SEE TEXT ONLY

High-Performance Computing & FreeBSD

By Johannes M. Dieterich

High-performance computing encompasses a variety of fields and domains, from science and engineering to finance and social studies. The common denominator is *the practice of aggregating computing power in a way that delivers much higher performance than one could get from a typical desktop computer or workstation to solve large problems* [1]. More concrete examples include, in order of descending compute share on civil installations: materials design, drug discovery, climate modeling, materials design, computational fluid dynamics, economic simulations, and big data analysis.

Arguably, we are living in a consolidated HPC world in terms of operating system and microarchitectures employed, and on a cursory glance, contemporary HPC installations look very much alike. A typical installation will feature a Beowulf cluster build from thousands of nodes connected by a fast interconnect. As of this writing, the TOP500 list of the fastest supercomputers in the world contains 91.5% `amd64` processors and 99.6% Linux operating systems (OSs) [3].

So where is the space for FreeBSD in this monoculture and why should the FreeBSD community care about HPC? First, HPC continues to be both the source of important basic technologies widely used in more general computing and a cauldron of innovative technologies, e.g., basic numerical libraries and more recently deep-learning applications. Second, a more in-depth view would reveal a much more complex ecosystem also containing communication and the aforementioned numerical libraries, toolchains, and programming languages, as well as auxiliary hardware solutions like network switches and storage. Lastly, there are plenty of systems, such as developer workstations, that are more diverse, have different capability requirements, and are more accessible to OS alternatives; e.g., Ubuntu Linux is a prime choice for developer workstations, not so much for cluster installations.

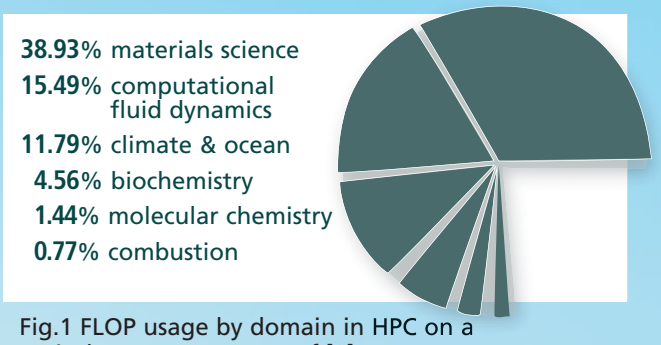

Fig.1 FLOP usage by domain in HPC on a typical supercomputer. Ref.[2]

To some extent, one may even argue that cloud computing is repackaged or broken-down HPC for customers not traditionally having or being able to afford a stake in or access to large installations. Even though FreeBSD already excels as an appliance solution, other capabilities (hypervisor choice, storage, security, network, etc.) matter more than the compute OS. I will focus here on capabilities for HPC, either the compute or developer OS, how they map onto more general computing needs, and try to highlight how FreeBSD would benefit from some HPC-targeted improvements.

As FreeBSD users and developers, we are aware of and value the inherent advantages of FreeBSD: a consistent experience of applications and libraries in the ports system, the easy build and deployment of custom packages potentially with architecture-specific optimizations, and, in general, a very stable operating system experience. Most importantly, the harmonized ports experience allows us to propagate changes across all parts of the OS interacting with the user or developer consistently.

Languages

What are the tools of HPC—the applications used by researchers to transform FLOPS into knowledge—made from? If we remember that HPC was one of the original purposes of computers, the answer that the majority of computing time is used by Fortran codes is less of a surprise. More than 65% of compute cycles are spent on Fortran codes on a typical installation. This share increases if one considers the numerical Fortran libraries used in mixed-language codes. Unfortunately, this shows that Fortran is alive and unlikely to go anywhere in the near-term to mid-term due to the size and complexity of existing code bases. Hence, HPC support absolutely requires Fortran support. Due to the effective absence of the typically used commercial compilers on FreeBSD, we fall back to the open-source alternatives.

Currently, if a port specifies `USES=fortran` all architectures will use the `gfortran` compiler from `lang/gcc`. `gfortran` is a good Fortran compiler that produces stable, fast binaries and supports all relevant Fortran standards. Currently, using `gfortran` requires explicit specification of the `rpath` to the linker to search for GNU libraries for both that port and, if it is a library, all dependent ports (or out-of-ports tree applications, for that matter). While trivial, it significantly increases the maintenance burden of porters and developers. Remedies are being discussed currently, but no patches have landed in the tree yet. Also recall that all possible architectures (among them `amd64`) use the LLVM-based `clang` compiler as the base compiler for licensing reasons and, hence, by extension, as the default ports compiler. Instead of sorting out mixed-compiler environments, it is then quite often easier to simply rely fully on the GNU compilers for mixed-language programs. This is a mildly unsatisfying situation.

Is there a more fundamental improvement possible? Maybe. `flang` is an open, LLVM-based, new Fortran compiler. [4] Its frontend is derived from the well-established Portland Group's compiler and was open-sourced and continues to be maintained by NVIDIA. It since has found support and use by AMD in its AOCC package as well. `flang` supports only up to Fortran 2003 language features and is limited to 64-bit architectures. FreeBSD now contains `devel/flang` as a preview. Certainly, some time and effort are required to make this port competitive with `gfortran` and vet it properly for HPC uses. Just very recently, plans from NVIDIA became public to rewrite large parts of `flang` to improve its feature set and likelihood of being accepted under the official LLVM umbrella. Uptake within the HPC community will be interesting to observe, but the added competition should prove beneficial to the GNU compiler either way.

Let us think beyond the single core. As noted above, a typical cluster contains tens of thousands of cores, and jobs are typically required to use tens to hundreds of them in parallel. Two major approaches exist for scaling out in HPC: OpenMP [5] and the Message Passing Interface (MPI) [6].

OpenMP mostly targets shared-memory machines with later standards having support for accelerator offloading and is discussed below. It is a relatively simple, pragma-based approach, supports C/C++ and Fortran, and is commonly used to parallelize over loops in a data-parallel fashion. Currently, our ports tree will again default to the `lang/gcc` port if a `USES= compiler:openmp` is encountered. For this reason, ports typically will not enable OpenMP by default (including some commonly used ones like `graphics/ImageMagick`) and hence will be limited to a single core.

A better alternative exists for at least `amd64` and `i386`; the `libomp` library associated with the LLVM project since the initial open-source release by Intel. It has not been imported into base yet, but a review for an older library version exists. In its hopefully temporary absence, the ports system should use one of the `devel/llvm` ports that do include `libomp`. Multiple integration tests have been done and the feature should soon be ready to land. Hopefully this will also incentivize fellow developers

to add the necessary FreeBSD bits for other architectures supported upstream. My tests indicate that `libomp` integration in LLVM is not ideal from a performance perspective; in particular, LLVM's vectorizer does not work (well) if OpenMP's `simd` pragma is used in conjunction with standard OpenMP thread parallelization (e.g., `parallel for`). But certainly, suboptimal parallelization is preferable to no parallelization.

MPI on the other hand operates through function calls into the MPI communication library. It features both simple send/receive/broadcast features as well as more advanced operations like reductions. MPI is used both for process-based intra-node, as well as inter-node, parallelization, and on the hardware level, typically uses a fast interconnect such as InfiniBand. Typically, MPI-enabled software packages are compiled using `mpicc/mpif90` wrapper scripts which configure the underlying compilers to find MPI headers and link against the MPI library. Within the ports tree, multiple MPI choices exist and we are only limited by the underlying compiler toolchain for Fortran.

Numerical Libraries

A large part of the *high performance* in HPC does not result from application code, but instead, in the judicious and abundant use of highly-optimized numerical libraries implementing standardized APIs. The arguably most important APIs are BLAS, a collection of basic dense linear algebra operations such as matrix-matrix multiplications, LAPACK, a collection of more complicated solvers such as Cholesky or LU decompositions, and Fast Fourier Transformations (FFTs), typically in the FFTW3 API incarnation. Their fundamental nature also makes them a common dependency in our ports system, e.g., for audio and graphics applications.

Choice in libraries implementing these APIs is much less important than performance and features of a single set of them. Most HPC clusters provide vendor-tuned, assembly-optimized BLAS and LAPACK libraries. For FreeBSD, we have multiple options exposed through the `blaslapack` selector. By default, this selector will use `math/blas` and `math/lapack`. These are Fortran-source, reference implementations and, as such, not competitive with optimized alternatives such as `math/blis` and `math/libflame`. Both the reference implementations and `math/openblas` have a Fortran dependency unlike the FLAME project's BLIS and `libflame` [7].

As we can see from Figure 3, both the OpenBLAS or BLIS implementations show a commanding lead in performance, starting from small to medium-sized matrix-matrix multiplications if their CPU architecture optimized kernels are used. More importantly, even if BLIS's source-only reference implementation is used, the implementation and blocking scheme results in an up to 80% performance advantage. Additionally, BLIS provides the ability to use `pthread` parallelization, which is of less impact in this test case. The FLAME project has expressed interest in working with us, accepting pull requests of FreeBSD changes and implementing features such as runtime kernel selection which we need for generic packages. Hence, BLIS is a good candidate to be our default BLAS implementation and rids us of a low-level Fortran dependency.

The `math/libflame` port has recently been updated to a recent development snapshot and configured to expose a LAPACK interface for `amd64` and `i386` CURRENT. More vetting is required before the FLAME libraries can be added as a `blaslapack` option for recent releases. Work in this regard is ongoing.

The status of other numerical, engineering, and scientific libraries on FreeBSD is already excellent: the `math/fftw3` port is in great shape; we also have a variety of development libraries from a range of domains including quantum physics/chemistry ready to use and multiple active ports committers in that realm.

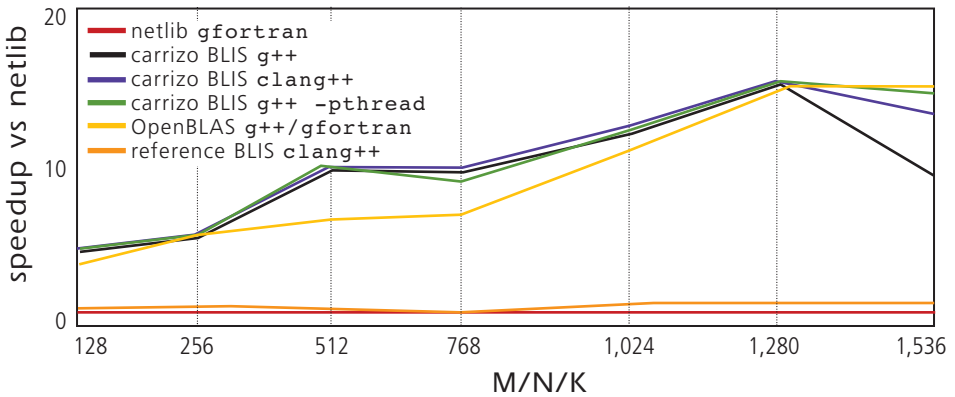

Fig. 3. Speedup of BLIS and OpenBLAS over the reference netlib BLAS for matrix-matrix multiplications (`dgemm`) of different sizes M/N/K on an AMD A12-8800B CPU. Data averaged over 1,000 evaluations on FreeBSD HEAD, compilation with -O2, all libraries compiled with `-mavx -mavx2 -msse -msse2 -msse3 -msse4 -msse4a -msse4.1 -msse4.2 -mmmx -maes -mbmi -mbmi2 -mf16c -mfsgsbase -mtune=bdver4`, BLIS compiled with `clang++` 5.0.0, netlib/OpenBLAS with `g++` 6.4.0.

Developing on and Porting to FreeBSD

With an overall, relatively positive status of the languages and libraries on FreeBSD, how hard is porting HPC applications to and developing them on FreeBSD? Typical challenges arise from the use of nonstandard make systems (typically combinations of shell and other scripts as well as make files), the prevalent use of Linuxisms [8] such as hardocded system paths and other assumptions, the generally nonstandard compliance of code (e.g., it only compiles and yields the correct result with a specific proprietary compiler [version]), and also our own BSD-isms such as the `rpath`. All but the last challenge are arguments in favor of porting: code bases typically improve and lingering bugs are exposed and fixed. Porting to FreeBSD has tenable advantages if done properly: just as with any other application, changes should be upstreamed, explained, and continuously maintained. In the meantime, we should consider reducing our BSD-isms to ease the porting task.

Continuous maintenance or development on FreeBSD is straightforward. Even though the normal commercial toolchains for profiling, debugging, etc., are absent, the base system and ports collection includes excellent free alternatives. `dtrace` and `hwpmc` in conjunction with `benchmarks/flamegraph` make analyzing application performance from the user down to the kernel level straightforward. `devel/gdb` and `lldb` may not have the same level of graphical user interface support as the commercial alternatives but do get the job done. Even though most HPC code is still developed with `vi` or `emacs`, modern editor/IDE alternatives are present with `java/eclipse`, `java/netbeans`, and the `linuxulator`-using `editors/linux-sublime3`.

Recently, I have found `bhyve` provides an invaluable addition to the HPC developer's toolkit. No matter whether bugs are to be located only occurring on a particular platform configuration, specific Linux releases need support, or a full, continuous integration setup is needed, `bhyve` provides an excellent solution for all these use cases.

Accelerators

Probably the most disruptive change to the HPC landscape in the last decade was the introduction of accelerators into the HPC mainstream and its subsequent trickling into the workstation and mainstream market. Even though most TOP500 installa-

tions still do not contain accelerators, the importance of accelerators for HPC and workstation applications cannot be overstated. Various means to exploit their potential exist, with the most important ones being direct programming via NVIDIA's proprietary CUDA language, the open alternative OpenCL, or through offloading via OpenMP.

These programming frameworks all rely on three fundamental parts: kernel driver support, a compiler or library, and a runtime environment. Through the FreeBSDDesktop project, we now have access to more recent open drivers for AMD and Intel GPUs. NVIDIA GPUs continue to be supported by the binary driver in the ports. There is no official CUDA support for FreeBSD; however, some people have reported success in the past compiling CUDA applications on Linux and running them through the Linuxulator on FreeBSD. Our OpenCL support is in reasonable shape: we include the nonofficial OpenCL clover library from the Mesa project for AMD GPUs. Its performance is absolutely not competitive but it works relatively reliably. For Intel, we include their official beignet implementation; however, Intel's GPUs are less competitive for compute tasks. Developing OpenCL applications is well supported; we include the CPU OpenCL emulator `lang/pocl` and the sanity checker `devel/oclgrind`. The exploitation of accelerators through OpenMP offloading is not supported currently but is, in general, not widespread yet.

A major improvement could be to include the Radeon Open Compute (ROCm) project [9]. It needs an open companion kernel driver, `amdkfd`, for the regular open `amdgpu` driver and provides a large open ecosystem centered around LLVM compiler technology to support both OpenCL and an open competitor to CUDA called HIP on AMD's GPUs. The FreeBSDDesktop team is now actively working on porting `amdkfd`, and the large ROCm stack should be straightforward if somewhat labor-intensive to port.

What's Next?

The highest priorities should be to vet FLAME's BLAS and LAPACK libraries and add them as an option to the `blaslapack` selector. Secondly, libomp should be used as the default for `amd64` through `devel/llvm`. Subsequently, some stabilization and extension of these major changes to architectures other than `amd64` will be needed. In the mid-term, I am hoping that we will be able to have ROCm working on FreeBSD. Another big-ticket item is proper SIMD/vectorization support in our `libm` and from LLVM. Together, these should already be an interesting HPC platform for developers. Hopefully, in the long-term, we can improve the Fortran situation and make FreeBSD a truly compelling HPC alternative.

It is also important to realize that improving FreeBSD for HPC will not hurt it, either as a server or workstation system. On the contrary, it will likely be a boon for these use cases.

Acknowledgments

I wish to express my gratitude to all FreeBSD developers and users for making FreeBSD the platform it is. In particular, I would like to thank the FreeBSDDesktop team and my mentors Matthew Macy, Niclas Zeising, Steve Wills, and Rene Ladan. I also extend my deep gratitude to the BSDTW conference and organizers where the content of this article was first presented as a lecture. ●

REFERENCE LIST

[1] InsideHPC: https://insidehpc.com/hpc-basic-training/what-is-hpc/

[2] Data source: UK supercomputer ARCHER application usage of the last month, http://www.archer.ac.uk/status/codes

[3] Data source: TOP500 list, June 2017, https://www.top500.org/statistics/ list/

[4] Flang github: https://github.com/flang-compiler/flang

[5] OpenMP https://www.openmp.org/

[6] MPI forum https://www.mpi-forum.org/

[7] FLAME project https://github.com/flame

[8] Linuxisms discussion https://wiki.freebsd.org/AvoidingLinuxisms

[9] Radeon Open Compute https://github.com/RadeonOpenCompute/

JOHANNES DIETERICH started using FreeBSD with the 6.1 release and became a ports committer a year ago. During the day, he has spent the last nine years in academic research working on high-performance computing for a range of problems from global optimization to quantum chemical methods. Recently, he started working on GPU-accelerated deep learning for AMD.