A Design Methodology for Data-Parallel Applications

Lars S. Nyland, Jan F. Prins, *Member*, *IEEE*,
Allen Goldberg, *Member*, *IEEE*, and Peter H. Mills, *Member*, *IEEE*

Abstract—A methodology for the design and development of data parallel applications and components is presented. Data-parallelism is a well-understood form of parallel computation, yet developing simple applications can involve substantial efforts to express the problem in low-level notations. We describe a process of software development for data-parallel applications starting from high-level specifications, generating repeated refinements of designs to match different architectural models and performance constraints, enabling a development activity with cost-benefit analysis. Primary issues are algorithm choice, correctness, and efficiency, followed by data decomposition, load balancing, and message-passing coordination. Development of a data-parallel multitarget tracking application is used as a case study, showing the progression from high to low-level refinements. We conclude by describing tool support for the process.

Index Terms—Software design, high-level programming languages, parallel algorithms, prototyping, software templates, multitarget tracking algorithms.

1 Introduction

Data-parallelism can be generally defined as a computation applied independently to each of a collection of data, permitting a degree of parallelism that can scale with the amount of data. By this definition, many computationally intensive problems can be expressed in a data-parallel fashion, but the definition is far more general than the data-parallel constructs found in typical parallel programming notations. For example, divide and conquer strategies are data parallel under this definition, but implementation of an efficient parallel divide-and-conquer algorithm, such as quicksort (with full parallelism at each level of the division), is quite challenging using the parallel programming languages shown in Fig. 1. Thus, the development of a data-parallel application can involve substantial effort to recast the problem to meet the limitations of the programming notation and target architecture.

From a methodological point of view, the problems faced developing data-parallel applications are:

- **Target Architecture**. Different target parallel architectures may require substantially different algorithms to achieve good performance, hence the target architecture has an early and pervasive effect on application development.
- **Multiplicity of Target Architectures.** For the same reasons just cited, the frequent requirement that an application must operate on a variety of *different* architectures (parallel or sequential) substantially complicates the development.
- **Changes in Problem Specification or Target Architecture(s)**. Changes in problem specification and/or target environment must be accommodated in a systematic fashion because of the large impact that either causes for parallel applications.

By far the most important reason for developing data-parallel applications is the potential for scalable performance. Even in the face of the longer development times, a performance improvement of one or two orders of magnitude on current parallel machines may well be worth the effort. For example, storm-cell prediction at NCSA [48], [49] using 256 processors runs 12 times faster than real-time (a six hour prediction takes 30 minutes to compute), making the computation valuable for saving lives, property, time, and money. The same code could be run sequentially, but the results would not be available until after the predicted time had passed, relegating them to academic interest only. Real-time applications that are not possible with sequential computers, such as real-time video processing, may be realized with parallel processors. Modern multimedia processors, or multimedia instruction set extensions to generic processors, typically operate in SIMD fashion and require data parallelism in some form to access their performance potential.

Data-parallel computations may themselves be components that are composed in parallel. For example, a number of different data-parallel computations can be arranged into a pipeline to process successive problem instances concurrently. In this paper, we do not address this type of concurrency, although other techniques, such as object-based concurrency, can be used along with the approach we

- *L. Nyland and J.F. Prins are with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175. E-mail: {nyland, prins}@unc.edu.*
- *A. Goldberg is with the Kestrel Institute, 3260 Hillview Ave., Palo Alto, CA 94304. E-mail: goldberg@kestrel.edu.*
- *P. Mills is with Orielle, LLC, PO Box 99081, Raleigh, NC 27624. E-mail: phmills@orielle.com.*

Manuscript received 22 July 1998; revised 3 Apr. 1999; accepted 5 Dec. 1999. Recommended for acceptance by M. Jazayeri.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 110091.

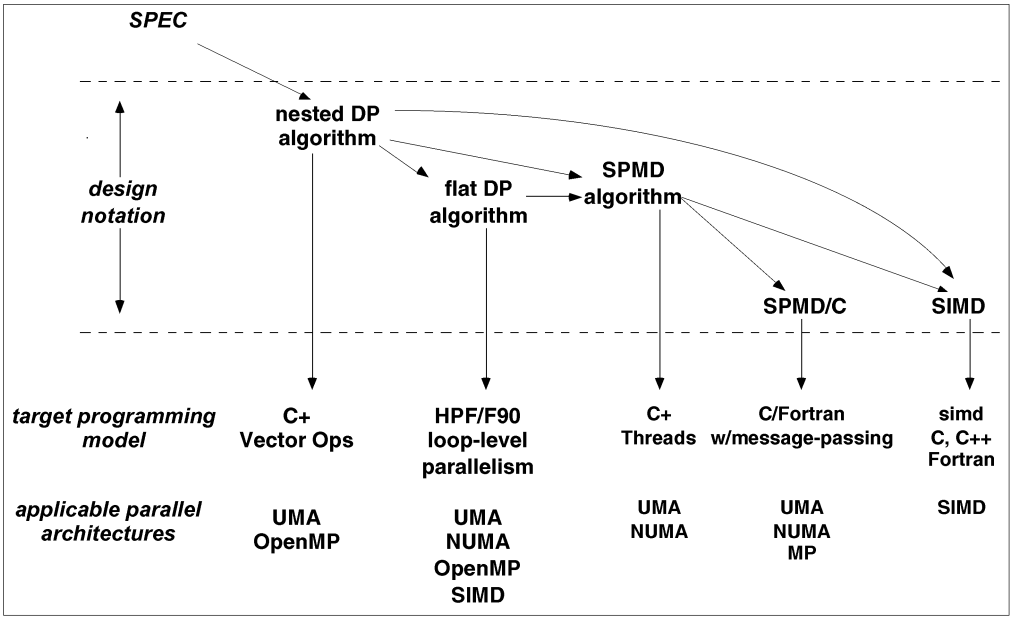


Fig. 1. The process of data-parallel design starts with the selection of a good algorithm initially expressed in a high-level nested data-parallel paradigm. Further architecture-independent exploration continues with perhaps a flat data-parallel model or migration to an SPMD model (multiple processes performing roughly the same operations). More performance can be obtained with migration to architectue-specific development models.

describe here [1], [47]. Other reasons for using concurrency, such as the development of reactive systems or fault-tolerant systems, are also better treated using other techniques [2].

Our design methodology is based on successive refinement yielding a tree-structured collection of data-parallel designs whose level of development effort varies with the parallel architecture(s) targeted and the required level of performance. The methodology explicates activities whose time and expense can be regulated via a cost-benefit analysis. The main features of the methodology are:

- **High-Level Design Capture**, an executable description of the design that can be evaluated for its complexity and scalability.
- **A Data-Parallel Design Notation,** a high-level, architecture-independent, executable notation that incorporates the fully general definition of data-parallelism.
- **Prototyping**. Disciplined experimentation with the design at various levels of abstraction to gain information used to further direct the refinement process.
- **Analysis and Refinement** of the design based on fundamental considerations of complexity, communication, locality, and target architecture.

This paper is a summary of our work in software design methodologies for data-parallel applications. The design methodology has been under development for some time, being first proposed in [3] and further refined in [4], where radar coverage applications were demonstrated. In [5], we briefly sketched the process and outlined the test cases described more fully in this paper. A comprehensive report is available in [6].

1.1 Overview of the Design Methodology

1.1.1 High-Level Expression of Designs

In current practice, parallel architectures typically influence software design early and heavily and, in doing so, overconstrain it. Since parallel architectures continue to evolve, architecture-specific designs have a limited lifetime. Alternatively, architecture-independent environments allow the exploration of families of designs without the constraints imposed by specific architectures. When developed without constraint, they can be gradually refined to classes of architectures (such as shared-memory parallel vector processors or distributed-memory cache-oriented multiprocessors) and, finally, be transformed to architecture-specific instances. As architectures change, or as the relative costs of operations and communication change in architectural implementations, the original higher-level designs and prototypes will still be useful as a basis for developing new versions.

Many computational problems can be naturally expressed using data-parallel computations, but may use sophisticated algorithms that are subject to improvements and refinements. Thus, not only is the target architecture undergoing change, but so is the algorithm. By starting the design activity at a high level, many variations and alternative algorithms can be explored and new algorithms considered. This exploration is not haphazard or aimless, but directed. The exploration answers questions prescribed in the methodology that guide subsequent design and development.

The proposed methodology diminishes the dependence on specific parallel architectures, putting emphasis on high-level design without premature architecture-dependent design decisions. Good algorithm choice and refinement can obtain much better performance than low-level optimization of a poor algorithm.

1.1.2 Data Parallel Design Notations

Central to our methodology is the high-level, architecture-independent expression of designs in a uniform notation. We follow the approach of high-level prototyping languages such as SETL. Languages such as this directly support arbitrary collections of data, in particular nested collections of data such as sets of sets or sequences of sequences. In the presence of such nested collections, we can define data parallelism in a completely general form: the application of an arbitrary function to each element of a collection. This yields *nested data-parallelism*, where, for example, a data-parallel function is applied in parallel to each collection in a sequence of collections. If quicksort is a data parallel function, then application of quicksort, in parallel, to each of the subproblems obtained by partitioning the input around a pivot constitutes nested data parallelism. Nested data parallelism arises naturally in many efficient scientific computing algorithms [8].

Several existing languages or notations support high-level expressiveness and general data parallelism. These include languages such as APL, FP [7], and the Bird-Meertens notation that posit a rich algebra of predefined data-parallel operations. SETL, Nesl [8], Sisal [9], and Proteus [10], and many other programming languages provide a smaller set of data parallel operations, but include a very general data-parallel constructor such as set comprehension. Generally, we favor the approach of the latter category since it more directly expresses specifications. In this paper, we use the Proteus notation (described in Section 4.5). A sequence constructor is the sole source of data parallelism in this notation. It has simple semantics, and can be refined by algebraic methods.

Beyond a certain level of refinement, at the leaves of the tree of designs, we must switch to lower-level notations with restricted collections. For example, HPF collections are arrays. Lower-level languages also allow explication of further refinement choices, for example by making communication and processor scheduling explicit.

1.1.3 Prototyping

Our methodology is based on the spiral model of software development with prototyping, a model favored by the software engineering community [12]. During the risk analysis phase of development, development relies on concise high-level, executable notations for data parallel problems. In this paper, we use the Proteus notation, but others that may be suitable are ISETL, MATLAB, Nesl, and Sisal. The next phase is migration and integration (perhaps automatic) to achieve an efficient application that can be tested and deployed. As new requirements or target architectures are introduced, the development repeats the cycle.

Information discovered in later stages of the design process often leads to revision of previous design choices or even modification of requirements. Prototyping is an effective means of quickly gathering information early in the design process, thus limiting the impact of changes. The design notation for data parallel computation discussed above is succinct and executable, making prototyping an attractive methodology.

1.1.4 Design Analysis and Refinement

Explicit engineering design principles guide designers to develop useful, resilient systems. The proposed methodology insures that designers consider fundamental problem characteristics from the outset, by explicating design levels and the design issues associated with each level:

- **Problem Definition and Validation.** Problem definition and validation insure that the problem solved is the problem that needs to be solved and that the initial description is complete and consistent. This is best achieved by prototyping a simple solution and running experiments against the prototype.
- **Algorithm Selection and Parallel Complexity Analysis.** Alternative algorithmic solutions are considered and their work and step (parallel) complexities are analyzed.
- **Performance and Hardware Constraints.** Environmental factors and performance are considered. This includes problem size, timing and performance requirements, and, if relevant, a prescribed target architecture. The purpose of this analysis is to obtain confidence that the performance goals can be met within the specified constraints based on the algorithmic analysis performed in the previous step.
- **Analysis for Parallel Architectures.** Computation/communication ratios are calculated, data locality, sensitivity of the algorithm to input values, and other factors that determine the best architectural choice are studied.
- **Design Refinement.** Based on the problem and environmental characteristics determined in the preceding stages, designs are refined to reflect data and/or process distribution, and perhaps communication architecture. Refinement may be performed manually or as technology matures with increasing automated assistance.
- **Instantiate on a Machine Architecture.** In some (increasingly unusual) cases, a "design" might require constructing a low-level implementation of a critical component whose performance is dependent on low-level architectural detail. For example, a design in which the data flow-based communication is mapped closely to the communication topology so that all communication is to an adjacent processor and tightly synchronized might be prototyped to this level.

1.2 Taxonomy of Parallel Architectures

Four parallel architecture classes are defined—two shared memory classes and two distributed memory classes. The target architecture will have an influence on many aspects of the design, thus it is important to categorize for later consideration.

1.2.1 Shared Memory Parallel Architectures

Shared-memory multicomputers present a global address space. On a machine with uniform memory access (UMA), all references take approximately the same amount of time to read or write, regardless of address or location, provided they are issued "in bulk," e.g., as vector operations or

through multithreading. The Cray Parallel Vector Processors, the NEC SX-4, and the Tera computer are examples of UMA computers. The additional hardware required to create this capability is one of the larger costs in in a UMA computer.

It is far less expensive to build a shared-memory parallel computer where the memory access time varies according to the physical location of the memory. This defines the concept of nonuniform memory access (NUMA), where there can be multiple levels of memory (cache, local memory, cluster memory, noncluster memory) accessible to the program. Examples of such computers are the SGI Origin 2000, Sun UE10000, and the HP V-Class servers. The nonuniform nature of memory access complicates program development.

1.2.2 Distributed Memory Parallel Architectures

Distributed memory architectures are explicit about local and remote memory. Local memory can be accessed in a normal manner, while remote memory values must be exchanged with messages. There are large-grain and small-grain distributed memory machines.

Large-grain message-passing (MP) MIMD computers are often built by connecting conventional processors with a fast (or existing) communications network. The IBM SP-2 is an example, using IBM RS/6000 processors with a network to support data transfers between any pair of processors. The communications network topology is often abstracted away, avoiding such issues of mapping processes to specific processors (for minimal delay). These machines typically have high overall computational power since they are using large numbers of the fastest processors available. However, it can be difficult to utilize to their full potential since data transfers must not only be arranged in lieu of computation, but sending messages (including synchronization messages) tends to have high overhead and latency. Thus, good performance requires large transfers (to minimize the latency and overhead), prefetching, and/or minimal communication and synchronization. The general nature of MP machines allows the use of a standard message-passing interface, such as those provided by MPI and PVM [12], [13].

Small-grain message passing refers to the closer integration of the communications network with the processors to provide low-latency message passing by absorbing protocols into hardware. Typically, this permits one-sided communication that allows one processor to efficiently read or write remote memory without involving the remote processor. The Cray T3E is a commercial MIMD system with this capability.

1.2.3 Architectural Conclusions

UMA shared-memory architectures offer the simplest development model, along with high performance, but typically at great hardware expense. NUMA shared-memory architectures are less costly in exchange for a slightly more complicated programming model and reduced performance. MP-MIMD architectures offer the highest performance with a widely applicable model of programming, enough to develop architecture-independent programs, but often have difficulty achieving their peak performance and use a more complicated programming model. This model spans a wide variety of architectures, so MP applications can be run easily on many machines. SIMD architectures (and the more recent multimedia processors), while presenting the most specific programming model, can achieve excellent performance on an important class of problems and, thus, have their place in parallel computing.

Machines from all four architectural classes have their merits and drawbacks with respect to ease of programming, performance, and expense; no single architecture dominates the others. The proposed development methodology is sensitive to and driven by these different architectural classes.

1.3 Demonstration Problem

To demonstrate the methodology described here, we explore the development of multitarget tracking (MTT) algorithms. The problem is described, a set of algorithms is examined, and a list of questions is formulated about the differing implementations. With questions in hand, high-level development begins determining the answers to the questions. As answers are found, particularly promising implementations are transformed from the high-level model to more architecture-specific implementations. Most of the difficult work has been done at the high level, leaving only the machine-specific details to be negotiated at the low level. The result of this exploration is not only a family of MTT implementations, but also a demonstration of architecture-independent parallel software development. Additional results are characterizations of the different algorithms explored; for instance, memory use, processor mapping, and execution time can be predicted, given some characteristics of the input data and the underlying computer architecture. The information and the implementations developed demonstrate that the methodology to achieve long-term parallel solutions is worth the time and effort required for the exploration that takes place.

1.4 Tools and System Support

The proposed methodology can benefit from additional automated support and tools that would aid the process are proposed in Section 5. Some tools, such as debuggers, and performance analyzers are useful to any methodology, while others, such as refinement and high-level analysis tools, are specific to the methodology proposed.

2 Design Methodology

The methodology of software development that is being proposed is one that involves prototyping at several different design levels to discover the characteristics of different solutions to a problem. Efficiency of implementation is important (good coding and optimization), but much more important is the efficiency of the solution (good algorithm, good parallelization) and, so, it is important to first find an efficient solution. The efficiency of some algorithms can be judged on paper, prior to implementation, but the performance of most complex algorithms is difficult to analyze and must be done by measuring characteristics of implementations when run on a variety of "typical" data sets. As a simple example, consider the

unexpected conclusion in the implementation of the *joint probability data association* (JPDA) algorithm [14], [15] that an efficient mapping to a hypercube is not the primary performance driver due to serialization of message transmissions. This result was not evident to the authors from the description of the algorithm; the question had to be asked, the experiment performed, and measurements taken before the conclusion could be drawn. Of primary importance in the proposed design methodology, then, is the explication of a set of design issues to be addressed, and data to be gathered through prototyping exercises.

Fig. 1 describes the proposed refinement paths into the parallel architectures described earlier. While the emphasis is on high-level, architecture-independent concerns, the issues are addressed in stages and some design issues may only arise in certain contexts (e.g., data locality is not an issue in a uniform shared memory model).

Next, each stage is described in more detail and questions to be considered at each point within the methodology are enumerated.

2.1 Problem Definition and Validation

As a first step, a description of the problem and validation of the problem against functional requirements is sought. The contextual assumption, based on the fact that the focus is on data-parallel problems, is that the problem is a computationally intensive, algorithmically rich problem, but one of limited scope. The resulting component will be perhaps thousands of lines of code, certainly not millions. Thus, the driving problem is not to organize a complex system with subtle interactions, but to focus on obtaining high-performance for a succinctly described problem.

"*Does the description of the algorithm to be implemented provide enough information to produce the expected results?*" While this may seem like an obvious question to ask, many published algorithms have subtle flaws or depend on knowledge not currently at hand. The MTT algorithms depend on Kalman filters for predicting the locations of targets, but knowledge of Kalman filters may not be common and is rather complex itself. There is often this sort of background knowledge required in the development of complex algorithms. In addition, it is not uncommon to find errors in the descriptions of algorithms. And, finally, there are subtleties of the algorithm that may not have been fully examined by the authors, thus, despite the best hopes, there tend to be failures when the algorithm is stressed. Deciding how to repair an algorithmic error is often quite difficult. Again, the best mechanism for answering this question is to prototype.

2.2 Algorithm Selection and Parallel Complexity Analysis

Solutions to complex problems are often expressed as highly algorithmic, mathematically sophisticated descriptions. This makes the high-level analysis proposed here both necessary and feasible; necessary because these algorithms are often very computationally intensive and, hence, subject to parallelization and optimization; feasible because they are often succinctly specified at a high level. This enables asymptotic analysis, prototyping, and exploration of algorithm variants. Indeed, the mathematical richness of the problems allows many alternative algorithmic solutions with widely differing complexity, accuracy, communication, and synchronization requirements to be explored.

For some problems, the solution emanates from a single, clear-cut, best algorithm, for others, perhaps the choice is not so clear or perhaps the algorithm has been preselected. Regardless of how the choice is made, this stage simply boils down to asking the following question about the algorithm:

"*Consider the Work & Parallel Step Complexities: What are the best, worst, and average cases?*" For parallel processing, it is of extreme importance to know what the best possible parallelism is that can be achieved. The *work* reflects the amount of computation done, while the *step* complexity is a measure of how many steps the program will take on a parallel processor with "enough" processors. Note that this measure uses a PRAM model of computation in which communication is free. While this is not realistic and ignores important details, the approach is to separate concerns and formulate tractable, staged analysis methodology. At a later stage, other issues come to the forefront, but at this point, it may be discovered that there is insufficient opportunity for a parallel solution or even that a sequential one may be adequate (taking into account performance requirements as described below). Larger differences between the work and step-complexities give better justification for using parallel computers.

Of great benefit is the ability to determine the work and step complexities of a parallel algorithm using *language-based* models, as described in [9]. In high-level languages, the work complexity of an operation is typically the sum of the work-complexities of its subexpressions, while the step complexity is the maximum of the step complexities of its subexpressions. For example, the element-wise addition of two sequences has work complexity based on the length of the sequences, while the step complexity is simply $O(1)$. The step complexity is increased by sequential steps, such as recursion, let-expression evaluation (including parameter evaluation), and conditional expressions. If all the operations in a language have step and work-complexity measurements, then programs written in the language can have measurable step and work-complexities.

A *work-efficient* parallel algorithm is one whose total work matches that of the best sequential algorithm. In this stage, work-efficiency of the parallel algorithm is the primary means of selecting algorithms to advance into the next stages. The emphasis is on work efficiency rather than parallel step complexity ("fast algorithms"), as the primary selection criterion reflects an increasing understanding in the theoretical community that very fast algorithms with suboptimal work complexity have poor scaling behavior when the number of processors is limited (which ultimately is true of any setting).

While work-efficient algorithms are generally the best choice, other constraints may lead to a choice of a suboptimal algorithm. Some work-efficient algorithms are very complex and difficult to code correctly. Others are theoretically shown to be work-efficient, but have high constant factors that demand large problems to

demonstrate their performance improvements. In light of this, it is often better to choose an algorithm with slightly inferior complexity that aids in development and/or performance for the size of problem being solved. An example is the parallelized bitonic sort which has $O(n \log^2 n)$ work (worse than an $O(n \log n)$ sequential sort) and a step complexity of $O(\log^2 n)$, but is difficult to outperform for small problem sizes.

Understanding complexity measures is an important aspect in choosing an algorithm. Real-time applications are analyzed with worst-case analysis to meet hard deadlines. Other applications may be analyzed with average-case analysis. To understand the average case performance, it may be easier to measure an executing prototype, rather than formulate a probability distribution of input cases for a more formal analysis.

The coarse data from experimentation with a naive algorithm gives insight into whether the more sophisticated algorithms need to be employed and what impact parallelism will have. The most important single step in developing a data parallel application is choosing an algorithm. No amount of low-level optimization can overcome a poor algorithmic choice. Also, low-level optimization produces brittle, hard to maintain code. It is of the utmost importance to choose a parallel algorithm that does no more work than the best serial algorithm. It may be necessary to refine and analyze alternative algorithms until their behavior on an architecture class and given problem size becomes clear.

2.3 Performance and Hardware Constraints

The analysis in the previous step does not consider anything but the work and step complexities of the algorithms. As we move forward, other factors are considered: the performance requirements, special input/output characteristics, and mandated architecture choices. The questions to be asked are:

"*Based on the analysis performed above, does it appear likely that the performance constraints can be met?*" This can be a difficult question to answer because not all of the factors contributing to the performance have been determined. However, the work complexity or step complexity may be sufficient on its own to identify problems in meeting performance criteria. That is, based on the most optimistic assumptions on the time required to perform primitive operations, the required work may overwhelm the computational resources.

"*Is the work data-dependent, or is the algorithm oblivious?*" When an algorithm's worst case, best case, and average case performance differ, running the prototype on actual data is often the best way to get insight into projected performance. Without prototypes to evaluate the performance, average case analysis requires probability distribution functions, which may be difficult to formulate.

2.4 Analysis For Parallel Architectures

The goal of the next stage is to analyze the algorithm to determine communication requirements and other problem characteristics needed to select an appropriate refinement path among those described in Fig. 1. This analysis is organized by posing questions about the design that relate to its communication requirements and the regularity of the computation. In particular, the goal is to establish the ratio of nonlocal communication to computation and insure that the parallel architecture can match the ratio. Thus, the fundamental question is:

"*What are the communications requirements compared to the computational requirements?*" To perform computations, each processor must have the data it requires locally. If the data is not local, it must be retrieved from the remote location prior to computation. It is possible to explore the ratio of local-operations compared with remote-words-delivered for both an algorithm and a parallel computer. For example, in a matrix calculation, on what values does the (i, j) entry depend? The next step is to partition the data and computation to minimize nonlocal communication and still exploit the available parallelism.

Parallel hardware can be compared in the same manner, by considering the ratio of operations-per-second (floating-point or integer, as appropriate) to remote-words-delivered-per-second. If the ratio is 1, the machine is a uniform shared-memory computer since remote references can be delivered at processor speed. For other architectures, the ratio is not this good (indicated by a larger value), but the closer the communications speed is to local processor speed the better. If the ratio is large, then there must be substantial computation per datum communicated, otherwise, the program execution will spend most of the time communicating and will not scale. The fundamental goal is to achieve a ratio of local computation to nonlocal communication that insures a compute-bound, not communication-bound, computation in order to maximize processor usage. Thus, this ratio is an essential factor in matching an algorithm with an architecture.

The communication pattern must also be analyzed for scalability: *As the problem size grows, does the amount of remote data required increase? If so, what is the growth rate? (Logarithmic, linear, or worse?)*. That is, this ratio must be considered both for the problem sizes of interest and, more abstractly, for varying problem sizes.

For example, an all-pairs algorithm (simple N-body codes) performs $O(n^2)$ computation while only needing $O(n)$ remote data. For each remote value delivered to a process, there is $O(n)$ work, so the ratio of computation to communication is $O(n)$. On the other hand, if an algorithm requires about the same number of remote data words as it performs operations on, the ratio is closer to $O(1)$, a cause for concern in parallelization. If the communication increases faster than computation with increasing problem size, then, for large problem sizes, communication will dominate and the algorithm will not scale.

Measuring locality can be done analytically in some cases, but is too difficult to analyze in others. In particular, for irregular computations (see below), determination of locality is difficult since the values needed by an irregular computation cannot be statically determined. The data dependencies among these results may have to be characterized by a schema or pattern. A suitable formal notation for representing such schema has not been found and substantial tool support is not foreseen to support this analysis.

By examining the computations required compared to remote fetches in an algorithm and comparing that to the ratio of computation speed against communication speed of a parallel architecture, one can gain insight whether or not to pursue a certain line of development. For example, this sort of analysis for a molecular dynamics simulation with a fixed cutoff radius suggests that, for typical problem sizes (20,000 atoms) with a modest number of processors (8-16), about 200 floating-point operations are performed for every remote value fetched, making most current parallel architectures attractive for this kind of simulation. For smaller problems or larger parallelism, higher-performance interconnects may be needed. For particular architectures, locality may be increased by caching, batching communication together, and processor virtualization.

"*Is the work regular or irregular?*" Regular data is much easier to decompose for parallel execution than irregular data; there is often some regular decomposition scheme that has good locality and low communications costs. Regular problems are those that assign consistently sized subproblems to each processor, such as "a row of a matrix," or "a cubic region of space," where all the rows or regions require similarly complex calculations. It is not too surprising that irregular problems far outnumber regular problems since they often arise from data-dependent optimizations for work-efficient solutions. Simple examples are quicksort, whose subproblems depend on the value of the pivot, or variable-density triangularizations that depend on the domain being simulated. Indeed, asymptotic improvements are often the result of dynamic algorithms sacrificing the simplicity of a static processor and communication mapping to overcome the disadvantage of performing too much work. Irregular problems can often be solved with a recursive structure in which subproblems are decomposed until they are small enough to be solved directly. This decomposition can be captured as a nested aggregate data structure or as dynamic process creation (see [16] for a full-scale example of modeling and implementing an irregular problem).

2.5 Design Refinement

This section describes the refinement of programs shown in Fig. 1. The design notation is a high-level language where certain modifications are made to reflect the different programming models outlined.

2.5.1 Data Parallel Design Notation to Vector Model

A nested data-parallel design can be transformed to a vector model with automatic refinement, where the tools rewrite the program to use inter- and intrafunction parallelism. The vector model, such as that presented by the C Vector Library (CVL) [17], consists of C augmented with a vector data type, possibly segmented, whose elements must be scalar data types (integers, reals, booleans). CVL can execute on many parallel machines, but runs best on UMA because transformation to the vector model makes no provision for the nonuniform delays accessing memory. Thus, this is an appropriate choice when all accesses are local or when the target architecture has uniform memory access times.

A key aspect of the refinement requires flattening of nested vectors to vectors over scalar data types. The required transformations are essentially distributive laws that can be applied automatically and exhaustively. For details of a transformational approach see [18], [19], [20].

2.5.2 Nested Data-Parallel Design Notation to Flat Data-Parallel Notation

Flat data-parallel programming languages allow no nesting of aggregates, providing flat, rectangular arrays instead. As a result, iterator variables are bound to scalar values rather than sequences and no nested sequence operations are permitted. Flat data parallel programs are suitably expressed using languages such as HPF, Fortran90 with OpenMP [22], and C. Concurrency in flat data-parallel languages is limited to the concurrent operations on sequences of scalars, where nested data-parallelism allows concurrency on sequences of sequence values.

Many compilers for flat data-parallel languages partition computation based on memory decomposition using an "owner-computes" rule (the processor storing an array element is responsible for computing its value). For instance, in an array model such as Fortran, there is a certain confidence that two similarly dimensioned arrays indexed at the same location will be placed in the same memory (the statement `a(i) = b(i) + c(i)` has all local references).

Typically, there are many more values to compute than there are processors, so several methods of combining partitions are provided. HPF incorporates data decomposition declarations that provide memory decomposition strategies to the compiler, using *cyclic* or *block* distributions. It also allows description of how multiple arrays are overlaid on the processor space so that differing index sets for different arrays will still lead to local references (matrix-multiply is a common example showing both row and column decompositions). Other languages support compiler directives or explicit code for assigning data to processors.

Thus, to refine into this model, general nested vectors must be transformed to array-like data aggregates. Arrays are less general than nested parallel aggregates in the following respects: They are flat, that is, arrays must be indexed fully (all dimensions indexed) for the expression to make sense, where partial indexing of a nested sequence yields another sequence. They have uniform dimension and are rectangularly shaped, e.g., each dimension has a constant range. In addition to simply obtaining the required syntactic form, many optimizations and reformulations, including iterator inversion and index set transformation, can be introduced at this stage. These transformations can alter the final data partitioning.

2.5.3 Nested Data-Parallel to SPMD

From either a nested or flat data-parallel program, the next refinement is the creation of a Single-Program, Multiple-Data (SPMD) program, where a single program is executed by many processors. The benefit of this kind of program is that the work and memory decomposition is under program control, varying from techniques such as simple

memory-decomposition models to complex, perhaps even adaptive, load-balancing techniques. SPMD programming allows the most flexibility in terms of controlling the decomposition of work since ownership of computation can be dynamically adjusted. At this level, finer-grained complexity models, such as the LogP [22], BSP [23], and PMH [24] models can be used to direct the refinement process. As Fig. 1 illustrates, SPMD designs can be implemented on platforms for which the computation/communication ratio varies considerably.

2.5.4 SPMD to SPMD/C

This model includes explicit communications in a SPMD program. After development on shared-memory parallel computers (especially NUMA), the developer has deep knowledge about which memory references are local and which are not. This step makes the sharing of data explicit with the introduction of messages.

The conversion to messages also gives the developer an opportunity to replicate heavily used data, such as tables, that is required for all processes. By replicating data, memory space is traded for communication delays, which is the key consideration when deciding what to replicate. It also allows prefetching and batching of messages, which is important when communication latency is high.

Some compilers for distributed-memory architectures present a global shared-memory model to the programmer, and implement schemes that mimic cache memory [25]. Multiple read-only instances of data migrate on demand and, if the single, write-only copy is changed, then all the read-only copies are invalidated. These schemes often reduce the amount of work required by the programmer, but he/she must be aware of what is occurring.

2.5.5 Moving to a SIMD Model

The nested data-parallel version of an algorithm (or the vector-based code derived from it) is often the most suitable starting point for SIMD code, rather than the SPMD instances. The problematic part of the transition is in nonlocal memory references. Different indexing of vectors generates a permute operation to align data prior to further operations. General route instructions exist on most SIMD computers, but, in many cases, the full generality (and slow speed) of the router can be replaced with specific knowledge about the movement of data, allowing use of the faster interconnection network. The refinement of general route commands to specific transfer commands will have the biggest impact on SIMD programs.

The communication operations derived in the conversion from SPMD to SPMD/C may be helpful in deriving an SIMD refinement. For SIMD code, nonlocal data must be communicated, but, on an SIMD machine, the latency and overhead are not nearly as large, allowing efficient communication of smaller messages.

2.6 Managing the Results of the Design Process

During the entire design process, all the questions posed about the problem and any prototypes developed should be kept in the forefront. The questions should be reviewed periodically to ensure that answers to them are forthcoming. As the questions are answered, they should be documented for further activities, such as transfer of knowledge to software engineers, justification to upper-level management, or simply to state what can or can't be done with regard to high-performance implementations of particular algorithms.

3 Existing Parallel Software Development Tools

In considering the impact of the proposed design methodology, it is worthwhile to explore not only the proposed methodology, but also to consider what is currently available for developing high-performance software. Both the design phase and the implementation phase of software development are examined.

3.1 Design Tool Support for Parallel Software

During the design stages of software, many tools can be used to assist the process. Higher-level graphical CASE tools allow the user to "draw" a program or system, defining many characteristics of a system, but also very common is a set of specifications accompanied by programming.

3.1.1 Software Engineering Tools for Parallel Applications

In the sequential programming realm, many tools exist to aid in the design of software. These tools are referred to as CASE tools (computer-aided software engineering) and are often graphical in nature. The number of useful CASE tools for parallel program development is small but growing, and each tool tends to target a very specific development paradigm. For example, the Code and Hence tools [26] aid in the development of coarse-grain MIMD programs that are not only displayed, but edited and debugged graphically.

3.1.2 Machine-Independent Parallel Programming Languages

The next level of support is that of programming languages and their supporting tools. The justification for this focus is that programming languages present a model of computation. An excellent overview of parallel program development in several different languages is [27].

While many parallel programming languages allow the specification and execution of parallel applications, the level of support from tools to achieve good performance varies widely. Some of the languages mentioned are indeed the lowest level (generally available) for programming the underlying hardware, have little room for optimization, and rely on the parallel support for enhanced performance (e.g., C with MPI). Some languages use higher-level constructs, such as arrays and array operations, allowing stronger support for concurrency. And, several of the languages mentioned (Proteus, Nesl, Sisal, HPF, Fortran90, OpenMP directives) provide substantial optimization support to obtain a high level of concurrency from the developer's source code.

3.1.3 Libraries and Algorithm Templates

Highly optimized and extensively distributed mathematical and communication libraries have aided parallel software development for many years despite the poor notation and lack of automatic support. The most widely used include the ScaLAPACK, BLAS, LAPACK, LINPACK, NAG, and IMSL mathematical libraries and the PVM and MPI communications libraries [12], [13], [28]. The mathematical libraries allow users to program in a familiar setting, calling vector and matrix routines that have been carefully written and scrutinized for the fastest possible execution on a variety of architectures.

The communications libraries enable a general model of multiprocessor, message-passing execution. They support distributed computation by introducing libraries that, for instance, allow differing styles of message communication (broadcast, point-to-point) and support control over process groups. On machines with relatively slow communications, only very large-grain computations will perform well using these libraries. However, highly tuned versions of these libraries exist on high-performance computers, enabling good concurrency for medium or even fine-grain applications. The Cray T3E and SGI Origin2000 are both examples that have fast, hardware-dependent implementations of PVM and MPI.

A more progressive method of support suggested for parallel software development is that of using predefined archetypes or templates [29], [30], [31]. This method has many benefits; among them are highly optimized, architecture-specific implementations and software reuse. The use of templates has been suggested for sequential software without much success; perhaps the additional complexity of code and difficulty in achieving high-performance in concurrent systems will aid in their adoption.

Templates work well for solving problems that have a well-known, common structure, especially if the common structure is one that is difficult to parallelize. Templates may not be useful for exploring new algorithms (since an applicable template may not exist), but it may be worthwhile to explore using templates, if possible, due to the level of effort, high quality, and wide-spread architecture applicability of templates. Templates also have the potential to increase the number of people developing efficient high-performance applications since the people who understand the details of concurrent algorithms can distribute their knowledge in a form that is directly useful by many others.

Besides the fine-tuning available with libraries and templates, another bonus is that of software reuse. If a general construction method for templates is forthcoming, then there's bound to be a boon in high-quality reusable software. However, the current number of templates available is small and the number of template specification systems is even smaller. For example, the *eText* project at CalTech has reports on linear algebra, spectral methods, and mesh-spectral archetypes, but the templates are supported by writing and linking Fortran-M, rather than a system that is expressly built for templates. In time, however, we feel that templates will be a rather simple method of creating efficient, high-performance codes.

3.2 Postdevelopment Support of Parallel Software: Analysis Tools

As software is developed, there is a need for tools to help analyze performance. Debuggers aid developers in finding incorrect actions in programs, while performance analysis tools point out execution bottlenecks. The support for postdevelopment tools in parallel programming is nearly up to par with the tools for sequential programming. Debuggers have been ported and extended appropriately to aid in building correct programs and performance analysis support comes from many sources, compiler messages, tools that show states vs. time, and message logging all provide useful information to analyze the performance of an application.

3.2.1 Debuggers

Debuggers allow a developer to monitor a program as it executes, stopping it along the way, allowing the inspection of data and flow-of-control. Debuggers for sequential machines commonly allow the developer to make queries about the program in terms of the original source code (rather than the compiled machine code that is being run). On some parallel machines, the serial debuggers have been ported, with the few additional commands that are needed to support the underlying architecture. For instance, the dbx debugger under IRIX has added support for multiple threads and thread-private memory. An example of a sophisticated concurrent program debugger is TotalView from Cray. It has powerful graphical interfaces (X-windows) combined with an interpretive command interface to create a useful tool for debugging Fortran, C, and C++ programs. Unfortunately, since many users of supercomputers are remote, graphics interfaces are often not usable, either because of the limited bandwidth or for security reasons (with X-windows). This often forces the developers to resort to text-based debugging techniques, a less than ideal situation since recompiling code is often extremely time-consuming (compilers for new architectures tend to perform poorly).

In debugging parallel software, there is a distinction among compilers for the supercomputers: those that have excellent compiler optimizations for the architecture and those that don't. Vectorizing compilers have extremely good optimization capabilities as opposed to, say, compilers for an MIMD system that make use of the more pedestrian optimizations. Heavy optimization and debugging are often at odds with one another as it is often not possible to debug optimized code that is running incorrectly. Many debuggers do not function with code that is optimized, leaving the developer to perhaps debug apparently correct code (it is a common problem to have optimized code fail and unoptimized code perform with expected results; this is often due to not-quite-correct code running successfully until it is optimized, where the assumption that it is correct is false).

A debugger developed some time ago by the Convex Computer Corp. was a remarkable achievement in this regard [32] and has been ported to other vector platforms. It relates heavily optimized code back to the original source, understanding such optimizations as loop-fusion, loop-unrolling, strength-reduction, dead-code removal, etc. This

allows the developer to inspect the executing code in its optimized form. This helps alleviate code problems that are often blamed on the optimizer, allowing better code to be developed. Hopefully, debugging technology such as this will migrate to more architectures over time.

3.2.2 Performance Monitors

Once parallel software computes the proper result, the focus changes from achieving functionality to that of efficient performance; after all, that is usually the point of using parallel hardware. Analyzing the performance of functioning code is an area where tool support is of extreme importance since it is not possible to improve performance unless the impediments can be found. Another area that is equally important to the performance of parallel codes is the implementation of the primitive parallel operations, such as semaphore or other synchronization primitives. Analysis tools can often point out where the supporting software is impeding high-performance, rather than the user's code.

Accurate timers are the first step in measuring the performance of parallel software. Most systems offer timers that are accurate to the processor clock (counting clock ticks). Additionally, registers may exist that count different kinds of operations executed on the machine, including local and nonlocal memory operations (cache hits) and floating-point operations, giving the performance-conscious developer powerful inspection capabilities.

In conjunction with accurate timers, statistics-gathering capabilities help a developer achieve good performance, but only if the statistics gathering minimally impacts the execution of the program. The kinds of information gathered may include events such as the delay between waiting for a message and its actual arrival, or when each process is ready to synchronize, or the number of processors performing useful work over time. The ability to gather statistics such as these depends on accurate and consistent clock data available to all processors.

Running a program to gather statistics about its execution can generate a large volume of data. Rather than analyze the data in its raw form, data displayed graphically can convey characteristics of the program almost immediately. Visual performance tools are available for analyzing both architecture-specific and architecture-independent software. Cray provides analysis tools for T3E programs; SGI provides the CaseVision Debugger; KSR provided state-transition timelines; included with PVM is *xpvm* to monitor PVM programs; MPI has the *nupshot*, jumpshot, and xmpi program to display logged MPI events; Paragraph [33] is used for debugging PICL message-passing programs; SvPablo displays performance data collected from programs using the Pablo toolkit. All of these tools are useful for making large improvements in the performance of parallel (and distributed) software. An excellent source describing many of these tools is [34].

4 Case Studies: Multitarget Tracking

In this section, an example will be presented that demonstrates the use of the proposed methodology and the reliance on particular tools used for the process. The problem chosen to solve here is that of multitarget tracking (MTT) and, through the development process, it will be shown that there is a need for additional support that is currently nonexistent.

4.1 The Multitarget Tracking Problem

The problem to be solved is conceptually simple: A set of targets is being tracked when new location data arrives (from radar, sonar, or other locating devices). Prediction models compute the expected locations of the tracked targets, but the new data does not necessarily coincide with the estimated positions. The problem then is to find the probability that a target t (for a total of n targets overall) is represented by a measurement j (for each of m measurements). It is a joint probability; that is, it is to be computed in the presence of all other targets and measurements.

While there are several algorithms for solving the multitracking target problem, the results presented here focus on two. The first is the joint probability data association (JPDA) algorithm since it has been the subject of other studies. The second is the tree-search joint probability data association filter (ZB-JPDAF) algorithm; it is an algorithm which, in the worst case, takes much more time than the JPDA, but has claims of better performance in the average (and highly likely) cases, making it interesting for prototyping.

The JPDA is a specific association strategy that uses a weighted average of returns. Targets are not treated independently; if two (or more) targets have nonzero probability of being the same return, then the JPDA calculation of each target is dependent on the other(s). A report from MITRE [14] describes a *column-recursive* implementation that is exponential, but improved by a factor of $n(m+1)$ over the computation of all permanents of a matrix, yet another method of multitarget tracking.

Zhou and Bose have presented several papers on efficient algorithms for data association in multitarget tracking [35], [36] and, in their most recent paper, they present an improved version of their depth-first search joint probabilistic data association filter (which is referred to as ZB-JPDAF). It is a tree exploration based on a recursive formulation, requiring the same a priori probability input as the JPDA. They claim that it typically performs more efficiently than the JPDA and that it is more amenable to parallel execution than their previous algorithms.

4.2 Undocumented Characteristics about Parallelism in MTT Studies

In the descriptions of the JPDA and the ZB-JPDAF, there are claims about the parallelism available in each of the algorithms and hints about what the developers did in terms of parallelization, but architecture-independent discussion about parallelism is lacking. Games et al. describe implementations written in C, Sisal, and C*, with parallel execution on a CM-2. The discussion focuses on the implementations with regard to the machines that executed them. This leaves questions about future implementations and expectations of the JPDA, such as how well it will run on today's newer architectures. Zhou and Bose present their ZB-JPDA algorithm, but do not describe its parallelism. They give indications about how it can be implemented

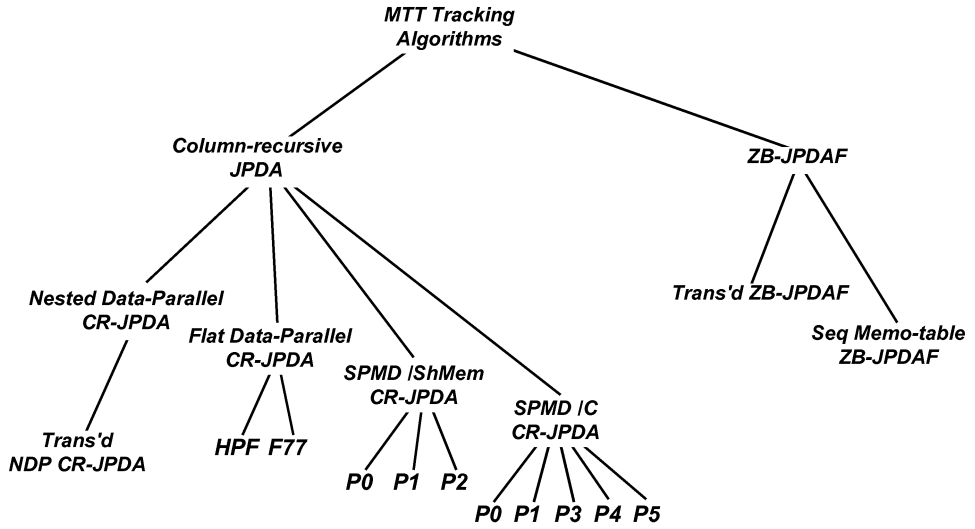


Fig. 2. The development tree of the multitarget tracking (MTT) algorithms.

using Fortran, but, again, there is no analysis of the parallelization outside of a particular architecture. Both of these studies have the groundwork completed for analyzing the parallel performance of their algorithms for multiple architectures, but do not do so; rather, they focus on the specific architectures chosen.

4.3 What Support Exists (Templates, Libraries, Predeveloped Software)?

Support for implementing MTT algorithms exists in many forms. Publications have been the primary source of information leading to development. This includes several journal articles, several technical reports, and several books. No software templates have yet been sought out, yet templates would be of help for a high-performance solution. The template repositories are young, so no real support exists today, but, in the future, this could be a substantial resource.

The reasons that a template would help for this problem are that the column-recursive JPDA is a dynamic programming solution and may fit well into a dynamic programming template. The ZB-JPDAF is a tree-search algorithm, which is a common algorithmic solution that is not trivial to parallelize, thus good templates for parallel tree-search would be very helpful. Templates were also not sought out since the code that has been written is quite short.

4.4 Development Hierarchy to Solve Multitarget Tracking Problems

Fig. 2 shows the development hierarchy of MTT solutions using the proposed methodology. The development of the multitarget tracking algorithms begins at the root with a specification of the problem to be solved. This is written in English and is not executable. Below it are the two algorithms under consideration, the JPDA and the ZB-JPDAF. These two implementations are written in Proteus[1] and serve the purpose of achieving a baseline implementation with no initial concern for parallelism. The first descendant of the JPDA targets nested-data-parallelism, which is suitable for translation to C with vector operations. The initial version of the ZB-JPDAF, written in Proteus, is also suitable for translation to C with vector operations. At this level, all implementations are architecture-independent, although they are tending toward particular architectures.

Additional studies were done according to the development path outlined in Fig. 1. A flat data-parallel version of the *column-recursive JPDA* (CR-JPDA) was developed in Proteus and translated (by hand) into HPF and Fortran77, where memory decomposition and vectorization could be examined. An SPMD version of the CR-JPDA was developed to explore assignment of work to processors and further memory decompositions. Each of these is discussed in the following sections. There was also a paper study performed to estimate the performance of a variety of message-passing implementations of the CR-JPDA, the results of which are briefly summarized.

4.5 The Proteus Programming Language, Briefly Explained

All the prototypes in this study were developed using Proteus, a high-level parallel programming language. A brief introduction is given here.

Proteus is an imperative, procedural language with some additions from functional languages (let-expressions). There are local and global variables, assignment statements, for- and while-loops, function definition, and calling, as well as a full set of operations on sequences. It is block-structured, and uses "{" and "}" to demarcate a block. Conditional constructs (if) can either be statements or expressions (where both the true and false branches must exist).

The biggest difference between Proteus and other languages is the inclusion of nested aggregate data types, namely sequences (sets are included as well). Sequences are first-class data, they can be assigned, passed as parameters, and returned from functions. Sequence members may either

1. Proteus was chosen for its expressiveness of nested sequences and the ability to translate nested sequence notation to vector-based execution. Any language with first-class sequences would have been suitable.

TABLE 1
Proteus Syntax for Sequence Expressions, Let-Expressions, and Infix Function Calls

[1..n]	A sequence from 1 to n, inclusive
a[i]	Indexing. Return the ith element of a.
[i in D : expr] ex: [i in [1..10] : -a[i]]	Build a sequence of values, evaluating the expression for each value of the sequence-expression D.
[i in D | predicate(i) : expr] ex: [i in [1..10] | prime(i) : i]	Filtering an expression based on a predicate. This restricts where the expression is evaluated.
#S	Size of a sequence (number of top-level elements).
let x1 = e1, x2 = e2, ... in expr ex: let d = sqrt(x*x + y*y) in [x/d, y/d]	let-expression. sequentially evaluate (and assign) e1, e2, etc. for use in the final expression.
a 'op' b ex: a 'merge' b	Infix function call for binary functions, identical to op(a, b).
+/S ex: +/[1..5] is 15	Summation (add-reduction) of sequence S. Other binary operators can also be used ('*', min, max, etc.).

be scalars (numbers, Booleans, etc.) or sequences themselves. Table 1 shows some sequence syntax and operations.

The nested data-parallel nature of Proteus permits sophisticated compilation and optimization techniques for parallel execution. Typically, functional programs written with nested sequence expressions, even those crossing function boundaries, can have all their operations run concurrently. As an example, consider a nested merge operation, or nested summations that are used in the MTT algorithms. The syntax for each is:

```
[i in [1..n] : a[i] 'merge' b[i]];
— where a and b are depth-2 sequences

+/[t in members(A) :
  +/[j in members(B) : p[j][t] ] ];
 — to sum all remaining probabilities
```

In the first line, an additional function has been automatically created from the user-defined "merge" function. It has been modified to take sequences of parameters and returns a sequence of results. The transformed code would then be "x_merge(a, b)." Any pure function, whether it applies to scalars or sequences, can be *promoted* in this fashion to concurrently execute on sequences of parameters to compute a sequence of results.

The second line specifies a sum of several nested sums and the transformations rely on a segmented-sum function that can compute all the inner sums concurrently (the outer sum is still computed after the inner sums). The "hoisting" of concurrency is a powerful technique to achieve highly concurrent programs from nested data-parallel programs. For a complete description, see the references [18], [19].

4.6 The JPDA, as Described by Games et al. and Bar-Shalom

4.6.1 Description

The CR-JPDA algorithm has a short mathematical description with many subtleties. The description is shown in Fig. 3 and provides enough information to develop a prototype. The input data is described by shape only, but, as stated earlier, the matrix P with entries p_{tj} represents the a priori probability that target t is associated with measurement j. The output is the joint probability that target t is associated with measurement j.

From the description, it is apparent that the for-loop executes $m+1$ sequential steps to calculate the elements of f. The space required for f is $(m+1)m2^n$, however, analysis reveals that only two rows of f are needed at any time during the calculation (indices l and $l-1$), so the space required for f is reduced to $2m2^n$. The number of operations described is larger than the space required since each value of f depends on several values, on the order of $m(m+1)n2^{n-1}$.

4.6.2 Prototyping the Column-Recursive JPDA

The description of the algorithm is straightforward mathematics, with some ordering information akin to program control (for loops). We created a working, high-level prototype in Proteus,[2] where it was quickly discovered that the specification had errors (there are indexing bugs in the normalization step). The occurrence of bugs in the high-level specification is not uncommon since specifications are typically written without formal verification. After the bug was repaired, we generated several test-cases on which the prototype was run and the results not only convinced us that the joint probabilities were correctly computed, but that the computational complexity was as large as estimated. The prototype is shown in Fig. 4.

The column-recursive JPDA always performs the same amount of work for given values of m and n. As pointed out earlier, the amount of work is $O(nm^2 2^n)$, a complexity that can be calculated from the number of iterations of the for-loop, the range of j within the for-loop, and the sum of the sizes of all subsets of Z_n. The parallel step complexity is estimated at $\mathrm{O}(m)$ (or $\mathrm{O}(mn)$ if reduction and scan

2. Translation into Proteus is a straightforward step. The summations are written with add-reduction operations over sequence expressions that are identical to the expressions in the description. the for-statement is written as a for-loop in Proteus and the assignment to f is written as a nested sequence expression that enumerates the values of l, j, and a.

$$f_{ja}^{0} = \begin{cases} \prod_{i \in a} p_{ij} & j = 0 \text{ and } a \subset Z_n \\ \text{undefined} & 0 < j \le m \end{cases}$$

$$Z_n = \{0, \ldots, n-1\}$$

$$\text{Input: } P = [p_{ij}] \in R^{n \times (m+1)}$$

$$\text{Auxiliary: } F = [f_{ij}] \in R^{(m+1) \times (2^n - 1)}$$

$$\text{Output: } B = [\beta_{ij}] \in R^{n \times (m+1)}$$

$$\text{For } l = 1..m$$

$$f_{ja}^{l} = \begin{cases} f_{ja}^{l-1} + \sum_{i \in a} p_{il} \cdot f_{j,a \setminus i}^{l-1} & j < l \text{ and } a \subset Z_n \\ f_{0a}^{l-1} & j = l \text{ and } a \subset Z_n \\ \text{undefined} & l < j \le m \end{cases}$$

$$\beta_{ij} = \frac{\alpha_{ij}}{\sum_{i \in Z_{m+1}} \alpha_{ij}} \quad i \in Z_n \text{ and } j \in Z_{m+1}$$

$$\text{where } \alpha_{ij} = p_{ij} \cdot f_{j,Z_n \setminus i}^{m}$$

Fig. 3. The JPDA calculations specified in the MITRE report.

```
#include "dpl.pro"

-- set of integers from 0 to n-1
function Z(n) return 2**n-1;

-- take the int i out of the set of ints n
function without(n, i) return n - 2**i;

-- Return the members of the set numbered n.
-- This is the bit indices where there are 1 bits in n.
function members(n:int):[int]
return
     [i in [0..n > 0 ? lg(n): -1] | (n div 2**i) % 2 == 1 : i];

function cr_jpda(p:[[real]]) : [[real]]
{
     var f : [[[real]]];
     var B : [[real]];
     var n, m;

     n = #p;
     m = #p[1] - 1;

     f[0] = [j in [0] : [ a in [0 .. Z(n)] :
                          */[i in members(a) : p[i][j] ] ] ];
     for el in [1 .. m] do
     {
          f[el] = [ j in [0 .. el] : [ a in [0 .. Z(n)] :
                     if j < el
                     then f[el-1][j][a] +
                          +/[i in members(a) :
                               [el] * f[el-1][j][a 'without' i] ]
                     else f[el-1][0][a] ] ];
     }
     B = let
          a = [i in [0 .. n-1] : [ j in [0 .. m] :
                     p[i][j] * f[m][j][Z(n) 'without' i]]]
     in
          [i in [0 .. n-1] : [ j in [0 .. m] :
                     a[i][j] / +/[k in [0 .. n-1] : a[k][j]]]];
     return B;
}
```

Fig. 4. Initial implementation of the column-recursive JPDA as described in Fig. 3.

```
function compute_f(p:[[real]], f1:[[real]], el, m, n) : [[real]]
{
     return
     let
       f = [ j in [0..el] : [ a in [Z(0)..Z(n)] :
                if j < el
                then f1[j][a] +
                       +/[i in members(a) :
                            p[el][i] * f1[j][a 'without' i] ]
                else f1[0][a]
           ] ]
     in
       if el < m
           then compute_f(p, f, el+1, m, n)
           else f;
}

function cr_jpda(p:[[real]]) : [[real]]
{
     return let
                m = #p - 1,
                n = #p[0],
                f0 = [j in [0] :
                      [ a in [0..Z(n)] :
                           */[i in members(a) : p[j][i] ] ] ],
                f = compute_f(p, f0, 1, m, n),
                B = let
                      a = [i in [0..n-1] : [ j in [0..m] :
                           p[i][j] * f[j][Z(n) 'without' i]]],
                      sum = [i in [0..n-1] : +/[k in [0..m] : a[i][k]]]
                    in
                      -- reversed indices for compatibility with Zhou-Bose.
                      [ j in [0..m] : [ i in [0..n-1] : a[i][j] / sum[i]]]
           in B;
}
```

Fig. 5. The cr_jpda routine from Fig. 4 is replaced with a functional recursive implementation that is amenable to nested data-parallel translation.

operations require logarithmic time) since all calculations required at each step of the algorithm can all be computed simultaneously, given enough processors. The difference between the step and work complexity is enormous, indicating that there is substantial work per step. This is the first successful characteristic in determining that it is sensible to run an algorithm in parallel, but there are several more issues to address when considering a parallel implementation (except on an ideal machine, such as the PRAM, that has no communication or memory-contention costs).

The computations in this algorithm are independent of the values in the input data; however, the computations are not quite regular. Consider the summations over each subset of Z_n. The subsets range in size from 0 up to n, so care must be taken to ensure that the summations over all subsets of Z_n are not performed as if they were all of cardinality n. The predictable nature of the computations can be used to distribute the work more evenly.

The locality analysis at this level of program description can be surmised from array indexes. The majority of the work is performed in computing $f[l][j][a]$, where it can be seen that the indexing on the left and righthand sides of the assignment are similar only in the second array index. If all data with the same value of j is local, then the majority of the calculations rely on local values, but with parallelism limited to a maximum of m tasks. Other locality schemes are possible, but locality is a detail for later refinement in a development paradigm where better data-placement control exists.

In the original description of the calculations, there is a description of what to do in the case where the indices of f are not legal. Presumably, these are described to explicitly show what isn't calculated. The prototype ignores this constraint, using nested sequences of differing lengths. Since no indexing operations refer to the "undefined" locations, there is no need to describe their values.

This version of the code runs serially under the Proteus interpreter and is almost amenable to translation by our tools for data-parallel execution. Our model of nested data parallelism [18] (and those of others [8]) does not include iteration, thus the focus of further modification is to rewrite the loop that calculates f, with the goals of making it functional, and storing data only at levels l and $l+1$.

4.6.3 Nested Data-Parallel Implementation of the JPDA

The column-recursive JPDA has iterative constructs. Our automatic translation of nested data parallel programs depends on functional, not iterative, code, so the iteration is rewritten as recursion (Fig. 5 shows the updated function cr_jpda and the added function $compute\_f$). The recursive nature keeps only the two most recent levels of f active;

```
function compute_f(p:[[real]], f1:[[real]], el, m, n) : [[real]]
{
      return
      let
        is_member = member_table(n),
        f = [ j in [0..el] : [ a in [Z(0)..Z(n)] :
                 if j < el
                 then f1[j][a] +
                       +/[i in is_member[i][a] :
                            p[el][i] * f1[j][a 'without' i] ]
                 else f1[0][a]
            ] ]
      in
        if el < m
           then compute_f(p, f, el+1, m, n)
           else f;
}
```

Fig. 6. The recursive "compute_f" function from Fig. 5 with the nested data-parallel features removed (differences highlighted).

other levels can be discarded as analysis reveals their usefulness is over.

Without the looping construct, the entire program can be written as functions defined by a list of single-assignments used for computing a return value. The assignments can be part of a *let*-expression that computes the return value. When the program is written this way, it is amenable to the nested data-parallel transformations and optimizations of the Proteus system. This version performs no less work than the iterative version; it is simply rewritten to use single-assignment and recursion. It does, however, use less memory (factor of $m/2$), which allows programs of larger size to be run. This version is a descendent of the previous version of the refinement tree of MTT algorithms.

This version of the CR-JPDA is suitable input to our transformation tool *ptrans*, based on transformation rules in [37], that generates a C program of vector operations from the Proteus input. Over 2,000 lines of C code are generated from less than 100 lines of Proteus, with calls to the CVL library to perform the calculation in parallel. The important aspects of parallelism in the generated C program are:

- All members of all sets are generated in parallel by a parallelized "members" function (takes a sequence of integers and returns a sequence of sequences of member values). The function is automatically generated in the transformation to C.
- All "without" function calls are evaluated in parallel.
- All summations within a sequence are calculated in parallel.

These automatic benefits, while attainable manually in other languages, promote the rapid development of parallel code at a high-level.

4.6.4 Flat Data-Parallel Implementation of the CR-JPDA Algorithm

To achieve a "flat" data-parallel version, the code was modified to precompute data and store it in a table (modifications shown in Fig. 6). Flat data-parallel programs have the following characteristics: no nested sequence structures except to mimic rectangular multidimensional arrays; iterator variables are only bound to scalar values; and only pure function calls are allowed in sequence expressions (where they can be inlined by the compiler). This model is similar to that allowed by HPF and Fortran90.

The target of reformulating the JPDA to meet this model is again the calculation of f. In the nested prototype, the call to "members" returns a sequence that is bound to an iterator variable and the nested sequence structure must be made rectangular.

If an $n * 2^n$ table indexed by $[i, a]$ indicates whether the integer i is in set a is calculated upon entry to cr_jpda, then the iterator "i in members(a)" can be changed to the conditional iterator "i in [0..n] | is_member[i][a]." This change makes the nested structure regular *and* eliminates the binding of an iterator to a sequence value. The added statement prior to loop entry is "is_member = member_table(n);."

Another point to consider in developing a flat data-parallel implementation is that, even though it may be possible to describe nested calculations that meet the above guidelines, it is often the case that compilers can only parallelize the innermost iteration. This forces the developer to consider the nesting order, putting the longest sequences as the innermost expression.

Given these considerations, versions were quickly developed in HPF and Fortran77. As stated, a function was written that computes the membership table, which is then used as a mask within some nested HPF "forall" loops. This version ran with comparable results to the previous Proteus versions, but the supporting tools were only in the early stages of development, so some parallel operations are not executed in parallel (e.g., the "pack" operation which extracts and compresses values based on a Boolean vector). A Fortran77 version was written to explore the vectorization capabilities of compilers for different architectures, such as the Cray vector machines, where the loops were inverted, since vector-parallelism typically means that only the innermost loop is vectorized. Fortunately, with the changes required for the flat data-parallel version already described, the loops had been made independent of one another, so loop inversion was a trivial change. The resulting code was fully vectorized by the compiler with the commensurate increase in performance.

HPF provides directives for regular decomposition of arrays for the purpose of assigning work to processors with an "owner-computes" rule. The decomposition of f along the longest axis (all subsets of $\{0..n\}$) assigns the memory uniformly to processors, but not the work. The sets with fewer elements have fewer operations to compute their values. This is a limitation of the HPF model: Uniform memory decomposition does not always yield uniform work decomposition.

4.6.5 SPMD Implementation of the CR-JPDA Algorithm

A Single-Program Multiple-Data implementation allows more control over the decomposition of work because a wide variety of assigning computational responsibilities is possible, limited only by the expressiveness of the programming language used. In the CR-JPDA, the sharing of data and strict ordering requirements force the introduction of "barriers" (or other synchronization mechanism) at the end of the for-loop (or recursive call) in each process to ensure that no process gets ahead of the other processes, where it would read uninitialized memory values. Once the proper constraints are introduced to insure correctness, the choice of decompositions within an SPMD program is greater than with HPF.

An SPMD version of the JPDA was developed using C with Posix threads, running on up to 32 processors on an SGI Origin200 (originally developed on a KSR-1 shared memory computer). The Posix threads interface is a standard for shared-memory SPMD programs and, thus, the developed code will run on other shared-memory architectures, such as SGI Challenge, HP V and K-class, and DEC Alpha cluster architectures.

The first attempt was to mimic the HPF version (the hand translation required less than one hour). The program was written in such a way that it was simple to change between block and cyclic layouts of the f array. Block layouts are defined by contiguous regions of size n/p with a delta of 1, while cyclic layouts have a larger loop delta (n/p) and starting points are contiguous. On shared-memory computer systems, block-style layouts tend to perform better since the unit of shared memory is often larger than one word (such as a page, subpage, or cache line). If multiple threads are attempting to update different locations in the same shared block, the memory system must handle update requests from many processors. Block layout leads to a single thread writing into the same shared memory block and generates few concurrent requests from the memory system.

As pointed out previously, the JPDA does not perform the same number of operations per value computed, so regular decompositions of memory will not balance work. Two efforts were made in achieving a better load balance, the first of which was to couple remote memory references with values that required less work (giving more local memory references to values that require more work) and partitioning the array based on estimated work.

The computation of $f[l][j][a]$ depends on several values of $f[l-1][j][x]$, where x is a set of values computed from a where a single one-bit is set to zero. This implies that all values of x are less than a, so, to compute a value in f, only f-values at lower locations are required. The upshot of this when using a block-style distribution scheme is that each thread accesses its own memory and that belonging to lower numbered threads. The lower numbered threads already have less work and, with the increase of local-to-remote memory references, the imbalance is exaggerated. As stated earlier, cyclic decomposition creates false-sharing, reducing performance. Another decomposition was sought to aid this computation.

One solution is to remap the data into small blocks that are laid out cyclically in hopes of redistributing the local/remote memory references. As a first attempt, this was done by rotating the bits of the a index of f. While this succeeded in rearranging the memory references, the performance was reduced by a factor of two because the calculation of the index (several logical operations) is as costly as the single multiply-add operation that follows.

Instead of address recalculation, introducing an additional for-loop can also create a block-cyclic layout. The outer loop iterates over each block, while the inner loop iterates over the data in a block. This style of looping uses no other instructions than add and index to find the required values (as opposed to the previous attempt, or double indirection of a precomputed map, such as a gray-code). The C-style for-loops appear as:

```
/* PE = total number of threads or processors
 * my_pe = thread number;
 * N = total size of array;
 * block_size = N / (PE**2)
 */

for (block = my_pe * block_size; block < N; block +=
       (PE-1)*block_size) {
     for (i = block, high = block+block_size; i < high; i++) {
          f[j][i] = ...;
     }
}
```

The block size used is N/P^2 to give each thread as many blocks as there are threads. This gives each thread enough work to do to make good use of the scalar processor (and pipeline), local cache, and shared memory locality, while also redistributing the memory references more evenly for this algorithm. Some preliminary performance charts are shown in Fig. 7. In our final implementation on 16 processors, the idle time of the least loaded thread was reduced from over 60 percent to less than 20 percent.

4.6.6 Analysis of Implementations of the CR-JPDA Algorithm

Multiple parallel implementations of the CR-JPDA have been developed and, as described earlier, the focus of interest in each changes as progress in the development is made. The original prototype was developed to ensure that a baseline, correct version could be developed. The high-level, collection-oriented language allowed transcription in almost a line-for-line manner, without eliminating any possibility for parallel execution. The prototype also served to introduce the developers with the actual calculations performed, as well as requiring the construction of test data.

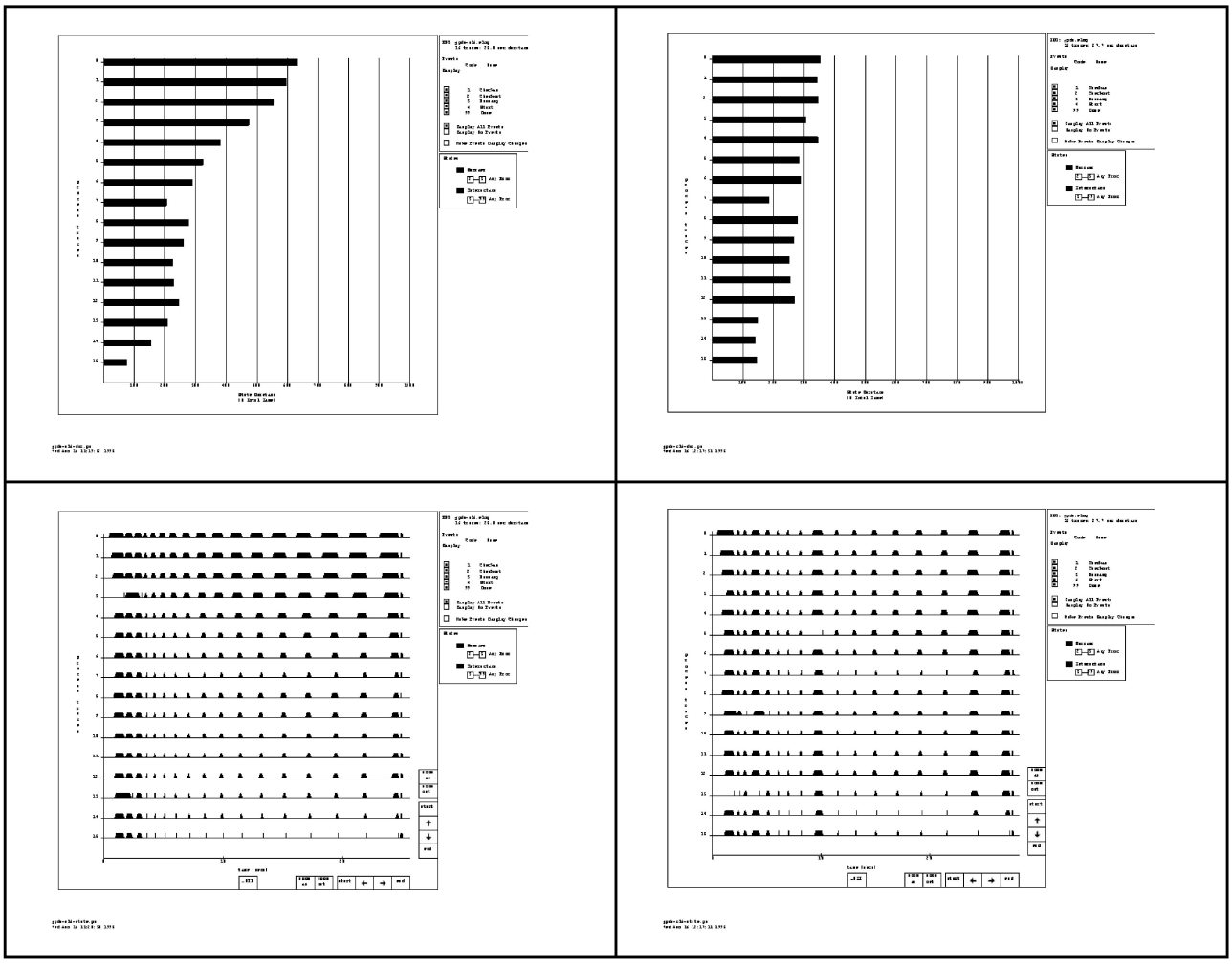


Fig. 7. These screen-shots demonstrate the ability to improve performance by examining state durations in the execution of SPMD JPDA application. The upper graphs show how much time each processor spent waiting in barriers as a percentage of total time, while the bottom graphs display a state-vs-time strip for each processor (the black areas represent waiting periods). The point to note here is the imbalance with the strategy on the left (block decomposition) vs. the more balanced version (block-cyclic decomposition) on the right. This output is from the Gist tool on the now-defunct KSR, but shows how analysis tools can aid in understanding performance.

Because of tool restrictions, it was refined into a nested data-parallel version that could be (and was) transformed into vector-class C code. The goal behind this version was to specify as much parallelism as possible without being burdened with any of the low-level details of architecture-specific parallel programs.

Subsequent development used the Proteus prototype as a model for development, while focusing on the specifics of the architectures on which the code was run. For the flat data-parallel (HPF and Fortran77) versions, an irregular structure was made regular and sequence manipulation was converted to simple integer calculations. The conversions not only made both Fortran versions possible to develop, they also introduced some efficiency improvements that should be reflected in the higher-level version (so that subsequent development based on the prototype will not have to develop the same modifications and miss the efficiency opportunity). It was hoped to experiment with the memory layout directives in HPF, but, with immature tools (in 1996), runs of this version were limited to a single processor. Additional concerns with the Fortran77 version had to do with tool support for that language, namely only the innermost sequence operation is run in parallel, so manual iterator inversion was used to achieve the highest-performance.

The shared-memory SPMD version allowed us to examine work allocation strategies, as the cyclic and block decomposition strategies used in higher levels were not efficient. The introduction of an extra for-loop to cyclically execute small blocks added virtually no overhead while improving locality to create a much more efficient program.

The development of the CR-JPDA has shown a progressively refined set of versions, from a high-level prototype to a variety of lower-level, portable versions where specific issues were explored. Each instance of the program targeted a single issue (correctness, block/cyclic parallelism, load balancing, memory locality, etc.) allowing for quick and focused development. The issues were resolved and the resolutions are migrated back to the high-level description so that further development (a message-passing

implementation, for instance) builds upon the knowledge previously gained.

4.6.7 A Paper Study of the CR-JPDA on a Message-Passing Architecture

A related study exploring parallel implementation of the CR-JPDA was performed separately [38]. In this study, five different parallelization schemes were developed in an effort to attain efficient parallelization on machines with thousands of processors. The underlying assumption of the study explored the main computational component as it might be executed on an IBM SP-2. The study relied on some simple optimization and analysis tools that revealed the peak performance of the code would be 71 percent of the machine's maximum speed. Using this value and published communication latency and throughput values, five decomposition schemes were developed which are designated as the SPMD/C branch of the CR-JPDA development. Please refer to the full report for details.

4.7 ZB-JPDAF, as Described by Zhou and Bose

This section briefly describes an alternate solution to the MTT problem. Rather than explore the details, as was done in the previous section, we only highlight key aspects of this development subtree, referred to as ZB-JPDAF in Fig. 2. Full details can be found in [39].

4.7.1 Description

An alternate algorithm for solving multitarget tracking problems has been developed Zhou and Bose in [36] in which they refer to their algorithm as "improved" over their depth-first-search algorithm [35] for a joint probabilistic data association filter (hereafter referred to as ZB-JPDAF). The algorithm is a tree-exploration described with recursive equations (and appropriate base cases); as such, it is straightforward to implement at a high-level in Proteus. The independence of node expansions in the tree is a key feature that enhances parallelism, as there are relatively few ordering constraints to with which to comply.

The tree exploration terminates upon matching all measurements to targets or encountering a zero-probability scenario (a target has no probability of being any of the remaining measurements). Therefore, the amount of work performed varies with the input data: If there is a high spatial density of measurements, then the work complexity is large and, as the density of measurements goes down, so does the work. The work is $\mathrm{O}(\max(m,n)) \leq \mathrm{T}(m,n) \leq \mathrm{O}(m!n)$. The authors claim that there are typically no more than three probable measurements for each target, thus the average case has a complexity $\mathrm{O}(n3^m)$.

The parallel step complexity is determined by the sizes of the target and measurement sets. These sizes control the depth of the tree since each level of the tree (usually) eliminates one target and one measurement. All the nodes at each level can be computed simultaneously; the only dependency is that all children must finish prior to a parent node finishing (thus, the computation is not complete until the root receives data from all of its children).

The work this algorithm performs is irregular, being based completely upon the input data, and, thus, will have a significant impact on the parallel implementation. Each subproblem solved will be of different size, so allocation of subproblems to processors will result in all solutions running as slowly as solving the slowest problem. The tree exploration may also terminate early along some paths, leading to more irregularity. Unbalanced, irregular computation such as this is exactly the style of computation where conventional parallel programming languages offer little support. Automatic support for developing efficient parallel implementations of this algorithm is manifested in two ways: programming languages that support irregular decompositions and parallel program templates.

Nested data-parallel programming languages (Proteus, NESL) can automatically generate functions that apply to all nodes at a single level of the tree, using the function that applies to a single node (which is all the developer writes) as a template. This automatic extension provides substantial leverage by providing interfunction parallelism, especially in the case of irregular decomposition. In this algorithm, the step complexity is related to the depth of the tree since all node expansions at each level can be executed in one step. There is no doubt that code could be written and parallelized similarly to the output generated by the Proteus (or NESL) translation tools [18], [40], but the resulting code would be large, complex, difficult to understand, and difficult to modify.

Software templates are an alternate solution. Tree-search algorithms have been studied for efficient parallel implementation and the resulting algorithms are complex [41], [42]. Fortunately, there are systems such as ZRAM [43], [44] that implement many parallel tree-search algorithms. ZRAM not only provides a library interface, but, in addition, provides a "skeleton" (or template) where the user specifies some of his own routines to be called by the parallel search engine ("upcalls" or "call-backs"). The ZRAM search uses the node-expansion and evaluation routines in one of its many customizable search strategies (backtrack, branch-and-bound, reverse-search on graphs) to perform the search in parallel. It also provides automatic checkpoint-restart capabilities to enable long-running searches. This is a recently developed system that we look forward to using.

The Proteus version of the ZB-JPDAF was developed in such a way that it could be directly compared with the CR-JPDA implementation and it was found that, on the sparser cases (branching factor of 3), the execution times were nearly identical. On denser input data, the runtime increased for the ZB-JPDAF, but not for the CR-JPDA. However, an implementation based upon the specifications ignores key efficiency short-cuts. The authors note that many of the nodes calculate identical values since all matchings of a subset of targets with a subset of measurements yield an identical subproblem of remaining targets and measurements to match. We modified the code so that the results are tabulated as calculated (using the *memoize* capabilities of Proteus), reducing the worst-case work to $\mathrm{O}(n2^m)$. Even the suggested average case (branching factor of 3) benefits substantially from memoization, enough for the implementation of the ZB-JPDAF to almost always run faster than the CR-JPDA, regardless of density. Unfortunately, our model of nested-data-parallelism does not

include memoization, thus the memoized version was not translated to C+CVL. At least one template solution supports memoization; ZRAM has a parallel implementation of *reverse search* which avoids expansion of nodes previously visited. Experimentally, the ZB-JPDAF is the fastest MTT algorithm that was developed, and should be considered for parallelization.

4.8 Conclusion of MTT Case Studies

Even though the MTT application developed in this paper is relatively small, we feel that development was aided by using the design methodology proposed here. The most obvious gains arise from a separation of concerns in that a thorough understanding of the algorithm is developed during prototyping, while performance concerns are left for subsequent development. Additionally, refinements made in low-level applications can be integrated into the high-level prototypes, providing a clearer understanding of an improved algorithm.

The Proteus implementations showed the process of starting with a high-level functioning prototype that is carefully refined and eventually translated to production programming languages. Each level of refinement focused on more specific problems where solutions were found and, often, the solutions were useful in improving their ancestral codes.

The work also shows that the choice of the proper algorithm (work-efficient) is of utmost importance. In the ZB-JPDAF algorithm, substantial gains were made by introducing memo-functions to avoid recomputation of complex values. In runs comparing the ZB-JPDAF to the CR-JPDA, it ran at about the same speed in dense conditions to over a factor of 100 times faster with sparser (branching factor of 3) data. A parallel solution of the CR-JPDA would have to achieve a factor of 100 speedup just to be equal with the sequential ZB-JPDAF. Additionally, since it falls into a well-studied problem domain, templates exist that support its parallel execution.

5 Tools to Support Parallel Software Development

5.1 Powerful Parallel Programming Prototyping Language

While the brevity, clarity, and analyzability of the Proteus implementations shown in this report make a strong argument for using high-level languages for prototyping, so do other efforts, most notably the Sisal implementation of the JPDA in the MITRE report and the growing availability of MATLAB templates (e.g., [29]). It is our belief that the strength and longevity of the Sisal JPDA implementation were unexpected to the authors in [14] and, even so, their implementation will still perform well on today's fastest computers. Their other implementations have become unusable due to the fading architectures on which they ran.

More powerful optimization models must be developed. Current high-level parallel programming languages rely on functional (side-effect free) descriptions, limiting the developer to exploring only those problems that fit within this domain. There are additional programming techniques that can be formalized and optimized for parallel computation; they simply are not yet developed. Examples are localized state changes for iteration, automatic parallel hash-table management, or high-level task parallel models. If these programming structures were available, recognizable, and translated for parallel execution, then the benefits of prototyping could extend much further.

5.2 Repository Version Manager

The development of a tree of program versions is not a one-way process where each descendent relies upon a perfect ancestor; rather, there is constant flow among all program instances in both directions. As an example, consider what begins as a tree-style development. Version A is developed and, in doing so, several support routines are developed. As version B is developed, it initially relies upon some code from version A, but, later, better code is developed that would have an impact on version A as well. Some changes, must be propagated up and down the tree, while certain parts of the code must remain fixed (perhaps the improvements in one version are only beneficial in certain models).

While consolidating routines in an additional file that both versions share might solve the problem for this particular example, it is a primitive solution that is bound to be inadequate for large applications. Instead, it must be possible to develop new versions on the development tree by taking parts of other versions, or perhaps new parts, and composing them. Currently, no such tool exists, so the development of all of the versions in this report was managed manually, with support from the Unix RCS and CVS tools.

5.3 Program Transformation Tools

Why was MITRE's Sisal version of the JPDA only run serially? Probably due to a lack of compiler support for Sisal programs on parallel architectures to which the developers at MITRE had access. The development groups working on Proteus, NESL, and Sisal are all research groups and lack the manpower to have new versions available as machines are delivered. Furthermore, the translations and optimizations being performed are complex and require significant development tool capabilities. This all takes time and manpower for successful tool development.

The translation and optimization strategies developed by at UNC, the Scandal group at CMU, and the Sisal group at LLNL give the developer powerful parallel programming constructs, unequaled in other parallel programming languages. This is demonstrated by the concise, clear implementations that run competitively with less clear, detail-packed implementations at a lower level.

The usefulness of good transformation tools can be extended even further as more translation and optimization schemes are developed. But, these are often difficult to implement due to current limitations in tool support. As the tool support grows, currently unused optimization schemes will become available to developers, enabling a wider range of high-level parallel applications.

5.4 Multilingual Programs

As prototypes are refined, it is often the case that one small part could be substantially improved if rewritten in a low-level language. Rather than commit the entire prototype to

the lower-level language, only the targeted parts need to be rewritten if multilingual programming capabilities exist. This allows the majority of the program to remain at a high-level, while also giving the developer the chance to use very efficient implementations.

Additionally, prototypes often solve only one small problem, which is exactly the situation here with the MTT algorithm prototypes. Once the prototypes are developed, it would be ideal if they could be integrated in a framework that already exists, such as a larger tracking system. Other examples are prototypes to perform nonbonded electrostatic interactions in MD simulations while avoiding redevelopment of an entire molecular dynamics simulation prototype.

Thus, language interaction is important for enhancing prototypes, as well as prototyping pieces of existing systems. To be widely adopted, a prototyping system must support calling out to other languages, as well as being called by other languages. This capability is available with Sisal and MATLAB, where key components of programs are moved into low-level parallel programming domain while leaving the noncomputationally complex code at a high-level.

5.5 Information Gathering Tools

Of course, a key goal of developing parallel programs is achieving high performance. As such, it is important to know where the program is functioning as expected and where it is not. This includes help from optimizers about when the code can and cannot be successfully optimized and, rather than bombarding the user with all the information about the entire program, the tool would be better if the user could explore only those areas of concern.

Other information that is useful is data gathered about the successful executions of the program. What is the efficiency? How many messages were sent? Where is the program spending all of the time? What is the parallelism profile of the program? All of these questions can be answered with automatic or semiautomatic support and tools that display this type of data, such as Paragraph and XMPI, are indeed popular.

5.6 Tools for Equational Verification and Refinement

One basic theme that is emerging is that, in the arena of data-parallel design, equational optimization (that is, making equations work efficiently according to some simple operational model) is often tantamount to effecting work and step efficient data parallel design. Moreover, the use of nested data-parallelism is invaluable in direct expression and prototyping of recursive equational specifications in aggregate (data-parallel) form.

The above points argue for tools that exploit the extremely constrained form of data parallel programs—that is, that they are functional in nature with the attributes of simple semantics and amenability to algebraic rewriting. Two classes of interactive tools are envisioned:

- Equational verification can assist programmers by verifying the equivalence of manually derived improvements. Tools here can leverage off ongoing work in semantic verification, term rewriting, and functional program transformation.
- Refinement tools to semiautomatically guide equational optimizations.

Indeed, the technique of algebraic rewriting is already explicitly used in the translation of Proteus in which nested data-parallelism is transformed to flat data-parallelism. It is envisioned that related techniques can be further used to guide derivation, but at the refinement level.

5.7 Portable Compilation Targets

As stated earlier, complex translation and optimization schemes take significant time to develop. Having different output models (languages or libraries) lengthens this time. Blelloch has demonstrated, with the C+CVL target, that an architecture-independent compilation target ensures the longevity of the NESL project [17], [40], [45]. The Proteus translator also relies on the CVL library, thus Proteus' longevity currently depends on support from NESL. Other systems, such as Fortran-M, depend on portable message-passing, allowing quick migration to new computer architectures.

6 Conclusions

It is difficult to develop reliable implementations of complex algorithms that run efficiently on parallel computers. Adding to the difficulty is the constant turnover of high-performance computing hardware with differing models of programming. To alleviate this problem, we have proposed and demonstrated a development methodology for data-parallel programs that separates the concerns of algorithm complexity, efficient use of hardware, and the variety of hardware targeted. The methodology relies on prototyping at a high level, followed by translation (either automatic or "by hand") to low-level implementations. The low-level implementations are often necessary to allow more precise control of work and storage allocation.

Prototyping computationally complex algorithms with high-level languages allows rapid development without overconstraining the implementation for parallelism. Languages such as Proteus and MATLAB are ideal for specifying data-parallel applications, as they have excellent support for specifying a single path of control over aggregate data. Unfortunately, programs written in such high-level languages often fail to meet performance criteria. Still, the process of developing a high-level, compact implementation is an extremely useful exercise in that it both builds and conveys knowledge about solutions to the underlying problem. Given that prototypes do not generally have adequate performance, there are two possible approaches to solve this problem. The first is to use tools that automatically translate high-level prototypes to actual parallel code. While this is not currently applicable to all problems, tools do exist for problem subdomains where high-level specifications can be run with high performance. Broader support in compiling a larger domain of solutions to a more comprehensive set of parallel architectures is a topic of current research.

The second option is the hand-translation from the prototype to a particular model of parallel programming

that has high performance. This option is transitional as we await more tools that support the first option. While hand-translation sounds work-intensive, the applications considered here are of moderate size, so the actual work is small; our experience is an afternoon or a full day. This is relatively minor when compared to the time spent in original development of the algorithm and test cases, followed by the subsequent performance tuning, which, in our test cases, amounted to many weeks.

The largest benefits of systematically developing a tree of versions are the wide-applicability and longevity of the application. The benefits arise from the rapid initial development, the discussion that can occur about the initial development, the development of versions for particular parallel programming models, and the continued execution ability despite the aging and decommissioning of parallel hardware. This has been shown in our case, where our original parallel computers were taken out of service and replaced with similar equipment from another vendor. The ability to run the MTT algorithms on the new hardware was carried forward by having a reference model and out-of-date artifacts to build a new application. The downside of beginning development at a low level is shown in the studies cited in this paper, where implementations for particular architectures are no longer relevant. The software that remains has far too many architecture-specific details to be a foundation from which new versions of the application can be developed.

In time, indeed it is beginning to occur already, the need for low-level implementations will be reduced. This is occurring in two ways. The first is the growth of available templates for common algorithms with parallel execution. Linear algebra has had this support for some time and there are tools appearing that support sophisticated tree-search methods on a wide variety of parallel architectures, of which the ZRAM system is an excellent example.

The second is the automatic translation of high-level programs to efficient, machine-specific implementations. The Proteus and NESL translation tools have demonstrated the capability for a narrow range of applications targeting parallel computers with good memory performance. Further work in the area of efficient translation of nested data-parallel programs to message-passing machines is ongoing [46].

ACKNOWLEDGMENTS

This work was supported in part by Rome Laboratories #F30602-94-C-0037 and by ARPA via US Office of Naval Research #N-00014-92-C-0182.

REFERENCES

[1] L.V. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," *Proc. OOPSLA '93,* 1993.

[2] D.C. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz, "Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems," *J. Systems and Software,* vol. 21, pp. 253-265, 1993.

[3] P.H. Mills, L.S. Nyland, J.F. Prins, and J.H. Reif, "Software Issues in High-Performance Computing and a Framework for the Development of HPC Applications," *Developing a Computer Science Agenda for High-Performance Computing,* U. Vishkin, ed., pp. 110-117, ACM Press, 1994.

[4] A. Goldberg, P. Mills, L. Nyland, J. Prins, J. Reif, and J. Riely, "Specification and Development of Parallel Algorithms with the Proteus System," *Specification of Parallel Algorithms,* G. Blelloch, M. Chandy, and S. Jagannathan, eds. Am. Math. Soc., 1994.

[5] L. Nyland, J. Prins, A. Goldberg, P. Mills, J. Reif, and R. Wagner, "A Refinement Methodology for Developing Data-Parallel Applications," *Proc. Europar '96,* 1996.

[6] L. Nyland, J. Prins, P. Mills, and J. Reif, "Design Study of Data-Parallel Multitarget Tracking Algorithms," *Proc. AIP Design Meeting (7/96),* 1996. ftp://ftp.cs.unc.edu/pub/projects/proteus/rome/taskII.pdf.

[7] J. Backus, "Can Programming Be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM,* vol. 21, pp. 613-641, 1978.

[8] G.E. Blelloch, "Programming Parallel Algorithms," *Comm. ACM,* vol. 39, 1996.

[9] D.C. Cann, "SISAL 1.2: A Brief Introduction and Tutorial," technical report, Lawrence Livermore Nat'l Laboratory, 1993.

[10] G. Levin and L. Nyland, "An Introduction to Proteus, Version 0.9," Technical Report TR95-025, Univ. of North Carolina-Chapel Hill, Aug. 1993.

[11] B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Software,* pp. 61-72, 1985.

[12] A. Geguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "A User's Guide to PVM: Parallel Virtual Machine," technical report, Oak Ridge Nat'l Laboratory, July 1991.

[13] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[14] R.A. Games, J.D. Ramsdell, and J.J. Rushanan, "Techniques for Real-Time Parallel Processing: Sensor Processing Case Studies," Technical Report MTR 93B0000186, MITRE Corp., Apr. 1994.

[15] J.K. Antonio, "Architectural Influences on Task Scheduling: A Case Study Implementation of the JPDA Algorithm," Technical Report RL-TR-94-200, Rome Laboratory, Nov. 1994.

[16] L. Nyland, J. Prins, R.H. Yun, J. Hermans, H.-C. Kum, and L. Wang, "Achieving Scalable Parallel Molecular Dynamics Using Dynamic Spatial Domain Decomposition Techniques," *J. Parallel and Distributed Computing,* vol. 47, pp. 125-138, 1997.

[17] G. Blelloch, S. Chatterjee, J. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zahga, "CVL: A C Vector Library," Technical Report CMU-CS-93-114, Carnegie Mellon Univ., Feb. 1993.

[18] J. Prins and D. Palmer, "Transforming High-Level Data-Parallel Programs into Vector Operations," *Proc. Fourth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* 1993.

[19] G. Blelloch and G. Sabot, "Compiling Collection-Oriented Languages into Massively Parallel Computers," *J. Parallel and Distributed Computing,* vol. 8, pp. 119-134, 1990.

[20] P.K.T. Au, M.M.T. Chakravarty, J. Darlington, Y. Guo, S. Jähnichen, G. Keller, M. Köhler, M. Simons, and W. Pfannenstiel, "Enlarging the Scope of Vector-Based Computations: Extending Fortran 90 with Nested Data Parallelism," *Proc. Int'l Conf. Advances in Parallel and Distributed Computing,* 1997.

[21] OpenMP Architecture Review Board, "The OpenMP API," OpenMP http://www.openmp.org, 1997.

[22] 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. Fourth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* 1993.

[23] L.G. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM,* 1990.

[24] B. Alpern, L. Carter, and J. Ferrante, "Modeling Parallel Computers as Memory Hierarchies," *Proc. Workshop Portability and Performance for Parallel Processing,* 1993.

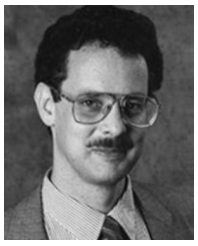
[25] H. Lu, Y.C. Hu, and W. Zwaenepoel, "OpenMP on Networks of Workstations," *Proc. Supercomputing '98,* 1998.

[26] J.C. Browne, S.I. Hyder, J. Dongarra, K. Moore, and P. Newton, "Visual Programming and Debugging for Parallel Computing," *IEEE Parallel and Distributed Technology,* vol. 3, 1995.

[27] I. Foster, *Designing and Building Parallel Programs.* Addison Wesley, 1995.

[28] G.C. Fox, S. Ranka, and P.C.R. Consortium, "Common Runtime Support for High-Performance Parallel Languages," Draft Technical Report PCRC-001, Northeast Parallel Architectures Center, Syracuse Univ., July 1993.
[29] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. v. d. Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.
[30] K.M. Chandy, "Concurrent Program Archetypes," Computer Science Report 256-80, California Inst. of Technology, Pasadena, http://www.etext.caltech.edu/Papers2/ArchetypeOverview/ArchPaper.html, 1995.
[31] E. Johnson, D. Gannon, and P. Beckman, "HPC++: Experiments with the Parallel Standard Template Library," *Proc. Int'l Conf. Supercomputing,* 1997.
[32] L.V. Streepy Jr., "CXdb: A New View on Optimization," *Proc. Supercomputer Debugging Workshop,* 1991.
[33] M.T. Heath, "Performance Visualization with ParaGraph," *Proc. Second Workshop Environments and Tools for Parallel Scientific Computing,* 1994.
[34] S. Browne, "Cross-Platform Parallel Debugging and Performance Analysis Tools," *Proc. EuroPVM/MPI '98,* 1998.
[35] B. Zhou and N.K. Bose, "Multitarget Tracking in Clutter: Fast Algorithms for Data Association," *IEEE Trans. Aerospace and Electronic Systems,* vol. 29, pp. 352-363, 1993.
[36] B. Zhou and N.K. Bose, "An Efficient Algorithm for Data Association in Multitarget Tracking," *IEEE Trans. Aerospace and Electronic Systems,* vol. 31, pp. 458-468, 1995.
[37] D.W. Palmer, "Efficient Execution of Nested Data-Parallel Programs," Univ. of North Carolina, 1996.
[38] R.A. Wagner, "Task Parallel Implementation of the JPDA Algorithm," technical report, Dept. of Computer Science, Duke Univ., Durham, N.C., June 1995, ftp://ftp.cs.unc.edu/pub/projects/proteus/rome/wagner-jpda.pdf.
[39] L. Nyland, J. Prins, A. Goldberg, P. Mills, and J. Reif, "A Design Methodology for Data-Parallel Applications," *Proc. AIP Design Meeting (12/95),* 1995.
[40] G.E. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a Portable Nested Data-Parallel Language," *Proc. Fourth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* 1993.
[41] R.M. Karp and Y. Zhang, "A Randomized Parallel Branch-and-Bound Procedure," *Proc 20th Ann. Symp. Theory of Computing,* 1988.
[42] L.V. Kale, B. Ramkumar, V. Saletore, and A.B. Sinha, "Prioritization in Parallel Symbolic Computing," *Lecture Notes in Computer Science,* T.I.a.R. Halstead, ed., vol. 748, pp. 12-41, Springer-Verlag, 1993.
[43] A. Brüngger, A. Marzetta, K. Fukuda, and J. Nievergelt, "The Parallel Search Bench ZRAM and Its Applications," *Annals of Operations Research,* 1999.
[44] A. Marzetta, "ZRAM: A Library of Parallel Search Algorithms and Its Use in Enumeration and Combinatorial Optimization," ETH Zürich, 1998, http://nobi.ethz.ch/ambros/zram/zram.html.
[45] G.E. Blelloch, "NESL: A Nested Data-Parallel Language," Technical Report CMU-CS-93-129, Carnegie Mellon Univ., Jan. 1992.
[46] G. Keller and M. Chakravarty, "On the Distributed Implementation of Aggregate Data-structures by Program Transformation," *Proc. Fourth Int'l Workshop High-Level Parallel Programming Models and Supportive Environments (HIPS '99),* 1999.
[47] G. Agha, "Concurrent Object-Oriented Programming," *Comm. ACM,* vol. 33, pp. 125-141, 1990.
[48] H. Korab, "Access Feature: Stormy Weather," http://access.ncsa.uiuc.edu/CoverStories/StormPrediction/Storms.html, 1999.
[49] K. Droegemeier, "Performance of the CAPS Advanced Regional Prediction System (ARPS) during the 3 May 1999 Oklahoma Tornado Outbreak," http://geowww.ou.edu/~kkd/May 3 Case/, 1999.

Lars Nyland received his BS with highest honors from the Pratt Institute in 1981, his AM from Duke University in 1983, and his PhD from Duke University in 1991. He is a research associate professor of computer science at the University of North Carolina at Chapel Hill. His research interests include languages and algorithms for high-performance computing, 3D model acquisition, and image-based rendering.



Jan F. Prins obtained a BS in 1978 in mathematics from Syracuse University and an MS in 1983 and a PhD in 1987 in computer science from Cornell University in the area of formal methods in program development. He is an associate professor in the Department of Computer Science at the University of North Carolina at Chapel Hill. His research interests center on parallel computing, including algorithm design, computer architecture, and programming languages. He is a member of the IEEE.

Allen Goldberg received the PhD degree in computer science from the Courant Institute, New York University in 1979. He is currently a senior researcher at the Kestrel Institute, Palo Alto, California. Previously, he was an assistant professor of computer science at the University of California, Santa Cruz. His current interests include formal methods, program transformation, and mobile code security. He is a member of the IEEE.

Peter Mills received the BA in mathematics from Duke University in 1978 and the MS in computer science from the University of North Carolina at Chapel Hill in 1989. He is chief technology officer of Orielle. Prior to founding Orielle, he was a research associate in the Department of Computer Science at Duke University from 1991 to 1996 and a senior scientist at Applied Research Associates from 1997 to 1999. His research interests include parallel computing, real-time systems, and distributed object technologies. He is a member of the IEEE.