we may not have the luxury of being able to wait, for example if power consumption needs to be reduced immediately.

Our strategy is to allow applications to set breakpoints in the execution of the dataflow graph. At one such breakpoint, an actor may accept a new code update, an actor body that is activated and replaces the running actor, binding to the communication channels that the original actor was communicating on. The replacement actor has the responsibility of continuing to support the external interface provided by the original actor. We enforce this using the notion of *flowstate*, a type-level mechanism for checking that dataflow actors provide a given firing behavior, described in the next section.

One simplifying assumption might be to restrict such updates to each iteration of the top-level loop of an actor, but the example in Fig. 1 demonstrates that the definition of the top-level code for an actor is a malleable notion. For example, even for a single-threaded atomic actor, loop unrolling will duplicate the top-level, and preserving the original update behavior will require duplicating a top-level update point for each loop unrolling. Our approach is not to place any restriction on where such an update point occurs in an actor, only that the update must preserve the interface of the original actor.

We use the term "actor" to refer not just to a single atomic dataflow actor, but also to an arbitrarily complicated dataflow graph resulting from connecting several actors together, using a compositional notion of dataflow [1]. In general, an actor will offer data channels on which it sends and receives data. In addition in our model it offers several points at which the actor may be updated. Each update point has a flowstate reflecting the constraints on the replacement actor (which may be atomic or composite). Data channels are internalized as actors are linked together, and cannot be elided. Update channels remain available on a "control plane" to controller actors that trigger updates. Unlike data channels, update channels can be elided from actor interfaces, reflecting a willingness to forego the ability to perform some updates.

As the example in Fig. 1 demonstrates, in general it is not sufficient to simply replace a single actor. Rather we may wish to replace a section of a dataflow graph in parallel. For example, in the graph in Fig. 1(b), we may want to set update points at the intermediate points of execution in the three "A" agents, and between the executions of the "B" and "C" agents. The general rule is that a set of update points in an dataflow graph should be a "consistent cut," with in particular no dependencies between the update points. Updates then become a form of barrier synchronization: a collection of updates is offered in parallel on a consistent cut of update points in a dataflow graph. The actors executing in the graph block at an update point if an update has been offered, and the update is finally executed when all the affected actors have reached their update points.

References

[1] D. Duggan and J. Yao. Static sessional dataflow. In *European Conference on Object-Oriented Programming (ECOOP)*, Beijing, June 2012.

[2] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of the IFIP Congress*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[3] E. Lee and D. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75 (9):1235–1245, September 1987.

[4] W. Thies. *Language and Compiler Support for Stream Programs*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2009.