The need for software deployability: Broadening community tools for industry use

Benjamin Cowan, Tech-X Corporation

Scientific software has taken great strides towards portability and sustainability in recent years. Portability libraries such as Kokkos provide a means for application developers to enable their codes to run on a variety of compute architectures. The widespread adoption of CMake has allowed codes to build for many combinations of platforms and compilers, and package managers such as Spack reduce the effort needed to build a toolchain required by a particular code. Tools such as these have broadened the reach of open-source, community-developed scientific software. Still, sustainability of community codes remains challenging, and requires the broadest possible user and developer base. Increasing the adoption of community scientific software tools by commercial software developers would increase both the user base of those tools and the resources available for development.

However, the low-level tools supporting scientific software development for HPC systems are not yet sufficient to attract commercial adoption. This is because commercial software has a more stringent set of requirements than community software. Community software need only be *portable*, i.e. be able to run with good performance on a reasonable range of systems. Such systems generally include only Unix-like platforms such as desktop Linux distributions, specialized Linux distributions for HPC, and possibly Mac OS. In addition, users are often responsible for building the codes themselves (along with any dependencies), especially when using an HPC system. They might need to make minor modifications to the build system or the code itself to get the software working on their system. In fact, some codes even require rebuilding for different use cases.

Commercial software, in contrast, has more stringent requirements, which we refer to as *deployability*. For software to be deployable, is must be able to be provided to end users as a binary package. One reason for this is that to maintain integrity of their intellectual property (and hence their revenue stream), commercial developers often use a closed-source model, so cannot distribute the source code for end users to build. More importantly, commercial developers cannot expect or require their customers to have the skill or the time to build a software toolchain. This would severely limit the potential customer base. Instead, commercial software must be provided to customers with an installation process no more complicated than double-clicking an installer or decompressing a tarball. Then, the installed software must run with good performance on the customer's system, without the developer knowing the customer's hardware configuration beforehand.

Deployability presents several unique challenges. The first of these is Microsoft Windows. Customers of commercial scientific software often use Windows workstations on their desks. Thus, the software must be built to run natively on Windows. The challenge here is that the Windows build environment is markedly different from those on Unix-like systems. Many packages require Microsoft Visual Studio—it is the only supported host compiler for CUDA, for instance—which has different semantics than commonly used compilers on other platforms. It also tends to lag in performance features such as OpenMP, necessitating the use of mixed-compiler toolchains. In addition, the Windows scripting environment is different, complicating package management scripts. Even with bash environments like Cygwin or WSL, some tools require

Windows paths rather than Unix-style paths. Community codes often make assumptions about the system environment that do not apply to windows.

Another challenge of deployability is the need for fat binaries. The developer does not know what hardware the customer will use—it could be an ancient Opteron box that's limited to SSE instructions or a brand new Intel Xeon system with AVX-512. There could be one or more GPUs in a variety of hardware generations. On the GPU side, the CUDA toolkit provides a simple mechanism for generating device code for all desired NVIDIA architectures with automated dispatch. For CPUs, the situated is more complicated. Some compilers support automatic dispatch, GCC through attributes and Intel through compiler options. Others provide attributes to optimize for a single architecture, but this would require a dispatch mechanism in the code. Performance portability libraries have not yet addressed this issue to our knowledge, since portability assumes that users can tailor the compile options to build specifically for the system they plan to use.