Programming Language Requirements for the Next Millennium

William G. Griswold, Richard Wolski, Scott B. Baden, Stephen J. Fink
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{wgg,rich,baden,sfink}@cs.ucsd.edu

Scott R. Kohn
Department of Chemistry
University of California, San Diego
La Jolla, CA 92093-0340
skohn@chem.ucsd.edu

1 Problem: The New World Disorder

Our technology infrastructure has become so highly evolved that the changes it induces are perceptible daily, subjecting our lives to continuous, often tumultuous change. Change is now so rapid that the decisions that we make to enhance our livelihoods—from education to product development to technology research itself—can become obsolete before they are made.

As a result, the application software that helps us make these decisions is becoming increasingly performance limited. Moreover, the applications themselves need to be quickly evolvable both to address these rapidly arising problems and to incorporate the newer, better solution methods that are continually being introduced. Because problems are arising more frequently, less resources are available for solving each problem.

The performance of many of these applications is sensitive to the architectural platform. To compound this problem, the push for ever greater performance at lower cost has led to building computational systems from locally managed computational resources (e.g., workstations on desks) that are accessible over high-speed enterprise-wide LANS and the Internet. As a consequence of resource sharing and the constraints imposed by local management, an application must dynamically adapt its execution to account for contention in the environment. Similarly, for maximum performance an application must be able to adapt itself to the structure of the data as it evolves throughout the execution of the application.

2 Coping with the Problem: An Example from Scientific Computing

One example where the performance requirements imposed by dynamism are most evident is in scientific and engineering computing, which encompasses particle simulation, stock market prediction, weather forecasting, drug design, and product engineering. The scientific computing community has turned to concurrency (i.e., parallelism) in response to the need for maximum performance.

The language approach: High Performance Fortran. An important focus for the scientific computing community has been the development of improved programming languages to address the tension between performance and backward compatibility, both of which are vital in a rapidly changing environment. The High Performance Fortran (HPF) development effort is the most visible example. HPF is intended to provide data-parallel abstractions that can be efficiently targeted to a variety of high-performance computer architectures [High Performance Fortran Forum 93].

The central abstraction in HPF is the array, although it also provides derived type facilities for defining new data abstractions. Arrays are sophisticated entities, supporting array section manipulation, vector operations, and parallel control structures. To control computational loads and communication costs amongst processors, an HPF programmer can use compiler directives to manipulate the partitioning of arrays across processors. However, directives are restricted to combinations of block and cyclic partitions.

The scientific computing community has found HPF lacking in many ways. For one, HPF's definition of data parallelism is somewhat narrow. Programs with data that is sparse, irregular, or evolving in structure cannot be adequately optimized using the existing array abstraction and its directives. Since these issues are just beginning to be understood, supporting them in HPF has been deemed premature.

Another problem is HPF's focus on data parallelism, since achieving the best performance requires exploiting high-level task parallelism and low-level instruction parallelism as well. High-level task parallelism arises in large calculations such as long-term weather prediction, which can use two major tasks to calculate atmospheric and ocean effects [Mechoso et al. 94]. To address the need for task parallelism, the HPF community has considered the addition of mechanisms such as MPI [Message-Passing Interface Standard 95]. Instruction-level parallelism is successfully exploited in sequential instruction streams by today's RISC processors. However, naive data parallelization can destroy instruction-level parallelism by disturbing spatial and temporal locality in the sequential instruction stream. There is no direct support for instruction-level parallelism in HPF, although there is an "escape mechanism" called the `HPF_LOCAL` extrinsic environment, which allows a programmer to work directly with the code and array partitions residing on an individual processor. Consequently, a programmer can use `HPF_LOCAL` to call sequential routines that have been hand-optimized to make the best use of the memory hierarchy and instruction-level parallelism. However, because of the way `HPF_LOCAL` works, any change to data partition directives—which normally affect only performance—can require non-trivial changes to `HPF_LOCAL` code. As a result, performance tuning and retargeting can be hampered.

The bigger problem facing HPF, however, is that the nature of the problems being solved and the computational methods being used are changing so rapidly that a widely acceptable language definition has proved elusive. The HPF community has frequently discussed freezing the language definition, but has been sensitive to losing some important subcommunity's support or perhaps failing altogether. However, without a stable language definition, a stable compiler of commercial value is not possible. To compound the problem, like other technology, the architectures of high-performance platforms are changing so rapidly that timely retargeting of a highly optimizing compiler for a large language like HPF may not be possible.

The application-specific library approach. Many computational scientists have responded to language inadequacies by constructing special-purpose computational libraries. These libraries are developed in a traditional programming language extended with a vendor-specific message passing interface. In a rapidly changing environment, libraries have the advantage of being very high-level and easily retargetable. Just as importantly, their special-purpose nature allows the use of application knowledge to design highly optimized abstractions. Libraries also have proved to be easier to enhance with new algorithms and technology than compilers. Moreover, these libraries are usually available as source code, so they can be tailored by a sophisticated user of the library to meet unanticipated needs. Additionally, many of the most popular libraries have evolved an unusual but common set of structural features—including open layering, limited interoperability, and a separate performance-tuning interface—that can help programmers accommodate the changes driving the development of their software. We briefly discuss each in turn.

These libraries have developed a layered structure for a number of reasons besides layering's support for incremental development. For one, the generic features of a library can be used to develop a variety of application-specific interfaces. For instance, consider KeLP, a C++ class library that manages data layout and data motion for dynamic irregular block-structured applications [Fink et al. 96]. KeLP (and its predecessor LPARX [Baden & Kohn 95]) not only provide a layer for the solution of dynamic irregular problems, but also provide specialized layers for adaptive finite difference codes, particle problems, and HPF-like data decompositions. By distinguishing between the generic and specialized abstractions, new application-specific interfaces can be introduced as new computational methods are discovered. In a slightly different vein, not all application programmers can get adequate performance at the highest level of abstraction, so sometimes they desire to program parts of their codes at a lower layer to get the control they need. However, as with using `HPF_LOCAL`, circumventing the top-level abstractions can violate guarantees of useful invariants such as the consistency of data.

Another benefit of layering in these systems is retargetability. The lowest layer of these layered systems is typically a machine-dependent layer that resolves basic architectural issues such as communication, synchronization, and memory management. These libraries can be retargeted by reprogramming this layer only. Success of this approach, however, requires not only hiding the syntax of the architectural details, but providing the best possible performance of the features provided by the layer. If the lowest layer's interface cannot be adequately matched to the underlying architecture, then the performance of the entire library will suffer. Historically, highly application-specific approaches such as the BLAS and BLACS linear algebra libraries have been the most successful [Dongarra et al. 90, Dongarra et al. 93]

These libraries also support some basic form of interoperability, which is a necessity because the libraries are so specialized. For instance, because KeLP focuses on managing distributed data and interprocessor communication, it provides no special support for serial numeric computation. The intention is that a programmer provides finely tuned Fortran numeric kernels for the actual computations. Since these kernels already exist and are expensive to develop, it is more valuable for KeLP to support interoperability than kernel development. However, inter-library interoperability has not occurred, in large part because each library makes substantial assumptions about the formatting of data and its presentation in abstractions. Some of these differences are essential properties of the underlying algorithms and their assumed usage, others are artifacts of expedience.

Another property shared by many libraries is support for directly managing performance. A typical numeric problem has several algorithmic solutions, each suited to a particular pattern in the data, such as regularity or sparseness. Ideally, there is not a separate abstraction for each kind of data pattern, but rather a separate interface that informs the library what kind of data to expect [Kiczales 94]. Alternatively, the library might actually sense the kind of data at load time [Brewer 95] or even adapt during execution (e.g., by performing load balancing). However, a library such as ScaLAPACK [Dongarra & Walker 95] blends the performance interface into the functional interface in part because it is implemented in FORTRAN, which lacks adequate abstraction facilities.

These libraries have been successful, but they all lack the ability to detect when their abstractions are used in a stylized way that can be optimized. Instead, the library user must call special "composite" routines that encapsulate that stylized usage, or may even have to extend the library to provide such routines as needed [Stankovic 82]. Moreover, neither current libraries nor languages sense changes to the computational environment during execution.[1] When running in a globally shared computing infrastructure, such sensing is necessary in order to manage load balance and communications costs in the presence of contention for resources.

3 Solution for a Chaotic Planet: Meta-Minimal Languages

We observe, then, that high-performance programming languages and layered libraries each have their distinct advantages, but also disadvantages that must be overcome in order to support both high performance and high flexibility at low cost. In particular, languages lack the flexibility of layered libraries, and layered libraries lack the optimization capabilities of languages. Optimization itself is complicated by the rapid

[1]Some load balancing mechanisms can indirectly react to some changes, however.

evolution of computer architectures, and programs must now be able to optimize on the fly to adapt to contention in the computing environment. Interoperability is also a pervasive concern.

However, we see a trend that suggests that the needs of the high-performance programming community will be met by combining the best features of programming languages and layered libraries. In particular, we believe that the community can benefit from a small, stable programming language with abstraction features that support the development of self-tuning, optimizing, easily adaptable, integrable layered systems. The language must be *small* in order to facilitate rapid retargeting of its optimizing compiler to new high-performance systems. The language must provide *sophisticated abstraction facilities* so that the required performance interfaces and self-adapting data types can be defined. To provide the best performance, it must also allow *uses of the library to be optimized* just like the primitive features of the language. Moreover, we believe that these facilities will blur the distinction between programming, compiling, and executing an application. In particular, the sharing of information amongst these activities may be significantly enhanced. Finally, if history is a good teacher, a migration path for old code helps retain software assets and facilitates adoption of the new infrastructure.

The implication, then, is that there may be no standard high-performance programming language *per se*—only a language for defining high-level high-performance libraries. Standard languages of a sort—that is, these libraries—will naturally emerge as coalitions of library developers and users facing the same problem. We now highlight just a few of the key features of a language for defining libraries, unfortunately omitting discussion of issues such as library interoperability, due to space limitations.

Metalanguage features. Emerging languages are developing techniques to support our proposed model. For example, OpenC++ and MPC++ offer abstraction facilities through a meta-object protocol (MOP) that support the addition of new language features along with optimization directives for those features [Chiba 96, Ishikawa et al. 96].[2] Optimization directives are represented as feature usage patterns that, when matched, direct the compiler to substitute specialized implementations of the appropriate library features.

As observed with HPF and several libraries, performance interfaces in the form of directives or tuning parameters are a valuable mechanism for improving performance when an optimizer cannot. OpenC++ and MPC++ can support the development of performance interfaces and even first-class performance objects. And unlike earlier approaches using MOP's, these are compile-time mechanisms, and so do not necessarily incur runtime overhead.

Using a specification language to specify algebraic properties of a layer is an alternative mechanism for supporting layer-level optimization [Vandevoorde & Guttag 94]. This technique can also be used to provide layer-level static semantic checking [Evans et al. 94]. However, a specification-based approach is probably not sufficiently powerful to support the definition of performance interfaces.

Perhaps the greatest challenge facing current metalanguage efforts will be keeping the language small and stable. After ten years of standardization effort, C++ is still undergoing radical and somewhat expansive change, frustrating compiler writers, library developers, and application specialists.

Layering. A critical feature lacking in current and emerging languages is explicit support for layers. Module features can be used to define layers, but classes are not appropriate, as a layer is not a unified data abstraction. Rather, a layer is a virtual machine—that is, a complementary combination of data abstractions, control structures, performance parameters, event handlers, and the like. It also has been observed that layers and information hiding modules serve different purposes, and that one structure does not necessarily nest inside the other [Habermann et al. 76, Parnas 79, Griswold & Notkin 95]. If these distinctions are ignored, software can become difficult to evolve and performance can suffer. These issues become more important when the users of a layered system are not the developers, as any implicit layered structure is prone

[2]The developers of OpenC++ specifically mention the importance of their mechanism in library development. MPC++ actually provides a more general mechanism than a MOP to support extension of procedural language features.

to be misunderstood, leading to errors and performance problems. Moreover, a mechanism for library-level optimization could benefit from layering abstractions that could be named and manipulated.

Event support. Our new language's runtime system must also support the integration of sensing capabilities to adapt the program's computational strategies to the environment. For instance, the Network Weather Service (NWS) is a runtime mechanism that tracks network, processor, and memory contention in a globally shared system so that a parallel distributed application can make appropriate task allocation (and perhaps migration) decisions [Berman et al. 96]. A straightforward application of data abstraction in order to integrate these facilities into a program is unwieldy, as an application must essentially poll the NWS and change its allocation strategy periodically. A more proactive abstraction would allow the NWS to asynchronously change the allocation strategy without the application's intervention. However, some sort of event mechanism is required for asynchronously propagating information upward from the network to the scheduler. Such mechanisms are rare in languages, and usually lack the flexibility required by modern systems [Sullivan & Notkin 92, Notkin et al. 93].

Providing data flow information to the runtime system. Compile-time optimization is limited to making static decisions about what type of input data to expect for the application. On the other hand, runtime systems like the NWS often lack direct knowledge of the structure of the application itself. However, it could be beneficial if our new language could save data flow information for the runtime system [Grimshaw et al. 93, Grunwald & Vajracharya 94]. Consider, for example, a runtime system attempting to dynamically load-balance a performance-critical application. When it discovers a load imbalance, it attempts to migrate a task from an overloaded processor to an underutilized processor. But which task? Preferably one whose movement will not only improve load balance, but also hold down communication costs. Such communication costs can be estimated from information culled by the compiler.

4 Conclusion

Because our rapidly changing world has sent us on a quest for application-level programming with maximum performance, intermediary agents—library programmers—are developing new application abstractions incorporating the newest technology. Consequently, the next-millennium programming language should be designed for library developers, not application programmers. Because it is impossible to anticipate the particular needs of application developers, programming languages should make few design commitments, yet provide abstraction and runtime facilities so that library programmers have the ability to make those commitments at will. In some respects, however, the stability problem has simply been moved from language and compiler developers to library developers. Semantically informed programming tools will prove valuable in evolving both libraries and user code [Griswold & Notkin 93, Chow & Notkin 96].

References

[Baden & Kohn 95] S. B. Baden and S. R. Kohn. Portable parallel programming of numerical problems under the LPAR system. *Journal of Parallel and Distributed Computing*, pages 38–55, May 1995.

[Berman et al. 96] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. 1996.

[Brewer 95] E. A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, July 1995.

[Chiba 96] S. Chiba. A metaobject protocol for C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 285–299, October 1996. SIGPLAN Notices 30(10).

[Chow & Notkin 96] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. Technical Report UW-CSE 96-03-01, Department of Computer Science and Engineering, University of Washington, 1996. to appear in ICSM '96.

[Dongarra & Walker 95] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, June 1995.

[Dongarra et al. 90] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[Dongarra et al. 93] J. J. Dongarra, R. A. van de Geijn, and R. C. Whaley. A users' guide to the BLACS. Manuscript. Department of Computer Science, University of Tennessee, Knoxville, TN 37996., 1993.

[Evans et al. 94] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: a tool for using specifications to check code. In *ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, December 1994.

[Fink et al. 96] S. J. Fink, S. B. Baden, and S. R. Kohn. Flexible communication mechanisms for dynamic structured applications. to appear in IRREGULAR '96, August 1996.

[Grimshaw et al. 93] A. S. Grimshaw, J. B. Weissman, and T. Strayer. Portable run-time support for dynamic object-oriented parallel processing. Technical Report CS-93-40, University of Virginia: Department of Computer Science, July 1993.

[Griswold & Notkin 93] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[Griswold & Notkin 95] W. G. Griswold and D. Notkin. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Transactions on Software Engineering*, 21(4):275–287, 1995.

[Grunwald & Vajracharya 94] D. Grunwald and S. Vajracharya. The DUDE runtime system: An object-oriented macro-dataflow approach to integrated task and object parallelism. Technical Report CU-CS-779-95, Dept. of Computer Science, University of Colorado, 1994.

[Habermann et al. 76] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, May 1976.

[High Performance Fortran Forum 93] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1993.

[Ishikawa et al. 96] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and implementation of metalevel architecture in C++ – MPC++ approach. In *Proceedings of the Reflection '96 Conference*, April 1996.

[Kiczales 94] G. Kiczales. Why black boxes are so hard to reuse: A new approach to abstraction for the engineering of software. Videotape. Selections from OOPSLA '94, 1994. See also `http://www.xerox.com/PARC/spl/eca/oi.html`.

[Mechoso et al. 94] C. R. Mechoso, J. D. Farrara, and J. A. Spahr. Running a climate model in a heterogeneous, distributed computer environment. In *Proceedings of the Third IEEE International Symposium on High Performance Distributed Computing*, pages 79–84, August 1994.

[Message-Passing Interface Standard 95] Message-Passing Interface Standard. MPI: A message-passing interface standard. University of Tennessee, Knoxville, TN, June 1995.

[Notkin et al. 93] D. Notkin, D. Garlan, W. G. Griswold, and K. Sullivan. Adding Implicit Invocation to Languages: Three Approaches. In *Object Technologies for Advanced Software*, volume 742 of *Lecture Notes in Computer Science*, pages 489–510. First JSSST International Symposium, November 1993.

[Parnas 79] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, March 1979.

[Stankovic 82] J. Stankovic. Good system structure features: Their complexity and execution time cost. *IEEE Transactions on Software Engineering*, SE-8(4):306–318, July 1982.

[Sullivan & Notkin 92] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[Vandevoorde & Guttag 94] M. T. Vandevoorde and J. V. Guttag. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 121–127, December 1994.