SPECIFICATION AND DEVELOPMENT OF PARALLEL ALGORITHMS WITH THE PROTEUS SYSTEM

ALLEN GOLDBERG, PETER MILLS, LARS NYLAND, JAN PRINS, JOHN REIF, AND JAMES RIELY

ABSTRACT. The Proteus language is a wide-spectrum parallel programming notation that supports the expression of both high-level architecture-independent specifications and lower-level architecture-specific implementations. A methodology based on successive refinement and interactive experimentation supports the development of parallel algorithms from specification to various efficient architecture-dependent implementations. The Proteus system combines the language and tools supporting this methodology. This paper presents a brief overview of the Proteus system and describes its use in the exploration and development of several non-trivial algorithms, including the fast multipole algorithm for N-body computations.

1. INTRODUCTION

Practical implementations of parallel algorithms that access the performance potential of current computers are difficult to develop, too often fail to deliver the expected performance, and lack portability to other platforms. 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 (e.g., shared vs. distributed memory, SIMD vs. MIMD, exposed vs. hidden interconnection network). These notations lack portability across architectures and are too low-level to support the exploration of complex designs. Higher-level notations, on the other hand, trade reduced access to architecture-specific features for improved abstract models of computation, but this trade is often not the right one: the whole point of parallelism, for most applications, is performance.

The problem is a fundamental one: abstract models of parallel computation lead to impractical implementations, whereas machine-specific models lead to intractable analysis of even the simplest programs. The goal of our work is to provide tools for exploring the design space of a parallel application by a process of prototyping and successive refinement.

The Proteus system comprises:

1991 *Mathematics Subject Classification.* Primary: 68Q10,68N15,68N20; Secondary: 68Q22.

Key words and phrases. Parallel program development, specification, prototyping, refinement, data parallelism, process parallelism, parallel fast multipole algorithm.

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

- a wide-spectrum parallel programming notation that allows high-level expression of specifications,
- a methodology for (semi-automatic) refinement of architecture-independent specifications to lower-level programs 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.

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

This paper gives a brief overview of the Proteus system and experiences with its use. In the next section, we describe the Proteus programming language in more detail. Section 3 then discusses the methodology and environment. Section 4 reviews some of our experiments with the system. Finally, we conclude in Section 5.

2. Proteus Programming Notation

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 [15].

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 provide aggregate values (such as sets or sequences) and the ability to apply functions independently on every element. This sort of expressive capability is found 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.

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 $\{\{(p,q) \mid (q \in A) \wedge (p \geq q)\} \mid \{p \in A\}\}$ denotes the set

$$\{\{(1,1)\},\{(2,1),(2,2)\},\{(3,1),(3,2),(3,3)\}\}.$$

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 gives rise to a great potential for concurrent evaluation since the number of independent sub-expressions can be very large.

The Proteus iterator construct captures the essence of comprehensions. 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 widely applicable, as we demonstrate here by writing a data-parallel quicksort algorithm (adapted from [3]). 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 or empty. The final value is obtained by concatenating the (sorted) `lesser`, `equal` and `greater` lists. In Proteus, this may be coded as:

```
function qsort(list)
   return if #list <= 1
            then list; — if empty or singleton
            else let
                   greater = 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.

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 expressed conveniently in languages such as High Performance FORTRAN, in which all aggregates must be rectangular and non-nested.

The utility of nested data-parallelism has long been established in high-level languages like SETL and APL2. Blelloch [3] showed that nested and irregular data-parallelism can be vectorized. We have developed a set of transformations that translate Proteus data-parallelism to the portable low-level vector model CVL [24, 2].

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.

2.2.1. *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 server(j);}  || master(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.

2.2.2. *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 [19]. 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 [13].

2.2.3. *Shared Objects.* Communication and synchronization between dynamic processes is more generally provided within the framework of object classes through three simple techniques.

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. Object values are always references; only method-invocation dereferences such values. Under this scheme the private memory model and merge mechanism will apply uniformly to variables, whether they hold object references or private 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 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 for one process wait for another to reach a given point or to provide a result. 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, much like a multilisp future. The `|>` construct causes the spawned function to write a value of type sync when it completes. For example, given a function f`:int->int`, then

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

spawns `f(y)` and invokes `x.write` with the result. If `x.read` is attempted before `f` has completed, the caller is suspended until the value is available.

The class `shared(t)` provides mutually excluded 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 [16]. 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 in user-defined classes to build higher-order abstractions such as buffered channels.

Related work on concurrent languages which embody the notion of sync variables includes Compositional C++ [7] and PCN [6]. 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 [5] (which however exclude only concurrent invocations of a single method). Our linear operators attempt to achieve the goals of CML [26] 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 [1] with guarded blocking communication.

3. Program Development Methodology and Tools

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 [10]; of other work on program transformation, our approach is closest to the "step-by-step" refinement approach of [29]. The relation to software development issues unique to high-performance computing is described in [18].

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 interest 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.

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 1, and the key components are described next.

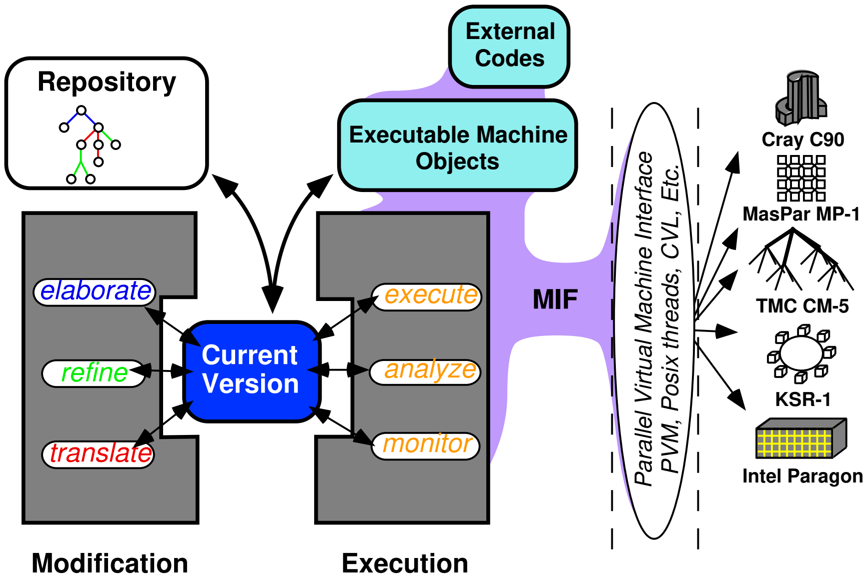

FIGURE 1. The components of the Proteus System

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 [28] 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 CVL is the furthest along. The steps of the translation process are shown in Figure 2. 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 an intermediate notation that can easily be manipulated by the Kestrel system. The program is then vectorized using source-to-source transformations (iterator elimination). Finally the code is translated into C with nested sequence operations. This process is described in detail in [24].

The C Vector Library (CVL) [2] implements operations on vectors of scalar values. CVL provides a consistent interface for vector computation on a variety of parallel architectures, allowing Proteus code to be run today on workstations, the

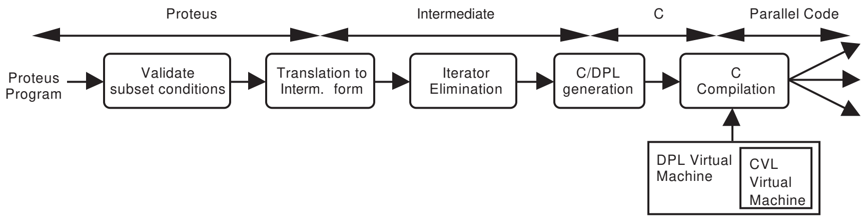

FIGURE 2. Translation of Proteus programs to parallel (vector) code

Connection Machines CM2 and CM5, the Cray Y-MP and C90 and the MasPar MP1 and MP2. 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) [23] is built using operations in CVL and, thus, is also highly portable.

We are investigating transformations for refining Proteus to other parallel virtual machines, implementing asynchronous parallelism with shared or distributed memory. For multi-processor 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) [9] or MPI (Message Passing Interface) [20].

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. A performance model provides a basis for predicting the performance of a program. It is difficult to define an accurate model for high-level code, but as code is refined, so is the performance model; increasingly detailed models become necessary as program refinement progresses. In addition, different models are appropriate for code segments following different paradigms, such as data-parallelism and message-passing.

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 is to use, as program refinement progresses, increasingly detailed parallel computational models. The accuracy and confidence of assessment thus increases as the level of architectural detail incorporated into the program increases. Moreover, to support the assessment of multi-paradigm programs we use different models for analysis of code segments following different paradigms, such as the VRAM [3] for data-parallelism and LogP [8] for message-passing, with suitable instrumentation to "attach" the model to the program. Support for such multiple refined performance-prediction models is under development.

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 [25] is one such system). The Proteus programming system provides a limited MIF capability giving developers the power to build upon, rather than ignore, 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. Examples

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 [17, 22]. 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 [11], 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 [14, 30, 27], but none have explored adaptive parallelization for arbitrary non-uniform distributions in 3D. The reason is 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 a requirement to specify the algorithm with nested sequence expressions. For instance, each region at a particular level (of which there are 8^{level}) 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 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 varying-sized regions (but still rectangular), 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.

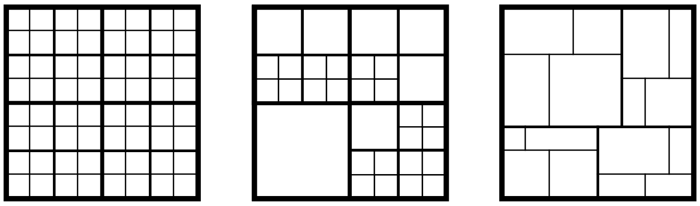

FIGURE 3. A *uniform*, *variable depth adaptive*, and *uniform depth median cut* decomposition in 2-space.

A pictorial representation of the different decompositions is shown in Figure 3 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.

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 [12].

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. It is the high-level notation of Proteus that enables such exploration; 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 [22].

4.2. Geo-server Prototype. Our effort in developing the Proteus system is one of several projects in the ARPA/ONR *ProtoTech* program. As a member of the community, 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 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 2 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, and is documented in [21].

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 (over time) 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 4.

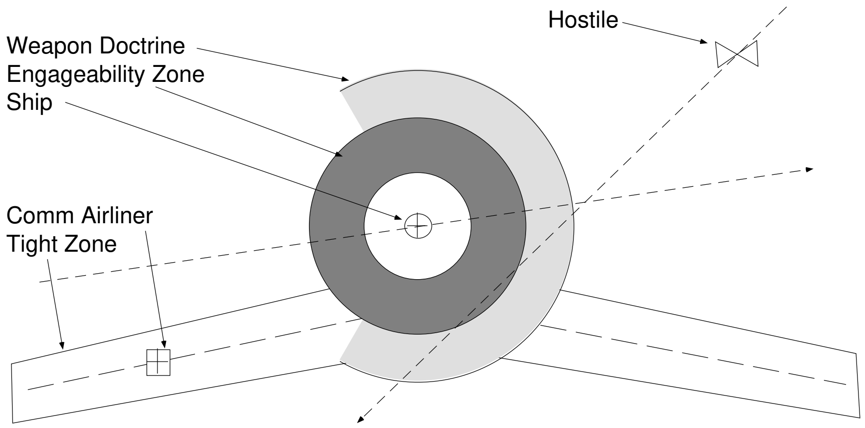

FIGURE 4. An example of regions and radar data for the geo-server prototype

The imagined environment for the geo-server is that it calculates 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. Conclusions

In the Proteus system, we stress 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.

Generally speaking, it is best to to avoid new languages whenever the equivalent capabilities may be achieved by the introduction and implementation of appropriate abstractions in widely used conventional languages such as C with libraries or C++ with classes. There are many hurdles to overcome for a new language: obtaining a clean design, tool support, and most of all, wide-spread adoption. We believe that the constructs and concepts needed in a wide-spectrum parallel programming language of the sort we advocate here can not easily be expressed in current languages. Thus we have developed the Proteus notation, adding a few key concepts to a standard base.

Automated support for refinement and translation of Proteus programs is less far along, and thus 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 to subsets that can be automatically translated. The individual refinement steps in these efforts had well-defined objectives and were quite manageable. After each step, there was great value in executing the new version to compare it with the previous version.

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 non-trivial translation; only Nesl [4] 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 incorporate sophisticated compilation strategies relatively easily as they are developed in the optimizing compiler community.

Prototyping is essential in 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 cannot often be justified. Therefore, we emphasize the use of refinement as a means of evolving prototypes into production code.

6. Acknowledgments

We gratefully acknowledge the help and insight of Rickard Faith, Daniel Palmer and Stephen Westfold in this work.

References

1. P. S. Barth, R. S. Nikhil, and Arvind. *M-structures: Extending a parallel, non-strict functional language with state*, volume 523, pages 538–68. Springer-Verlag, 1991.
2. G. Blelloch, S. Chatterjee, J. Sipelstein, and M. Zahga. CVL: A C vector library. Technical report, Carnegie Mellon University, 1993.
3. G. E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.
4. G. E. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 102–111. ACM, 1993.
5. R. Chandra, A. Gupta, and J. Hennessy. Integrating concurrency and data abstraction in the COOL parallel programming language. Technical report, Stanford University Computer Systems Laboratory, 1992. To appear in IEEE Computer, Feb. 1994.
6. K. Chandy and S. Taylor. *An Introduction to Parallel Programming*. Jones & Bartlett, 1992.
7. K. Mani Chandy and C. Kesselman. Compositional C++ : Compositional parallel programming. In *Proc. of the 4th Workshop on Parallel Computing and Compilers*. Springer-Verlag, 1992.
8. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 1993.
9. J. Dongarra, G. A. Geist, Robert Manchek, and V. S. Sundaram. Integrated PVM framework supports heterogeneous network computing. *J. Computers in Physics*, 7:166–75, 1993.
10. Allen Goldberg, Jan Prins, John Reif, Rickard Faith, Zhiyong Li, Peter Mills, Lars Nyland, Daniel Palmer, and James Riely. The proteus system for the development of parallel applications. Technical report, UNC-CH, May 1994.
11. L. F. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems*. MIT Press, 1987.
12. Ziyad Hakura, Bill Elliot, and John Board. Parallel multipole treecode algorithm. Anonymous FTP: egr.duke.edu, Feb. 1994.
13. Lisa Higham and Eric Schenk. The parallel asynchronous recursion model. In *IEEE Symp. on Parallel and Distributed Processing*, 1992.
14. James F. Leathrum. *The Parallelization of the Fast Multipole Algorithm in Three Dimensions*. PhD thesis, Duke University, 1992.
15. Gary Levin and Lars Nyland. An introduction to Proteus, version 0.9. Technical report, UNC-CH, 1993.
16. P. Mills. Parallel programming using linear variables. Technical report, Duke University, 1994.
17. Peter Mills, Lars Nyland, Jan Prins, and John Reif. Prototyping high-performance parallel computing applications in Proteus. In *Proceedings of 1992 DARPA Software Technology Conference*, pages 433–42. Meridian, 1992.
18. Peter Mills, Lars Nyland, Jan Prins, and John Reif. Software issues in high-performance computing and a framework for the development of HPC applications. Technical report, UNC-CH, May 1994.
19. Peter Mills, Lars Nyland, Jan Prins, John Reif, and Robert Wagner. Prototyping parallel and distributed programs in Proteus. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 10–19. IEEE, 1991.
20. MPI: A message passing interface. In *Supercomputing 93*. MPI Forum, 1993.
21. Lars S. Nyland, Jan F. Prins, Peter H. Mills, and John H. Reif. The Proteus solution to the NSWC prototyping problem. Technical report, UNC, 1993.
22. Lars S. Nyland, Jan F. Prins, and John H. Reif. A data-parallel implementation of the fast multipole algorithm. In *DAGS '93*, 1993.
23. Daniel W. Palmer. DPL— data parallel library manual. Technical report, UNC, 1993.
24. Jan Prins and Daniel Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–28. ACM, 1993.
25. James M. Purtilo. The POLYLITH software bus. *TOPLAS*, 16(1):151, 1994.
26. J. H. Reppy. CML: A higher-order concurrent language. In *Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

27. Jaswinder Pal Singh. *Parallel Hierarchical N-Body Methods and their Implications for Multiprocessors.* PhD thesis, Stanford University, 1993.
28. Douglas R. Smith. KIDS: A semi-automatic program development system. *IEEE TSE*, 1989.
29. W. M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.
30. Feng Zhao and S. Lennart Johnsson. The fast multipole method on the connection machine. *SIAM J. Sci. Stat. Comput.*, 12:1420–1437, 1991.

(L. Nyland, J. Prins and J. Riely) Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, USA
E-mail address: {nyland, prins, riely}@cs.unc.edu

(P. Mills and J. Reif) Department of Computer Science, Duke University, Box 90129, Durham, NC 27708-0129, USA
E-mail address: {phm, reif}@cs.duke.edu

(A. Goldberg) Kestrel Institute, 3260 Hill View Avenue, Palo Alto, CA 94304, USA
E-mail address: goldberg@kestrel.edu

All of our Proteus system papers, current implementations, and demonstrations are available from The Proteus WWW information server `http://www.cs.unc.edu/proteus.html`.