

## Reduce parallel programming pain with dataflow languages

By Casey Weltzin

[EE Times](#)

(06/28/10, 12:01:00 AM EDT)

Whether modifying an existing application or writing entirely new code, parallel applications can be much more challenging to work with than their sequential counterparts.

Without a doubt, the high-level abstractions and APIs commonly used today greatly simplify the process, but most methods still require manual identification of parallel code sections, as well as consideration of such issues as race conditions and synchronization among parallel tasks. And as the number of CPU cores on a single chip grows, the corresponding applications that take advantage of this hardware will be even more likely to result in parallel programming pain.

A class of programming languages based on a principle called dataflow not only can greatly simplify the process of developing code for multicore processors today but also could become a key strategy for leveraging the many-core CPUs of the future.

### Sequential searches

In seeking solutions to the parallel programming challenge, it is helpful first to consider how the current-day mismatch between programming languages and parallel processor architectures came about.

As processor hardware has evolved, embedded engineers and computer scientists have customarily written programs that directly relate to how that hardware is structured. At the most basic level, this concept is highlighted in assembly language, where processor registers themselves are directly manipulated.

Modern programming languages have grown in their level of abstraction by offering task-based APIs and parallel structures in addition to manual thread operations. But one aspect of embedded programming has remained largely unchanged:

Programming is done in a sequential line-by-line fashion to mimic the sequential behavior of single-threaded execution on most microprocessors.

This is notably different from the flow-chart type of approach that many engineers pursue during the brainstorming process for an application. Instead of immediately concentrating on a set of sequential steps as mandated by CPU architecture, a flow-chart approach lets programmers more directly address the problems they are trying to solve by focusing on the algorithms needed to manipulate data, as well as the dependencies between those algorithms.

In addition, flow charts are a natural choice for expressing parallel processes (those that do not have data dependencies) in a way that is visually intuitive.

Traditionally, programmers have been required to translate flow-chart components into sequential statements before implementation on a CPU—a task that can be both time-consuming and especially difficult for parallel tasks. Decades of R&D, however, have yielded programming languages that can directly encode both function and data dependency information, rendering the translation step unnecessary. Referred to as dataflow languages, they have one important benefit that cannot be ignored: the ability to identify parallel sections of code automatically and execute them on multicore processors.

In short, traditional programming languages require programmers to adapt their ideas to a sequential execution model and manually identify parallel sections of code. Dataflow solutions are available today that use intelligent compilers to detect parallelism (**Figure 2**) and then choose how to best schedule sequential CPU instructions. That's a game changer in that it lets developers concentrate on the problem at hand, rather than on the underlying hardware.

Although dataflow applications can be written in text (one common example being VHDL), dataflow source code is often expressed graphically to most clearly convey data relationships and parallelism to programmers.

Consider the source code for a basic arithmetic operation,

$$\text{Result} = ((B * C) + A) / (D - E),$$

written using the National Instruments LabView dataflow language show in **Figure 1**.

This graphical dataflow representation has two significant features compared with sequential languages. First, intuitive visualization of parallel operations lets programmers quickly identify how program elements relate to each other. Examining the diagram in **Figure 1**, it is apparent that the add and multiply nodes can potentially be run at the same time as the subtract node.

Second, an intelligent compiler can make the same observation and divide sections of the source code diagram into independent pieces that can be assigned to threads for execution on a multicore processor.

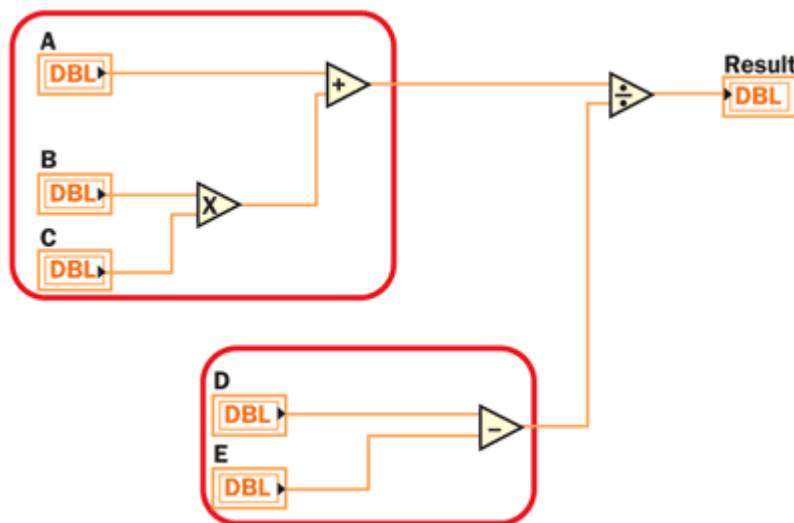
This is not an academic exercise; such dataflow technology is commercially available today, and a growing number of languages are adding dataflow extensions to give developers access to dataflow's parallel programming benefits.

The arithmetic example shown in **Figure 1** is simplified for explanation purposes. In reality, dataflow source code may feature signal processing algorithms, library calls, and most other structures and operations that are commonly found in any modern programming language.

Whether creating a small program with two parallel paths or a large application with a hundred, however, the inherent parallelism of dataflow programming languages can be used to take advantage of the multicore CPU hardware available.

Compilers' ability to map dataflow code automatically to parallel processors, however, does not remove all responsibility and control from the developer. Fundamental parallel programming techniques such as task parallelism, data parallelism and pipelining can be implemented naturally in dataflow code to optimize application performance. Manual threading and synchronization may also be supported, depending on the specific dataflow programming tool.

**Figure 3** shows an example of a pipelining implementation in a dataflow language. Here, pipelining is represented using unit delay elements (arrows) that store the result of a computation and pass it to another node in a subsequent loop iteration. The gray border surrounding the diagram elements represents a while loop.



**Figure 1. Sample dataflow diagram illustrating a basic arithmetic operation. Note that the add, multiply and subtract nodes do not depend on each other and can be executed in parallel.**

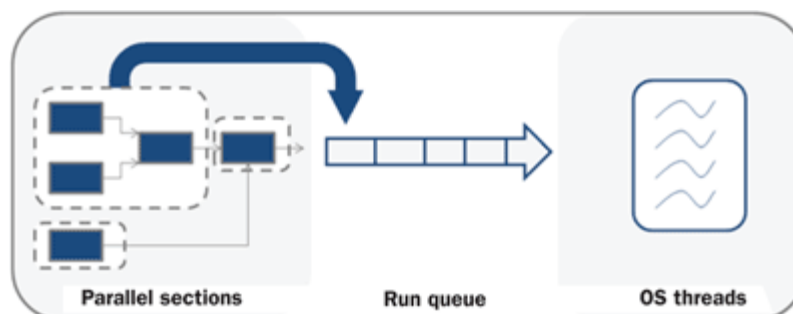


Figure 2. Dataflow compilers can automatically identify parallel sections of code to be scheduled for execution on multicore CPUs at runtime.

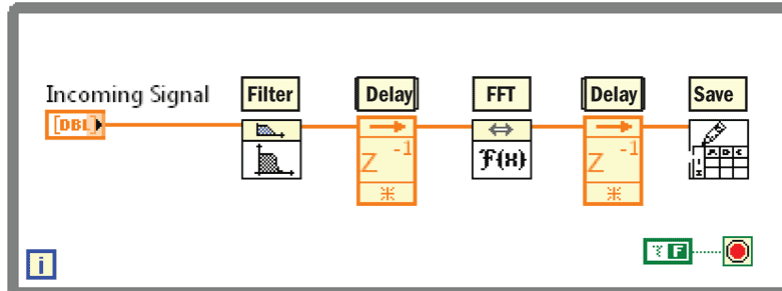


Figure 3. Using parallel programming patterns such as pipelining (shown here), task parallelism and data parallelism can help optimize dataflow applications for execution on multicore CPUs.

[Click on image to enlarge.](#)

As with all technologies, there are also some drawbacks associated with dataflow languages. First and foremost, because the dataflow paradigm differs significantly from traditional sequential languages, programmers accustomed to the sequential languages must overcome a learning curve to make the transition.

To lower the barriers associated with that transition, dataflow languages like LabView incorporate different ways to import existing C code and even .m file scripts so that a large amount of existing code can be reused.

Additionally, dataflow languages rely on the absence of by-reference memory accesses to map parallel code sections automatically to multicore CPUs; only by-value accesses are typically allowed. While that lets programmers take advantage of parallel hardware with minimal effort, it also has the potential to increase the memory footprint of applications.

Advanced dataflow compilers attempt to mitigate the impact on footprint effect by using various techniques that minimize the number of memory locations used.

For many individuals, though, the similarities between dataflow languages and familiar flow charts make them easier to learn and use than conventional languages and tools.

### Looking ahead

A good way to think about the languages that will be used to program tomorrow's many-core CPUs is to consider one of the most parallel hardware devices in widespread use today: the field-programmable gate array. Hardware description languages such as VHDL and Verilog are commonly used to program FPGAs, and they rely on developers' specifying both logic algorithms and data dependencies—an example of dataflow.

As CPUs continue to evolve, both organizations and individual programmers should look carefully at dataflow as a potential solution to the multicore programming challenge. Regardless of the exact form assumed by the next generation of parallel CPUs, one thing is certain: Developers will need to continue looking for new approaches, such as dataflow, to take advantage of these CPUs and ease their parallel programming pain.



*Casey Weltzin is a product manager for real-time solutions at National Instruments. He holds a degree in electrical engineering from the University of Wisconsin-Madison.*

Please [login or register here](#) to post a comment or to get an email when other comments are made on this article