How to Performance Portability for CSE Applications

The IDEAS Scientific Software Productivity Project
ideas-productivity.org/resources/howtos/

Motivation: Many applications have long development and deployment cycles where code capabilities and complexity grow with time. The code lifecycle outlasts the platform lifecycle by several generations. Furthermore, CSE applications are used in similar or different configurations on many different platforms at any given time. A code may need to run on a cluster with or without accelerators, or it may need to work on all the latest leadership computing platforms, each of which has a unique architecture and software stack. Therefore a baseline performance across a range of platforms is a fundamental requirement for these codes. When combined with the necessity of using scarce HPC resources well from the systems perspective, and time to solution and therefore scientific discovery from the scientific perspective, performance portability becomes a critical issue, especially in medium to large code bases.

Software design approach: A code designed for a detailed specific architecture is unlikely to be portable or performance-portable. A good practice has been to design for an abstract machine model with distributed memory and relatively shallow memory hierarchy. Solvers focused on maintaining spatial and temporal locality of data as much as possible without hard-coding any machine-specific parameters. Designing for abstract machine models is still a good practice, although more than one type may be needed. An option is to broadly characterize the target machines into as few abstract models as feasible, and even from those extract the commonalities for design considerations. For example, data can be organized so that compilers can vectorize, or hierarchical parallelism can be used to exploit coherence domains. C++ template programming provides one way of using abstractions.

Focus on performance objectives: A software project should have a clear outline of the performance objectives of the code that are important for scientific discovery. Performance considerations should be at the full application level, facilitated by tuning knobs. In general the tuning space of applications is large. Exposing tuning knobs and making them easy to set allow exploration of the performance space more quickly. For example, if accuracy requirements are known, then one can trade off accuracy within certain bounds for faster time to solution.

Separation of concerns: Designing software such that different expertise can concentrate on different aspects of the software is a good practice for many reasons; performance portability is among the most important. For example, isolating parallelism from the performance considerations of local sequential kernels has been useful. Similar encapsulation of functionalities so that different kinds of optimizations may apply to different sections of the code helps with portable performance.

Composability: A composable code is one that can select and combine existing components in the code base in many different ways to generate different applications. Composability also allows for multiple alternative implementations of select code capabilities. This feature can be exploited to limit the amount of platform-specific implementation that needs to exist in a code.

Using Libraries: Many numerical technologies are available as libraries, many of which have portable performance. Libraries are also becoming more interoperable with one another.

Programming Model Choice: Several tools and programming environments are at various levels of maturity in providing options for portability. For example, OpenCL offers portability across GPU hardware, OS software, and multicore processors; but it does not offer performance portability (see study *here*). Similarly OpenMP has been a mainstay in hybrid or hierarchical parallelism. It can provide some degree of performance portability but with nontrivial effort in designing parallelism. OpenACC 2.0 is a directive-based programming standard, which can generate OpenCL and CUDA

This material is based upon work supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.

Version 0.2, April 25, 2016

code. Kokkos and RAJA use C++ metaprogramming to provide performance-portable programming models. Tiling abstractions such as *TiDA* provide a way of incorporating hierarchical parallelism. Some domains and codes have taken the approach of domain-specific languages (DSLs); examples are *OP2* for unstructured meshes and stencil-based languages such as *Stella* and Nebo (see also *Physis*). Another approach for providing performance portability is taken by task-based programming models such as *Charm++* and *Parallex*, which are independent of the number of processors, automate resource management, and support concurrent composition.

Dealing with Existing Large Code Base: The most important step is to know -- before starting the refactoring -- the final objective in terms of how deep the changes are going to be. The next step is to profile the code in order to find the hot spots and, if possible, to create a performance model that will help estimate possible performance improvements and help make a decision on the first target kernels to be refactored. One should then look for the best match among the available abstractions. Having a good test suite before starting refactoring is critical. The test suite should not depend on bitwise no-change tests. If possible the test suite should also monitor performance. Also important is having a strategy for verifying intermediate stages of refactoring, because waiting until the whole code is refactored (sometimes called "on-ramping") will make verification of correctness and performance much more difficult and time consuming.

FAQs:

Q: How do I choose which programming model to use in my application?

A: That depends on the target architecture and whether you want to be able to run on multiple platforms. At present not all programming models are compatible with all languages, so that might be another constraint. See *here* and *here* for examples. **(I'd prefer to see something specific rather than "here and here" - perhaps, See examples with a mesh smoothing algorithm and with a spectral-element-based code.)**

Q: When should I use libraries for performance portability?

A: If the bulk of your computation time is spent in solvers that are available from a library, you should consider using it, even if you have to adjust your data structures. You are likely to get state-of-the-art performance portability and better-quality solutions.

This document was prepared by Anshu Dubey with key contributions from Ulrike Yang, Michael A. Heroux, Todd Gamblin, and Irina Demeshko.

This material is based upon work supported by the U.S. Department of Energy Office of Science, Advanced Scientific Computing Research and Biological and Environmental Research programs.