acmqueue

The Ideal HPC Programming Language

Maybe it's Fortran. Or maybe it just doesn't matter.

Eugene Loh, Oracle

The DARPA HPCS (High Productivity Computing Systems) program sought a tenfold productivity improvement in trans-petaflop systems for HPC (high-performance computing). This article describes programmability studies undertaken by Sun Microsystems in its HPCS participation. These studies were distinct from Sun's ongoing development of a new HPC programming language (Fortress) and the company's broader HPCS productivity studies, though there was certainly overlap with both activities.

These programmability studies started with a focus on programming languages, but the focus quickly shifted to other topics. Existing languages—notably Fortran, which is arguably still the primary language in HPC—proved remarkably adequate. Programming challenges stem mostly from other factors.

BACKGROUND

What if *programming* did not mean having to learn a language someone else devised and then wrestling with the limitations of that language, its compilers, and computers to implement your task? What if it meant, in a sense, the opposite? You could write your program in whatever way was most expressive for you, without regard for language rules imposed by someone else. Then it would be somebody else's job to define the programming language that would make sense of what you wrote, write the compilers to digest the program, and build the computers that would efficiently run the task you specified.

We undertook such an exercise to get a feel for what an "ideal" programming language for HPC applications might look like. Our approach was to take existing HPC programs and have someone rewrite them in whatever way suited that individual, not bound by the constraints of any existing computer, compiler, or language. Rather, he was invited to write whatever seemed most expressive. We might not be able to compile or run these programs, but we could at least see what the writer wanted.

Almost immediately, we were struck by what we were seeing. Of course, the rewritten code was much more compact and readable than the original, but, surprisingly, the "ideal" programming language was basically Fortran.

My first job here is to convince you that this finding is not ridiculous. I admit, the experiment was biased in that we were starting with existing code, mostly written in Fortran, and used a human subject who was not only familiar with Fortran but indeed embraced it. The main point, however, is less that *every* programmer would have ended up preferring Fortran and more that the problems with the original source code have more to do with reasons other than the limitations of existing programming languages. We look at some of these reasons here.

The DARPA HPCS program also sponsored the development of new programming languages:

Chapel from Cray, Fortress from Sun, and X10 from IBM. Proponents of those languages would show early on how rewriting familiar HPC benchmarks in the new languages could reduce source-code volume substantially—tenfold reductions were not surprising—but rewriting these benchmarks even in Fortran achieved similar source-code reductions and corresponding improvements in expressivity.

New programming languages still have much to offer, for example, in the areas of expressing concurrency and especially data distribution. It's just that the bloat we see in current HPC source code stems not so much from inadequacies in current languages as from other factors.

WHAT WE DID

We rewrote a number of HPC benchmarks and applications using modern Fortran in a way that took into account the human costs of software development: programmability and associated characteristics such as readability, verifiability, and maintainability. These are important considerations; although copy-and-paste is a fast way of writing lines of code, it degrades readability and increases maintenance costs.

Part of this effort included working with the Sun HPCS productivity group to quantify programmer productivity in general and to study human subjects in our rewriting exercises in particular. A human subject's activities could be observed passively with the Hackystat telemetry tool or actively via interviews or having the subject keep a journal. The team included a cultural anthropologist who guided these observations.

In this article we focus on the output of the rewriting activity, examining the rewritten HPC programs and causes of source-code bloat. The particular HPC test codes used here are the NPBs (NAS Parallel Benchmarks) CG, MG, and BT; the plasma fusion application GTC; and the 3D hydrodynamics code sPPM.

A key metric was the number of SLOC (source lines of code). This is admittedly a crude and often deceptive metric, but it served as a convenient starting point for quantifying readability and expressivity of source code.

Since the generated computer programs were in Fortran, they could be compiled and run. Thus, we were able to study their performance relative to the original code, test automatic parallelization with currently available tools, and speculate on the potential for improvements in autoparallelization.

Table 1 lists SLOC and performance comparisons between original and rewritten versions of some of the HPC codes we studied:

TABLE 1. HPC Code Comparisons

	Lines of Code			
Code Name	Before	After	Reduction	Performance Slow Down
NPB CG	839	81	10x	1x
NPB MG	1701	150	11x	2x-6x
NPB BT	4234	594	7x	2.7x
GTC	6736	1889	3.6x	2.7x
sPPM	13606	1358	10x	2x

AUTHOR FEEDBACK

We saw remarkably large reductions in source-code volume. The smallest reduction was in GTC, which already used relatively modern Fortran constructs, had relatively little MPI (Message Passing Interface) parallelism (distributed memory), and had computation and I/O formatting that the human subject was uncomfortable modifying.

We saw various indications that the rewritten programs not only had fewer lines of code, but also were easier to read, verify, and modify. It was not simply our judgment, however, that concluded that expressivity could be improved tremendously. In the case of GTC, we solicited feedback on the rewritten program from one of the application's maintainers. Here are selected comments:

> At first glance, I was impressed by how small and compact the code had become. I always thought that GTC was as small as it could get, but I was obviously wrong. I was also pleasantly surprised to discover that the programming language was still standard Fortran 90/95, and not a totally new language.
>
> The new code is clear, concise, and easy to read.
>
> The fact that all the MPI calls and OpenMP directives have been removed makes the physics represented in the code easier to follow.
>
> [The rewrite introduced elegant] code reuse in CHARGE and PUSH.

But there was this warning:

> [Expect a] performance hit unless the compiler can perform very good interprocedural optimization and/or automatic inlining.

This warning arose because there were many transformations from continuous (ζ,r,θ) coordinates to discretized mesh indices. The readability and maintainability of the source code benefited greatly from encapsulating these many transformations into a few functions, but the performance suffered from the extra procedure calls and loss of many specializations and optimizations of the transformations.

SINGLE-CPU PERFORMANCE

Much of HPC is performance, including parallelization. The code we examined showed many familiar HPC characteristics: loop unrolling, vectorization, cache blocking, multithreading, data distribution, and so on. One might argue that, while it may be possible to reduce code volume dramatically, the cost in overall performance would be intolerable.

We were pleasantly surprised that single-CPU performance degradation wasn't too bad in general. Indeed, for NPB CG, most of the work is performed by low-level sparse-matrix routines, and overall performance really didn't change at all. We expect similar results whenever the computationally intensive kernels—sparse-matrix routines, dense matrix multiplies, FFTs (fast Fourier transforms), etc.—are performed in library or other well-tuned kernels.

In other cases we saw slowdowns but expected to recover much of the performance with judicious,

tactical (few-line) optimizations. For example, the rewrite of the NPB MG code saw a 2x speedup by converting stencil operations from array syntax to (arguably more readable) DO loops. In GTC, one section of code ran four times faster when the Fortran MODULO intrinsic was replaced by a suitable substitute. Such optimizations, of course, place one on a slippery slope. Code bloat creeps back in, and maintainability of the code degrades. Indeed, even performance can suffer. We have seen cases where simplifying the source code by removing "optimizations" actually improved performance, presumably because the "optimizations" originated on sufficiently different hardware or targeted sufficiently different compilers.

Meanwhile, the battle to deliver good performance on expressive HPC source code must still be waged. Compiler optimizations must be augmented with ongoing hardware improvements. There is much work to be done on latency-hiding techniques such as prefetch, chip multithreading, and scout threads. To some extent, this simply moves the pressure from memory latency to memory bandwidth; thus, some system designers tackle other problems, such as efficient use of partial cache lines.

PARALLELIZATION

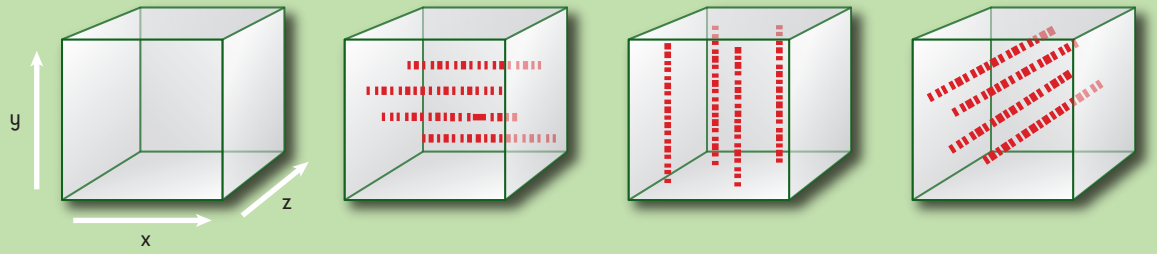
HPC parallelization often falls into two categories: finding concurrency and distributing data. Finding concurrency is much simpler than distributing data. Our guarded optimism regarding existing languages extends even to parallelization if, by that, we mean finding concurrency. If data distribution is needed to achieve high-end performance, however, new programming languages or constructs seem that much more crucial.

HPC seldom uses locks. More typically, concurrency is related to data-parallel loops—for example, time stepping all particles or grid elements concurrently. Meanwhile, clusters of commodity computers have become the price-performance winners in HPC. Therefore, parallelization also involves the decomposition of data over cluster nodes. Nodes share data in HPC typically through explicit message passing, for example with MPI.

Consider the ADI (alternating direction implicit) method for solving partial differential equations. Specifically, consider a 3D rectangular grid, such as that shown in figure 1. Physically, the information on any grid cell propagates throughout the 3D volume, ultimately influencing all other cells. Computationally, we can restrict data propagation to be along only the X axis in one phase of computation, later along the Y axis, and finally along the Z axis. Ultimately, the computed physics should remain unchanged.

Such an algorithm organizes computation along "pencils" of cells. For example, in the first phase, all cells in an X-aligned pencil can be updated based solely on data values within this pencil. Indeed, all X-aligned pencils can be updated concurrently; then all Y-aligned pencils; and finally, all Z-aligned pencils. If there are N^3 elements in the grid, then there are N^2 pencils in each of the X, Y, or Z phases. That is to say, there is considerable concurrency. The BT and sPPM codes are both organized like this, as are multidimensional FFTs.

FIGURE 1 **3D Grid Showing Pencils of Cells**



The pseudocode might look like this:

```
INTEGER NX, NY, NZ, X, Y, Z
DIMENSION MYDATA(NX,NY,NZ)
FORALL ( X = 1:NX, Y = 1:NY ) CALL UPDATE(MYDATA(X,Y,:))
FORALL ( X = 1:NX, Z = 1:NZ ) CALL UPDATE(MYDATA(X,:,Z))
FORALL ( Y = 1:NY, Z = 1:NZ ) CALL UPDATE(MYDATA(:,Y,Z))
```

Each subroutine call can be made concurrently with all other calls in the same FORALL statement. There is, however, no way of distributing the elements of MYDATA onto multiple processors so that each processor has all the data it needs for all stages of computation. If a particular processor "owns" MYDATA(X,Y,Z), then to process an X-aligned pencil of data it needs all MYDATA(:,Y,Z) values. Then, to process Y-aligned pencils of data, it needs all MYDATA(:,:,Z) values. Finally, to process Z-aligned pencils of data, it needs all MYDATA(:,:,:) values.

Therefore, while concurrency in this example is rife, distributed-memory systems face a great challenge both in exchanging data between processors explicitly and in distributing data so that such costly exchanges are minimized.

Similar issues arise even in shared-memory systems. It may be possible for all processors to access all elements in place, but these accesses must be coordinated, whether to prevent race conditions or to deal with cache coherency. Even shared-memory systems benefit from spatial locality since processors can then deal with complete cache lines.

If we focus on the relatively easier problem of concurrency, we could in the long term help keep the HPC programmer from having to parallelize explicitly. We would benefit from improvements in software. Existing compilers already identify some opportunities for automatic parallelization. This includes progress on autoscoping—that is, automatically analyzing source code to determine the usage (private, shared, read-only shared, replicated, etc.) of variables so that a loop could be parallelized. Automatic analysis would be aided by whole-program or interprocedural analysis.

Runtime management of concurrency would also help. Loops might be nested, or loop iterations might be unbalanced. Loop counts and processor counts might not be known until runtime. Static analysis alone cannot balance computational loads or judge the balance between fine-grained

parallelism (for maximum concurrency) and coarse-grained parallelism (to amortize the costs of parallelization).

Simpler concurrency for the HPC programmer would also benefit from hardware improvements. Large, globally addressable memories help. Processors run faster with cached data, however, so coherency must be managed. Hardware can support concurrency with atomic operations, transactional memory, and active messages.

While concurrency seems relatively simpler, managing data distribution seems a much more difficult task. This is one area where new programming languages could really offer help.

For example, the NPB BT benchmark has a cousin, BT I/O, which adds I/O to the test. This offers a test of adaptive maintenance—that is, adding functionality to an already written program. The comparison was almost a joke: setting up I/O in the original, distributed-memory version of the code added 144 source lines, while the rewritten, shared-memory version needed only one extra line!

ALGORITHMIC COMPLEXITY

Performance and parallelization are not the only pressures causing large source code. Another issue is that the ideas the computational scientist wants to express are rather low level. For example, the fusion code GTC models the Lorenz force, which a physicist could succinctly write as

$$F = q(E + v \times B)$$

but which the computational physicist transforms into many pages of bewildering equations and commensurately large volumes of computer code. Since charged particles travel in very tight spirals in plasmas, the computational physicist starts by transforming to a "guiding-center" formulation. Then coordinates are transformed to align with the magnetic fields in a tokamak. Such transformations introduce considerable complexity, but they also improve the numerical properties and performance of the code by several orders of magnitude, an advantage that cannot be overcome just by buying more computer equipment.

Generally, computational scientists remove high-frequency components, discretize grids, use sophisticated time stepping, introduce crucial approximations, expand terms, transform coordinates, add dissipative terms and upwind differencing to control numerical stability, and otherwise turn a few simple equations into pages of mind-numbing algorithms that represent the essence of what they're trying to do computationally. To forgo that algorithmic complexity would increase computational cost by many unaffordable orders of magnitude. A computational scientist's bread and butter is not simply the equations of mathematical physics (the Lorenz force, Schrödinger's equation, Navier-Stokes equations, etc.) but algorithmic specifications that make computation possible within a particular set of conditions. Fortran is rather good at expressing computational rules. Modern Fortran with array syntax, generic interfaces, optional arguments, recursive subroutines, MODULEs, array-valued functions, and other features, is even more so. The ability to have typeset mathematical syntax, as with Fortress or Mathematica, would also be nice.

Other areas that seem to have complexity that would be hard to express regardless of the programming language include high-level algorithmic control flow and detailed I/O formatting.

IMPLEMENTING BAND-AIDS

Source-code volume also expands as a result of limitations—even defects—in current software and hardware. One example is portability. HPC programmers must account for different vendors, MPI implementations, threading models, compilers, Fortran-C interoperability conventions, default word sizes, etc. In other cases, code takes pains to reproduce particular floating-point numerics (regardless of whether those numerics are right). Inconsistent library availability, whether a result of licensing or installation and bundling issues, also is an issue: while libraries offer all sorts of functionality, HPC code often has its own random-number generators, matrix multipliers, sparse-matrix support, linear solvers, and FFTs to ensure that these capabilities will be available regardless of where the application is run.

HPC code sometimes also implements capabilities that might be better provided by tools. Examples include performance instrumentation, debugging code, and checkpointing.

Source code also reflects workarounds to transient bugs or to limitations in compilers. An example is Fortran array syntax. We have found many instances where array syntax allows much higher-level programming. Many programmers, however, have avoided the elegant syntax because its implementation in many compilers is immature. Arguably, developing new programming languages would exacerbate rather than solve such a problem.

Despite our rosy view of existing programming languages, we admit encountering areas where language improvements would have been nice. Type inference, including the inference of array extents, would allow one to forgo tedious boilerplate declarations. Better support of stencils (computations on grids where each element is updated based on nearby elements) is useful for HPC.

SPECIFICATION, VERIFICATION, AND VALIDATION

We started with the software development model in which a computer program starts from a written specification. Then, it must be verified (checked against the spec) and validated (checked that it fulfills its intended purpose over some range of parameters).

It is possible that the program is written without verifiability in mind. Here is a striking example from the NPB BT code:

```
rhs(2,i,j,k) = rhs(2,i,j,k) + dx2tx1 *
      (u(2,i+1,j,k) - 2.0d0*u(2,i,j,k) +
      u(2,i-1,j,k)) +
      xxcon2*con43 * (up1 - 2.0d0*uijk + um1) -
      tx2 * (u(2,i+1,j,k)*up1 -
      u(2,i-1,j,k)*um1 +
      (u(5,i+1,j,k)- square(i+1,j,k)-
      u(5,i-1,j,k)+ square(i-1,j,k))*
      c2)
```

This code is basically supposed to implement the following from the NPB1 specification:

$$[RHS2] = \ldots$$
$$- (\partial / \partial \xi) ([u^{(2)}]^2/u^{(1)} + \phi)$$
$$+ (\partial^2 / \partial \xi^2) (d_{\xi}^{(2)} u^{(2)} + (4/3) k_3 k_4 [u^{(2)}/u^{(1)}])$$

There is little correspondence between the source code and the specification it is supposed to implement. This is not so much a limitation of the programming language but of human intention. Here is how we rewrote the code, with the purpose of improving readability and verifiability:

```
RHS2 = RHS2 - deriv(1,1,u2**2/u1+phi)
RHS2 = RHS2 + deriv(2,1,dx2*u2 + 4*k3*k4/3*u2/u1)
```

It is more likely, however, that there isn't even a spec to verify the code against. When we attempted to verify GTC and asked for a specification for the application, we received this somewhat humorous reply: "There is one physicist at the lab who actually went through the code line by line and took some notes. Unfortunately, these notes are not in electronic format, and worse... they're in Chinese."

There may have been a specification originally, but the source code evolved over time, while the spec was never updated. To mitigate the divergence of spec and source code, we looked at making source code, even Fortran, as readable as possible and interleaving source code with specification or documentation. We tried an implementation of the HPCS graph analysis benchmark, SSCA #2, where the "source code" was HTML from which a script could extract Fortran code to compile and execute. This approach to having a single artifact to maintain, instead of disjointed specification and source code, is similar to ideas found in Mathematica notebooks, Donald Knuth's *Literate Programming*, and Scientific WorkPlace.

Validation is also difficult. One must compare results in particular parameter regimes to results that might be known from analysis or predecessor codes. Since validation is so expensive and depends so critically on experienced scientific understanding and intuition, over most of an HPC application's lifetime one simply checks software modifications by comparing results with an earlier version of the code. Whereas the science is meaningful to only limited precision (say, 1 percent or even 10 percent), checking numerical results in HPC usually means checking fickle floating-point arithmetic out to the least significant digit. We found cases, for example, where we refrained from changing the source code because changing `((2*pi)*k)/N` to `2*((pi*k)/N)` or changing `X*(1/deltat)` to `X/deltat` changed floating-point results subtly. We do not know if the results were more accurate or less, only that they were slightly different. These differences prevented us from making the source code more readable or run faster.

PROGRAMMERS' PRIORITIES

Project deadlines force software to be written quickly. Many expedient writing styles, however, cause programs to become longer and therefore more difficult to read, understand, verify, and maintain. Meanwhile, many programming habits develop in a culture of fast prototyping, where programmers avoid advanced language features since their support is immature, and focus instead on the last drop of performance. As presented in Donn Seeley's *ACM Queue* article, "How Not to Write Fortran in Any Language" (December/January 2004/2005), examples of poor programming practices abound.

Programming for verifiability is often not a priority, as the BT example illustrated.

As another example, in sPPM we found thousands of lines of code for handling boundary conditions. The rewritten code used only about a dozen lines. There are many reasons for this

astounding reduction, but one issue is that the original code attempted to fill in "ghost cells" only when their values would be needed. (Ghost cells are replicas of real computational cells, where such replication can simplify the handling of boundary conditions.) In the rewrite, we would routinely fill in all ghosts cells. Eliminating checks on whether such updates were needed facilitated the programming logic immensely, with nearly no overall performance loss in the cases we studied. In HPC, the mindset is usually to program for performance rather than programmability even before establishing whether a particular section of code is performance sensitive or not.

The ISO/IEC standard on software maintenance adopted the term *perfective maintenance*. Modifying source code simply to improve its maintainability, however, often receives scant attention when other objectives—such as fixing defects, implementing new features, tuning performance, and migrating to new platforms—clamor for attention.

The NPB BT source code takes hundreds of lines of code to compute the time derivative d*U*/d*t* to form the "right-hand side." This computation appears to have been implemented from scratch twice, once in file rhs.f and again in exact_rhs.f. Even if this duplication of effort was overlooked originally, perfective maintenance should weed out such redundancy to benefit the generations of HPC workers who have had to look at this source code since it was first written—provided, of course, that this is important for the software's owners.

WHERE DO WE GO FROM HERE?

Repeating some of these programmability studies on larger HPC programs would be interesting. In particular, it would be nice to move from self-contained programs that are small enough for one person to have written—what DeRemor and Kron would term "programming in the small"—to larger pieces of software, written by many people and where interfaces among many parts are important ("programming in the large"). Like nature, source code looks different at different scales: from fast prototyping, to self-contained applications, to multi-decade legacy code. Further work to relate source-code characteristics empirically to human productivity metrics would also be interesting.

Most of all, the HPC community could well benefit from a community-wide effort to emphasize programmability and human productivity. No one piece comes first. Progress is required on all fronts: language development, compiler maturity, hardware innovations, HPC software development practices, and even procurements and competitive benchmarking.

When we start with an existing language, however, we benefit from available compilers, systems, reference codes, experience, and programmers. Q

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

EUGENE LOH (eugene.loh@oracle.com) is a principal software engineer at Oracle Corporation and worked on programmability, performance, and productivity studies as part of Sun's HPCS activities. His current focus is performance of MPI-based HPC applications.