SOFTWARE ISSUES IN HIGH-PERFORMANCE COMPUTING AND A FRAMEWORK FOR THE DEVELOPMENT OF HPC APPLICATIONS

PETER H. MILLS, LARS S. NYLAND, JAN F. PRINS, AND JOHN H. REIF

ABSTRACT. We identify the following key problems faced by HPC software: (1) the large gap between HPC design and implementation models in application development, (2) achieving high performance for a single application on different HPC platforms, and (3) accommodating constant changes in both problem specification and target architecture as computational methods and architectures evolve.

To attack these problems, we suggest an application development methodology in which high-level architecture-independent specifications are elaborated, through an iterative refinement process which introduces architectural detail, into a form which can be translated to efficient low-level architecture-specific programming notations. A tree-structured development process permits multiple architectures to be targeted with implementation strategies appropriate to each architecture, and also provides a systematic means to accommodate changes in specification and target architecture.

We describe the *Proteus* system, an application development system based on a wide-spectrum programming notation coupled with a notion of program refinement. This system supports the above development methodology via: (1) the construction of the specification and the successive designs in a uniform notation, which can be interpreted to provide early feedback on functionality and performance, (2) migration of the design towards specific architectures using formal methods of program refinement, (3) techniques for performance assessment in which the computational model varies with the level of refinement, and (4) the automatic translation of suitably refined programs to low-level parallel virtual machine codes for efficient execution.

1. HPC Software Issues

While large scale parallel processors have greatly increased the performance potential for HPC, they have also introduced substantial new software development problems. We identify three problems that we see as the largest obstacles to the development of HPC software.

Currently, architecture-specific notations are largely used to program parallel machines. These low-level notations reflect specific features of a target architecture such as shared vs. distributed memory, SIMD vs. MIMD control organization, and different forms of memory and communication locality.

This work was supported by ARPA via ONR contracts N00014-91-J-1985 and N00014-92-C-0182, and by Rome Labs Contract F30602-94-C-0037.

Although such low-level notations are needed to provide the detailed access to the machinery necessary to orchestrate high performance, they are not well suited to sustaining a large design and development activity. They are unable to generalize the expression of concurrency with the consequence that each software development step involves large and tedious low-level programming effort whose effect may be difficult to analyze. On the other hand, higher-level parallel computing models that might be more suitable for the development of parallel software tend to rely on generalizations of concurrency that are unrealistic or inaccurate with respect to actual machine performance. Such models can easily lead to designs that are completely impractical.

Thus the first problem is that in order to construct complex parallel applications that achieve high performance, developers must bridge the gap between these two levels of expression in some fashion.

The second problem has become more apparent as we enter the second or even third generation of parallel computers: different architectures and different-sized machines are now available to researchers and hence existing applications need retargeting in order to "track" the HPC revolution. However, the low-level notations in which applications have been developed lack portability between architectures. It is not a matter of simple translation between notations: the effect of the target architecture can be pervasive. At a high level, different architectures may require fundamentally different algorithms to achieve optimal performance; at a low level, overall performance exhibits great sensitivity to changes in communication topology and memory hierarchy. The retargeting problems involved are sufficiently complex that automatic translation and optimization are unlikely to offer a comprehensive solution.

Consider, for example, a molecular dynamics simulation package with which we have experience (the Cedar system, developed by J. Hermans at UNC-CH). Molecular dynamics simulation is a grand-challenge problem, of great importance to areas such as drug-design. The molecular dynamics package in question was originally developed to run on Cray computers. It was subsequently adapted for use on a number of other HPC platforms including mini-supercomputers, high-performance workstations, the MasPar Computer family, the Kendall Square family, and workstation clusters supporting the PVM services. Most of these versions remain important because users of the Cedar software at different sites have access to different kinds and sizes of HPC machines. Fundamentally different algorithms are involved in all of these versions to achieve the best performance. This is a result not just of architectural differences but also of the degree of parallelism sought relative to the problem size.

The third problem emerges when we consider that a scientific application such as the Cedar system is continuously evolving as new scientific and algorithmic ideas need to be incorporated. In this case small changes in the functional specification can lead to large and very different program changes in each of the architecture-specific implementations. The maintenance of all these implementations quickly becomes intractable with the result that some architecture-specific versions become scientifically obsolete while other scientifically current versions are not able to take advantage of the full range of HPC resources. We see similar problems appearing in the development and maintenance commercial and military applications.

Thus, to summarize, the key problems we see in the development of HPC software

are:

- bridging the gap between high-level parallel design models and low-level execution models,
- targeting a single application to multiple HPC platforms, and
- managing evolution in both the application and the target architectures.

We believe that high level programming models and notations are critical to the expression and exploration of complex designs. They also promote portability across architectures (or at least make explicit at a higher level the design differences for different architectures). However, for a programming methodology based on high-level notations to be practical it must eventually be connected to the kinds of detailed notations that access the lower-level performance issues. We believe that it is a mistake to use only high-level or only low-level models and notation; a useful framework must accommodate both views.

The growing crisis in software version management as high performance applications evolve and are ported to a variety of HPC architectures over their lifetime also suggests that different views of the development are needed: a high-level view is preferred for changes in specification, while lower-level views are more appropriate for architecture and machine changes.

The remainder of this paper is organized as follows. In Section 2 we outline a software development methodology that addresses these problems by providing for the architectural specialization of high-level designs couched in a single wide-spectrum notation, the exploitation of parallel virtual machines as translation targets, and the early assessment of prototypes using a hierarchy of parallel computational models matched to the level of design. In section 3 we describe the *Proteus* system, an effort under way within our group to support the software development methodology. Section 4 contains a brief overview of related work. We conclude the paper with a discussion of requirements for realizing the above framework.

2. A Refinement-Based Approach to HPC Software Development

To address the problems raised in the previous section, we propose a refinement-based development methodology. Informally, by *refinement* we mean the inclusion of additional detail. The approach starts with a specification that is initially refined into a high-level architecture independent design and from there successively brought closer to a target architecture through refinement steps which incrementally incorporate architectural details. The most refined version corresponds directly to an implementation in a low-level architecture-specific notation.

2.1. Tree structured program development. We represent the successive versions developed as nodes in a graph and add an edge between versions when one is developed from another via refinement, as shown in Figure 1. A set of architecture dependent implementations of a single problem can be constructed using a tree structured graph. At internal nodes of the tree, each different refinement reflects a design decisions that essentially directs the development toward one or a group of specific architectures (e.g. shared memory or distributed memory machines).

If we express the refinement steps in a formal manner, for example as program transformations, then there is the possibility of applying these transformations

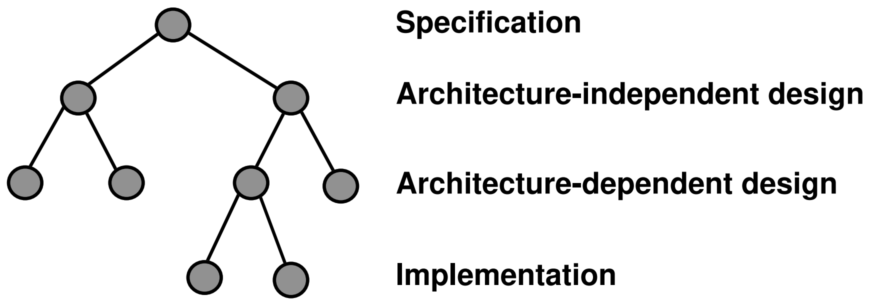

FIGURE 1. Program development tree

automatically in the form of development "tactics", as is done in program synthesis systems such as KIDS [22], and the CIP system [19].

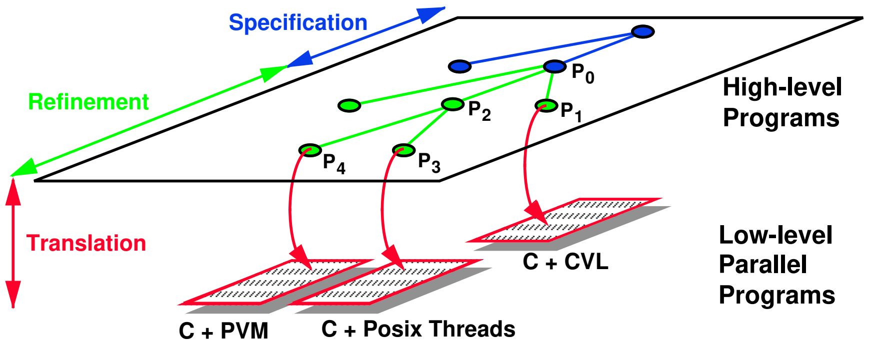

FIGURE 2. Development tree with translations to parallel implementations

An automated approach can be particularly important in this setting because there are more versions to develop from a specification than is typical in the synthesis of a conventional (sequential) application. Targeting a new architecture requires the addition of a new sequence of refinements starting from an appropriate level of design to an architecture-specific implementation. Generating a new set of implementations following a change to the high-level specification (for example, adding a new component in a simulation), in principle requires that we "replay" all refinement steps starting with the new specification to generate a new tree of versions that incorporates the changes for each of the targeted architectures.

Of course, automated program synthesis technology and design capture are research areas that require a great deal of additional development to be generally applicable. A more pragmatic view is that a tree structured development is an organizational concept and that the actual synthesis, refinement and replay steps are conducted using a mixture of manual and automatic techniques. In particular, it is likely that automated techniques may apply only near the leaves of the development tree.

2.2. Wide-spectrum concurrent programming notation. We believe that the best way to support the capabilities described is to represent *all* versions in the development tree using a single wide-spectrum concurrent programming language.

Such a language, by unifying the various parallel programming paradigms, would both be able to capture concurrency in an abstract high-level fashion and to provide a uniform vehicle for refinement towards particular architectures demanding various paradigms such as data and process parallelism.

Using such a language, prototypes can be constructed at an architecture-independent level and evaluated using an interpreter and tools. Refinement of such prototypes consists of program modifications or transformations that result in restrictions in the use of the concurrency constructs. Such restrictions express the adaptation of a high-level design to constructs efficiently supported on a specific architecture. Since the resulting program would still be in the same notation, the interpreter can again be used to assess its functionality and some performance measures. Programs that are suitably refined should be automatically translatable to efficient parallel programs in low-level architecture-specific notations and run directly on the targeted parallel machines.

2.3. Low-level parallel virtual machines. The final translation techniques can gain wider applicability by targeting low-level parallel virtual machines that are efficiently implemented on classes of parallel architectures, rather than machine-specific languages. For example the C language along with libraries such as the vector library CVL [1], PVM [11] or POSIX threads might be appropriate as low-level parallel virtual machine targets.

2.4. Software development process. Figure 2 illustrates the development process we have in mind. Starting with an initial specification, programs are successively transformed to incorporate specification changes, to restrict the expression of concurrency and to translate versions to architecture-specific low-level notations. We thus differentiate transformation steps into

- *elaborations*, which alter the meaning of a specification,
- *refinements*, which preserve the meaning of the specification but narrow the choices for execution, for example by restricting the form of concurrency employed, and
- *translations*, which convert the program from the high level notation to a low-level notation.

In Figure 2 an initial executable specification P_0 is developed (after some elaboration). This formulation may or may not include any explicit concurrency. Here we assume P_0 includes only implicit data parallelism.

Several refinement paths are shown. Each refinement is simply a rewriting of the program text that preserves meaning but changes the detailed form of the program and the constructs used. For example, the refinement from P_0 to P_1 might restrict data-parallel expressions so that the resulting program is translatable to the CVL model. Alternatively, the refinement from P_0 to P_2 introduces explicit concurrency in the program. In version P_3 this concurrency is expressed in a very general form from which it is not possible to determine communication points. For this version, a C program spawning Posix threads and using semaphores to synchronize would have to generated. P_4 is an alternate refinement of P_2 in which the form of the explicit concurrency is sufficiently restricted that all communication can be statically identified and a PVM program can be generated from P_4. If P_4 also includes a component that uses data-parallelism, that component can be translated to CVL. In general, a program version can translate to any number of program segments in different virtual machine models.

These refinement operations could be accomplished by manual rewriting or by a semi-automatic program transformation system. As one gains insight into the principles of these refinements, it may be possible to express them in the form of tactics or to incorporate them into the automated translations.

2.5. Prototyping and evaluation of designs. It is critical to this development methodology that the intermediate versions in a development can be assessed in some form. Through the use of an interpreter for the high-level notation, all versions can be executed, so that some empirical measurement of functional and performance characteristics is possible. In general it is unlikely that efficient parallel execution could be achieved for high-level specifications, and in fact it may well be that the only way to execute such versions is by a sequential simulation performed by the interpreter. Even with this limitation important performance and functionality measurements could still be obtained, e.g. total work performed by a parallel algorithm and its load distribution under some particular data decomposition.

The execution capability would support the rapid prototyping of designs and permit the exploration of a large and complex space of alternatives in which significant design trade-offs exist. There is extensive evidence in many engineering domains, including both the software and hardware, that information obtained by disciplined experimentation with prototypes reduces risk and improves productivity. In the domain of parallel computation where design principles are not well understood, the knowledge acquired from prototyping can be particularly valuable.

By refining prototypes toward specific implementations rather than throwing them away, we improve the ability to carry information from the prototype into implementation.

2.6. Performance prediction. A significant aid in the design of efficient parallel programs is the ability to predict the performance on actual parallel machines, allowing the early assessment of algorithmic variations without the cost of full implementation. Performance analysis here encompasses both empirical measurement of performance (e.g. through simulation) as well as static estimation of complexity measures of time and resource utilization. For both these cases the measurements of behavior are defined in terms of a mathematical model of a computing machine. Indeed, parallel computation models which underlie both static and dynamic performance analysis give an operational semantics to programs which provides an intuitive framework that guides the very design of the algorithm. In this sense there is little technical distinction between formal models of computation and what are typically termed programming models. Thus it is not surprising that it remains difficult to assess parallel program performance for many of the same reasons it is difficult to construct efficient and portable parallel programs in the first place – the gap between high-level models and diverse low-level machines hinders the the accuracy of performance estimation. To combat this problem we propose a refinement-based approach to performance prediction in which the computing model, chosen from a hierarchy of recently developed more detailed models, matches the level of program refinement.

Historically, the PRAM is the most widely used parallel model. However, for current parallel machines, the PRAM is often inaccurate in predicting the actual running time of programs since it hides details which impact performance such as the time required for network communication and synchronization as well as issues of asynchrony and memory hierarchy. For example, this model does not reflect the current trend toward larger-grained asynchronous MIMD machines whose processors each may have their own sophisticated memory hierarchies and which communicate over relatively slow networks. This necessitates (1) a pragmatic refinement of parallel machine models, that is, the development of models which incorporate realistic aspects such as communication costs and memory hierarchy while still remaining abstract enough to be machine-independent and amenable to reasoning, and (2) the practical application of these theoretical models to performance analysis, that is, the development of better techniques and tools for performance prediction.

Models and resource metrics for parallel computation. In response to the first need there have been proposed a variety of models which extend the PRAM to incorporate realistic aspects such as *asynchrony* of processes (e.g., the APRAM [9]), *communication costs*, such as network latency and bandwidth restrictions (e.g., the LogP model [10]), and *memory hierarchy*, reflecting the effects of multileveled memory such as differing access times for registers, local cache, main memory and disk I/O (e.g., the P-HMM [23]). The most prevalent and promising recent models are *parameterized* (or generic) models, which abstract the architectural details into several generic parameters which we call *resource metrics*. Typical resource metrics include the number of processors, communication latency, bandwidth, block transfer capability, network topology, memory hierarchy, memory access method and degree of asynchrony. Using such a parameterized model one can design broadly applicable parameterized algorithms that can be tailored to specific machines by instantiating the parameters, such as latency and bandwidth, to match machine characteristics.

Refinement of models. We argue for an approach to performance analysis that in answer to the second need – practical application of recent refined models – allies performance assessment with the incremental refinement of design. Our approach for performance prediction is based on (1) the use of increasingly detailed models as the program refinement progresses, gaining accuracy and confidence as development progresses, (2) the use of different models for analysis of code segments following different paradigms, such as data-parallelism and message-passing, to support the assessment of multi-paradigm programs, and (3) the extension of an already emerging hierarchy of refined models as needed to support the above goals, following principles derived from a careful examination of key issues in the design of models of parallel computation.

The key notion in the first point is that the computing model used for assessment varies with the level of refinement. At each point in the stepwise refinement the design can be assessed; the accuracy of assessment increases with the level of architectural detail incorporated into the design and the correspondingly more detailed model used for analysis. Moreover, in terms of resource metrics, the model should "fit" the refined program not just in level of detail but also in the choice of resource metrics with which it approximates machines. As more detailed architectural commitments are made in the specific expression of concurrency (for example incorporating notions of message-passing) models with appropriate resource metrics (for example with notions of latency) can be attached to the program. Thus a hierarchy of models expressing increasingly detailed resource metrics are used which matches the tree-like structure of program refinement.

For example, at the coarsest level performance prediction may be done using the interpreter to derive simple approximations of total work. As the program is refined into data-parallel code one might employ the VRAM vector model [2], or for a shared-memory version a model akin to the APRAM might be used for performance evaluation. As the program is further refined, for example from a shared-memory program to a more sophisticated form which corresponds to message-passing, the LogP network model may be employed to gain more accurate assessment, with suitable instrumentation that identifies low-level units of communication (and work) in order to "attach" the model to the program.

At a further stage in the refinement process it becomes important to model the several layers of memories which exist in many machines, since differing access times to local cache and disk may strongly effect performance. Yet to accommodate these more detailed performance measures, further refinements of parallel models might be required, since a void exists in models that accurately treat both network communication and parallel multi-level memory. As a simple example of the process of developing improved performance models, consider a new hybrid model of parallel computation, the LogP-HMM model [14], which extends a network model (the LogP) with a sequential hierarchical memory model (the HMM). Such a refined model could be instrumented into parallel code through the use of annotations which incorporate explicit details of memory locality. A related approach has been used for cache-coherent shared-memory multiprocessors in the CICO project [13], where annotations serve both for performance prediction and to guide more efficient code generation.

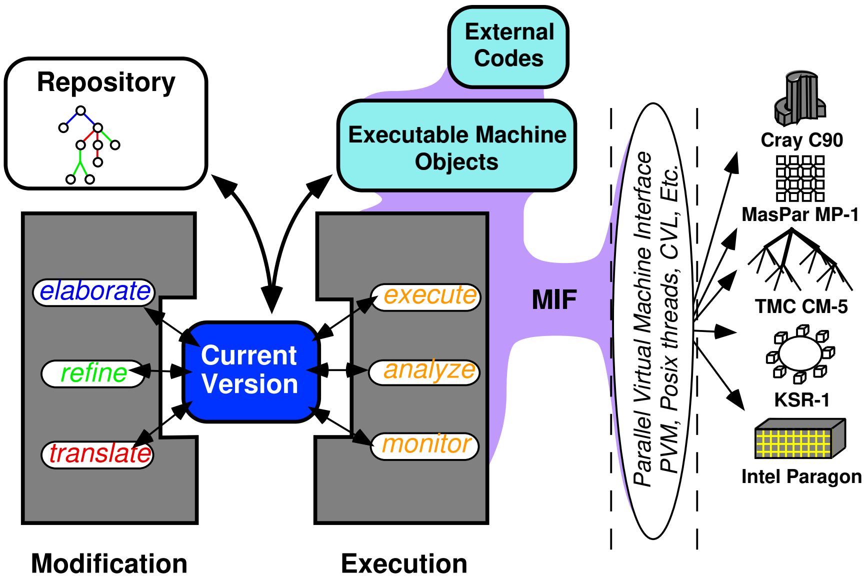

FIGURE 3. Refinement-based framework for software development

3. The Proteus System for the Development of HPC Software

We now briefly describe Proteus, a refinement-based system for parallel software development which embodies the principles earlier presented. The Proteus system is under joint development by Duke University, the University of North Carolina at Chapel Hill, and the Kestrel Institute. Its goal is to provide improved capabilities for exploring the design space of a parallel application using prototypes, and for evolving a prototype into a highly-specialized and efficient parallel implementation.

The Proteus system, illustrated in Figure 3, comprises:

- a wide-spectrum parallel programming notation that allows high-level expression of specifications,
- a methodology for (semi-automatic) refinement of architecture-independent prototypes to lower-level programs optimized for specific architectures, followed by translation to portable intermediate languages,
- an execution system consisting of an interpreter, a Module Interconnection Facility (MIF) allowing interoperability of Proteus with other programming languages and run-time analysis tools, and
- a methodology for prototype performance evaluation integrating both dynamic (experimental) and static (analytical) techniques with models matched to the level of refinement.

We believe that, in the absence of both standard models for parallel computing and adequate compilers, this approach gives the greatest hope of producing useful applications for today's computers. It allows the programmer to balance execution speed against portability and ease of development.

3.1. The Proteus language. The Proteus language is an imperative language that provides high-level notations for expressing several fundamental forms of parallelism including implicit concurrency found in data-parallel expressions, and explicit concurrency in the form of tasks and controlled access to shared state.

Data parallel operations are expressed using the familiar mathematical notations of set, sequence, and map comprehension. The ability to specify irregular and nested data parallelism is a natural consequence of providing nested aggregate data types such as sequences of sequences, sets of sets, etc. Like SETL, many of the powerful and flexible mathematical types are predefined in Proteus. Additional user-defined data types may be specified algebraically in Proteus and packaged as parameterized theories – parameterization both generalizes polymorphism and enhances reusability.

Process (or task) parallel computations can also be succinctly expressed with a small set of process creation and synchronization primitives similar to those adopted in recent languages such as PCN [7], CC++ [6], and COOL [5]. In particular, communication is through a shared object model in which the access to shared state is controlled through object methods and class directives which constrain mutual exclusion of methods [16]. Predefined classes such as for single-assignment objects which synchronize a producer with a consumer [12], together with provisions for private state with barrier synchronization [17], allow the expression of a wide range of parallel computing paradigms.

3.2. Transformation of data and process parallelism. Proteus data-parallel expressions in a functional subset involving nested sequence datatypes can currently be transformed and translated to C with vector operations (CVL) using the Kestrel Data-Type Refinement System (DTRE3) [20]. This includes irregular and nested data-parallel expressions (as found in the parallel application of a function to each of a collection of argument sequences of differing lengths), recursive parallel computations (as found for example in divide-and-conquer algorithms), and high-order parallel function application (as found in the parallel reduction of a sequence of values using an arbitrary function).

The resultant CVL program can be efficiently executed on diverse parallel machines such as the Cray C-90, the TMC CM-5, and the MasPar MP-2. Translation of process parallelism is under development.

3.3. Performance prediction and measurement in Proteus. The Proteus interpreter currently provides a rudimentary per-process clock that measures computational steps. This, in conjunction with explicit instrumentation of Proteus code is, used to develop resource requirement measures and to predict performance. Support for multiple performance-prediction models is under investigation.

3.4. Applications of the Proteus system. Several small demonstrations and larger driving problems have been used to assess and validate our technical approach, focusing on such aspects as the prototyping process and methodology, the expressiveness of the Proteus language, and the effectiveness of the Proteus tools.

One demonstration problem involves the prototyping and implementation of algorithmic variants of the Fast Multipole Algorithm (FMA) for N-body simulation. These algorithms promise performance and accuracy advantages for computationally challenging problems such as molecular dynamics simulations, yet are complex and time-consuming to implement. The FMA has many variants which generate a design space which is not well understood. The goal of our experiments with Proteus has been to explore this space. Our experiments have identified new adaptive problem decompositions that yield good performance even in complex settings where bodies are not uniformly distributed [18].

Further descriptions of the language, implementation, and demonstrations are available from the Proteus WWW information server at
`http://www.cs.unc.edu/proteus.html`.

4. Related Work

There are a variety of efforts which seek to address the problem of parallel software development through high-level languages capable of expressing programs executable on a broad range of parallel architectures. These efforts may be distinguished in the approach they take to dealing with the tradeoffs of expressiveness, efficiency, and sophistication of compilation strategies. For example, some approaches restrict the forms of concurrency usable in order to achieve good performance on all platforms. This is the approach of High-Performance Fortran (HPF), for example, which is limited to flat data-parallelism, and might be said to also characterize the current intent of HPC++. However, by restricting the notation to expressing only what is readily compilable, a measure of expressiveness, for example nested data-parallelism, may be sacrificed.

Other languages attempt to provide a higher level of expression, but then face difficulty in achieving good performance because of very general programming model that can not take full advantage of architectures. This might be said to characterize some coordination languages with simple but widely translatable logical models such as the distributed data structures of Linda [4]. In this camp might also be said to fall several functional (or equational) languages. The parallelism is typically implicit and is primarily data-parallelism. For example, a notable effort in this area is NESL (Nested Sequence Language) [3], a data-parallel language that supports the expression of nested data parallelism and is compiled to a widely implemented lower-level vector language VCODE. Id and SISAL are other functional languages which employ a single-assignment property to enforce determinate behavior. Fairly sophisticated translation strategies are used in these cases to bridge the gap from high-level language to machine and so achieve a measure of architecture independence.

Several high-level parallel languages rely on transformation from high-level specification to realize efficient execution. Notable efforts include Crystal [8] and variants of the Bird-Meertens functional formalism [21]. Another noteworthy effort is Maude [15], a language based on rewriting logic which can be transformed into a parallel sublanguage (Simple Maude) which can then be compiled. In these cases the refinement steps are justified formally through inference steps or algebraic transformations.

While both Crystal and the Bird-Meertens formalisms pursue a transformational approach in which parallel specifications are refined to parallel programs for a particular class of machine, in both cases the languages are to a degree insufficiently expressive. In the case of Crystal, where concurrency is implied by independence in the equational specification, not only is the equational notation somewhat restrictive but the user must have some knowledge of the architectural mapping in order to guide efficient implementation. The Bird-Meertens formalism is also restrictive as a design notation as it cannot express many forms of parallelism such as process-parallelism.

Although our approach to software development is also based on the use of a high-level language and program transformation, our language is wide-spectrum in order to cover the intended hierarchy of designs. In contrast to other approaches, we rely on manual refinement to bring the design closer to the machine before translation and so bridge a gap that can not be reliably crossed using compilation techniques, thus obtaining a language which simultaneously is expressive and capable of specifying highly efficient concurrent programs.

5. Conclusions

We propose a methodology for parallel software development based on the use of a wide-spectrum concurrent programming language together with refinement as a means of prototyping and evolving initial designs into implementations.

A realization of the above approach entails the development of a framework with the following components, which we suggest forms part of an agenda for HPC software development environments.

(1) wide-spectrum parallel languages which provide a uniform high-level notation that unifies the typically disjoint paradigms of data- and process-parallelism,

(2) formal methods of program transformation which migrate the design towards specific architectures,
(3) parallel virtual machines forming efficient portable execution targets,
(4) models and tools for performance prediction that utilize realistic parallel computational models matched to the level of design.

In our current efforts we have developed the Proteus notation as a candidate wide spectrum concurrent programming language and have constructed a framework integrating manual, automated, and semi-automated transformation of programs to allow exploration of the design space as well as development and maintenance of efficient implementations.

References

1. G. Blelloch, S. Chatterjee, J. Sipelstein, and M. Zahga. CVL: A C vector library. Draft Technical Note, Carnegie Mellon University, December 1990.
2. G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.
3. G. E. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 1993.
4. Micholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.
5. Rohit Chandra, Anoop Gupta, and John Hennessy. Integrating concurrency and data abstraction in the COOL parallel programming language. Technical Report CSL-TR-92-511, Computer Systems Laboratory, Stanford University, Ca., 1992.
6. K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proc. of the 4th Workshop on Parallel Computing and Compilers.* Springer-Verlag, 1992.
7. K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, 1992.
8. Marina C. Chen, Young il Choo, and Jinke Li. Crystal: Theory and pragmatics of generating efficient parallel code. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 7, pages 255–308. ACM Press, 1991.
9. Richard Cole and Ofer Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proc. of the First ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178. ACM Press, 1989.
10. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 1993.
11. Jack Dongarra, G. A. Geist, Robert Manchek, and V. S. Sundaram. Integrated PVM framework supports heterogeneous network computing. *J. Computers in Physics*, 7(2):166–175, 1993.
12. Allen Goldberg, Jan Prins, John Reif, Rik Faith, Zhiyong Li, Peter Mills, Lars Nyland, Dan Palmer, James Riely, and Stephen Westfold. *The Proteus System for the Development of Parallel Applications.* April 1994.
13. James R. Larus, Satish Chandra, and David A. Wood. CICO: A practical shared-memory programming performance model. In Ferrante and Hey, editors, *Portability and Performance for Parallel Processors.* 1994.
14. Zhiyong Li, Peter H. Mills, and John H. Reif. Models and resource metrics for parallel and distributed computation. Technical Report, Department of Computer Science, Duke University, 1994.
15. JoseḾeseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
16. Peter H. Mills. Parallel programming using linear variables. Draft Technical Report, Department of Computer Science, Duke University, 1994.

17. Peter H. Mills, Lars S. Nyland, Jan F. Prins, John H. Reif, and Robert A. Wagner. Prototyping parallel and distributed programs in Proteus. In *Proc. of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 10–19. IEEE, 1991.
18. Lars S. Nyland, Jan F. Prins, and John H. Reif. A data-parallel implementation of the adaptive fast multipole algorithm. In *Proc. of the 1993 DAGS/PC Symposium*, Dartmouth College, Hanover, NH, June 1993.
19. Helmut A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer-Verlag, Berlin, 1990.
20. Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proc. of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–128. ACM, May 1993.
21. D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.
22. Douglas R. Smith. KIDS – a semi-automactic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.
23. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 1993.

(P. Mills, J. Reif) Dept. of Computer Science, Duke University, Durham, N.C. 27708-0129.
E-mail address: {phm,reif}@cs.duke.edu

(L. Nyland, J. Prins) Dept. of Computer Science, University of North Carolina, Chapel Hill, N.C. 27599-3175.
E-mail address: {nyland,prins}@cs.unc.edu