Separation of Concerns in Scientific Programming

Catherine Olschanowsky

Efficient applications minimize the volume of data moved and avoid complex interactions with memory, however, designing an application around these goals obfuscates the primary computation. The obfuscation frustrates maintainability and testing challenges. Usable design and execution efficiency are simultaneously achieved when code design includes a specific kind of separation of concerns. This document provides a brief definition of the design pattern and describes a class of optimizations possible.

Separation of concerns generally refers to breaking an application into sections that address specific concerns (modular programming), but in this context, it has a different use. The concerns we propose separating are the primary computation, the execution schedule, and the storage mapping. The *primary computation* is the algorithm or mathematics being performed; the order the expressions composing the primary computation are executed is the *execution schedule*; and the *storage mapping* is how the values required and written by the computation are stored in memory. Each can be abstracted from the application using a variety of methods.

Loop transformations reschedule the execution of an application. Memory traffic can be reduced and simplified through transformations and vectorization depends heavily on loop transformations. The loop transformation done by general purpose compiler are limited due to the generality of the heuristics used to make choices. When the computation, execution schedule, and storage mappings are abstracted, source-to-source optimizing tools have more transformation choices.

Many scientific and data-intensive applications will perform computations as a series of operations. The operations, especially the more complex ones, are often within a deep loop nest. Memory is required to save the output of one operation for consumption by the next. Loop transformations can reduce the distance between producer and consumer and in the process, reduce the need for temporary storage in memory, reducing traffic.

Scientific applications often use embedded domain specific languages (eDSLS) that achieve separation. As an example, consider the Parflow application. It uses an eDSL that abstracts the execution schedule. Rather than writing code that manually traverses the execution domain, programmers choose from a predefined set of macros that traverse the specific portions of the domain. This greatly simplifies the appearance of the code, but importantly, it also provides a hook for optimization.

Combining eDSLs with optimizing transformation tools will increase application performance and energy-efficiency. The applications maintain a modular design and the optimizations are kept outside the source code. However, each project cannot develop these tools in an ad-hoc manner. A language agnostic transformation tool should be applied to multiple well coordinated eDSLs serving multiple applications.