Scientific Programming 16 (2008) 275–276
DOI 10.3233/SPR-2008-0277
IOS Press

Editorial

Complexity in scalable computing

Damian W.I. Rouson
Scalable Computing Research and Development Department, Sandia National Laboratories, Livermore, CA, USA

The rich history of scalable computing research owes much to a rapid rise in computing platform scale in terms of size and speed. As platforms evolve, so must algorithms and the software expressions of those algorithms. Unbridled growth in scale inevitably leads to complexity. This special issue grapples with two facets of this complexity: scalable execution and scalable development. The former results from efficient programming of novel hardware with increasing numbers of processing units (e.g., cores, processors, threads or processes). The latter results from efficient development of robust, flexible software with increasing numbers of programming units (e.g., procedures, classes, components or developers). The progression in the above two parenthetical lists goes from the lowest levels of abstraction (hardware) to the highest (people). This issue's theme encompasses this entire spectrum.

The lead author of each article resides in the Scalable Computing Research and Development Department at Sandia National Laboratories in Livermore, CA. Their co-authors hail from other parts of Sandia, other national laboratories and academia. Their research sponsors include several programs within the Department of Energy's Office of Advanced Scientific Computing Research and its National Nuclear Security Administration, along with Sandia's Laboratory Directed Research and Development program and the Office of Naval Research. The breadth of interests of these authors and their customers reflects in the breadth of applications this issue covers.

Multicore challenges and benefits

This article demonstrates how to obtain scalable execution on the increasingly dominant high-performance computing platform: a Linux cluster with multicore chips. The authors describe how deep memory hierarchies necessitate reducing communication overhead by using threads to exploit shared register and cache memory. On a matrix–matrix multiplication problem, they achieve up to 96% parallel efficiency with a three-part strategy: intra-node multithreading, non-blocking inter-node message passing, and a dedicated communications thread to facilitate concurrent communications and computations. On a quantum chemistry problem, they spawn multiple computation threads and communication threads on each node and use one-sided communications between nodes to minimize wait times. They reduce software complexity by evolving a multi-threaded factory pattern in C++ from a working, message-passing program in C.

Components for collaborative quantum chemistry

This article describes the use of component-based software engineering (CBSE) and object-oriented design patterns to surmount several social hurdles to collaborative software development. For example, they stress the importance of compromise in the interface standardization process that lies at the heart of CBSE. When compromise proves elusive, they employ the adaptor pattern to allow components to function in environments that require a non-standard interface. They also quantify the modest overhead (12%) associated with low-level operations such as data re-ordering at the component interface layer for packages that do not conform to the standard. The article includes Scientific Interface Definition Language (SIDL) code snippets based on a generic chemistry package from the Quantum Chemistry Scientific Application Partnership (QCSAP).

Multiscale modeling of micro- and nano-fluidics

This article presents two multiphysics applications formed from single-physics software components that must be advanced at disparate rates. One application involves a continuum model of flow in microchannels patterned into a solid, the boundaries of which support electro-osmotic flows simulated via molecular dynamics. A second application combines Brownian motion of solute ions interacting with nano-porous membranes and a continuum model of system-scale effects in electrodialytic water desalination. The authors' Python "puppeteer" components orchestrate the behavior of subordinate C/C++/Fortran 77 components, each of which remains incognizant of its peers, while automatically generated bindings bridge between the puppeteer and its subordinates. The flexibility afforded by localizing shared operations such as data transfer and convergence management inside the puppeteer easily justifies its cost: 1% of runtime.

Managing complexity with Bocca and CCA

This article presents Bocca, the first rapid prototyping tool for CBSE tailored to high-performance computing. Based on the Common Component Architecture (CCA), Bocca operates in an implementation-language-agnostic manner to automate the task of writing component glue code. By leveraging CCA's Babel SIDL interpreter, Bocca supports two-way calls between each of the most common HPC languages: Fortran 77/90/95, C, C++, Python and Java. The authors provide several polynomial scaling arguments to demonstrate the complexity reduction Bocca affords. After providing a Bocca script that creates a complete, componentized application in only a handful of lines, they detail how Bocca simplifies the software life cycle from code creation to maintenance, application generation, refactoring, and importing existing SIDL and implementation code.

Analysis-driven architecture: Abstract data type calculus

This article charts three paths for analyzing scalable development. One employs object-oriented design (OOD) metrics. Another adapts concepts from computational complexity theory wherein the "computation" is a developer's algorithm for finding a bug. A third analysis uses information theory wherein the information takes two forms: that added when extending an abstract base class and that added to a collection of interfaces when introducing a new single-physics class into a multiphysics package. Each analysis considers abstract data type (ADT) calculus, or the construction of arithmetic, integral and differential operators on software abstractions of scalar, vector and tensor fields. The analyses demonstrate the low complexity of ADT calculus in terms of high package stability, short bug search times and minimal information entropy.

While a common thread of software complexity considerations runs through each article, their ordering delineates a trend from greater focus on scalable execution to sole focus on scalable development. The articles in the middle consider the interplay between the two. A unique emphasis on quantifying software complexity also binds most of the issue. The research accomplishments represented by these articles warrants optimism regarding the prospects for taming complexity. The challenges that persist warrant optimism that many years of vibrant research remain.