Prototyping N-body Simulation in *Proteus*[‡]

Peter H. Mills[†], Lars S. Nyland[*], Jan F. Prins[*], John H. Reif[†]

[†]Department of Computer Science,
Duke University,
Durham, N.C. 27706

[*]Department of Computer Science,
University of North Carolina,
Chapel Hill, N.C. 27599-3175 USA

Abstract

This paper explores the use of *Proteus*, an architecture-independent language suitable for prototyping parallel and distributed programs. *Proteus* is a high-level imperative notation based on sets and sequences with a single construct for the parallel composition of processes communicating through shared memory. Several different parallel algorithms for N-body simulation are presented in *Proteus*, illustrating how *Proteus* provides a common foundation for expressing the various parallel programming models. This common foundation allows prototype parallel programs to be tested and evolved without the use of machine-specific languages. To transform prototypes to implementations on specific architectures, program refinement techniques are utilized. Refinement strategies are illustrated that target broad-spectrum parallel intermediate languages, and their viability is demonstrated by refining an N-body algorithm to data-parallel CVL code.

1. Introduction

Over the past twenty years many diverse parallel machine architectures have been developed with different models of computation, which in turn have given rise to a great variety of parallel programming languages which to some degree reflect the underlying machine organization. For example, *distributed systems* and *distributed-memory multiprocessors* such as the Intel iPSC and its descendants are typically programmed using the concepts of processes and message-passing. Languages for these asynchronous distributed-state systems include CSP [Hoa85] and Strand [FT90]. *Shared-memory multiprocessors*, like the Multimax or the Sequent, are typically programmed using languages that support shared variables with access-exclusion and synchronization mechanisms like monitors, such as found in Concurrent Pascal, or threads such as found in Mach [BRS+85]. *Highly-parallel processors* such as the TMC CM-2 or the MasPar MP-1 are programmed using data-parallel operations and barrier synchronization. Families of abstract computational models for these classes of synchronous and asynchronous shared-memory machines may be found in the PRAM and APRAM respectively [CZ89].

The proliferation of languages following different concurrent programming paradigms targeting different architectures, together with the emergence of heterogeneous systems and mixed-mode architectures, pose problems for the development of parallel software. The diversity of languages and paradigms increases the complexity of programming and reduces software portability, reuse, and reliability. One solution is an *architecture-independent* approach, in which parallel applications can initially be developed independently of the target machines, and then specialized to run on particular target architectures as desired. This approach can be realized using a multiparadigm concurrent language which provides a foundation supporting diverse paradigms, together with algorithmic and data refinement techniques that target specific classes of architectures.

A high-level architecture-independent language, when coupled with a strategy to transform and specialize parallel programs, can be particularly useful in an evolutionary approach to software development. Early working models, or prototypes, serve to rapidly validate requirements and explore parallel execution strategies without incurring the cost of expressing programs in low-level machine-specific languages. Prototypes that exhibit the correct behavior are then further refined and transformed into parallel programs on desired machines.

In this paper we briefly review the salient features of *Proteus*, a language we are developing specifically to support the prototyping of parallel and distributed programs. A more detailed description of the language and comparison with other approaches can be found in [Nyl91, MNP+91]. *Proteus* provides a high-level set-theoretic notation together with a sparse but powerful set of mechanisms for controlling parallel execution. A shared-memory model is the basis for communication between processes: this memory can be partitioned into *shared* and *private* variables. Parallel processes operate on individual copies of private variables, which are independently updated and may be *merged* into the shared state at specifiable barrier synchronization points. These mechanisms support diverse concurrent programming styles within a single logical framework.

To demonstrate the expressive power of the language, we use *Proteus* to express several prototype parallel algorithms for the N-body problem in compu-

[‡]This work was supported under DARPA / SISTO contract N00014-91-C-0114 administered through ONR.

In *Proceedings of the Sixth International Parallel Processing Symposium*, (Beverly Hills, Ca., March 23-26), pp. 476–482, IEEE, 1992.

• Statements:	*assignments, procedure calls*	
• Guarded commands:	<expr> → <stmt>	
• Operators over statement sequences:		**Syntactic abbreviation**
Sequence:	**seq** $[\boldsymbol{S}_1, \ldots, \boldsymbol{S}_n]$	$(\boldsymbol{S}_1; \ldots; \boldsymbol{S}_n)$
Choice:	**alt** $[\boldsymbol{B}_1 \rightarrow \boldsymbol{S}_1, \ldots, \boldsymbol{B}_n \rightarrow \boldsymbol{S}_n]$	$(\boldsymbol{B}_1 \rightarrow \boldsymbol{S}_1 [] \ldots [] \boldsymbol{B}_n \rightarrow \boldsymbol{S}_n)$
Repetition:	**rep** $[\boldsymbol{B}_1 \rightarrow \boldsymbol{S}_1, \ldots, \boldsymbol{B}_n \rightarrow \boldsymbol{S}_n]$	$(\boldsymbol{B}_1 \rightarrow \boldsymbol{S}_1 [] \ldots [] \boldsymbol{B}_n \rightarrow \boldsymbol{S}_n)*$
Parallel Composition:	**par** $[\boldsymbol{P}_1, \ldots, \boldsymbol{P}_n]$	$(\boldsymbol{P}_1 \| \ldots \| \boldsymbol{P}_n)$

Figure 1: **Control primitives in *Proteus***

tational physics. We then investigate strategies for refinement of the algorithms to intermediate languages suited for particular classes of machine architecture. One variant is targeted to a data-parallel SIMD execution model, while another variant is evolved towards an MIMD execution model. We conclude the paper with a discussion of directions of ongoing research.

2. Basic features of *Proteus*

Our language starts with rich data models and operators along the lines of SETL [SDDS86] and REFINE [Ref88], which employ the high-level mathematical notions of sets, sequences, and maps. The core of our language is a conventional imperative notation to the degree that it is assignment-based and block-structured; program state is maintained in typed, lexically-scoped variables, and assignment statements or procedure calls modify this state. Sets and sequences may be constructed by enumeration or by generation based on another set or sequence. Generators are of the form:

{expr(x) : x **in** set | pred(x)}   *(set)*

[expr(x) : x **in** sequence | pred(x)]   *(sequence)*

For example,

{i*i : i **in** {0..5} | (i < 3)}

has value $\{0, 1, 4\}$. Standard operations on sets and sequences are present, such as concatenation (##) and indexing on sequences, and union and arbitrary choice on sets. Also present is the APL-like reduction operation f/S which applies a binary function f between the elements of sequence S. In addition *Proteus* supports segmented reduction, written as $(f, D)/S$, which performs reduction separately on each subsequence (or *segment*) of a sequence S partitioned by another sequence D of segment lengths. For example,

$$(+, [2, 3])/[1, 2, 3, 4, 5]$$

yields the value $[3, 12]$.

Functions and statements are also values in *Proteus*. For example, the assignment

f := **func**(n) (**return** n+x);

yields as a value for f the closure of the function in the lexically-scoped environment. As a result, higher-order functions such as the reduction operation can be defined directly, as is the case in ISETL.

However, unlike SETL or ISETL, statement values can also be formed. This allows the expression of familiar control constructs – such as sequential composition – as operators over sequences of statements, yielding a flexible and extensible control regime. Figure 1 summarizes a number of control operators over sequences of statements and the familiar syntax that may be used when all of the statement values are explicit rather than generated. While the guarded command constructs behave similarly to those of Dijkstra and Hoare [Dij78, Hoa85] — for example, the **rep** operator repeatedly executes one command selected arbitrarily from those with true guards until all guards are false — *Proteus* provides greater expressive power by permitting operands to be dynamically generated by sequence construction.

2.1. Constructs for concurrency

Our language supports parallelism with one simple parallel composition operator. The statement $(P_1 \| P_2)$ specifies "**cobegin/coend**"-like concurrent execution of the two statements P_1 and P_2 which we call processes. No assumptions about atomicity, interleaving, or relative rates of progress of P_1 and P_2 are made. Processes communicate through global state, which admits a potential for interference problematic to many shared-memory models.

To control interference in our model of unconstrained parallelism, *Proteus* models a division of state into distributed and shared memory through the introduction of *private* and *shared* variables. Our technique exploits the standard scope rules for block-structured languages. Within a parallel composition each process can reference any variable visible according to these scope rules; but now each non-local variable is further specified to be either private or shared. A **shared** variable is a single entry in the state, whereas a **private** variable has an entry in each process in the parallel construct which shadows the entry in the enclosing scope. The initial value of a private variable v is the same in all processes in the construct and is the value of v in the enclosing scope. Operations on shared variables may interfere with each other since they all refer to the same state, but operations on private variables can never interfere.

The following example illustrates how this concept of private variables naturally fits with standard scoping rules.

var a, b, c;

(**private** c; **var** b; P_1) || (**private** c; P_2)

We assume that by default all non-local variables are shared, and hence the names of private variables must be declared in each process. In this example, the shared variable a is seen by both P_1 and P_2, but private copies of c are held by each. Analogous to other *Proteus* control constructs, the syntax in this example is an abbreviation for an operator over statement sequences, of the form:

par [(**private** $v_1, \ldots, v_k$; P_1) ,...,
(**private** $v_1, \ldots, v_k$; P_n)]

Since private declarations are most often identical in each parallel process, they may be compactly introduced into sequence generators; *Proteus* also allows abbreviation of the enumerated parallel operator as:

(**private** $v_1, \ldots, v_k$ **in** $P_1 \| \ldots \| P_n$)

The mechanism by which processes can communicate information from the private state back up into the global state is a simple primitive combining two-way communication and synchronization. The *barrier-merge* operation

merge $v_{i'}, \ldots, v_{k'}$

may be invoked within the processes P_i, and delays the process containing the operation until all other processes in the composition have reached a **merge** operation. This effects *barrier synchronization.*

At this point, the private state is *merged* into the global state. Each private variable has its values in all processes combined using a specified merge function f, and the result updates the corresponding variable in the enclosing scope. This combining action is similar to that used to resolve conflict in message collisions [Sab88], although *Proteus* applies the reduction of f only across the *changed* values from all processes. While in our examples the merge function defaults uniformly to arbitrary selection, *Proteus* also permits merge functions to be individually specified for each variable through declarations of the form:

private $v_1, \ldots, v_k$ **using** f

The last step in the **merge** operation is to copy the global state back into each private state. If so specified, only a subset $v_{i'}, \ldots, v_{k'}$ of the private variables will be updated. Furthermore, for safety a **merge** operation implicitly occurs at the end of every parallel composition.

Proteus provides one further mechanism for synchronization. The conditional await construct:

await $[B_1 \rightarrow S_1, \ldots, B_n \rightarrow S_n]$

waits for a true guard B_i and then executes the guard and statement $B_i \rightarrow S_i$ atomically, i.e., while excluding all other processes. It follows that **await** $[true \rightarrow S]$ is equivalent to atomic execution of S, which we abbreviate as $\ll S \gg$.

2.2. Related work

A variety of parallel languages are cited as being useful for programming broad classes of concurrent systems. These languages might be roughly divided into the following categories.

- Languages with widely translatable logical models, such as Linda's distributed data structures [CGL86], the synchronization-variable methods of Strand [FT90] and PCN [CT92], or the data-parallel abstraction of the Paralation model [Sab88].
- Languages which incorporate a large variety of parallel primitives, such as Ease [Zen90].
- Wide-spectrum parallel languages that rely on refinement from architecture-independent specification. Notable wide-spectrum parallel language efforts include Crystal [Che86] and variants of the Bird-Meertens functional formalism [Ski90]. UNITY [CM88], although not a wide-spectrum notation, is, as its name suggests, a particularly elegant notation for describing a large range of parallel and distributed computations.

We see *Proteus* as falling into the last category. All of these wide-spectrum languages support a methodology in which parallel specifications are refined to parallel programs for a particular class of machine. In the case of Crystal, UNITY, and the Bird-Meertens formalism the refinement steps are justified formally through inference steps or algebraic transformations.

In comparison with these languages *Proteus* supports fundamental parallel abstractions at a higher level than UNITY and at a lower level than Crystal, where concurrency is implied by independence in the equational specification. While *Proteus* incorporates some declarative features, for utility in prototyping it is oriented towards procedural specification and can refer to shared state explicitly. UNITY programs also manipulate shared state, but the control of interference is implicit by constraining execution to statement-level interleaving.

3. N-body simulation

We now present a simple example to illustrate the diversity of parallel computations that can be accommodated in *Proteus*, and to examine refinement strategies that can target specific parallel architectures. The problem is N-body simulation — that is, given a collection of N particles (or *bodies*) distributed in space, to simulate the motion of the particles over time due to gravitational or electrostatic interaction. Each step of the simulation over time consists of calculating for each body the sum of forces due to pairwise interaction with all other particles, and then updating the position of each particle as a function of this force. The N-body problem characterizes physical phenomena that arise in many important applications in fields such as astrophysics, plasma physics, and molecular dynamics [App85, Gre90]. While numerical solutions for the N-body problem are thus critically needed, they unfortunately require large amounts of computation due to the typically large number of particles and nature of the N^2 pairwise interaction.

Many algorithmic refinements have been proposed to render N-body simulation more tractable. These include methods which approximate interaction of a particle with a cluster of particles that are far away by modeling the cluster as a single particle (so-called *far-field* interactions) [App85], and *tree-code* methods which compute far-field interactions by recursively decomposing the spatial domain [BH86]. A further optimization is obtained by the Fast Multipole Method [Gre90], which uses multigrid techniques and multipole approximations for far clusters to yield a faster and more accurate algorithm. To further decrease computational complexity, parallel implementations of these algorithms have also been explored [GG89], in particular on data-parallel architectures such as the Connection Machine [ZJ89].

In following sections we consider two parallel algorithms for N-body simulation. We first present a simple N^2 interaction per step simulation, and refine this algorithm toward a highly-parallel SIMD architecture. Next a variant of the algorithm is considered that utilizes far-field interactions. This algorithm is refined to target large-grain MIMD machines with SIMD or vector processors, such as the CM-5 or the Cray Y-MP. In both cases we target intermediate languages rather than machine-specific low-level languages.

3.1. Direct algorithm in *Proteus*

The most naive and direct solution to the N-body problem is to accumulate for each particle the force due to *all* pairwise interactions, and then calculate the new position of the particle as a result of this force. This treats all particle interactions as *near-field* interactions. We consider the direct solution for the general case of an arbitrary interaction function f. Let P be a vector of N bodies, each body described by a tuple (position, velocity, mass). Furthermore, assume we are given the following functions:

f: body × body → force
g: force × force → force
h: force × body × time → body

where f computes the force vector between two bodies (we assume $f(x,x) = 0$), g adds force vectors, and h computes the body's position and velocity given the force acting on the body and its duration. We will not present the details of f, g, and h, other than noting that they consist of scalar operations (e.g. addition) and scalar operations extended over vectors.

The algorithm for performing one iteration of the simulation, with step duration d, is succinctly expressed in Proteus as:

P := [h(g/[f(P(i),P(j)): j **in** [1..N]], P(i), d)
  : i **in** [1..N]];

The outer sequence generator computes a new position for each particle $P(i)$ using the function h. The total force on a particle $P(i)$ is obtained using g to reduce (via addition) the sequence of forces between $P(i)$ and every particle in P.

3.2. Execution of prototypes

While programs in *Proteus* should be able to run on parallel platforms, it is not our intention that any single program execute well on all parallel platforms. Early prototypes that explore specifications are likely to be expressed independent of a specific class of platforms, and initially executed on sequential machines. Prototypes can then evolve to use *Proteus* in more restricted ways that are in close correspondence with a particular architecture or programming model. In common with other architecture-independent languages, refinement can help achieve this architectural specialization. However, it is important in prototyping to distinguish between refinement of prototypes, which refers to meaning-preserving transformations, and evolution of prototypes, which may include broader algorithmic changes: both may be required to effect execution on specific architectures.

Refinement strategies whereby a program is specialized to a particular subset of the language and mechanisms for the translation of such a subset to run on a parallel platform are being developed in conjunction with our colleagues at the Kestrel Institute, building on their environments for transformational program development. The KIDS system (Kestrel Interactive Development System) [Smi90] has been used to develop programs from specifications, and includes a number of algorithm design tactics and data refinement transformations [BG90].

Providing refinement techniques to target many specific architectures is likely to be prohibitive, hence our strategy is to refine to existing or proposed intermediate languages which permit reasonably efficient execution on a class of parallel platforms. For example, we intend initially to reduce data-parallelism to the set of parallel vector operations provided by the CVL library [BCSZ90], developed by Guy Blelloch and colleagues at Carnegie-Mellon as a machine-independent library used in the interpretation of the data-parallel intermediate code VCODE [BC90]. Likewise, we intend to reduce process parallelism to the set of procedures provided with the threads facility of Mach [BRS+85].

3.3. Refinement to SIMD

We now apply these strategies to our N-body program to yield execution on an SIMD architecture. Although sequence generators that evaluate simple scalar functions of their index sets are well-suited to SIMD execution, the N-body program given in section 3.1 would not achieve a large degree of parallelism with this approach. The problem is that the nested sequence generators correspond to *nested parallelism*, which can not be implemented directly under the SIMD execution model. In order to target SIMD execution, we must refine the program to separate the reduction operation from the nested sequence generators, and combine the nested sequence generators into one. We do this by following techniques outlined in [BS90], yielding a form of the program that can be translated to vector operations. In this case we performed the refinement and translation manually, but based on insights gained from this experiment we are developing tools in the Refine system to perform these steps semi-automatically.

The refinement step rewrites the N-body program to evaluate f on the Cartesian product of the particles with themselves, and to replace N reduction operations with a single segmented reduction operation. The new version of the program, shown below, remains an executable *Proteus* program, so that we can validate the refinement experimentally.

Q := [P[i] : i, j **in** [1..N]]
R := [P[j] : i, j **in** [1..N]]
F := [f(Q[i],R[i]): i **in** [1..N^2]]
V := (g, S) / [i **in** [1..N^2] : F[i]]
    where S = [N: i **in** [1..N]]
P := [h(V[i],P[i],d) : i **in** [1..N]]

The form of this refinement is driven by the facilities and limitations of the targeted language with which the refined operations must be compatible. In this case this language was C with calls to CVL. This library supports elementary scalar operations extended elementwise between vectors, as well as segmented reduction and several simple forms of creating a vector by replication. The translation yields a series of vector operations:

```
S := distribute(N, N)
Q, R := cartesian(P, P)
F := elwise(f, Q, R)
V := seg_reduce(g, F, S)
P := elwise(h, V, P)
```

However, further refinements of the *Proteus* program are needed to accommodate some additional limitations of CVL. These limitations are:

1. CVL vector operands must be vectors of scalars.
2. CVL does not support elementwise extension of user-defined scalar functions.
3. CVL does not support reduction using user-defined combining functions.

These limitations require transformations which:

1. Flatten operations on structures, that is, transformations that turn P into scalar vectors P-x, P-y etc., and others which break up f into operations on these scalar vectors.
2. Transform elwise(f,...) operations into applications of the vector extension of f derived by converting scalar operations in f to vector operations. This corresponds to compiling Paralation "elwise" forms into vector operations, a topic addressed in detail in [BS90].
3. Transform segmented reduction (using g) into vector operations, either by deriving from the scalar operations of g a sequence of segmented reductions, or by using the vector extension of g to implement the reduction using well-known techniques such as doubling.

In conjunction with Kestrel Institute, we are investigating ways to formalize and mechanize these refinement strategies. One promising avenue for data type and algorithmic refinement relies on techniques to recognize the presense of CVL patterns such as elwise and product operations, as well as transformational strategies based on algebraic laws for functional languages [Ski90].

4. Far-field approximation

We now consider an evolution of the original *Proteus* N-body simulation program that yields a parallel algorithm suitable for targeting asynchronous collections of SIMD processors. We take advantage of the far-field approximation mentioned earlier. The particle space can be partitioned into clusters of near particles, and any two clusters are either *well-separated* — meaning their separation exceeds some accuracy criterion so that far-field approximation can be used — or they are *neighbors*. Interaction of any particle and those in in a well-separated cluster can be approximated by interaction of the particle with a monopole approximation of the cluster, for example a point-mass placed at the cluster's center of mass.

The spatial decomposition into clusters naturally supports parallelization by mapping one cluster to each of p parallel processes. In each step of the simulation, each process computes interactions for its particles by examining other clusters, using far-field interactions if possible, and if not then computing roughly $(N/p)^2$ pairwise interaction as before. Next the particle positions are updated. Since particles may have to be migrated between neighboring clusters, a list of particle additions and deletions is built for each cluster and then used to perform migration. Lastly the monopole approximations are recomputed.

4.1. Refinement to MIMD

The *Proteus* program incorporating these refinements is shown in Figure 2. In this program, the **merge**s perform synchronization required to avoid interference when computing pairwise interactions or when completing the migration list before using it to move particles between clusters. To execute this prototype, the control-parallelism of the group of parallel cluster processes can be straightforwardly implemented, if needed, in terms of threads (for example in Mach [BRS+85]). A key point is that each cluster process can be vectorized since it uses the same techniques as the simple *Proteus* program for pairwise interaction. This makes this far-field program, although clearly not optimal, well suited for execution on an asynchronous collection of vector processors with shared memory, for example a CRAY supercomputer.

```
Let Cl =  [Particles(i) : i in [1..p]] be the sequence of clusters, i.e., sequence of sequences of particles.
Let Cen = [Centroid(i) : i in [1..p]] be the monopole approximations for each cluster.
Let ML =  [Migration(i) : i in [1..p]] be the migration-list for each
              cluster, indicating which particles move to or from that cluster.
Let nbor(i) = function returning indices of neighbors of cluster P[i], excluding P[i] itself.
Let pairwise = func(P,Q) return [ g/[ f(P[i],Q[j]): j in [1..#Q]] : i in [1..#P] ];     — cluster interaction

par[                                          — one process per cluster
    var V;
    V := [0 : u in 1..#Cl[i] ];               — initialize accumulated force
    par[                                      — accumulate force
      [ well_separated( Cen(i), Cen(j) ) → V := V + pairwise( Cl[i], [Cen[j]] ),
        not well_separated( Cen[i], Cen[j] )  → V := V + pairwise( Cl[i], Cl[j] ) ]
      : j in [1..p] ];
    merge;
    Cl[i] :=  [ h(Cl[i][j],V[j]) : j in [1..#Cl[i]] ]       — update position
    ML[i],[ML[k]: k in nbor(i)] := update_migration_list( Cl[i], ML[i] ## [ML[k] : k in nbor(i)] );
    merge;
    Cl[i] := migrate_particles ( Cl[i], ML[i] );            — add and delete migrated particles
    Cen[i] := compute_centroid( Cl[i] );
: i in [1..p] ]
```

Figure 2: **N-body simulation using far-field interaction**

4.2. Refinement to isolate state

In the case of a collection of asynchronous processors with distributed memory, we can reduce communication and synchronization overhead by using the **private** mechanisms of *Proteus* to copy and localize access to state otherwise shared. The information shared between clusters during one step of the simulation is the list of centroids, and the neighboring clusters together with their migration lists. These neighbors constitute the *boundary*. We make these items private in the *Proteus* program shown in Figure 2 by adding the following declarations after the first **par** construct:

```
private Cen;
private Cl;
private ML using ##;
```

The basic idea is that each cluster process has its own copy of its neighbors and their migration lists; it examines the former to compute pairwise interaction and may update the latter to reflect particle motion into neighboring clusters. Isolating this state removes the need for the first **merge** synchronization command, since the clusters cannot interfere. The migration lists, when completely updated, are then **merge**d using sequence concatenation and subsequently used to add and delete particles for each cluster. At the end of the simulation step the private clusters are merged: this effects exchange of boundary cells between clusters since they are effectively recopied as privates in the next **par** cycle. This corresponds to exchanging guard strips in other multigrid simulations [F$^+$88]. Note that, although the entire cluster array CL is declared as private, *Proteus* implementations need only copy on demand referenced boundary elements.

The same technique of state isolation can be applied to parallelize further optimizations of the N-body simulation, specifically for Barnes-Hut tree-codes [BH86] and the Fast Multipole Method [Gre90]. Both employ a hierarchical decomposition of cluster space, such as quad-trees for 2D or oct-trees for 3D, which can be used to recursively partition areas of otherwise near-body interaction so as to treat them as far-body effects. We are examining prototypes of parallel Fast Multipole algorithms in *Proteus* which isolate boundary cell communication while at the lowest level are vectorizable — how these can be efficiently implemented on asynchronous collections of SIMD processors, or can be refined into existing data-parallel Multipole methods [ZJ89] is still being investigated.

5. Summary and future work

In this paper we explored the use of *Proteus*, a prototyping language whose constructs for parallelism can serve as a foundation for expressing many concurrent programming models. In conjunction with refinement techniques for architectural specialization, *Proteus* can provide a powerful platform for the construction and execution of a wide spectrum of parallel prototypes. Prototyping and refinement of algorithms for N-body simulation were explored, illustrating the expressive power of *Proteus* and demonstrating the viability of refinement strategies for execution of prototypes, in particular techniques which target intermediate parallel languages to gain wide applicability.

Ongoing work in the area of the *Proteus* language design is concentrated in several areas. First, we are are investigating the inclusion of higher-level features for distributed programming, using the notion of concurrent objects as the basis of an approach to controlling process parallelism [Agh90]. Second, we are investigating the modeling of time-constrained computation in the form of annotations for the relative execution rates of processes.

Finally, we are currently involved in the implementation of key features of the language and refinement system to assess the suitability of the approach. The long-term goal of the work is to incorporate *Proteus* into a prototyping system that links several prototyping languages, targeting different problem domains, to form an effective vehicle for the development and assessment of full system prototypes.

Acknowledgements

The authors would like to thank Allen Goldberg and Dan Palmer for their constructive advice and comments.

References

[Agh90] G. Agha, "Concurrent object-oriented programming," *Comm. ACM*, vol. 33, pp. 125–141, Sept. 1990.

[App85] A. W. Appel, "An efficient program for many-body simulation," *Siam J. Sci. Stat. Comput.*, vol. 6, pp. 85–103, Jan. 1985.

[BC90] G. Blelloch and S. Chatterjee, "VCODE: a data-parallel intermediate language," in *Proc. Frontiers 90*, IEEE, 1990.

[BCSZ90] G. Blelloch, S. Chatterjee, J. Sipelstein, and M. Zahga, "CVL: A C vector library," Draft Technical Note, Carnegie Mellon University, Dec. 1990.

[BG90] L. Blaine and A. Goldberg, "Modules and types for a common prototyping language," Technical Report, Kestrel Institute, Palo Alto, California, Oct. 1990.

[BH86] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.

[BRS+85] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young, "Mach-1: An operating environment for large-scale multiprocessor applications," *IEEE Software*, July 1985.

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

[CGL86] N. Carriero, D. Gelernter, and J. Leichter, "Distributed data structures in Linda," in *Proc. 13th ACM Symp. on Principles of Programming Languages*, pp. 236–242, ACM, 1986.

[Che86] M. Chen, "Very-high-level parallel programming in Crystal," in *Proc. 1986 Hypercube Conference*, (Knoxville,Tn.), 1986.

[CM88] K. Chandy and J. Misra, *Parallel Program Design, A Foundation.* Addison-Wesley, 1988.

[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," in *Proc. 1st ACM Symp. on Parallel Algorithms and Architectures*, pp. 169–178, ACM, 1989.

[Dij78] E. Dijkstra, "Guarded commands, nondeterminacy and the formal derivation of programs," *Comm. ACM*, vol. 18, pp. 453–457, 1978.

[F+88] G. Fox *et al.*, *Solving problems on concurrent processors.* Prentice-Hall, 1988.

[FT90] I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[GG89] L. Greengard and W. Gropp, "A parallel version of the fast multipole method," in *Parallel Processing for Scientific Computing* (G. Rodrigue, ed.), p. 213, SIAM, 1989.

[Gre90] L. Greengard, "The numerical solution of the N-body problem," *Computers in Physics*, pp. 142–152, Mar. 1990.

[Hoa85] C. Hoare, *Communicating Sequential Processes.* Addison-Wesley, 1985.

[MNP+91] P. H. Mills, L. S. Nyland, J. F. Prins, J. H. Reif, and R. W. Wagner, "Prototyping parallel and distributed programs in *proteus*," in *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing*, IEEE, 1991.

[Nyl91] L. S. Nyland, *The Design of A Prototyping Programming Language for Parallel and Sequential Algorithms.* Ph.D. dissertation, Duke University, Feb. 3 1991.

[Ref88] Reasoning Systems, Inc., Palo Alto, California, *Refine 2.0 Language Summary*, Aug. 1988.

[Sab88] G. Sabot, *The Paralation Model: Architecture-Independent Parallel Programming.* MIT, 1988.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets, An Introduction to SETL.* Springer-Verlag, 1986.

[Ski90] D. Skillicorn, "Architecture-independent parallel computation," *IEEE Computer*, vol. 23, pp. 38–50, Dec. 1990.

[Smi90] D. R. Smith, "KIDS – a semi-automatic program development system," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1024–1043, Sept. 1990.

[Zen90] S. Zenith, "Programming with Ease: a semiotic definition of the language," Research Report RR809, Yale University, July 1990.

[ZJ89] F. Zhao and S. L. Johnsson, "The Parallel Multipole Method on the Connection Machine," Research Report CS89-6, Massachusetts Institute of Technology, Oct. 1989.