Chapter 5:
The Proteus System for the Development of Parallel Applications

Allen Goldberg, Jan Prins, John Reif

Rickard Faith, Zhiyong Li, Peter Mills, Lars Nyland

Daniel Palmer, JamesRiely, Stephen Westfold

1 Introduction

In recent years technological advances have made the construction of large-scale parallel computers economically attractive. These machines have the potential to provide fast solutions to computationally demanding problems that arise in computational science, real-time control, computer simulation, large database manipulation and other areas. However, applications that exploit this performance potential have been slow to appear; such applications have proved exceptionally difficult to develop and, once written, too often fail to deliver the expected speed-up.

This state of affairs may be explained by the proliferation of parallel architectures and the simultaneous lack of effective high-level architecture-independent programming languages. Parallel applications are currently developed using low-level parallel programming notations that reflect specific features of the target architecture (for example, shared vs. distributed memory, SIMD vs. MIMD, exposed vs. hidden interconnection network). These notations have significant drawbacks:

- they lack portability across architectures, and
- they are too low-level to support the expression and exploration of complex designs.

The problem is fundamental: abstract models of parallel computation lead to unrealistic algorithms, whereas machine-specific models slow development and analysis of even the simplest programs. The effect of the target architecture is pervasive: at a high level, different architectures often 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 result is that the developer of a parallel application must explore a complex and poorly-understood design space that contains significant architecture-dependent trade-offs. Moreover, this algorithmic exploration must be conducted using programming languages that offer low levels of abstraction in their mechanisms for expressing concurrency. While a reasonable solution to these problems is to trade reduced access to architecture-specific performance for improved abstract models of computation, this trade may not always be the right one: the whole point of parallelism, for most applications, is performance.

The goal of the DUNCK (Duke University, University of North Carolina at Chapel Hill and the Kestrel Institute) ProtoTech effort is to provide improved capabilities for

- exploring the design space of a parallel application using prototypes, and
- evolving a prototype into a highly-specialized and efficient parallel implementation.

1.1 Program Development Methodology

Ideally, one would like to write codes at a high level, having a compiler translate the codes to run on a specific parallel machine with acceptable performance. Such a system remains elusive, and given the intractability or undecidability of most of the relevant questions, significant advances will likely be slow in coming. If program optimization cannot be fully automated, the next best thing is a framework for computer assisted optimization. Such a framework should integrate manual, automated, and semi-automated transformation of programs to allow exploration of the design space and evaluation of the alternatives—the level of human interaction being commensurate with the complexity of the code and the state of the art in automated program transformation tools.

In Proteus, we propose such a framework:

- a wide-spectrum parallel programming notation that allows high-level expression of prototypes,
- a methodology for (semi-automatic) refinement of architecture-independent prototypes to lower-level prototypes optimized for specific architectures, followed by translation to executable low-level parallel languages,
- an execution system consisting of an interpreter, a Module Interconnection Facility (MIF) allowing integration of Proteus with other codes, 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.

Our methodology for the development of parallel programs is based on the successive refinement of high-level specifications into efficient architecture-dependent implementations, with early feedback on designs provided through interactive experimentation with executable prototypes. Figure 33 illustrates our development methodology. Here the upper plane consists of Proteus programs, related to one another through modification steps in the development process. The root of the tree is the original specification; this specification may be improved over time by incorporating changes prompted by experimentation with prototypes. In the figure, two “child” specifications are shown, one of which is abandoned at the prototyping stage, perhaps because it reflected an incorrect interpretation of the specification. Working prototypes may not provide acceptable performance, leading to further *refinement* of the prototype. Typically these refinements will use Proteus language constructs in a restricted way so that the code may be directly *translated* to efficient executables for a particular architecture. As different refinements may be required to specialize the program towards different classes of architectures, this gives rise to different paths in the development tree. For translation targets we use *hub languages*, as described in the introduction to this book, which provide portability over a broad classes of architectures. Shown in the picture are three such hub languages based on C: one augmented with vector operations (C + CVL), one augmented with message passing process-parallelism (C + PVM), and the another augmented with shared memory process-parallelism (C + Posix Threads).

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

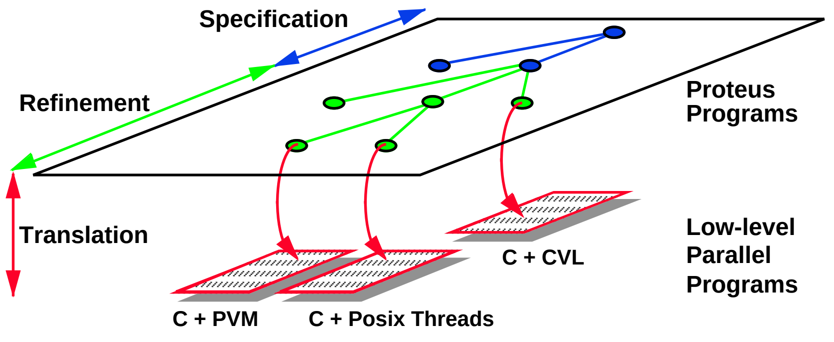


Figure 33 Proteus Program Development Methodology

1.2 Structure of the Chapter

The following six sections discuss many aspects of our work on the Proteus system. Here we briefly discuss the content of each of these sections.

Programming Notation. We describe the Proteus notation, concentrating on language constructs related to parallel execution. The notation allows expression of high-level architecture-independent specifications as well as lower-level specifications targeted for specific architectures. Parallelism is expressed using a small set of expressive constructs: aggregates and iterators for data-parallelism, processes and shared objects for process-parallelism.

Program Development Methodology and Environment. To support the programming methodology outlined above, a set of integrated tools are needed. We describe the current state of tools for Proteus program modification, execution, performance analysis, interoperation, and program version control.

Proof of Concept and Demonstrations. Our development of Proteus has been aided by many experiments that have been conducted to test both the methodology and the tools that have been implemented. Although we have designed the system for broad applicability, most of our experimental work has been in high-performance scientific applications. We have used Proteus to prototype several substantial applications, summarized here.

Refinement System. We describe the system that we use for program transformations. The system has its underpinnings in algebraic specification techniques and category theory.

Translation of Nested Data-Parallelism to Vector Operations. The translation of very high-level data-parallel programs to vector operations that can be executed on a broad class of parallel computers is a key demonstration of the refinement and translation approach that we advocate. Here we give the details on this sophisticated translation technique and report on its performance.

Performance Prediction and Measurement. Techniques for predicting program performance on actual parallel machines are valuable since they allow early assessment of algorithmic variations without the cost of full implementation Here we examine key issues in the design of models of parallel computation used for performance prediction and present an approach for performance prediction in Proteus based on the use of increasingly detailed models as program refinement progresses.

1.3 Availability

Implementations, demonstrations and papers are available from the Proteus World Wide Web information server at `http://www.cs.unc.edu/proteus.html`.

2 Programming Notation

In developing the Proteus programming notation, we began with language features that have proven their value in prototyping sequential programs and extended it to allow expression of concurrency. We have also emphasized the development of a highly interactive language interpreter and the ability to integrate with existing codes.

Proteus is a small imperative language with first-class functions, aggregate data types, and constructs for data and process-parallelism. The language is described in detail in [LN93, FNPP94]. The sequential core of Proteus includes features of proven value in specifying sequential programs. Experience with specification languages such as Z and VDM and prototyping languages such as SETL and APL indicates that an expressive set of predefined aggregate types are a key requirement for rapid, model-based prototyping; Proteus includes two such types, sets and sequences. In addition, the expression sub-language of Proteus is a strict higher-order functional language, allowing many algorithms to be written without resorting to imperative constructs. The sequential portion of the statement sub-language is standard.

This sequential core is extended with a few highly-expressive concurrency constructs, carefully chosen to support programming in most paradigms. We distinguish between two means of expressing concurrency: functional data-parallelism and imperative process-parallelism. *Data-parallelism* refers to the repeated application of a fixed operation to every element of a data aggregate, while *process-parallelism* denotes the parallel composition of two or more distinct processes.

2.1 Data-Parallelism

To support data-parallelism a language must include aggregate values (such as sets or sequences) and the ability to apply a function independently to every element of an aggregate. This sort of expressive capability can be found in a succinct form in the relative comprehension construct of set theory. For example $\{f(x) \mid x \in A\}$ denotes the set of values obtained by evaluating the function f on each element of set A. The potential for concurrency arises from the fact that these evaluations of f are independent and, therefore, may be performed simultaneously.

In set theory arbitrary functions are allowed in comprehensions, including set-valued functions that may themselves allow data-parallelism. Thus if A is the set $\{1, 2, 3\}$ then $\{\{(p,q) \mid (q \in A) \,\&\, (p \geq q)\} \mid \{p \in A\}\}$ denotes the set of sets $\{\{(1,1)\}, \{(2,1), (2,2)\}, \{(3,1), (3,2), (3,3)\}\}$.

Parallel execution of such expressions is termed *nested parallelism* because for each choice of p, there is a “nested” set of choices for q that may be evaluated in parallel. Nested parallelism provides great potential for concurrent evaluation since the number of independent sub-expressions can be very large.

The Proteus *iterator* construct is used to express such parallel execution. For example, if A and B are sequences of length n, then the iterator expression

```
[i in [1..n]: A[i] + B[i] ]
```

specifies the sequence `[A[1]+B[1]`, `A[2]+B[2]`, ..., `A[n]+B[n]]`. Note that unlike comprehensions, the bound variable of an iterator is written first, improving the readability of long expressions. Nested parallelism is specified by nesting iterators.

Nested-parallelism is widely applicable, as we demonstrate here by writing a data-parallel quicksort algorithm. Recall that given a sequence, quicksort works by choosing an arbitrary pivot element and partitioning the sequence into subsequences (`lesser`, `equal`, `greater`) based on the pivot. The algorithm is then applied recursively on the `lesser` and `greater` subsequences, terminating when the sequences are singletons. The final value is obtained by concatenating the (sorted) `lesser`, `equal`, and `greater` lists. In Proteus, this may be coded as follows.

```
function qsort(list)
  return
    if #list <= 1 then list;      — Return, if empty or singleton
    else let
           pivot   = arb(list);
           lesser  = [el in list| el <  pivot: el];
           equal   = [el in list| el == pivot: el];
           greater = [el in list| el >  pivot: el];
           sorted  = [s in [lesser, greater] : qsort(s)];
         in
           sorted[1] ++ equal ++ sorted[2];
```

While there clearly is data-parallelism in the evaluation of the `lesser`, `equal` and `greater`, if that were all the parallelism that were available, then only the largest sub-problems would have any substantial parallelism. The key to this algorithm is that the recursive application of `qsort` is also expressed using an iterator. As a consequence, all applications of `qsort` at a given depth in the recursion can be evaluated simultaneously. In Section 6 we give a general method for transforming such nested-parallel algorithms into programs using simple vector operations.

An important quality of nested sets and sequences (as opposed to arrays) is that they allow irregular collections of values to be directly expressed. In `qsort`, for example, `lesser` and `greater` will likely be of different lengths. Note that this algorithm cannot be so expressed in languages such as High Performance FORTRAN, in which all aggregates must be rectangular.

2.2 Process-Parallelism

Proteus provides a minimal set of constructs for the explicit parallel composition of processes which communicate through shared state. More sophisticated concurrency abstractions, such as buffered communication channels and monitors, may be constructed from these.

Process Creation. Process-parallelism may be specified in two ways. The static parallel composition construct

statement$_1$ `||` *statement*$_2$ `|| ... ||` *statement*$_n$`;`

specifies the process-parallel execution of the *n* statements enumerated. The lifetime of the processes is statically determined; the construct terminates when all component statements have completed. Static process-parallelism may also be specified parametrically using the `forall` construct

`forall` *variable* **`in`** *aggregate-expression* **`do`** *statement* `;`

which may be freely intermixed with the enumerated form, as in the following example.

```
{ forall j in [1..n] do client(j); }  ||  server(n);
```

Dynamic process-parallelism, on the other hand, generates a process whose lifetime is not statically defined. The *spawn* construct

`|>` *statement*

starts asynchronous execution of a child process to compute *statement* and immediately continues.

Memory Model. In order to control interference from parallel access, we make the provision that all variables outside the local scope of the parallel processes are treated as private variables. When a process is created, it conceptually makes a copy of each of the non-local variables visible in its scope; subsequent operations act on the now local private variables. Static processes interact by merging their private variables into the shared state at specified barrier synchronization points [MNP+91]. The merge statement

`merge` v_1 **`using`** f_1`,` v_2 **`using`** f_2`, ...;`

specifies a synchronization point which must be reached by all other processes created in the same `forall` or `||`-statement. At this barrier, the values of updated private variables (v_i) are combined to update the value in the parent process and this value is then copied back to all children. The default combining function is arbitrary selection of changed values, although a user-defined function (f_i) may be specified as shown above. A merge implicitly occurs at static process termination. In implementation, it is not necessary to make a complete copy of the shared state; efficient implementations of this memory model are possible [HS92].

Shared Objects. Communication and synchronization between processes is more generally provided within the framework of object classes. There are three principle aspects of this approach.

First, object references are the mechanism for sharing information: a process may interact with another process if both have a reference to the same (shared) object. Variables can only denote a reference to an object, not the object itself; method-invocation must be used to dereference object values. The private memory model applies uniformly to variables, whether they hold object references or other values.

Second, controlled access to shared state is provided through constraints on the mutual exclusion of object methods. The class definition specifies how to resolve concurrent requests for execution of methods of an object instance through the schedule directive

schedule $method_1$ # $method_1$, $method_1$ # $method_2$, ... ;

which specifies that, for each object which is an instance of that class, an invocation of $method_1$ must not execute concurrently with any other invocations of $method_1$ or $method_2$ (in other words, they must not temporally overlap). Intuitively, the construct # denotes conflict, and is used to control competition for resources. For example, we may define a class, parameterized by type `t`, which permits multiple readers and a mutually exclusive writer as follows.

```
class shared_reader (t) {
  var read:  void->t;
  var write: t->void;
  schedule read # write, write # write;   — exclusive writes
};
```

Third, we provide a number of predefined shared object classes. The class `sync` provides a simple way of specifying that one process wait for another. Intuitively, a sync object *x* consists of a datum that in addition to having a value is also tagged with an "empty/full" bit, initially empty. Any process attempting to read an empty datum (through the method *x*.`read`) is suspended until the value is filled, or "defined," by another process (through the method *x*.`write`). In addition, a process may inquire whether *x* is full or empty without blocking (through the method *x*.`test`). Sync variables may be set only once; that is, they possess a single-assignment property.

The sync class in conjunction with the `|>` construct can be used to wait for and obtain the result of an asynchronously spawned function. The `|>` construct promotes the spawned function to return a value of type sync. For example, given a function `f:int->int`, then

```
{ var x: sync(int); x |> f(y) ;  ...   ; z := x.read(); }
```

spawns `f(y)` and puts a placeholder for the result produced by `f` in `x`. When `f` completes it transparently invokes `x`.write on the returned value. If the request for the value of `x` (`x.read`) is made before `f` has completed, the caller is suspended until the value is available.

The class `shared(t)` provides mutually exclusive access to a value of type `t`. Other predefined synchronization classes are being considered. For example, methods can be based on so-called *linear operators* investigated in [Mil94]. Linear operators (as methods in a linear class) generalize the sync methods to model shared data as a consumable resource. In a linear object, the read method blocks until the object is defined, at which point the value is *consumed* and reset to empty; the write method waits until the object is undefined and then *produces*, or sets, the value. Linear operators succinctly model message passing in a shared memory framework, and moreover can be used to build higher-order abstractions such as buffered channels.

Related work on concurrent languages which embody the notion of sync variables includes Compositional C++ [CK92] and PCN [CT92]. We differ significantly from these efforts in our use of explicit operators for synchronization and the casting into an object framework. Our schedule construct bears resemblance to the "mutex" methods of COOL [CGH92], which allow exclusion, however, only over concurrent invocations of a single method. Our linear operators attempt to achieve the goals of CML [Rep91] in supporting the construction of composable high-level concurrency abstractions, but instead of making closures of guarded commands we combine primitive operators similar to those found in Id's M-structures [BNA91] with guarded blocking communication.

3 Program Development Methodology and Environment

Starting with an initial high-level specification, Proteus programs are developed through program transformations which incrementally incorporate architectural detail, yielding a form translatable to efficient lower-level parallel virtual machines. We differentiate between *elaborations*, which alter the meaning of a specification, and *refinements*, which preserve the meaning of the specification but narrow the choices for execution. Elaboration allows development of new specifications from existing ones. We also define *translation* to be the conversion of a program from one language to another. The formal basis of our work is described in Section 5. The relation to software development issues unique to high-performance computing is described in [MNPR94].

Refinement of Proteus programs includes standard compiler optimizations like constant-propagation and common sub-expression elimination. It has been the refinement of constructs for expressing concurrency, however, that most interests us. Such a refinement restricts a high-level design to use only constructs efficiently supported on a specific architecture, presumably improving performance. Since the refined program remains in the Proteus notation, the Proteus programming environment can be used to assess the functionality and performance of the restricted program.

Programs that are suitably refined in their use of the Proteus notation can be automatically translated to efficient parallel programs in low-level architecture-specific notations. These programs can then be run directly on the targeted parallel machines. Changes in the specification or in the targeted architecture can be accommodated by making alterations in the high-level Proteus designs and replaying the relevant refinement and translation steps [Gol90].

The Proteus prototyping environment is designed to support this framework. Many substantial software tools are needed to achieve this end. Of course, program modification must be supported with transformation and compilation tools, targeted to a number of intermediate virtual machines. However, to support experimentation, rapid feedback is necessary; thus we have implemented a highly interactive language interpreter with performance measurement tools. To allow integration with existing codes we also provide a module interconnection facility. Finally, a program repository is required for version control. The entire system is depicted in Figure 34, and the key components are described next.

3.1 Modification

The techniques used to compile programs for efficient parallel execution are complex and evolving. Currently, elaboration is a manual process; refinement is automated with respect to particular goals; and translation is fully automated. We use the KIDS system and related tools from the Kestrel Institute [Smi89] to translate subsets of Proteus language constructs.

The automated refinement strategies are defined to yield Proteus code that preserves the meaning of the code, but in a form that is either more efficient (as a result of high-level optimizations), has increased capabilities for parallelism (automatically extended code), or is more suitable for translation (certain subsets of Proteus notation are more efficiently translatable than others).

Of all our efforts, the translation of data-parallel Proteus code to the parallel virtual machine provided by the C Vector Library (CVL) [BCS+93] is the furthest along. This process is described in detail in Section 6. CVL implements operations on vectors of scalar values. It provides a consistent interface for vector computation on a variety of parallel architectures, allowing Proteus code to be run today on workstations, the Connection Machines CM2 and CM5, the Cray Y-MP and C90 and the MasPar MP1 and MP2 [FHS93]. To simplify our transformations of Proteus code we have implemented an intermediate abstract machine that supports nested sequences and the operations necessary to manipulate them. This Data-Parallel Library (DPL) [Pal93] is built using operations in CVL and, thus, is also highly portable.

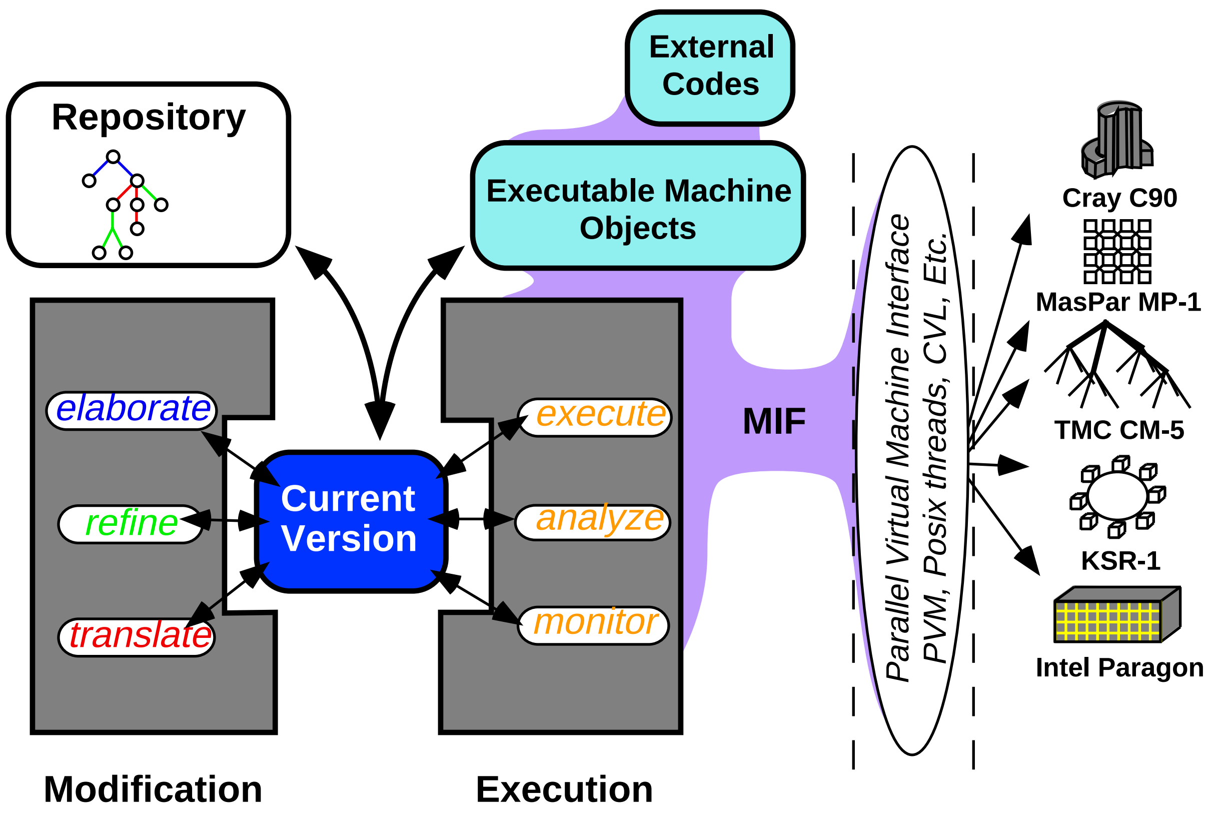


Figure 34 The components of the Proteus System

We are investigating transformations for refining Proteus to other parallel virtual machines, implementing asynchronous parallelism with shared or distributed memory. For multiprocessor shared memory computers, we intend to rely on POSIX threads, whereas for heterogeneous message passing systems, we intend to rely on PVM (Parallel Virtual Machine) [DGM+93] or MPI (Message Passing Interface) [MPI93].

3.2 Execution

For rapid feedback during development, an interpreter for the language is provided. The interpreter does not require variable and type declarations, speeding code development time and encouraging experimentation. This gives the developer some leeway during development, with subsequent refinement steps adding declarations as necessary. The interpreter runs sequentially, simulating parallel execution, including the effects of private memory and unpredictable ordering of execution.

3.3 Performance Analysis

The Proteus interpreter provides a rudimentary per-process clock that measures computational steps. This, in conjunction with explicit instrumentation of Proteus code is used to develop rough resource requirement measures and to predict performance at the higher design levels. However as the program is refined we would like to be able to include more accurate measures of the effects of locality and communication in our experimental and theoretical analyses.

Our methodology for performance prediction, further described in Section 7, is to use increasingly detailed models as program refinement progresses and to support the assessment of multi-paradigm programs by using different models for analysis of code segments following different paradigms—such as the VRAM model for data-parallelism and the LogP model for message passing.

3.4 Module Interconnection Facility

A Module Interconnection Facility (MIF) provides the ability to connect programs written in different languages, possibly running on different machines (Polylith [Pur94] is one such system). The Proteus programming system provides a limited MIF capability giving developers the power to build upon, rather than repeat, previous coding efforts. It also provides an interface for interpreted Proteus code to interact with the code produced by translation of some portion of the prototype. The Proteus MIF provides the interpreter with access to high-performance computers and a mechanism to gradually migrate codes between execution models.

3.5 Repository

A natural consequence of prototyping and refinement is that a derivation tree of programs is made explicit. A history of the transformation activities that created this tree can be kept, not only for reference purposes, but as a basis for re-deriving the program with the same transformation strategies when an ancestral version is changed.

We have such a repository at two levels. First, we keep a version-controlled, tree structured library of the various versions of the prototype. Second, within the KIDS system, all transformations and intermediate results are recorded. From a derivation history window any prior state can be restored by a mouse click, and a new branch of a derivation started. Also there is a replay capability that can replay steps on a modified program using some simple heuristics for maintaining an association between the old program and its modification. This capability has been useful more as a debugging aid and a system development tool than as a tool to explore the design space for a target problem. The reason is that the KIDS refinements are automatic and hence there are no derivation alternatives in this phase of refinement to explore nor any use for replay on an modified program (which can always be refined automatically). Nonetheless this has proved to be a very useful tool on problems which require manual selection of refinements.

4 Proof of Concept and Demonstrations

Several small demonstrations and larger driving problems have been used to examine, assess and validate of our technical approach, including such aspects as the prototyping process and methodology, the expressiveness of the Proteus language, and the effectiveness of the Proteus tools. This section describes prototype solutions for *N*-body calculations using the fast multipole algorithm and several solutions for a geo-server, a problem proposed by the Navy to better understand the usefulness of prototyping.

4.1 N-Body & FMA Calculations

A particularly interesting project is our work prototyping the fast multipole algorithm (FMA), an *O(N)* solution to the *N*-body problem [MNPR92b, NPR93]. This is a problem of extreme practical importance and a key component of several grand-challenge problems.

The foundation of the FMA prototype is the description of the algorithm by Greengard [Gre87], where solutions in two-dimensions using uniform and adaptive spatial decomposition strategies are described, followed by a much more complex algorithm for a uniformly decomposed three-dimensional solution. Greengard's 3D algorithm decomposes space in a hierarchical manner, using an oct-tree of smaller and smaller cubic regions to represent the simulation space. It has phases that sweep sequentially up and then down the oct-tree, with independent work for all the nodes at a given level.

Many others have developed parallel solutions for the FMA, but none have explored adaptive parallelization for arbitrary non-uniform distributions in 3D. The reason is the extreme complexity of the mathematical, algorithmic, and data decomposition issues. In our work, we developed several prototypes of the 3D FMA using Proteus, to explore parallelism issues and spatial decomposition strategies.

4.1.1 Process-Parallel FMA Prototypes

An initial process-parallel prototype was written that reflected a uniform depth spatial decomposition. This prototype was then further refined to accommodate the adaptive structure outlined by Greengard; it consists of an adaptive oct-tree where decomposition of each sub-cube continues until some threshold is reached (fewer than *k* bodies per cube).

Some definitions had to be extended for the adaptive 3D solution. In Greengard's description of the 2D solution, square regions in the plane are categorized as “adjacent” or “well-separated” with respect to one another. However, in 3-space there are some regions that are neither adjacent nor well-separated, so the definitions must be subtly extended. The extensions are not obvious, but Proteus made it simpler to develop and verify them.

4.1.2 Data-Parallel FMA Prototypes

Further prototyping led to a comparison of work performed by data-parallel versions of the uniform and adaptive algorithms. These versions not only allowed us to look at the differences between explicit and implicit parallelism, but also allowed us to examine the expressiveness of the data-parallel subset of Proteus slated for vector execution.

In the data-parallel implementations of the FMA, it was not only possible but almost required to specify the algorithm with nested sequence expressions. For instance, each region at a particular level l (of which there are 8^l) must generate a new expansion based on neighbor, child or parent expansions (depending on the phase). In this setting, an expansion is a truncated polynomial (over two indices) of complex coefficients. Nested iterators express the calculations on all of the regions and all of the interacting expansions over all of the coefficients of the expansions quite succinctly. The high-order functions in Proteus were of great benefit, allowing the definition of a function for adding two expansions, and then using that function as a reduction operation over a sequence of expansions (such as in the operation where all of the lower-level expansions used to create a higher-level expansion).

The adaptive variant of the algorithm developed using Greengard's description seemed inadequate for achieving good parallelism and maintaining reasonably sized data structures. The deeper levels of the spatial decomposition become sparse, and it is difficult to have a data structure that supports both dense and sparse data. In addition, much of the parallelism is gained by performing all of the calculations for a given depth at once, so sparse levels in the decomposition tree lead to less concurrency. Proteus has map data types (from any domain-type to any range-type) which were used for the sparse data structures in the prototypes, but refinement of maps to data-parallel execution must be performed manually. An alternative decomposition was sought to alleviate these problems.

4.1.3 An Alternative Adaptive Decomposition

If, instead of splitting the space in half (or equal octants), the space is cut such that an equal number of bodies end up in a region (a *median-cut* decomposition), then several characteristics change. First, the depth of the decomposition is the same everywhere. Second, the number of bodies in each region is the same (± 1). Third, the decomposition data structure is dense, allowing the use of sequences instead of maps. The data-dependent nature of this variant yields non-cubic (but rectangular) regions of varying-size, and calculating which regions interact in what manner requires re-examination. Once again, the changes are subtle, but the high-level nature of Proteus allowed rapid exploration and discovery to yield a running program in short amount of time. The result was a a variant of the FMA that performed less work overall and provides greater parallelism due to the regular depth of the decomposition.

Figure 35 gives a pictorial representation of the different decompositions in two dimensions. The uniform decomposition is regular, the simplest to parallelize, and the least applicable to non-uniform distributions. The variable-depth adaptive decomposition is generated such that there is a limited population in each region (square in the figure, cubic in 3-space). It performs better with non-uniform distributions, but has unbalanced depth, making it difficult to parallelize (since there is an order imposed by level). The third decomposition is called median-cut, since it cuts the space in half (in each dimension) such that there are equal populations on each side of the cut. The population of each region is the same, and so is the depth of the decomposition. The sizes of the regions are not predictable, nor are the interaction lists (distinguishing near regions from far regions). The lists must therefore be calculated (rather than looked up from precalculation), but this is not a major part of the 3D simulation.

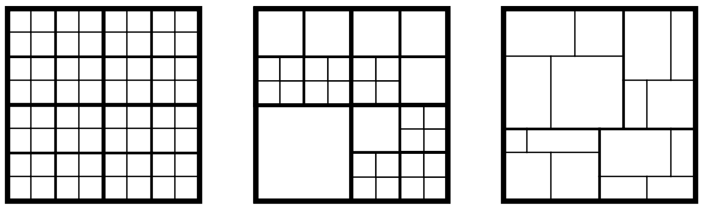


Figure 35 A *uniform*, *variable depth adaptive*, and *uniform depth median cut* decomposition in 2-space.

4.1.4 Execution of FMA Prototype

All variants of the FMA were developed using the interpreter running sequentially. The interpreter simulates process-parallelism with its own notion of threads, and data-parallel operations are executed sequentially. The data-parallel variants were refined to use the subset of Proteus that is automatically vectorized. The size of the data-parallel prototype (which implements all 3 decompositions) is 477 executable lines of Proteus. The most compact C implementation with which we are familiar consists of just under 2000 executable statements, and this only implements the uniform spatial decomposition.

Developing the mathematical code for the 3D variants is extremely complex. Fortunately, at the intermediate steps in the development, it was simple to make comparisons with the direct $O(n^2)$ force calculations. By using a high-level language in an interpretive environment, it is possible, for instance, to select a set of multipole expansions, evaluate them, and sum their results to compare with the direct calculation without having to write another program. Each time a new variant was explored, this type of comparison took place many times to avoid coding errors.

The number of calculations for any instance of the FMA is enormous, one step of a 1000 body simulation using the Proteus interpreter takes 108 minutes on a Sun workstation. Fortunately, there are several well-defined functions within the FMA calculations that could be developed as external code in a more efficient language. By developing C code using the Proteus code as a guideline, 7 external functions were developed. With these and the Proteus MIF, the high-level decomposition strategy that controls the execution and manages the data stays in Proteus, while the computationally intense code is written in C for much higher efficiency. By exploiting this capability, one step of a 1000 body simulation can be run in under a minute, giving quite acceptable performance for interactive algorithm exploration.

4.1.5 Conclusions from Prototyping the FMA

The most important conclusion to be drawn from prototyping the FMA is that many variants could be explored at a high level where decomposition strategies could be easily manipulated. The expressiveness and compactness is a major benefit; for example, the code to calculate the region interaction lists is 45 lines in Proteus compared with 160 lines of C.

A previously undescribed variant of the adaptive 3D FMA was developed that performs less work overall with more parallel execution. This was validated by running all variants of the FMA and recording significant operations performed by each. The high-level notation of Proteus makes this type of exploration possible; low-level specifications of the same algorithms would be far too complex to quickly modify. The FMA development demonstrates algorithm exploration, migration from prototype to efficient implementation (using refinement and the MIF), and translation to parallel code. The effort is documented in [NPR93].

4.2 Geo-server Prototype

Our group participated with others in a demonstration effort to show the capabilities of prototyping in a realistic environment—that of code development for Navy ships to be deployed in the year 2003. An initial experiment was coordinated by the Naval Surface Warfare Center (NSWC) in Dahlgren, VA. Each group agreed that no more than 40 man-hours would be spent over a two week period, and each group would submit their results at the end of that period. The

NSWC challenge problem, the geo-server, was quite naturally expressed in Proteus and developed using the interpreter. It is described in [NPMR93].

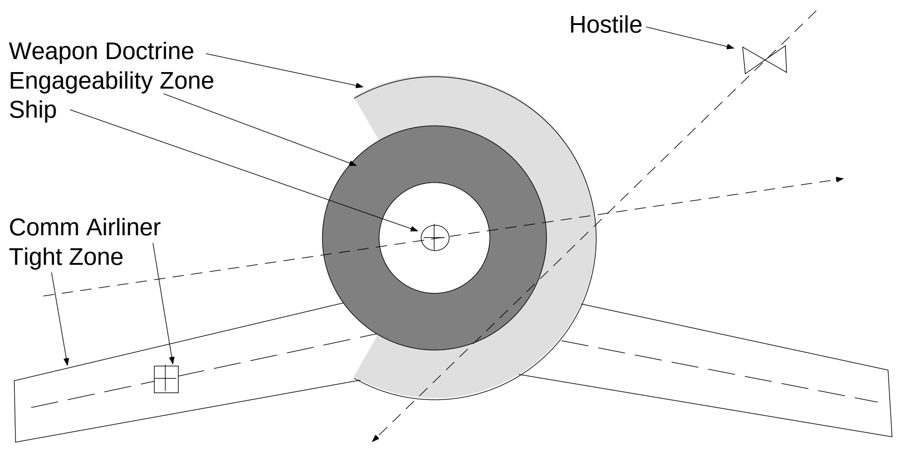


Figure 36 An example of regions and radar data for the geo-server prototype

4.2.1 The Geo-server Problem

A high-level description of this problem can be stated as follows: Given a list of regions and a changing list of radar returns, compute and report the intersections of regions and radar returns. Not all of the regions are fixed on the surface of the Earth, some are based upon the location of a radar datum (the region around an aircraft carrier or airplane, for instance). Each radar datum is a tuple consisting of an identifier, a location, a velocity vector and altitude. The regions, or doctrines, have an identifier, a location, and a shape (composed of a wide choice of shapes, such as arcs, circles, polygons). A pictorial example is shown in Figure 36.

The geo-server need only calculate the intersection information, making it available to a strategic planning system—no judgment is made about the importance of each intersection. This, of course, simplifies the task, making it one component of a larger system.

There is a possibility that the amount of data for this problem may be large, on the order of hundreds of regions and thousands of radar returns, thus the solution should be applicable in such regimes. An algorithm that examines all pairs of regions and radar returns may require too much time to execute in the short period available.

4.2.2 Proteus Solutions

Within the time limit of the exercise we were able to explore three solutions.

- A straightforward sequential solution. It loops repeatedly, gathering information, performing intersection calculations, and posting results. This allowed us to develop support routines for use in further prototypes without regard to concurrency.
- A process-parallel version representing a pipelined calculation. The first process gathers the data from the sensing devices, the second process receives that data and calculates the intersections, and the last process is a display process, showing the results of the calculations.
- A data-parallel rewrite of the intersection calculation using a spatial decomposition. Examining all pairs of radar data and regions is not a scalable activity. Instead, we developed a spatial decomposition, assigning radar data and regions appropriately, and then performing the intersection calculation on the smaller problems. This solution is scalable, the execution time goes up linearly with the total number of regions and radar data.

In the process-parallel implementations, the processes are running asynchronously, reading and writing data to shared data structures, simulating a distributed database environment. Each process runs at a different speed, so a “clock” process was introduced that distributes a “time” value, allowing new data to be posted at appropriate times. It simply scaled down the rate of the per-process clock; the amount of scaling was determined experimentally. We wanted to make sure that each process had enough time to perform its calculation prior to the next set of data becoming available. If the clock ran too fast, the data-gathering process outran the intersection process, causing some data to be missed. The clock process had to be adjusted downward as more functionality was added to the intersection routine to ensure all necessary computations occurred. The reason for not making the clock process run very slowly is that there is a small window between missing some data and seeing it twice. If calculations must only be performed once, then periodic scheduling will have to be done.

Our three related solutions showed rapid development of a parallel application. We used both process- and data-parallelism for different parts of the problem, finding success with both. The developed code was substantially smaller than NSWC's effort (in Ada), primarily due to the high-level nature of the language and was well-accepted. One advantage we had, and made use of, is that Proteus data values (sets, sequences, tuples) can be read directly, eliminating any need to structure the data as it is read or written. The entire geo-server activity was small and short, but many of the benefits found during this activity will save time, coding effort, and bug elimination in larger projects due to the comparative ease with which Proteus code can be developed.

4.3 Other Efforts

Most of our experiments have been performed by Proteus developers well acquainted with the language and the programming environment; however, others are also using the system to develop parallel applications and to predict their performance on various platforms. One such effort, by the medical image processing group at UNC, is to develop sophisticated new parallel algorithms for 3D image segmentation. In this case, the prototypes developed are operational and have been invaluable in locating problems in the mathematics and the parallelization of these algorithms. The Raytheon corporation also intends to use Proteus in this fashion to explore the implementation of multiple hypothesis tracking algorithms.

4.4 Conclusions from Experiments

Our results with the FMA clearly illustrate the utility of the prototyping methodology that we have defined. The parallel algorithms with the best theoretical asymptotic performance may not be most efficient for obtaining solutions on realistic problem sizes, due to costs and parameter limits not made explicit in the model supporting the preliminary design of these algorithms. The FMA is particularly sensitive to this effect since it is an asymptotically-optimal but highly complex algorithm. It has many variants which generate a design space which to date is not well understood. The goal of our experiments with Proteus is 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.

5 Refinement System

This section describes the architecture of the refinement system used to transform Proteus programs. The architecture defines a core language which can express the semantics of the full Proteus language. While we have not defined all of the Proteus core, the Proteus data types have been defined.

A module, referred to as a *specification*, is an axiomatic description of requirements, domain knowledge, architectures, abstract data types and implementations. The refinement environment includes capabilities to construct, parameterize, instantiate and compose specifications, and to map specifications into concrete representations in C and Lisp.

5.1 Specifications

This section informally describes the module language concepts. The main entities are:

- *specifications*, which are theories of the polymorphic lambda calculus with sums and products,
- *specification morphisms*, which are symbol translations between specifications such that the axioms of the source specification are translated into theorems of the target specification, and
- *diagrams*, which are directed multigraphs with the nodes labeled by specifications and the arcs labeled by specification morphisms.

The interconnection of specifications via diagrams is the primary way of putting systems together out of smaller pieces. In particular, diagrams express parameterization, instantiation, importation, and refinement of specifications. The power of the notation arises from the explicit semantics in the specifications and morphisms and in the ability of diagrams to exactly express the structure of specifications and their refinement.

5.1.1 Basic Specifications

A basic specification consists of sort declarations, operation declarations, axioms, and definitions. Below is a specification that defines the data type of finite sequences.

```
Spec CORE-SEQ is
Sorts (fa Z) Seq(Z)
Imports Nat
op empty[]: Seq(Z)
op append       : Seq(Z), Z -> Seq(Z)
op prepend: Seq(Z), Z -> Seq(Z)
constructors {empty, prepend}  construct Seq(Z)
constructors {empty, append}   construct Seq(Z)
op size #_: Seq(Z) -> Nat    prec  20
op  seq-apply _[_]: Seq(Z), Nat -> Z prec 10 assoc right
definition  definition-of-size
   #[] = 0
   #prepend(tail,head) = 1 + #tail
axiom seq-apply
   prepend(S, z) [1] = z
   #S >= i => prepend(S, z) [i + 1] = S[i]
end-spec
```

Sorts. Sort declarations introduce new sort names for *base* sorts and sort constructors; here the notation `(fa Z) Seq(Z)` introduces Z as a sort variable and Seq as a single argument sort constructor. Subsequent use of Z is understood as a sort variable not a sort constant. Sorts are then closed over sums, products, function, and subsorting. Lambda abstraction, let, application and equality are predefined operations on functions; tupling and projection are predefined defined operation on products. Sort terms are used as signatures to denote sorts.

Operations. An operation declaration introduces an operation symbol:

```
op append  : Seq(Z), Z → Seq(Z)
```

Each operation consists of a name, a signtaure and a syntax description. Operation names can be overloaded. An equality operation, with infix syntax is implicitly declared for each sort.

Extended Syntax. There are convenient notational extensions that enhance the general usability of the language. Operations can be given mixfix syntax as illustrated.

Example Declaration	Example Use
op size #_: Seq(Z) -> Nat	#S
op seq-apply _[_]: Seq(Z), Nat -> Z	S[10]

Operations that take a variable number of arguments of the same sort are definable. They are used to define constructors of string constants or literal sets, using, for example, the notation `[1, 2, 3, 5]`. Operations defined using bound variables are also allowed. Examples include iterator expressions, `let` constructs and enumerative loop constructs.

Axioms. The axioms define the semantics of the operations in the sense that a model of a specification must satisfy the axioms.

```
axiom seq-apply
   prepend(S, z) [1] = z  &
   #S >= i => prepend(S, z) [i + 1] = S[i]
```

Constructor sets specify an induction axiom schema. Thus constructors are useful for proving properties or defining operations by induction.

```
constructors{empty, append} construct Seq(Z)
```

Definitions. Axioms of a special syntactic form are called definitions. General axioms describe the behavior of operations implicitly, but definitions are explicit. The purpose of distinguishing definitions is to reduce the amount of work required to define implementations of specifications. There are two kinds of definitions: explicit definitions and inductive definitions over constructors. There is in general, no guarantee that a specification is consistent. Specifically inductive definitions may not be well-defined.

Semantics. The denotation of a specification is a model of a typed polymorphic lambda calculus with sums and products, built on the base types defined in the specification. In general, there can be many models which satisfy such a specification, and they are all valid implementations of the specification. This is known as *loose semantics*. Loose semantics is a desirable property for specification. Inessential detail, for example how error situations are treated (such as sequence index out of range) may be initially omitted and then filled in as part of the refinement process.

5.1.2 Specification Operations

Specifications are definable from basic specifications (defined above) and by two specification-building operations. One such operation is to rename the sorts and operations of a specification. A second way is by computing the colimit of a diagram.

Morphisms. Intuitively, a morphism, *m,* from specification *S* to *T* written $S \xrightarrow{m} T$ describes how the structure of *S* is embedded in *T*. A *morphism* from *S* to *T* is defined by

- A homomorphic association of sort expressions in *S* to sort expressions in *T*. Sort names in *S* are mapped to sort names in *T;* Sort constructors are mapped to sort constructors of the same arity.
- An association of each operation in *S* to an operation in *T* so that signatures are mapped consistently, that is, the signatures type check. Sort variables appearing in a signature may be bound to sort expressions over the sort variable.
- The translations (inductively induced by the symbol translations) of the axioms are theorems of *T*.

Semantically, if there is a morphism $S \xrightarrow{m} T$ then there is a uniform method for constructing models of *S* from models of *T*. Thus if *T* is a consistent theory then so is *S*. Often to successfully prove that a morphism preserves axioms, it is necessary to associate a sort in the source specification with a subsort in the target specification. This is one justification for including predicate subsorts. An alternative is to introduce relativization predicates as in [End72].

Diagrams. A diagram is a directed multi-graph whose nodes are labeled with specifications and whose edges are labeled with morphisms. The colimit operation takes a diagram and yields a new diagram with one new node, labeled with the colimit specification, and a minimal set edges from existing nodes to the new node that insures there is a path from every node to the new colimit node. Informally, the colimit specification is a “shared” union of the sorts and operations and axioms of the specifications associated with each node of the original diagram. Shared here means that, based on the morphisms sorts and operations are identified. More precisely, for sorts (respectively operations) *x* and *y*, define relation *x S y* (*x O y*) if there is a morphism which associates *x* to *y*. Let S^* (O^*) be the reflexive transitive closure of *S* (*O*). Then the sorts (operations) of the colimit specification are the equivalence classes of S^* (O^*). To refer to an equivalence class a representative may be used. If in specification *X* there is a sort *y* , then *X.y* is a (non-unique) name for the equivalence class that *y* is a member of. All of the axioms from all specifications are copied into the colimit specification, with the appropriate symbol substitutions of names to equivalence classes.

Imports. The notation

```
spec T is
import D
   sort s
   op f...
   axiom
   ...
end spec
```

where *D* is a diagram denotes a new diagram consisting of the colimit diagram of *D* in which the colimit specification is augmented with the additional sorts, operations and axioms appearing after `import D`. The notation is overloaded so that it also denotes the specification at the colimit node. Similarly, if a specification or list of specifications appears as an import, instead of a diagram, it denotes the edgeless diagram containing a node for each specification on the list.

The following text expresses the diagram

$$\begin{tikzcd}
\text{INT} \arrow[dr] & \\
& \text{SEQ} \\
\text{CORE-SEQ} \arrow[ur] &
\end{tikzcd}$$

where `SEQ` is the specification consisting of the union of the sorts, operations, axioms and definitions found in `SEQ,` `INT` and the operations `reduce-plus-int`, `reduce-times-int`, `range`, etc. In this case we say `SEQ` *extends* `CORE-SEQ` and `INT.`

```
Spec SEQ is
imports CORE-SEQ, INT
op literal-seq [`_'*`,']: List(Z) -> Seq(Z)
op concat           _++_: Seq(Z), Seq(Z) -> Seq(Z)
                     prec 130 assoc left
op iterator [`@in_'*`,':_]: List(Bool), Z -> Seq(Z)
op filter           _\|_: Seq(Z), Bool -> Seq(Z)
op member           _in_: Z, Seq(Z) -> Bool  prec 150
op reduce2          _./_: (fa Y)(Y, Z -> Y), Seq(Z) -> Y
op reduce3        _./_:_: (fa Y)(Y, Z -> Y), Z, Seq(Z) -> Y
op range          [_.._]: Int, Int -> Seq(Int)
op st-range     [_,_.._]: Int, Int, Int -> Seq(Int)
op reduce-plus-int   +/_: Seq(Int) -> Int
op reduce-times-int  */_: Seq(Int) -> Int
```

```
definition  definition-of-equal
    (S1 = S2) = (#S1 = #S2
           & fa(i) (i in [1 .. #S1] => S1[i] = S2[i]))
definition  definition-of-literal-seq
   empty-list = empty
   cons(x, S) = prepend(x, S)
definition concat
   empty ++ S = S
   prepend(x,S) ++ T = prepend(x, S ++ T)
axiom seq-axiom-concat-associativity
    x ++ (y ++ z) = (x ++ y) ++ z

... more definitions ...
end-spec
```

Definitional Extensions. Suppose a specification *T* extends specifications $T_1, ..., T_n$ exclusively with subsort declarations and operation definitions. Then *T* is a *definitional extension* of $T_1, ..., T_n$. For example, `SEQ` is a definitional extension (assuming all new operations are given a definition) of `INT` and `CORE-SEQ`. A key property of definitional extensions is that using the refinement notions described below, if an implementation for *T* can be mechanically constructed from implementations of $T_1, ..., T_n$.

Parameterization. Diagrams are used to parameterize specifications. In the diagram on the left below the specification `PS` is parameterized by the specification `FP.` The morphism `FP` $\rightarrow$ `PS` picks out of `PS` those symbols that vary by instantiation. Thus `FP` defines an interface for `PS` which includes semantic requirements stated in the axioms of `FP`. `AP` is the actual parameter. The morphism `FP` $\rightarrow$ `AP` asserts that `AP` satisfies the interface requirement defined by `FP`. Computing the colimit of the diagram on the left results in the diagram on the right. In this diagram, the specification `IS` is the instantiation of `PS` by `AP.`

$$\begin{tikzcd} \mathrm{FP} \arrow[r] \arrow[d] & \mathrm{AP} \\ \mathrm{PS} & \end{tikzcd} \qquad \begin{tikzcd} \mathrm{FP} \arrow[r] \arrow[d] & \mathrm{AP} \arrow[d] \\ \mathrm{PS} \arrow[r] & \mathrm{IS} \end{tikzcd}$$

These operations support a style of system building based on diagram construction and manipulation. The same basic operations can be used to formally describe the refinement of such system descriptions.

5.2 Refinement

Our system has two ways of refining specifications:

- Algebraic reasoning within a specification: for example, simplification, partial evaluation, lambda lifting, etc.
- Interpretations which refine one specification to another based on the semantic-preserving properties of morphisms.

The previous subsection described specifications and specification building operations. These operations allow for the construction of a system, described as a diagram of specifications. In this subsection we describe specification operations that may be used to refine diagrams into implementations in C and Lisp.

5.2.1 Interpretations

The essential information supplied by a morphism $S \rightarrow T$ is that it defines how the structure and semantics of *S* is embedded in *T*, that is, specification *T* is rich enough to represent the specification *S*. Thus morphisms serve as the basis for describing specification refinements, or lower-level representations of the specification.

It is usually the case that *T* does not precisely match the structure of *S*. Instead a definitional extension of *T* is constructed for which a morphism from *S* is definable. The definitional extension is called the *mediating specification*. Thus an interpretation is a diagram of the form following.

$$\begin{tikzcd}
\text{Source} \arrow[r] & \text{Mediating} & \text{Target} \arrow[l]
\end{tikzcd}$$

For example, we may define an interpretation whose source is `CORE-SEQ,` target is "dynamic array," and the mediating spec contains definitional extensions that define sequence operations such as `prepend` in terms of array operations.

A system description that uses the `SEQ` data type will transitively import the specification `CORE-SEQ` as definitional extensions. To refine such a specification to implement sequences as arrays the diagram is extended by "gluing" the interpretation of `CORE-SEQ` diagram as shown below.

$$\begin{tikzcd}
\text{CORE-SEQ} \arrow[r] \arrow[d] & \text{SEQ-AS-ARRAY} & \text{ARRAY} \arrow[l] \\
\text{SEQ} \arrow[d] & & \\
\text{Component 1} \arrow[dr] & \text{Component 2} \arrow[d] & \\
& \text{System} &
\end{tikzcd}$$

Computing the colimit creates a specification `REFINED-SYSTEM` in which sequences are implemented as arrays.

$$\begin{tikzcd}
\text{CORE-SEQ} \arrow[r] \arrow[d] & \text{SEQ-AS-ARRAY} \arrow[dddr] & \text{ARRAY} \arrow[l] \\
\text{SEQ} \arrow[d] & & \\
\text{Component 1} \arrow[dr] & \text{Component 2} \arrow[d] \arrow[dr] & \\
& \text{System} \arrow[r] & \text{Refined System}
\end{tikzcd}$$

This discussion has elided many technical details. In particular polymorphism creates complications since an interpretation need not map the equality symbol on one sort to equality on another. Thus monomorphic variants of polymorphic operations must are created as refinement proceeds.

5.2.2 Abstract Target Language

In our methodology we have identified a small set of specifications that comprise the abstract target language (ATL) of the refinement system. These are specifications of types such as arrays, lists, tuples, integers, characters, etc., that commonly appear in programming languages. The refinement expresses a system as a definitional extension the ATL specs. Thus by associating a model—a concrete type in a specific programming language—with each ATL specification the complete system specification is compiled.

5.2.3 Proteus to DPL Translation

The translation of Proteus to DPL consists of a series of major steps:

1. Expansion of iterator expressions into image and filter expressions.
2. Conversion to data-parallel form.
3. An interpretation of sequences into the nested sequence vocabulary of DPL.
4. Addition of storage management code.
5. Conversion into C.

Interspersed with these major steps are common sub-expression abstraction and simplification steps. Apart from the interpretation step and the conversion to C, the other steps consist of algebraic manipulation within a theory. Currently these steps are implemented using REFINE code that manipulates abstract syntax trees. In the future, we intend to implement these steps by instantiating inference procedures with particular sets of axioms. Below we show some of the axioms underlying the procedure for conversion to data-parallel form.

```
spec EXT-SEQ is
imports SEQ
op image: (fa D,R)(D → R), Seq(D) → Seq(R)
op prime _' : (fa D,R)(D → R) → (Seq(D) → Seq(R))

axiom prime-defn
   f'(s) = image(f,s)
axiom image-id
   image(lambda(x) x, s) = s
axiom image-constant
   image(lambda(x) c, s) = distribute(c,#s)
axiom image-lambda-binary
   image(lambda(x) f(g(x),h(x)), s)
     = image(f, zip(image(g,s), image(h,s)))
axiom image-zip-simplify
   image(lambda(x,y) f(x), zip(r,s)) = image(f,r)
. . .
end-spec
```

The axiom `prime-defn` defines the primed version of a function `f` in terms of image. It is used to generate the bodies of primed versions of defined functions, and as a right-to-left rewrite rule to replace expressions of the form `image(f,s)` by calls to `f'(s).` The other rules are used as left-to-right rewrite rules to eliminate calls to `image` by calls to data-parallel versions of functions. However, a naive application of rewriting using the axiom `image-lambda-binary` results in an exponential code-expansion which is contracted by `image-zip-simplify` and common subexpression abstraction. These make the transformation process extremely inefficient. To avoid this it is necessary to use a more sophisticated rewrite procedure that interleaves the application of the rewrites to control the expansion, or else to derive more specialized versions of the axioms that have less expansion, or both.

5.2.4 Interoperability

The development scenario described above is the most basic scenario possible. In this section we consider how an abstract data type may be given different implementations. Recall that a diagram is a graph whose nodes are labeled by specifications and edges by morphisms. Distinct nodes may be labeled by the same specification. Now if two nodes, say N_1 and N_2, labeled with the same specification, say `SEQ`, are imported, then (unless there is a morphism identifying the symbols in the specification) there are two distinct copies of `SEQ` in the colimit specification. Symbols in each copy of `Seq` are disambiguated by qualifying the symbol with the node name. Since the sorts N_1`.Seq` and N_2`.Seq` are different a function expecting a value of type N_1`.Seq` cannot be passed a value of type N_2`.Seq`. As a consequence each sort and its associated operations may be refined independently. So depending on how the diagram is structured operations may be defined that reference a single sort `Seq` or multiple, non-interacting copies of `Seq` that may be refined differently.

Now it is often desired that the same abstract value be represented differently when passed to as a parameter to different operations. This may be done to optimize performance or because different target languages are use in a multi-lingual environment. This behavior is achieved by defining conversion functions between N_1`.Seq` and N_2`.Seq`. Each conversion function must satisfy a homomorphism condition; the two together insure that the models (implementations) of each copy of `SEQ` are isomorphic. In the case of a type defined by a constructor set, as is the case with `SEQ ,` the homomorphism condition is just expressed as axioms involving the constructors. Let h_1`:`N_1`.Seq -> `N_2`.Seq`, and h_2`:`N_2`.Seq -> `N_1`.Seq.` Then the axioms are as follows.

$$
\begin{aligned}
&h_1(N_1.\mathrm{empty}) &&= N_2.\mathrm{empty}\\
&h_1(N_1.\mathrm{append}(S,x)) &&= N_2.\mathrm{append}(h_1(S),\ x)\\
&h_2(N_2.\mathrm{empty}) &&= N_1.\mathrm{empty}\\
&h_2(N_2.\mathrm{append}(S,x)) &&= N_1.\mathrm{append}(h_2(S),\ x)
\end{aligned}
$$

In fact, these axioms are an inductive definition of the conversion function, and so when the theory is refined it is not necessary to specify how to refine the conversion functions.

Now the two copies of SEQ may be imported into a specification along with the two homomorphisms. The specifier can then define operations choosing to use either N_1.Seq or N_2.Seq in the signature. Compositions of these functions can be formed using the appropriate homomorphism function to coerce values of one type into values of another as required by the signatures. As the specs for N_1.SEQ and N_2.SEQ are refined the homomorphism is refined into a conversion function between the representations.

5.3 Architectural Description

The categorical framework (specifications as objects, morphisms as arrows and systems as diagrams) described is well-suited as a notation for software architectures. We illustrate this by showing how the merge sort algorithm can be developed as an instance of a generic divide-and-conquer software architecture.

Typical algorithms such as divide-and-conquer, search, constraint satisfaction, etc., each have a characteristic structure. For example, search algorithms are based on some search space comprising a class of states, a next-state generator, predicates for initial and final states, and a set of filters. To obtain an algorithm for a problem, we attempt to impose the characteristic structure of some algorithm class on that problem (this may involve non-trivial inference). The components thus identified can then be put together in a different manner to obtain a program. We illustrate with divide-and-conquer.

The characteristic structure of a divide-and-conquer algorithm for a function $f:D \rightarrow R$ consists of a set of decomposition operations on the domain D and a set of composition operations on the codomain R connected together and extended with axioms which guarantee that the process of decomposing a domain value, recursively applying the function to the pieces, and composing the results is sound. Correspondingly, a divide-and-conquer program scheme puts together the decomposition and composition operations into a recursive program. Figure 37 shows how mergesort can be derived by imposing Divide-and-Conquer on its problem description.

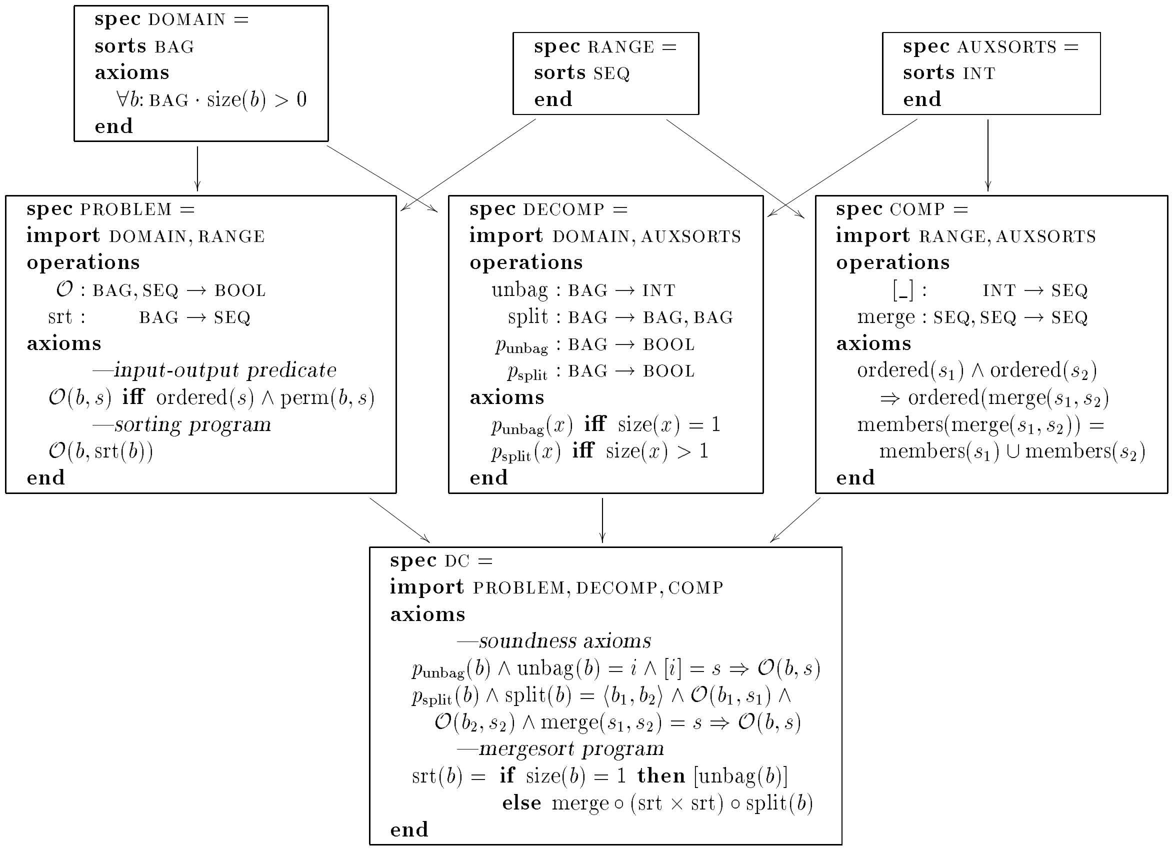
5.4 Related Work

Other work on transformations which comes closest to the spirit of our system is the step-by-step refinement approach of [MT84, TM87]. The difference is that our system is based on inference and is machine-supported.

The algebraic basis of our framework comes from the field of algebraic specification, which has been extensively studied [EM85, GTW78, GH78]. See [Wir90] for a survey. The organization of complex specifications into diagrams of small specifications was introduced in the language Clear [BG77], and is supported to a limited extent in the OBJ3 system [GW88] and the Larch system [GH86]. The formal basis in terms of institutions (a common framework for all logics) is described in [GB92, ST88a].

Our notion of interpretation comes from logic [End72]and is described in detail in [TM87]. A general view of implementations that is independent of the underlying logic is given in [ST88b]. The composition of interpretations (see Section 5.2) is studied in [VM92].

The refinement of (abstract) programs has been studied in the context of the language ML [ST89]. The basic operation is that of associating a module, possibly parameterized, with an interface specification. However, this approach does not formally handle the refinement of the semantics of interface specifications.



spec DOMAIN =
sorts BAG
axioms
$\forall b\colon \text{BAG} \cdot \text{size}(b) > 0$
end

spec RANGE =
sorts SEQ
end

spec AUXSORTS =
sorts INT
end

spec PROBLEM =
import DOMAIN, RANGE
operations
$\mathcal{O} : \text{BAG}, \text{SEQ} \rightarrow \text{BOOL}$
$\text{srt} : \text{BAG} \rightarrow \text{SEQ}$
axioms
—*input-output predicate*
$\mathcal{O}(b, s)$ **iff** $\text{ordered}(s) \wedge \text{perm}(b, s)$
—*sorting program*
$\mathcal{O}(b, \text{srt}(b))$
end

spec DECOMP =
import DOMAIN, AUXSORTS
operations
$\text{unbag} : \text{BAG} \rightarrow \text{INT}$
$\text{split} : \text{BAG} \rightarrow \text{BAG}, \text{BAG}$
$p_{\text{unbag}} : \text{BAG} \rightarrow \text{BOOL}$
$p_{\text{split}} : \text{BAG} \rightarrow \text{BOOL}$
axioms
$p_{\text{unbag}}(x)$ **iff** $\text{size}(x) = 1$
$p_{\text{split}}(x)$ **iff** $\text{size}(x) > 1$
end

spec COMP =
import RANGE, AUXSORTS
operations
$[\_] : \text{INT} \rightarrow \text{SEQ}$
$\text{merge} : \text{SEQ}, \text{SEQ} \rightarrow \text{SEQ}$
axioms
$\text{ordered}(s_1) \wedge \text{ordered}(s_2)$
$\Rightarrow \text{ordered}(\text{merge}(s_1, s_2))$
$\text{members}(\text{merge}(s_1, s_2)) =$
$\text{members}(s_1) \cup \text{members}(s_2)$
end

spec DC =
import PROBLEM, DECOMP, COMP
axioms
—*soundness axioms*
$p_{\text{unbag}}(b) \wedge \text{unbag}(b) = i \wedge [i] = s \Rightarrow \mathcal{O}(b, s)$
$p_{\text{split}}(b) \wedge \text{split}(b) = \langle b_1, b_2 \rangle \wedge \mathcal{O}(b_1, s_1) \wedge$
$\mathcal{O}(b_2, s_2) \wedge \text{merge}(s_1, s_2) = s \Rightarrow \mathcal{O}(b, s)$
—*mergesort program*
$\text{srt}(b) =$ **if** $\text{size}(b) = 1$ **then** $[\text{unbag}(b)]$
else $\text{merge} \circ (\text{srt} \times \text{srt}) \circ \text{split}(b)$
end

Figure 37 Mergesort Derivation

6 Translation of Nested Data-Parallelism to Vector Operations

To explore the feasibility and utility of our refinement and translation strategy in software development we have concentrated on an implementation of the following capability in the Proteus system. Consider a Proteus program *P*. The program is specified using the full facilities of the Proteus language. In particular, all ways to express concurrency may be used. Now we consider trying to express *P* as a data-parallel computation. The refinement requirements are that we refine (rewrite) *P* to a new Proteus program *P′* that restricts its use of Proteus to a subset of the language and that meets certain additional syntactic and semantic requirements.

In this case the subset is applicative, and hence excludes updating assignment. However, the single assignment construct (**`let`** ... **`in`** ...) and a value-returning conditional expression (**`if`** ... **`then`** ... **`else`** ...) may be used in an unrestricted fashion. The values in the subset are numbers, tuples, nested sequences and functions on these values, including, in particular, the data-parallel sequence iterator construct. While this subset definitely constitutes a restriction in the use of Proteus, the subset is itself highly expressive as a result of high-order functions, arbitrarily nested iterators and aggregate values. In particular, it is far more expressive than current data-parallel programming languages such as High Performance Fortran since in addition to the forms of data-parallelism found there, it can also express in a natural fashion irregular and dynamic parallelism as well as nested parallelism such as is found, for example, in parallel divide-and-conquer algorithms.

General strategies exist for the refinement process. For example, repetitive constructs are replaced by recursion, aggregate update is replaced by its functional form, and **`forall`** process-parallelism is replaced by a data-parallel iterator. If these strategies can be sufficiently described, they may be codified as a KIDS tactic. In the system we describe, these refinements must be performed manually.

Once a refined program is obtained that meets these restrictions, as validated by a subset condition checker, the data-parallel program is translated to a C program employing the vector computation parallel virtual machine CVL. In this section we give the details of this translation and describe how DTRE3 is used to implement this process.

6.1 Translation Overview

The translation process consists of the steps shown in Figure 38. First the Proteus program is parsed using a translator built using a parser shared with the Proteus interpreter. The presence and consistency of type declarations is checked and compliance with the subset restrictions is checked. Then the Proteus program is translated to a notation that can easily be manipulated by the Kestrel DTRE3 system.

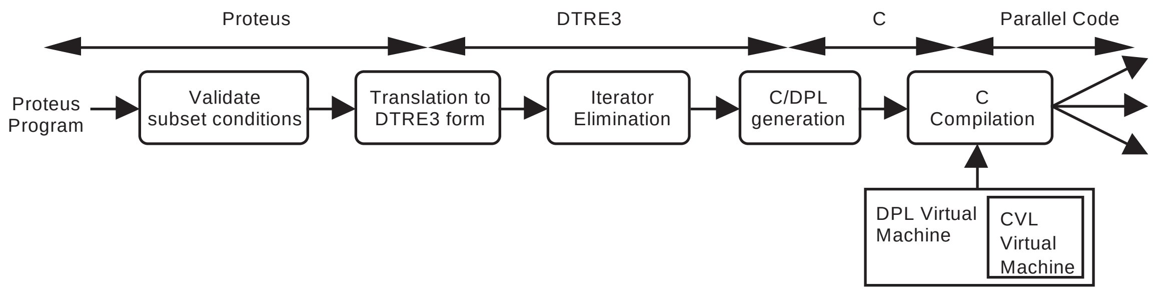


Figure 38 Translation of Proteus programs to parallel (vector) code

Since the translation is essentially a one-to-one mapping, the resulting DTRE3 program retains the nested parallelism expressed in the original Proteus program. The parallelism is then vectorized using source-to-source transformations implemented in DTRE3 as detailed in Section 6.3. The transformed DTRE3 is then translated into C with nested sequence operations. These operations are provided by a nested sequence library that is implemented using the vector operations of CVL. The remainder of this section provides a more detailed examination of these steps.

6.2 Transformational Vectorizing of Nested Parallelism

This section will give an overview of the transformation rules used in the initial implementation of the execution system described in detail in [PP93].

All parallelism in the data-parallel subset of Proteus comes from the iterator construct. The iterator [*i* **in** *D*: *e*] can be thought of as specifying the repeated evaluation of the expression *e* with successive values for *i* selected in turn from the domain sequence *D*. To vectorize a program the iterators must be replaced by direct operations on (nested) sequences. This can be accomplished by defining semantics preserving transformation rules to distribute the iterator through each of the constructs of the data-parallel subset and rules to replace an iterator surrounding a simple constants or variable with an iterator-free expression directly generating the required sequence result. If we visualize the expression *e* as an abstract syntax tree, the objective is to “push” the iterators to the leaves of the tree, where they are replaced with sequence expressions. A simple example is shown in Figure 39.

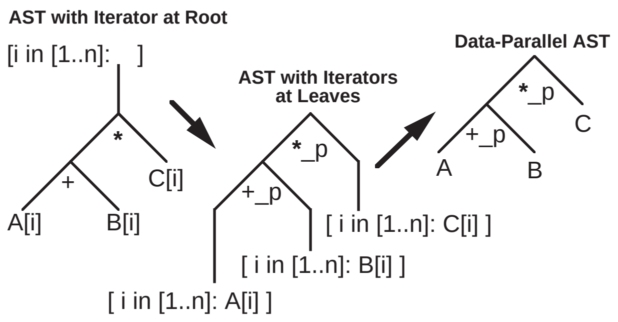


Figure 39 Iterator Eliminating Transformations

When iterators are distributed through a function invocation, the function applied must be altered. Instead of repeatedly evaluating single values, the function now must evaluate a sequence of values in parallel. In the case of, say, addition, the resultant parallel function is simply vector addition. For a small set of simple predefined functions, the resultant parallel functions are predefined functions of the vector library. For the remaining functions that appear in iterators, including in particular user-defined functions, the corresponding parallel versions must be generated. For an arbitrary function *f*, we use f^1 to denote the *data-parallel version* which applies *f* to all elements of a sequence in parallel. Generally speaking if *f* has the signature $\alpha \rightarrow \beta$, then f^1 has the signature $\mathrm{Seq}(\alpha) \rightarrow \mathrm{Seq}(\beta)$. The definition of f^1 is straightforward: an iterator is placed around the body of *f* and the transformations are applied, removing the introduced iterator to yield the body of f^1.

When *f* appears in a nested iterator expression, deeper parallel version is required. This version, f^d, applies f^{d-1} in parallel to all sequences in a nested sequence. Fortunately, we can avoid the need for an unbounded number of data-parallel versions. A depth *d* parallel extension of *f*, operates on values at depth *d* without altering the frame, hence it suffices to use f^1 in all contexts. To achieve the effect of $f^d(e)$, we flatten the frame around values in *e*, apply f^1, and restore the frame around the result of this application.

$$\begin{tikzcd}
V \arrow[r, "f^d"] \arrow[d, "\text{extract(V,d)}"'] & S \\
W \arrow[r, "f^1"'] & R \arrow[u, "\text{insert(R,V,d)}"']
\end{tikzcd}$$

Figure 40 Implementation of Deep Parallelism of User-defined Functions

This is accomplished by using the *extract* operation to remove the nesting frame of a sequence so the simple data-parallel function can be applied. The result is then re-attached to a frame using the *insert* instruction as shown in Figure 40. More details on the insert and extract operations are provided in Section 6.4. This approach eliminates the need for more than one data-parallel extension of any function.

Repeated application of the transformation rules alter the abstract syntax tree until all iterators are at the leaves surrounding either a constant, a bound variable or a free variable. Additional transformations replace such constructs with equivalent nested sequence representations that contain no iterator constructs. After all iterators have been eliminated and the code has been generated, aggressive common sub-expression elimination can make the resulting program much more efficient.

6.3 Nested Sequence Library

The Data-Parallel Library (DPL) [Pal93] is a collection of C-callable routines that provide the capability to treat nested sequences as primitive data types in an architecture-independent manner. The library is designed specifically to be used with C as the executable target notation for transformed Proteus programs. The table below provides a list of the principal nested sequence operations and their signatures. In addition to these, the library also provides a data-parallel version of each operation.

Since we represent nested sequences with a group of vectors, we can implement all nested sequence operations as series of vector operations, (scans, permutes, element wise operations, etc.) on those vectors that comprise the representation.

Name	**Signature**	**Action**
arith-op	$Seq(Num) \times Seq(Num) \rightarrow Seq(Num)$	basic math and logic operations
build	$\alpha \times ... \times \alpha \rightarrow Seq(\alpha)$	make a sequence from components
index	$Seq(\alpha) \times Int \rightarrow \alpha$	extract a value from a sequence
length	$Seq(\alpha) \rightarrow Int$	number of elements in a sequence
dist	$\alpha \times Int \rightarrow Seq(\alpha)$	replicates a value to form a sequence
range	$Int \times Int \rightarrow Seq(Int)$	enumerates integers between values
restrict	$Seq(\alpha) \times Seq(Bool) \rightarrow Seq(\alpha)$	packs a sequence according to a mask
reduce	$Seq(Num) \rightarrow Num$	reduce a sequence by associative operation
combine	$Seq(Bool) \times Seq(\alpha) \times Seq(\alpha) \rightarrow Seq(\alpha)$	merge two sequences based on a mask
promote	$Num \times Seq^k(Int) \rightarrow Seq^k(Num)$	replicate number to size and shape of sequence

Nested sequences are directly represented with vectors as follows. A collection of d vectors $V_1,...,V_d$ are used, where $V_1,..,V_{d-1}$ are *descriptor vectors*, V_1 is always a singleton vector and V_d is a *value vector.* A vector representation and its equivalent *nesting tree* representation are shown in the figure. Each descriptor vector indicates the partitioning for the vector on the level below. Only adjacent descriptor vectors are directly related to each other, so maintaining a consistent representation when performing sequence operations is relatively simple. Note that empty sequences at the leaves of the nesting tree are represented by a zero index in the lowest-level descriptor vector.

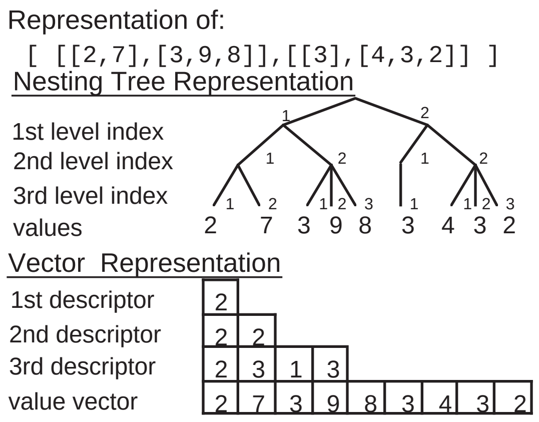


Figure 41 Vector Representation of a Nested Sequence

There are two operations that directly manipulate the representations of nested sequences: *extract* and *insert*. For V, a sequence of depth $d+k$, *extract*(V,d) flattens the top d nesting levels (see Figure 42) and can be implemented by replacing the top d descriptors by the singleton vector that is the sum of the values in the d^{th} level descriptor vector.

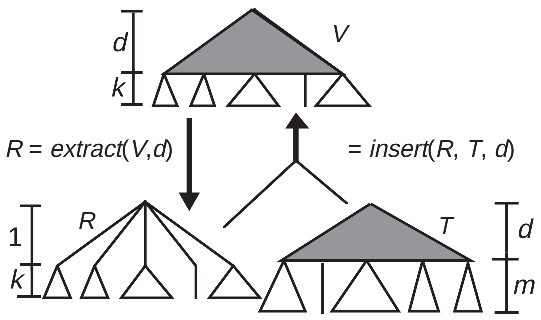


Figure 42 Nested Sequence Representation Manipulation

The *insert* operation forms a depth $d+k$ sequence from a sequence of depth $k+1$ and another sequence of depth greater than d. In practice, the second sequence is always the same as the sequence used in a corresponding *extract* operation, but this is not required. *Insert* (R, T, d) removes the top descriptor from R and replaces it with the top d descriptors from T . We require that the number of elements in the d^{th} level descriptor vector of T is equal to the number of top-level elements in R. This insures that the result of an *insert* operation is a consistent nested sequence representation. Note that V = *insert* (*extract* (V, d), V, d) for any d less than or equal to the depth of V.

6.4 Transformation Rules for Iterator Elimination

Several of the transformation rules make use of operations from the data-parallel library. A list of the operations and a brief description of the their meaning is presented in the next Section 6.4.

Rule 1. Transform function application.

```
[v in D: f^n (e_1,...e_k)] ≡  f^{n+1} ([v in D: e^1],...,[v in D: e^k])
```

Rule 2. Transform let statement.

```
[v in D:  let t = e^1 in e^2] ≡
   let
      T = [v in D: e_1]
   in
      [i in [1..#T]: e_2 ]  with every occurrence of t in e_2 replaced by T[i] and every
                            occurrence of v in e_2 replaced by D[i]
```

Rule 3. Transform conditional statement.

```
[v in D:  if e_2 then e_2 else e_3 ]..] ≡
   let
      M =  [v in D: e_1 ]..]
      T =  [ k in restrict(D,M): e_2 ]..]        with every v in e_2 replaced with k
      E =  [ k in restrict(D, not(M)): e_3 ]..]  with every v in e_3 replaced with k
   in
         combine(M,T,E)
```

Rule 4. Transforming iterator nests around bound variables.

```
[i in D : i] ≡  D
[i in D : j] ≡  dist(j, length(D))
```
where j is a bound variable of an enclosing iterator

Rule 5. Transforming iterator nests around constants and free variables.

```
[v in D: c ]                ≡ promote(c,D)
[v in Dn: promote(c,D) ] ≡ promote(c,[v in Dn: D])
```

Rule 6. Generating data-parallel functions. For function g defined as follows,

```
function g(x1, ... , xn) =
   return e
```

generate the following lifted version.

```
function g1(V1, ... , Vn) =
   return [i in [1..#V1]: e ]
```
with every occurrence of x_k in e replaced by $V_k[i]$

Rule 7. Implementation of deep data-parallel functions.

```
fd (e1, e2, ...,en) ≡
   let V1=e1, . . , Vn = en
   in
      insert
        (f1( extract(V1,d - 1),
                     ...
             extract(Vn,d - 1)),
      V1,d)
```

6.5 Example

We illustrate the transformations and translations described in the previous two sections on the following simple expression.

```
[i in [1..5]: sqs(i)]
```

using the function `sqs` defined as follows.

```
function sqs(n:[int]) = return [j in [1..n]: j * j];
```

This expression evaluates to

```
[[1],[1,4],[1,4,9],[1,4,9,16],[1,4,9,16,25]].
```

Apply Rule 1. The top-level expression is first transformed to the following.

```
sqs1[i in [1..5]: i];
```

Apply Rule 4. The iterator is then simplified.

```
sqs1([1..5]);
```

Apply Rule 6. Since the resultant expression has introduced `sqs`^{1}, we must define this function using `sqs` and transform it to remove iterators. The following transformation steps show the movement of iterators towards the leaves of the abstract syntax tree and the replacement of iterator expressions at the leaves with equivalent nested sequence expressions.

```
function sqs¹(V:[[int]]) =
   return
      [i in [1..#V] : [j in [1..V[i]]: j * j] ];
```

For clarity, we write all operations in the body in prefix form.

```
≡  [i in range1(length(V)) : [j in range1(index(V,i): mult(j,j)] ]
```

Apply Rule 1. First the inner iterator is lifted through the `mult` operation.

```
≡  [i in range1(length(V)) : mult¹([j in range1(index(V,i)): j],
                                    [j in range1(index(V,i)): j]) ]
```

Apply Rule 4. Then the inner iterators are simplified.

```
≡  [i in range1(length(V)) :
               mult¹(range1(index(V,i)),range1(index(V,i))) ];
```

Apply Rule 1. Next the outer iterator is transformed through the `mult`^{1} and both `range1` operations.

```
≡  mult²(range1¹([i in range1(length(V)): index(V,i)]),
         range1¹([i in range1(length(V)): index(V,i)]))
```

Simplify. Use the identity `[x in [1..#Y]: Y[x]] ≡ Y`.

```
≡  mult²(range1¹(V),range1¹(V))
```

Apply Rule 7. `Insert` and `extract` alter the depth of the inputs so `mult`^{1} can be used to implement `mult`^{2}.

```
≡  insert(mult¹(extract(range1¹(V),1),
                  extract(range1¹(V),1),range1¹(V),1) )
```

At this point we have removed the iterators from the program and it is fully vectorized. We use DTRE3 to generate C code for the remaining program, producing the following result.

```
nseq sqs¹(nseq V) {
   nseq  q = range1¹(V);
   nseq  r = insert(mult¹(extract(q,1), extract(q,1)), q,1)
   return r;
}
```

This C program can be compiled and linked against the DPL and CVL libraries. For each target architecture there exists a different implementation of the CVL operations, but since DPL is expressed in C using CVL, only one DPL library is needed.

6.6 Results

The C code presented in the previous section is a simplification, the actual code generated includes memory management operations and a main program to call the top-level expression to be evaluated. This program is fully vectorized and may be executed on any of a number of target platforms using the appropriate implementation of CVL. The vectorization is work-efficient in that all and only the operations specified in iterators in the Proteus programs are executed. Below a sevenfold speedup was observed in execution of the parallel version on a parallel machine.

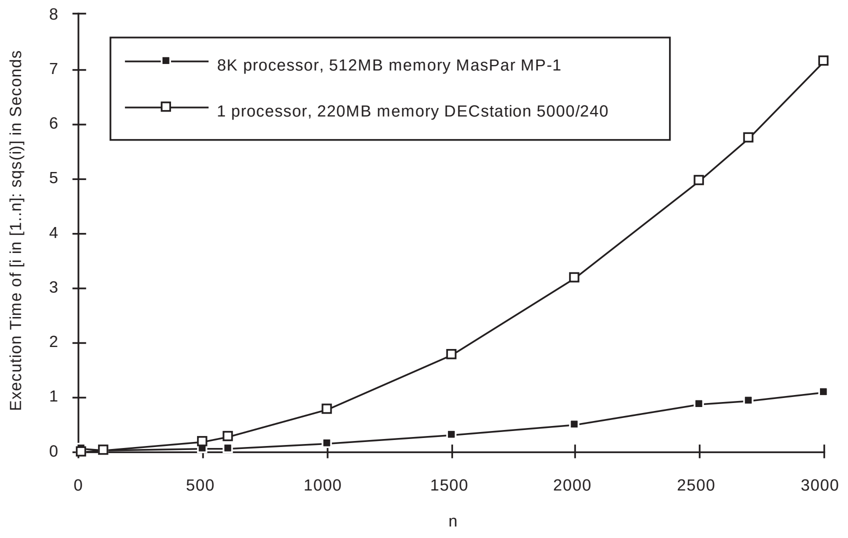


Figure 43 Comparative Performance of Serial and Parallel Evaluation of Generated Code

Note that the computation is O(n^2) in size, so when n = 3000, the size of the vectors being multiplied is about 4.5 million elements. We have shown that we can generate vector operations from high-level data-parallel expressions that can access the improved performance potential available from parallel execution.

6.7 Summary

The ability to specify a wide variety of data-parallel computations in Proteus is a natural consequence of the nested sequence aggregate data type and the iterator construct. These constructs permit us to specify work-efficient irregular data-parallel computations (as found in the parallel application of a function to each of a collection of sequences of different length), work-efficient recursive parallel computations (as found, for example, in parallel 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).

In this section we have shown how all these data-parallel expressions can be translated into vector operations that are well matched to the capabilities of modern highly-parallel computers, including vector machines, SIMD and MIMD machines, achieving excellent load-balance in all cases.

The possibility of vectorizing computations of this sort was initially explored in [Ble90]. In [BS90] a specialized compiler for vectorization was described. While the approach is valuable in providing parallel execution for a large class of sophisticated data-parallel algorithms, to date only NESL[Ble92] and Proteus implement this strategy. Part of the reason is the sophistication of the translation. Having developed a highly general transformational characterization of this process in [PP93] we were able to use the DTRE3 system to rapidly implement this translation. To generate good code, significant analysis and common subexpression elimination is required. Here the DTRE3 system again provides extensive leverage in achieving this level of sophistication.

Thus, while compilation for efficient parallel execution remains an elusive and ever moving target, we feel that our high-level transformational approach based on the Kestrel toolset permits us to incorporate sophisticated compilation strategies rather easily and effectively as they are developed.

7 Performance Prediction and Measurement

The ability to predict and measure the performance of programs on actual parallel machines is critical to a prototyping approach to the development of efficient parallel programs, allowing the early assessment of algorithmic variations without the cost of full implementation. Performance analysis here encompasses both static theoretical analysis as well as the dynamic monitoring of simulated or actual execution. Historically, mathematical models of parallel computation such as the PRAM hide details which impact performance and so are often inaccurate. In this section we examine issues in the design of improved models of parallel computation used for performance prediction and present our approach for the practical application of theoretical models to performance prediction in Proteus programs. As an illustration of the design and use of models supporting more detailed performance measures we present a new model of parallel computation we have developed, the LogP-HMM model, which is sensitive to both communication and memory hierarchy.

7.1 Models and Resource Metrics for Parallel Computation

A computational model is a mathematical abstraction of a computing machine that allows the estimation of the time complexity measures of an algorithm as well as its resource utilization such as memory space. A familiar computational model for the sequential domain is the Random Access Machine (RAM). The relation between language, model, and machine, is depicted in Figure 44. The accuracy of performance estimation depends on the correspondence between model and lower-level machine.

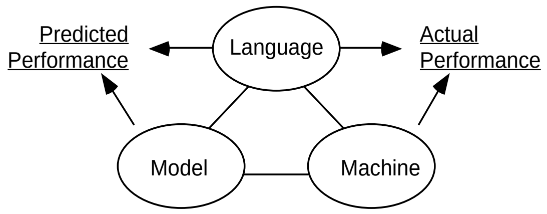


Figure 44 Performance prediction and computational models

Yet it remains difficult to use models to assess parallel program performance, for many of the same reasons it is difficult to construct portable efficient parallel algorithms in the first place. The problem arises from the diversity of complex parallel architectures and the induced wide gap between high-level design models and low-level machines and programming languages. Historically, the PRAM model [FW78] is the most widely used parallel model. It assumes that a collection of sequential processors work synchronously and that interprocessor communication through shared memory is essentially free. However, the PRAM model hides too many details which impact performance and so is often inaccurate in predicting the actual running time of programs. The PRAM does not reflect the growing trend towards larger-grained asynchronous MIMD machines whose processors each may have their own sophisticated memory hierarchies such as cache and which communicate over relatively slow networks. Examples of such machines are the Intel Paragon, the KSR-1 and the TMC CM-5. The PRAM ignores the times required for network communication and synchronization, as well as issues of asynchrony and memory hierarchy, and thus predicted performance is often not preserved in implementation.

To improve performance prediction we need:

- better theoretical models, that is, 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
- better tools and techniques for performance prediction, that is, the practical application of these theoretical models to performance prediction (as well as to the requisite tasks of compilation such as partitioning and mapping).

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* (for example in the APRAM [CZ89]),
- *communication costs*, such as network latency and bandwidth restrictions (for example, the BSP model [Val90], and the LogP model [CKP+93]), and
- *memory hierarchy*, reflecting the effects of multilevel memory such as differing access times for registers, local cache, main memory and disk I/O (for example, the PMH (parallel memory hierarchy) [ACF93], and the P-HMM (parallel hierarchical memory model) [VS93]).

The approach followed by these models is that of a *generic parameterized model*. That is, architecture details are abstracted into several generic parameters, which we call *resource metrics*, and the model is parameterized in terms of a number of these resource metrics. Typical resource metrics include the number of processors, the communication latency, asynchrony, bandwidth, block transfer capability, memory access method, network topology and memory hierarchy. For example, the LogP model [CKP+93] is an asynchronous parallel computation model which captures communication costs through the parameters of latency (“L”), overhead of the processor for handling a message (“o”), the minimum gap between successive message transmissions (“g”), and a parameter for the number of processors (“P”). The ratio L/g gives the network bandwidth. The LogP model is notable in that it ignores network topology, and is somewhat more suited for distributed memory machines.

Using such a parameterized model one can design parameterized algorithms (parameterized in the same metrics as the model) that can run efficiently on a broad class of machines. To derive the performance of such an algorithm (and its parameter-dependent behavior) on a specific machine, one tailors the model to match the specific machine characteristics, in other words instantiates the parameters with machine-specific values, for example choosing a specific value of latency and bandwidth to match the communication characteristics of the CM-5.

7.2 A refinement-based approach to performance prediction

In response to the second need—practical application of theoretical models to performance prediction—we are pursuing an approach for performance prediction of Proteus programs which is based on (1) the use of different models for analysis of code segments following different paradigms and (2) the use of increasingly detailed models as program refinement progresses.

At the coarsest level performance prediction in Proteus may be done using the interpreter to derive simple approximations of total work and parallel time: the code is instrumented to specify costs (steps) for sequential program segments, and using this a per-process “clock” measures computational steps. Of course, actual running time on a particular target is more complex and includes issues of communication and memory hierarchy. As the program is refined, for example from a shared memory program (using a weak access discipline such as the `shared_reader` class defined in Section 2.2) to a more sophisticated form which uses sync or linear classes (which might correspond to message passing on a distributed memory machine), we would like to be able include more accurate measures of these effects into our experimental and theoretical analyses.

Our strategy is to select a level of model appropriate for a given section of a Proteus program, which may follow one of several parallel programming paradigms. For data-parallel code using nested sequence operations (which can be transformed to vector operations) we employ analysis using the VRAM vector model [Ble90]. For shared memory code segments using “linear” producer-consumer access disciplines which resemble message passing, we employ the LogP network model. As the Proteus program, when using shared_reader variables or a more restrictive access discipline such as linear variables, reflects a programming paradigm with even less detail than the LogP model, we must instrument the code to identify low-level units of communication (such as message send and receive) as well as work. We are devising annotations for the program which re-incorporate details of memory access disciplines. An annotated program can then be semi-automatically analyzed statically or simulated using a sequential interpreter to derive performance measures which can then have the values of the work and communication units tailored to machine parameters.

7.3 LogP-HMM: A Network Model with Memory Hierarchy

It is important at some stage in the refinement process to be able to model the several layers of memories which exist in many practical machines, since the differing latency and transfer times to local cache and disk strongly effect the performance of parallel algorithms. Unfortunately, the LogP model, while accurately modeling network communication, does not address multilevel memories. Recent models such as the P-HMM (parallel hierarchical memory model) [VS93], on the other hand, while supporting a notion of hierarchical memory, use a simple characterization of the network (for example a hypercube). To support improved performance measures we have developed a new model of parallel computation, the LogP-HMM model [LMR94], which extends an asynchronous network model (LogP) with memory hierarchy.

The LogP-HMM parameterized model is unique in its approach in that it combines an existing network model (the LogP) with an existing sequential hierarchical memory model (the HMM) characterizing each processor connected by the network. As is the case in the P-HMM, the sequential hierarchical model chosen for the processors may be one of several existing proposed models, such as the UMH (Uniform Memory Hierarchy) [ACF90] where each memory level is parameterized by block size, block count, and transfer time, or the HMM (Hierarchical Memory Model) [ACS90] where each level i of 2^i locations has access time $i+1$. Our approach, however, differs from that of P-HMM in that we extend the parallel network model with memory hierarchy at each processor rather than connecting the base memory level by a simple network. The result more accurately captures both network communication costs and the effects of multilevel memory such as local cache and parallel I/O.

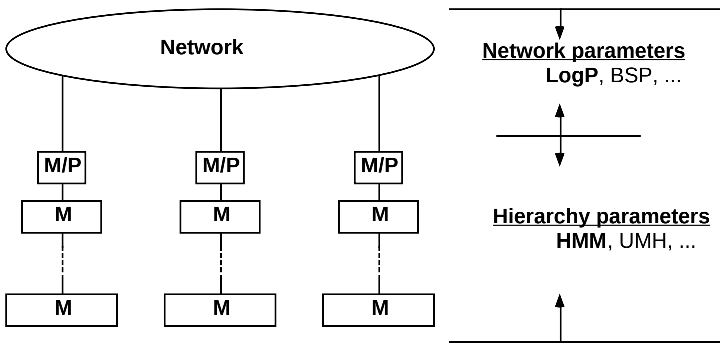


Figure 45 LogP-HMM: a parameterized network model with memory hierarchy

The LogP-HMM model we propose, depicted in Figure 45, thus consists of two parts: the network part, which in general can be any parallel network model such as LogP or BSP, and the memory hierarchy part, which can be any of the sequential hierarchical memory models such as HMM or UMH. Each computation node consists of several layers of memory with the processor sitting in the lowest memory level: the processors are connected by a network characterized by the LogP parameters, and the memory part is characterized by the HMM parameters which roughly charge log i cost to access memory location i.

Using this model we have designed a parallel FFT (Fast Fourier Transform) algorithm which takes into account improved strategies for locating and moving data between cache, main memory and disk [LMR94]. The FFT algorithm is within a factor $(1+g/(\log \log N/P))$ of what has been proven to be theoretically optimal [VS93]. Ongoing work includes the development of near optimal randomized sorting and geometry algorithms for our model.

7.4 Techniques for Performance Prediction using the Improved Model

We envision that the improved LogP-HMM can be “instrumented” into Proteus code in a number of ways. Most simply, disk reads and writes can use the more accurate costs of latency and transfer time for I/O to derive better performance measures. The problem of instrumenting the program to capture performance effects of lower levels of the hierarchy such as cache is more difficult. We are considering annotations for the program which incorporate explicit details of memory locality. A related approach has been used for cache-coherent shared memory multiprocessors in the CICO project [LCW93].

We have outlined, in this section, an approach to performance prediction of Proteus programs using a hierarchy of models expressing increasingly detailed approximations to resource metrics which matches the structure of program refinement. As more detailed architectural commitments are made in the specific expression of concurrency, more detailed approximations of resource metrics can be attached. Ongoing work includes pursuing extensions for mixed data-parallel and process-parallel code segments, detailed treatment of network and memory hierarchy using the LogP-HMM, and the development of a graphical user interface for performance analysis.

8 Conclusion

In the Proteus system, we emphasize the use of high-level specifications, the development of implementations by successive refinement and translation, and early feedback on designs through the use of executable prototypes.

The ability to compactly specify a wide variety of parallel algorithms in an architecture-independent fashion forms a convenient and comprehensible starting point for the development activity that eventually leads to implementations for different architectures. Our experience with the Proteus language has been that it is well-suited for the construction of executable specifications and their successive refinements.

Automated support for refinement and translation of Proteus programs is not yet complete, so it is premature to evaluate its strengths. In the development of the FMA and other trial projects, we have relied on manual refinement to bridge the gap from specification written in full Proteus to programs written in a subset of Proteus that could be automatically translated to efficient code. The individual refinement steps in these efforts had well-defined objectives and were quite manageable. After each step, comparison of the new version with the previous version through experimental execution was valuable.

With respect to automated translation, our main effort thus far has been the development of a translation that vectorizes arbitrary data-parallel expressions. This is a nontrivial translation; only NESL [Ble92] and Proteus provide this capability to date. We were able to use transformation tools from the Kestrel Institute to rapidly implement this translation; these tools are capable of generating good code and performing significant analysis. We are encouraged by this success, and believe that the refinement and translation approach will allow us to easily incorporate sophisticated compilation strategies as they are developed in the optimizing compiler community.

Prototyping is essential to the development of complex parallel applications. We have started from the premise that information obtained through disciplined experimentation with prototypes reduces risks 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. However, without the ability to migrate the prototype into an efficient implementation, the investment required to produce a working prototype often cannot be justified. Therefore, we emphasize refinement as a means of evolving prototypes into production code.

9 Acknowledgment

This chapter is a substantially revised and expanded version of [GMN+94], first published in *DIMACS: Specification of Parallel Algorithm*s in 1994 by the American Mathematical Society.

10 Bibliography

[ACF90] B. Alpern, L. Carter and E. Feig, “Uniform memory hierarchies,” *Proc. 31th Symp. on Foundations of Computer Science*, 1990.

[ACF93] B. Alpern, L. Carter and J. Ferrante, “Modeling parallel computers as memory hierarchies,” *Workshop for Portability and Performance for Parallel Processing*, Southampton, England, 1993.

[ACS90] A. Aggarwal, A. K. Chandra and M. Snir, “Communication complexity of PRAMs,” *J. Theoretical Computer Science*, 1990.

[BCS+93] G. Blelloch, S. Chatterjee, J. Sipelstein and M. Zahga, “CVL: A C vector library,” Draft Technical Report, Carnegie Mellon University, Mar 1993.

[BG77] R. M. Burstall and J. A. Goguen, “Putting theories together to make specifications,” *Proc. 5th Int. Joint Conf. on Artificial Intelligence*, pp. 1045-1058, 1977.

[Ble90] G. E. Blelloch, *Vector Models for Data-Parallel Computing*, The MIT Press, 1990.

[Ble92] G. E. Blelloch, “NESL: A nested data-parallel language,” Technical Report CMU-CS-93-129, Carnegie Mellon University, Jan 1992.

[BNA91] P. S. Barth, R. S. Nikhil and Arvind, “M-structures: Extending a parallel, non-strict functional language with state,” in *Functional Prog. Languages and Computer Architecture*, *Lecture Notes in Computer Science*, v. 523, pp. 538-568, Springer-Verlag, 1991.

[BS90] G. Blelloch and G. Sabot, “Compiling collection-oriented languages into massively parallel computers,” *Journal of Par. and Distr. Computing*, 8:119-134, 1990.

[CGH92] R. Chandra, A. Gupta and J. Hennessy, “Integrating concurrency and data abstraction in the COOL parallel programming language,” Technical Report CSL-TR-92-511, Stanford University, 1992.

[CK92] K. M. Chandy and C. Kesselman, “Compositional C++ : Compositional parallel programming,” *Proc. of the 4th Workshop on Parallel Computing and Compilers*, 1992.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. v. Eicken, “LogP: Towards a realistic model of parallel computation,” *Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1993.

[CT92] K. Chandy and S. Taylor, *An Introduction to Parallel Programming*, Jones & Bartlett, 1992.

[CZ89] R. Cole and O. Zajicek, “The APRAM: Incorporating asynchrony into the PRAM model,” pp. 169-178, 1989.

[DGM+93] J. Dongarra, G. A. Geist, R. Manchek and V. S. Sundaram, “Integrated PVM framework supports heterogeneous network computing,” *J. Computers in Physics*, 7:166-175, 1993.

[EM85] H. Ehrig and B. Mahr, “Fundamentals of algebraic specification: Equational and initial semantics,” in *EATCS Monographs on Theoretical Computer Science*, v. 6, Springer-Verlag, 1985.

[End72] H. B. Enderton, *A Mathematical Introduction to Logic*, Academic Press, 1972.

[FNPP94] R. E. Faith, L. S. Nyland, D. W. Palmer and J. F. Prins, “The Proteus NS grammar,” Technical Report TR94-029, UNC, 1994.

[FHS93] R. E. Faith, D. L. Hoffman and D. G. Stahl, “UnCvL: The University of North Carolina C vector library,” Technical Report TR93-063, UNC, 1993.

[FW78] S. Fortune and J. Wyllie, “Parallelism in random access machines,” *Tenth ACM Symp. on Theory of Computing*, pp. 114–118, 1978.

[GB92] J. A. Goguen and R. M. Burstall, “Institutions: Abstract model theory for specification and programming,” *J. ACM*, 39(1):95-146, 1992.

[GH78] J. V. Guttag and J. J. Horning, “The algebraic specification of abstract data types,” *Acta Inf.*, 10:27-52, 1978.

[GH86] J. V. Guttag and J. J. Horning, “Report on the Larch shared language,” *Sci. Comput. Programming*, 6:103-134, 1986.

[Gol90] A. Goldberg, “Reusing software developments,” *ACM SIGSOFT 4th Symp. on Software Development Environments*, 1990.

[GMN+94] A. Goldberg, P. Mills, L. Nyland, J. Prins, J. Reif and J. Riely, “Specification and development of parallel algorithms with the Proteus system,” *DIMACS: Specification of Parallel Algorithms*, AMS Press, 1994.

[Gre87] L. F. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems*, MIT Press, 1987.

[GTW78] J. A. Goguen, J. W. Thatcher and E. G. Wagner, “An initial algebra approach to the specification, correctness and implementation of abstract data types,” in *Data Structuring, Current Trends in Programming Methodology*, (R. T. Yeh, Ed.), v. IV, pp. 80-149, Prentice Hall, 1978.

[GW88] J. A. Goguen and T. Winkler, “Introducing OBJ3,” Technical Report SRI-CSL-88-09, SRI International, 1988.

[HS92] L. Higham and E. Schenk, “The parallel asynchronous recursion model,” *Third IEEE Symp. on Prallel and Districuted Processing*, 1992.

[Jul93] R. Jullig, “Applying formal software synthesis,” *IEEE Software*, 10:11-22, 1993.

[LCW93] J. R. Larus, S. Chandra and D. A. Wood, “CICO: A practical shared-memory programming performance model,” *Workshop on Portability and Performance for Parallel Processing*, Southampton University, England, 1993.

[LMR94] Z. Li, P. Mills and J. Reif, “Models and resource metrics for parallel and distributed computation,” Draft Technical Report, Duke University, Mar 1994.

[LN93] G. Levin and L. Nyland, “An introduction to Proteus, version 0.9,” Technical Report, UNC, 1993, 1993.

[McC93] W. F. McColl, “An architecture independent programming model for scalable parallel computing,” Draft Technical Report, Programming Research Group, Oxford University, Sep 1993.

[Mil94] P. Mills, “Parallel programming using linear variables,” Draft Technical Report, Duke University, 1994.

[MNP+91] P. Mills, L. Nyland, J. Prins, J. Reif and R. Wagner, “Prototyping parallel and distributed programs in Proteus,” *Proc. Third IEEE Symp. on Parallel and Distributed Processing*, Dallas, Texas, pp. 10-19, 1991.

[MNPR92a] P. Mills, L. Nyland, J. Prins and J. Reif, “Prototyping N-body simulation in Proteus,” *Proc. Sixth International Parallel Processing Symp.*, Beverly Hills, CA, pp. 476-482, 1992.

[MNPR92b] P. Mills, L. Nyland, J. Prins and J. Reif, “Prototyping high-performance parallel computing applications in Proteus,” *Proc. 1992 DARPA Software Technology Conf.*, Los Angeles, CA, pp. 433-442, 1992.

[MNPR94] P. Mills, L. Nyland, J. Prins and J. Reif, “Software issues in high performance computing and a framework for the development of HPC applications,” Technical Report, UNC, May 1994.

[Mor79] T. More, “The nested rectangular array as a model of data,” *Proc. APL79*, 1979.

[MPI93] MPI Forum, “MPI: A Message Passing Interface,” *Supercomputing ‘93*, 1993.

[MPR93] P. Mills, J. Prins and J. Reif, “Rate control as a language construct for parallel and distributed programming,” *IEEE Workshop on Parallel and Distributed Real-Time Systems (IPPS'93)*, pp. 164-170, 1993.

[MT84] T. Maibaum and W. Turski, “On what exactly is going on when software is developed step-by-step,” *7th Int. Conf. Softw. Eng.*, pp. 528-533, 1984.

[NPMR93] L. S. Nyland, J. F. Prins, P. H. Mills and J. H. Reif, “The Proteus solution to the NSWC prototyping experiment,” Technical Report, UNC, Nov 1993.

[NPR93] L. S. Nyland, J. F. Prins and J. H. Reif, "A data-parallel implementation of the fast multipole algorithm," *DAGS '93*, Dartmouth College, Hanover NH, 1993.

[Nyl93] L. S. Nyland, "Transfer of data and control between Proteus and other programming languages," Technical Report, UNC, Feb 1993.

[ORA91] O'Reilly & Associates, *Guide to OSF/1*, 1991.

[Pal93] D. W. Palmer, "DPL—Data-Parallel Library manual," Technical Report TR93-064, UNC, Nov 1993.

[PP93] J. Prins and D. Palmer, "Transforming high-level data-parallel programs into vector operations," *Proc. Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Diego, CA, pp. 119-128, 1993.

[Pur94] J. M. Purtilo, "The POLYLITH software bus," *TOPLAS*, 16(1), 1994.

[Rep91] J. H. Reppy, "CML: A higher-order concurrent language," *ACM Conf. on Programming Language Design and Implementation*, pp. 293-305, 1991.

[Sch70] J. Schwartz, "Set theory as a language for program specification and programming," Technical Report, Courant Institute of Mathematical Sciences, New York University, 1970.

[Smi89] D. R. Smith, "KIDS: A semi-automatic program development system," Technical Report, The Kestrel Institute, Oct 1989.

[ST88a] D. Sannella and A. Tarlecki, "Specifications in an arbitrary institution," *Inf. and Comput.*, 76:165-210, 1988.

[ST88b] D. Sannella and A. Tarlecki, "Towards formal development of programs from algebraic specifications: Implementations revisited," *Acta Inf.*, 25:233-281, 1988.

[ST89] D. Sannella and A. Tarlecki, "Toward formal development of ML programs: Foundations and methodology," in *Lecture Notes in Computer Science*, v. 352, pp. 375-389, Springer-Verlag, 1989.

[TM87] W. M. Turski and T. S. E. Maibaum, *The Specification of Computer Programs*, Addison-Wesley, 1987.

[Val90] L. G. Valiant, "A bridging model for parallel computation," *Comm. of the ACM*, 1990.

[VM92] P. A. Veloso and T. Maibaum, "On the modularization theorem for logical specification," Technical Report, 1992.

[VS93] J. S. Vitter and E. A. M. Shriver, "Algorithms for parallel memory II: Hierarchical multilevel memories," *Algorithmica*, 1993.

[Wir90] M. Wirsing, "Algebraic specification," in *Handbook of Theoretical Computer Science*, (J. van Leeuwen, Ed.), v. B, pp. 675-788, MIT Press/Elsevier, 1990.