Architecture Independent Parallel Design Tool: A Refinement-Based Methodology for the Design and Development of Parallel Software

July 1996

A DESIGN METHODOLOGY FOR DATA-PARALLEL APPLICATIONS

BY

LARS NYLAND
JAN PRINS

Dept. of Computer Science
Univ. of North Carolina
Chapel Hill, NC 27599

ALLEN GOLDBERG

Kestrel Institute
3260 Hillview Av.
Palo Alto, CA 95060

PETER MILLS
JOHN REIF
ROBERT WAGNER

Dept. of Computer Science
Duke University
Durham, NC 27708

Report for Task II
CDRL A005

for contract #F30602-94-C-0037

UNC TECHNICAL REPORT TR96-032

Abstract

Data-parallelism is a relatively well-understood form of parallel computation, yet developing simple applications can involve substantial efforts to express the problem in low-level data-parallel 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, supporting 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.

1. Introduction

Data-parallelism can be generally defined as the concurrent application of an arbitrary function to all items in a collection of data, yielding a degree of parallelism that typically scales with problem size. This definition permits many computationally intensive problems to be expressed in a data-parallel fashion, but is far more general than the data-parallel constructs found in typical parallel programming notations. As a result, 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 to increase performance by scaling the parallelism to a larger number of processors. Even in the face of the longer development times, the payoff of such increased performance is often worth the effort. For problems that are extremely time-consuming, the time taken to achieve results can be reduced by factors of 10's or 100's (week-long weather simulations take an afternoon) with sufficient processors. Real-time applications that are not possible with sequential computers can be achieved with data-parallel hardware (for instance, the 33 millisecond cycle of video signals represents a hard deadline). Other reasons for using parallel processing [Prins and Goldberg 1995], such as fault-tolerance or multi-function computing, do not drive data-parallel development, due to the paradigm shift necessary to express such applications.

Data-parallel programs may themselves be part of a larger process-parallel context in which their execution and mutual interactions are managed. Thus problems that can be decomposed into a number of successive data-parallel stages can be arranged into a pipeline to process successive problem instances concurrently. In such a context, issues of latency— the time between presentation of the input and delivery of the output for a problem instance, and throughput— the total rate at which problem instances are solved, are important measurements. For example in video image processing or tracking, the throughput determines whether the system operates in real-time, and latency determines the freshness of the resulting analysis. The interaction of data-parallel design with process parallel design is the subject of a future study.

A refinement methodology is proposed that generates a tree of data-parallel applications 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**, where an executable description of the design can be evaluated for its complexity and scalability.
- **A Data-Parallel Design Notation** that supports a fully generalized definition of data-parallelism and eliminates dependence on specific architectures.
- **Analysis and Refinement** of the design based on fundamental considerations of complexity, communication, locality, and target architecture.
- **Prototyping**. Disciplined experimentation with the design at various levels of abstraction to gain information used to further direct the refinement process.

1.1 Overview of the Design Methodology

1.1.1 High-level Expression

In current practice, parallel architectures typically influence software design early and heavily, and in doing so, overconstrain it. Since parallel architectures are rapidly evolving, architecture-specific software has a limited lifetime. Alternatively, architecture-independent environments allow the exploration of families of designs without the constraints imposed by specific architectures. Once developed without constraint, they can be slowly refined to classes of architectures (such as vector or message-passing), and can eventually be transformed to architecture-specific instances. Then as architectures change, and newer machines become faster, the original higher-level designs and prototypes will still be useful as a basis for developing new versions. Thus central to this methodology is the high-level, architecture-independent expression of the design in a design notation.

Many of the defense-critical computational problems, such as sensor processing, filtering, data fusion, track association, signal processing, fluid flow, etc., are naturally cast as data-parallel computations. Solutions to these problems use complex, sophisticated algorithms that are subject to improvements and refinements. Thus not only is the target architecture undergoing rapid and drastic changes, but so are the algorithms. 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 where the focus subsequent design and development.

The effect in current practice of architecture dependence is readily apparent; one relevant example is the Rome Laboratory sponsored <u>Sensor-processing Case Studies</u> report from MITRE [Games, et al. 1994] where the authors comment on one of their designs, saying, "... partitioning for the different MPL versions were all done 'by hand'...," indicating not only the use of an architecture-specific language, but a choice of manual data decomposition specific to the hardware chosen. While there is no doubt that their actions were initially led by high-level dataflow analysis, that they are experimenting with partitioning 'by hand' in a one-architecture programming language to achieve good results is a recent demonstration of the prevalent effect architecture has on software design.

Parallel architectures are rapidly evolving, and their limited lifetimes further reduce the applicability of architecture-dependent parallel software. Again, this is shown in the MITRE report by their choice of programming languages: MPL for programming the MasPar MP1 and C* for programming the Thinking Machines CM-2. The machines on which these languages are supported, while still available at some computation centers, are no longer preferred due to the difficulty of programming with only modest performance gains over RISC-based workstations. The specific programming models used have diminishing importance, as they are confined to the two architectures on which they run. The Sisal implementation described in the MITRE report, however, is not bound to a particular architecture[1], and can run on a variety of machines available today with comparable performance to the highest performance programming models available.

The proposed methodology eliminates dependence on specific parallel architectures, putting great emphasis on high-level design where there is much greater leverage. Much better performance can be obtained by good algorithm choice and refinement than by low-level optimization of a poor algorithm.

1.1.2 Data Parallel Design Notations

There is a general consensus on concise high-level notations for data parallel problems. These notations are expression-oriented, as exemplified in FP by Backus [Backus 1978] and the Bird-Meerten notation that posit an algebra of predefined data-parallel operations. Proteus, Nesl, Sisal [Blelloch 1996, Cann 1993, Levin and Nyland 1993] and other programming languages support this by providing a data-parallel constructor. These operations operate directly on data aggregates such as sets or sequences. Example operations are *map* and *reduce*. *Map* takes an aggregate (e.g., a sequence) and a function and creates a new sequence computed by applying the function to each element of the initial sequence. An example is `[x in V: -x]`, a Proteus expression to negate every element of a sequence *V*. *Reduce* takes an aggregate whose elements have a particular base type, and a function that combines two elements of that type, producing a value of the same type. An example is `+/V`, which sums the elements of *V*. In addition to reduction with standard operators, user-defined functions may also be used.

There are two variants of data-parallel notations. One uses iterators that introduce bound variables varying over an index set (e.g., a set former); in the other index sets are implicit. This paper will use iterator notation of the Proteus language (described in section 4.5). Parallelism is implicit in this notation. It has simple semantics, and can be refined by algebraic methods.

For all but the most trivial problems, specifications will employ *nested* parallel data structures. That is, the elements of aggregate data structures are themselves aggregates, and use sequence operations or sequence values within larger sequences for computing some result. By creating nested sequence expressions, large amounts of work are specified with few or no ordering restrictions. Algorithms operating on such data structures are called *nested data parallel (NDP)* algorithms.

In a refinement/translation context, lower-level notations introduce other aggregate abstractions. For example, HPF, C* and MPL aggregate data structures are arrays, shapes, and plural variables, where additional bookkeeping data is required to provide the functionality of the higher-level sets and sequences. Lower-level languages also allow explication of further refinement choices, for example by making communication and processor scheduling explicit.

---

[1] While Sisal is not architecture-specific, it supports a particular model of programming, namely that of a value returning functional domain. Message-passing paradigms, for instance, are not supported in Sisal. More information on Sisal is available in [Cann 1993]

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. During the risk analysis phase of development, development relies on concise high-level, executable notations for data parallel problems, such as FP, Sisal, Nesl and Proteus. 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 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.
- **Environmental and Performance 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 multi-computers present a global memory space, and allow all processors to access any memory address. Global variables are freely shared and an address refers to the same location, regardless of processor. The data model is very similar to a sequential programming model, with the addition of multiple processes (tasks or threads) with private memory.

On a machine with uniform memory access (UMA), all memory takes approximately the same amount of time to read/write, regardless of address or location. The Cray Parallel Vector Processors, the NEC SX-4 and the Tera computer are examples of UMA computers. The large amount hardware required to create this capability is one of the larger costs in buying a computer from Cray.

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 non-uniform memory access (NUMA), where there can be multiple levels of memory (cache, local memory, cluster memory, non-cluster memory) accessible to the program. Examples of such computers are the Convex Exemplar SPP, the SGI Power Challenge, and the Cray T3E. The non-uniform 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.

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 that will support 8mb/sec 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 the highest overall computational power, since they are use 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 be PVM and MPI [Geguelin, et al. 1991, MPI Forum 1993b].

The processors in a SIMD computer are more closely coupled than those in MIMD computers, and have a tightly integrated communications network that has minimal latency and overhead. This allows the exchange of messages at a frequent rate with low penalty. The MasPar MP-2 is an example, where one word of nearby data can be transferred in one instruction cycle by each processor. However, SIMD architectures use small, relatively slow processors, thus there have to be many of them before performance gains can be achieved. The interconnection network is also typically exposed (2D or 3D mesh, hypercube, etc.), thus the assignment of work to processors is sensitive to location.

1.2.3 Architectural Conclusions

UMA architectures offer the simplest development model, along with high performance, but typically at great hardware expense. NUMA 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, while presenting the most specific programming model, can achieve excellent performance on large, scalable 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 one 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, the development of multitarget tracking (MTT) algorithms is explored. The problem is described, a set of algorithms is examined, and a list of questions is formulated to be answered about the differing implementations. With questions in hand, high-level development begins, to determine 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 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, performance and analyzers are useful to any methodology, while others, such as refinement, and high-level analysis tools are specific to the methodology proposed.

2. 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 even 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. An example of this is the JPDA algorithm as implemented at MITRE [Games, et al. 1994], and then re-implemented on a hypercube [Antonio 1994], with the unexpected conclusion 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,

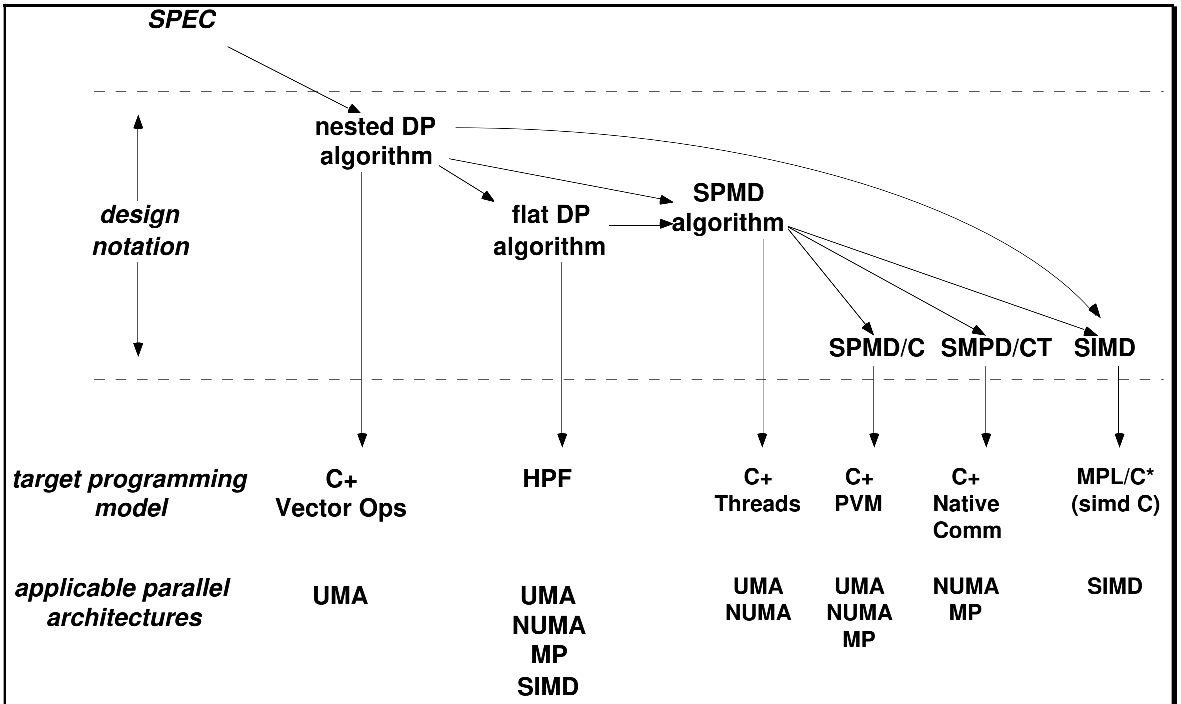

Figure 1. The process of data-parallel design starts with the selection of a good algorithm initially expressed in a high-level nested data-parallel paradigm. Further architecture-independent exploration continues with perhaps a flat data-parallel model, or migration to a SPMD model (multiple processes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

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.

Figure 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 Details of the Design Methodology

2.1.1 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 in 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.

2.1.2 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 algorithm to be used is a single clear-cut best algorithm, for others, perhaps the choice is not so clear, or perhaps the algorithm has been pre-selected. Regardless of how the choice is made, this stage simply boils down to asking the following question about the algorithm.

"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 (work and step complexities are the same for uniprocessor systems). Note 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, this issue comes 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 [Blelloch 1996]. In high-level languages, the work complexity of an operation is typically the sum of the work-complexities of its subexpressions (plus 1), 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 complexity is of primary importance, it is often the case that work-optimal algorithms are more complex than suboptimal ones. The complexity may be manifested in an algorithm that is more difficult to code and/or an algorithm with high constant factors associated with its step complexity bounds. The latter condition means that a theoretically better algorithm may not outperform an inferior one unless the problem size is very large. In such cases a parallel algorithm with suboptimal work efficiency, may still be a good target for parallelization at small problem sizes (e.g., an $O(n \log n)$ sequential sorting algorithm when parallelized as a bitonic sort will perform $O(n \log^2 n)$ work (suboptimal) with a step complexity of $O(\log^2 n)$, but is the algorithm of choice for sufficiently 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 the whether 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.

2.1.3 Environmental and Performance Constraints

The analysis in the previous step does not consider anything but the work and step complexities of the algorithms. At this stage, 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 executed the same way regardless of the data values?" 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.

2.1.4 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 figure 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 non-local 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 non-local communication and still exploit the available parallelism.

Parallel hardware can be compared in the same manner, by considering the ratio of floating-point operations-per-second 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 ratio), 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 non-local 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 can not 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 virtualization.

"Is the work regular or irregular?" Regular data is much easier to decompose for parallel execution than irregular data; there is usually some regular decomposition scheme that has good locality and low communications costs. Regular problems are those that assign consistently sized sub-problems to each processor, such as "a row of a matrix," or "a cubic region of space," where the rows or regions are all 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 sub-problems 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 making a difficult trade of the simplicity of a static processor and communication to overcome the disadvantage of performing more work. Irregular problems often have a recursive structure in which sub-problems 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.

2.1.5 Design Refinement

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

2.1.5.1 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 intra-function parallelism. The vector model such as that presented by the C Vector Library (CVL) [Blelloch, et al. 1993], 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 non-uniform 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 [Prins and Palmer 1993] and [Blelloch and Sabot 1990].

2.1.5.2 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, C* and MPL. 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. Beside 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.

2.1.5.3 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. An SPMD implementation is characterized by dynamically created threads of control in a single shared-memory program. The benefit of this kind of program is that the 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 [Culler, et al. 1993], BSP [Valiant 1990], and PMH [Alpern, et al. 1993] models can be used to direct the refinement process. As figure 1 illustrates, SPMD designs can be implemented on platforms for which the computation/communication ratio varies considerably.

Programs written in an SPMD style can be implemented in a shared-memory paradigm using C and Posix threads (or vendor-specific threads).

2.1.5.4 SPMD to SPMD/C

This model includes explicit communications in a SPMD program (same program running on different nodes, communicating with messages). After development on NUMA parallel computers, the developer has deep knowledge about which memory references are local, and which are not. This step makes the sharing of data explicit, by introducing messages. Synchronous messages provide shared data and synchronization, where non-blocking sends do not provide the sender with timing information.

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, and implement schemes that mimic cache memory. 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.

2.1.5.5 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. SIMD machines are often viewed as vector architectures with very long vector units. The problematic part of the transition is in non-local 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 (grid or hypercube connections). 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 a SIMD refinement. For SIMD code, non-local data must be communicated, but on a SIMD machine, the latency and overhead are not nearly as large, allowing efficient communication of smaller messages.

2.1.6 Instantiate on a machine architecture.

In the past, it was important to consider the distance non-local data must travel. The trend with parallel computers is to abstract the machine sufficiently so that the developer no longer has access to direct communication mechanisms. Indeed, for most current technology the delay caused by different physical distances are not reflected in the transmission time. Thus, the concern is usually limited to that of what is local and what is remote.

This point is demonstrated by an architectural study of mapping the JPDA onto a hypercube [Antonio 1994]. The patterns of communication required were noted, and a mapping derived from the dataflow requirements to the hypercube such that each message would only traverse one connection on the cube. The unexpected result of this careful mapping was that it had very little effect on the performance. The conclusion was that collecting several pieces of remote data serially outweighed the reduced communication time of each message. It was also reported that the time to read and write a message, regardless of the source and destination, were much larger than the transmission time. With the advantage of hindsight, it is clear that the impact of mapping could be judged prior to implementation by simply asking, "How much does message transmission time change with mapping on the nCUBE [or other architecture]?" The reported figures show that the overheads of writing and reading the message far outweigh the transmission time (with the message size used), so increasing the message distance by a factor of 10 does not even double the transmission time.

Still, some architectures do allow direct control over processor mapping and communication hardware, such as MasPar MP series. This question certainly applies when using hardware that allows this level of control. It also applies if an argument can be made that the algorithm will be significantly affected by the mapping. Library routines from the manufacturer often take the characteristics of the specific interconnect into account.

2.2 Translation of Prototypes to a Lower Level Language

High-level languages that support parallel computation may run as fast or faster than lower-level languages, but this is not often the case. After development has been explored at a high-level, where measurements have been taken, the process of achieving an efficient parallel code can occur. But care must be taken to use the knowledge gained from the early prototypes. If the prototype suggests several implementations based upon some machine metric, then some simple tests should be explored to determine what the value of the metric is for the machine in use. An example might be the overhead of sending a message. The time this takes varies substantially across high-performance hardware, and on machines with large message overhead, the messages should be reduced either by packing multiple messages into one, or using a solution that uses fewer messages. On the other hand, if message overhead is low, then no particular concern about them need apply.

Lower-level languages, such as any parallel Fortran, also give better metrics on actual resource usage. Most languages in this class do not provide automatic memory management (and garbage collection) and require the developer to use simple data structures of predictable size (arrays, for instance). The result of the low-level implementation is achieving reliable timing and memory utilization information.

One important aspect of high-level parallel prototypes is that it may be difficult to mimic their operation, even in a slightly lower-level language. This is due to the high-level semantics of the language, permitting excellent optimization and parallelization strategies. The best examples of this are irregular, data-sensitive applications, where the automatic derivation of parallelized functions creates extremely efficient parallel code. The same sort of data ordering and alignment, as well as higher-order function development *can* be done by hand by a developer, but it is complex, prone to error, and difficult to debug. It is not possible to put the same optimizations in lower-level languages, due to the more complex semantics and analysis required.

For data-parallel designs, translation can be to any of a number of target languages. First, nested parallelism may be translated to vector operations, regular (flat) data-parallelism may be translated to HPF or equivalent notation, and finally data-parallelism can be translated to an SPMD program using message passing or shared memory via techniques such as those describe in [Rogers and Pingali 1989]. Notations for the first two strategies are discussed next.

Examples of programming languages that support data-parallel programming are Proteus, Nesl, Sisal, C*, MPL, Fortran, and C. In each of these languages, the developer has a notion of what operations can run simultaneously. With the higher-level languages, collection-oriented operators specify the data-parallel operations, such as sequence quantification, sequence operations (reduction, scan), and using sequences as values (return values from functions). Expressions that define sequence values can typically be run concurrently, as can all the expressions in nested sequences.

Lower level parallel languages such as C* and MPL directly support data-parallel programming, by defining special data types. MPL defines "plural" variables, where each processor has a copy of the declared variable ("plural int I" defines the variable *I* on every processor, "plural float t[10]" defines an array *t* with 10 elements on every processor). Then in the MPL program, plural variables allow each processor to compute using its own data. Conditional statements are used to reduce the number of participating processors (e.g., "if (I > 0) Z = 1.0/I;"). If a conditional has both true and false clauses, both are run, first the true clause on the processors where the condition is true, and then the false clause on the processors where it is false. MPL also provides support for generalized data-movement (permute) or movement on a 2D grid (send/receive with neighbors in any of 8 compass directions). Scalar computations exist as C code, and specialized operations such as reduce and scan are specified with library calls. C* is similar, it provides shape declarations of data for concurrent computation. C* also provides automatic virtualization (the user defines the size of the data), where MPL only supports collections of data that are the size of the machine being used.

Serial languages used for data-parallel computation are Fortran77 and C (although predominately Fortran). This is done by recognizing loop structures free of serializing constraints, and translating them into data-parallel operations. This style of data-parallel programming has been the force behind powerful program optimization techniques used in vectorizing compilers. As only limited structures are concurrently executable, it is often the case that additions or changes to a program can drastically change the concurrency. Since the developer is relying on tools that recognize potential for concurrent operations, there must be a clear understanding of what structures can and cannot be recognized. This understanding is best developed by experimentation and legend (seasoned developers passing on the knowledge), and so the successful development of concurrent programs is not always straightforward.

In ideal programming models, programming for data-parallel execution is only slightly more complicated than programming for serial execution (the model fully supports collections of data). In lower level models that are typically used, data-parallel programming is much more difficult than serial programming, whether the user is manipulating a serial program (such as Fortran), or developing a program in a data-parallel language (such as MPL). The developer must be aware of many architecture-dependent details, such as the cost of different communication methods or expensive operations (the MasPar supports processor-specific addressing, but it is an expensive operation, and the cost affects choices of development).

2.3 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. Comparison of the Proposed Methodology to 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 available currently 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. While CASE tools would be beneficial for parallel software development, few are specifically designed to handle the additional needs of developing software for parallel execution. One area where this is the exception is support tools for developing concurrent real-time systems. There are pre- and post-design tools to analyze the timing requirements of a system, one graphical system is PERTS [Hull, et al. 1995], another is ControlCalc [Tetewsky 1994], a spreadsheet-based system from RTWARE in Durham, NC.

One CASE-style tool that aids in parallel program performance analysis is the PAWS tool developed at Syracuse University and Rome Laboratory [Pease, et al. 1991]. It has a graphical interface showing the dataflow dependencies of Ada and Sisal programs, allows the characterization of parallel hardware, and predicts the performance of a supplied code on a selected parallel architecture. Using the source language Ada, which is not a parallel programming language, the analysis performed finds concurrency available in the evaluation of expressions. When it is used with programs written in Sisal, a parallel language, the resulting analysis is substantially better, where PAWS predicts thousands of concurrent operations. Analysis and prediction tools such as PAWS combined with powerful parallel programming languages have the potential to drastically reduce development time by performing immediate analysis and displaying the results graphically.

3.1.2 Machine-Independent Programming Languages

With the relative scarcity of CASE tools for parallel program development, the next level of support is explored, namely that of programming languages. The justification for this focus is that programming languages present a model of computation. Parallel programming languages are often characterized by their ancestral language by what architecture’s that have compiler support. Some languages are vendor-specific, others are not. Table 1 categorizes some languages used for parallel programming.

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, and have little room for concurrency optimizations. Some use higher-level constructs, such as arrays and array operations, allowing stronger support for concurrency. And several of the languages mentioned (Proteus, Nesl, Sisal, Fortran90, Fortran-D) provide substantial optimization support to obtain a high level of concurrency from the developer’s source code.

Language	Programming Model	Architectures
Proteus, Nesl, Sisal, APL2	High-level, functional paradigm; nested vector creation and support; automatic vectorization of user-functions (except Sisal)	Cray, MasPar, CM-5, PVM-supported, serial.
PCN, CC++	Single-assignment, high-level MIMD	Distributed Memory Models
Fortran90, Fortran D, HPF	Mid-level, imperative; arrays and array operations, some have segmented arrays and operations.	Many
Fortran77	Low-level, global memory model	Almost universal support
Fortran-M	Low-level, straightforward message passing	Many
C*, MPL	Low-level, SIMD model, C* supports virtualization	All CM's, MasPar respectively
C+PVM , C+MPI	Low-level, portable message passing	Almost all

Table 1: Some Parallel Programming Languages, Models, and Architectures supported

Optimization is strongest in the vector models of parallelism, especially in the Fortran language. This is a result of the extremely high speed of multi-processor vector supercomputers, where there is a substantial body of code that can benefit from such powerful optimizations. Except for Proteus, Nesl, and Sisal, the further away from the vector model, the fewer optimizations there are for parallelism. That is, the execution of the program follows the code the user has written.

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 BLAS, LAPACK, LINPACK, NAG, and IMSL mathematical libraries and the PVM, MPI and ISIS communications libraries [Fox, et al. 1993, Geguelin, et al. 1991, MPI Forum 1993a, MPI Forum 1993b]. 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. A typical result is a program that relies on a library such as LAPACK; it will (probably) run very well on all machines that have a LAPACK library.

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

A more progressive method of support suggested for parallel software development is that of using predefined archetypes or templates [Barrett, et al. 1994, Chandy 1995]. This method has many benefits; among them are highly optimized architecture-specific correct 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, it is felt that templates will be a rather simple method of creating efficient, high-performance codes.

3.2 Post-development support of parallel software: analysis tools

As software systems mature, there is a need for tools to help analyze them. Debuggers aid developers in finding incorrect actions in programs, while analysis tools point out performance characteristics. The support for post-development tools in parallel programming is nearly up to par with the tools that exist for sequential programming. There are a variety of ways that performance issues can be determined. Some compilers will produce messages about statements that prohibit vectorization. Other systems include monitors that show different states of a parallel program (e.g., waiting or running) for each executing thread. Still other systems will record message transmissions with timing data in the hopes that analysis of such data will exhibit problematic areas of the program. This section examines the capabilities of different post-development tools.

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 debugger for the KSR is a port of the GNU debugger gdb with added support for multiple threads and thread-private memory. Debuggers developed specifically for debugging concurrent programs include MPPE from MasPar and TotalView from Cray. Both have powerful graphical interfaces (X-windows) combined with an interpretive command interface to create a useful tool for debugging Fortran, C, and MPL 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 a MIMD system that make use of the more pedestrian optimizations. Heavy optimization and debugging are often at odds with one another, it is often not possible to debug code that is optimized and 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).

The debugger developed by the Convex Computer Corp. is a remarkable achievement in this manner, and, along with their compiler, 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 is functioning, the focus changes from achieving proper functionality to that of efficient performance, after all, that is usually the point of using sophisticated 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 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 a micro- or millisecond, while the most accurate have registers that count clock cycles. Additionally, registers may exist that count local and non-local memory operations (cache hits) and floating-point operations, allowing the performance conscious developer powerful inspection tools.

In conjunction with accurate timers, statistics gathering capabilities can 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; KSR provided state-transition timelines; included with PVM is a set of X tools to monitor PVM programs; MPI provides the `upshot` program to display logged MPI events. All of these tools are useful for making large improvements in the performance of parallel (and distributed) software.

4. Case Studies: Multi-target Tracking

Up to this point, the discussion has focused on a model of developing parallel programs and the existing software development tools available. Now an example will be presented, to demonstrate 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, and through the development process, it will be shown that there is a need for additional support that is currently non-existent.

4.1 The Multi-target 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.

The specification of the solution to this problem relies on the definitions of the input and output parameters. An input matrix of *a priori* probabilities is supplied as input, and the calculation produces an output matrix of joint probabilities (or *a posteriori*) as described above.

While there are several algorithms for solving the multi-tracking 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.

The JPDA (Joint Probabilistic Data Association) is a specific association strategy that uses a weighted average of returns. Targets are not treated independently; if two (or more) targets have non-zero probability of being the same return, then the JPDA calculation of each target is dependent on the other(s). A report from MITRE [Games, et al. 1994] 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 [Zhou and Bose 1993, Zhou and Bose 1995], 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.

The MITRE study has been analyzed in detail to determine its exact relation to the ZB-JPDAF modified matrix permanent formulation. It turns out the MITRE parallel JPDA study is essentially an optimization of Zhou's matrix permanent written in a flat data-parallel style. The central formula by Zhou expresses the modified matrix permanent F as a recursive equation — it can be rewritten to sum over tracks instead of measurements (which corresponds to what is used in the MITRE report). The basic approach of the MITRE JPDA is to reverse Zhou's recursion into an iteration, and to recognize some of the common subexpressions of the $F(T, M)$ shared by the β_{jt} elements of the a posteriori track/measurement data-association probability matrix. The basic approach is sound, but as written in the MITRE report it is work inefficient and there are other ways to reorganize the use of common $F(T, M)$.

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 using Fortran on a CM-2, 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, Pre-Developed 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. Some pre-developed software [Bar-Shalom and Li 1993] was obtained for calculating the input probability matrices from the raw location data for the two algorithms under exploration, rather than redeveloping the data transformation pipeline.

4.4 Development hierarchy to solve multi-target tracking problems

Figure 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[2], and serve the purpose of achieving a baseline implementation, there is no initial concern for parallelism. The first descendant of the JPDA does target parallelism, it is a nested data-parallel implementation in Proteus, 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 towards particular architectures.

Additional studies were done, according to the development path outlined in figure 1. A flat data-parallel version of the CR-JPDA was developed and translated 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. And finally, a paper study was performed to estimate the performance of a variety of message-passing implementations of the CR-JPDA. Each of these is discussed in the following sections.

[2] Proteus was chosen for its expressiveness of nested sequences, and the ability to translate nested sequence notation to vector-based execution

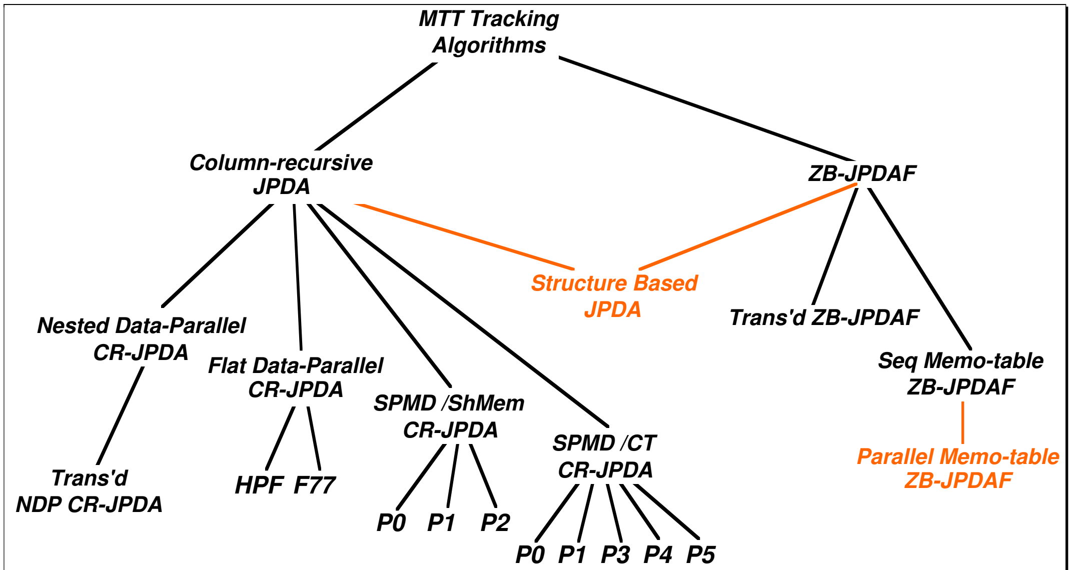

Figure 2. The development tree (thus far) of the MTT algorithms. The grayed instances have not yet been developed.

4.5 Further data-parallel refinements of the JPDA recursive matrix permanent formulation

The design hierarchy can be regarded as beginning with the recursive matrix permanent formulation given in [Zhou and Bose 1995] which can serve as a problem specification (amenable to naive parallelization) against which further designs may be validated and continues in the design of parallel algorithms based on optimizations of Zhou's matrix permanent. The refinements consist first of a nested data-parallel implementation of the MITRE study, then in improvements based on further equational optimizations of Zhou's formula involving (1) memoization, i.e., dynamic programming and (2) equational reformulation to share the computation of common subexpressions in an ascend-descend manner that can be extended to accommodate approximate heuristics ("near-optimal" JPDA).

After close inspection of the MITRE JPDA parallelization effort, Mills [Mills 1995] has developed a far faster parallel JPDA design based on further equational optimizations of Zhou's recursive matrix permanent formulation [Zhou and Bose 1995]. The algorithm reduces redundant computation of common subexpressions in an ascend-descend computation that avoids the overhead in powerset indexing, and which is amenable to nested data-parallel expression. Implementation of this algorithm is being pursued (and will be described in detail in a separate publication) in Proteus.

The algorithm can also be naturally extended to accommodate heuristic techniques for computing an approximate JPDA. The simple parallel (and response-time-adaptive) greedy heuristic modification to the Zhou formulation consists of computing a subset of full joint events by narrowing (based on a time-adaptive threshold) what branches are computed at each level of recursion. It can adaptively return approximate solutions to meet real-time demands.

However, the utility of approximations of the JPDA (or PDAF) is a questionable issue, as the JPDA is a probabilistic algorithm that is highly sensitive— indeed, even for small amounts of clutter (e.g., > 3) its accuracy rapidly degrades and it loses tracks. Certainly it has been established in the literature that approximating the JPDA using partial joint events [Roecker and Phillis 1993, Zhou and Bose 1993] is so inaccurate as to be essentially useless [Roecker 1994, Zhou and Bose 1995], a result that is not surprising considering it is antithetical to the underlying dependence on joint probabilities (indeed, Bar-Shalom [Bar-Shalom and Fortmann 1988] repeatedly

[1..n]	A sequence from 1 to *n*, inclusive
a[i]	Indexing. Return the *i*th 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.).

Table 2. Proteus syntax for sequence expressions, let-expressions, and infix function calls.

emphasizes that full joint probabilities must be computed). It is tentatively surmised that, due to the JPDA's sensitive nature, rather than giving primary emphasis to approximation techniques for subsets of full joint events, further effort is better focused on optimizing the exhaustive JPDA.

4.6 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 a block-structured, imperative, procedural language. 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 be scalars (numbers, booleans, etc.) or sequences themselves. Table 2 shows some sequence syntax and operations.

The nested data-parallel nature of Proteus (or any language) permits sophisticated compilation and optimization techniques for parallel execution. Typically, nested sequence expressions, even those crossing function boundaries, can have all their operations run concurrently, as long as the functions are pure. 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
```

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

$Z_n = \{0, \ldots, n-1\}$
Input: $P = [p_{ij}] \in R^{n \times (m+1)}$
Auxiliary: $F = [f_{ij}] \in R^{(m+1) \times (2^n - 1)}$
Output: $B = [\beta_{ij}] \in R^{n \times (m+1)}$

For $l = 1..m$

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

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

where $\alpha_{ij} = p_{ij} \cdot f^m_{j, Z_n \setminus i}$

Figure 3: The JPDA calculations specified in the MITRE report

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 [Blelloch and Sabot 1990, Prins and Palmer 1993].

4.7 The JPDA, as described by Games et al of MITRE and Bar-Shalom

4.7.1 Description

The CR-JPDA algorithm has a short mathematical description with many subtleties. The description is shown in figure 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 from the for-loop that there are m+1 sequential steps needed for calculating the elements of *f*. The space required for *f* is $(m+1)m2^n$, however, analysis reveals that only 2 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 $m(m+1)n2^{n-1}$.

```
#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;
}
```
Figure 4. Initial implementation of the column-recursive JPDA as described in figure 3

4.7.2 Commentary on the Column-recursive JPDA

The description of the algorithm is straightforward mathematics, with some ordering information akin to program control (for loops). It was translated into Proteus[3], and quickly found that there are indexing bugs in the normalization step. Once this was repaired, many test cases were generated and run, and it was indeed seen that slight increases in the number of targets resulted in substantial increases in the computation.

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 step complexity is

[3]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-loop 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.

estimated at O(m) (or O(mn) if reduction and scan 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 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 only 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 right-hand 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.

4.7.3 Commentary on Initial JPDA Implementation

A solution was developed from the description in figure 3. It was developed in Proteus, and is shown in figure 4. The implementation was straightforward, except for an indexing error in the normalization step of the mathematics.

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 implementation 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 under the Proteus interpreter, and is almost amenable to translation for data-parallel execution. The amount of parallelism is limited by the iterative nature of the *el* for-loop. The for-loop runs sequentially and is not translatable since it (re-) defines a sequence iteratively. The focus of further modification is to rewrite the loop that calculates f, with the goals of making it functional, and storing only the lth and l+1 levels.

4.7.4 Nested Data-Parallel Implementation of the JPDA

The column-recursive JPDA has iterative constructs. For the sake of clarity and parallelism, the iterative assignments are rewritten as sequence operations using recursion (figure 5 shows the updated function *cr_jpda* and the added function *compute_f*). The recursive nature keeps only the 2 most recent levels of f active, 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 amenable to transformation that generates a C program of vector operations from the Proteus input. The important aspects of parallelism in the generated C program are

```
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;
}
```

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

- 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.

4.7.5 Flat Data-parallel Implementation of the CR-JPDA Algorithm

To achieve a "flat" data-parallel version, the code was modified as shown in figure 6. Flat data-parallel programs have the following characteristics: no nested sequence structures except to mimic rectangular multi-dimensional arrays; iterator variables are only bound to scalar values; and only pure function calls are allowed in sequence expressions. This model is similar to that allowed by HPF.

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.

```
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;
}
```

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

If an $n*2^n$ table indexed by (i, a) that 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 parallelism is only applied to a single, innermost sequence expression. 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 are only in the early stages of development, so some parallel operations are not executed in parallel (e.g., 'pack'). 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.

4.7.6 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. 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

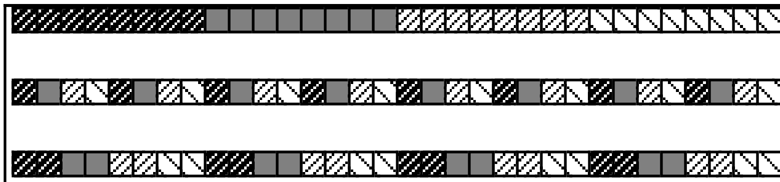

Figure 7. Memory decomposition with different shadings for 4 processors over 32 data elements. Blocks contain 8 contiguous elements, cycles have deltas of 8, and a block-cyclic layout has block of size 2 ($32/4^2=2$).

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 16 processors 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, Convex SPP, HP K-class, and DEC Alpha cluster architectures.

The first attempt was to mimic the HPF version, a simple enough task to require less than one hour of time for this problem. 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 or sub-page). 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 generate 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 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* with a single one-bit 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, a block-cyclic layout can be created by introducing an additional control loop. 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 approximately 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. The different memory layouts are shown in figure 7. When using 16 processors, the idle time of the least loaded thread went from over 60% to less than 20%.

4.7.7 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. It was refined into a nested data-parallel version that 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 converted sequence manipulation 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, runs of this version were limited to a single processor. Additional concerns with the Fortran77 version had to do with common tool support for that language, namely only the innermost sequence operation is run in parallel, so iterator inversion had to be considered to achieve the highest-performance.

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. 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.

4.8 The ZB-JPDAF, as described by Zhou & Bose

4.8.1 Description

Another algorithm for solving multi-target tracking problems is an algorithm developed by Zhou & Bose in [Zhou and Bose 1995] which they refer to as an *improved* algorithm over their depth-first-search algorithm [Zhou and Bose 1993] for a joint probabilistic data association filter (hereafter referred to as ZB-JPDAF). The algorithm is a tree-exploration described with simple recursive equations (and appropriate base cases), as such, it is straightforward to implement at a high-level in Proteus. The independence of node expansions is a key feature that enhances parallelism, as there are relatively few ordering constraints to with which to comply.

Zhou & Bose define the output, $\beta_{jt,}$ as the *a posteriori* probability that measurement j is associated with target t (j in $\{1..m\}$, t in $\{1..n\}$). For $j = 0$, β_{0t} is the a posteriori probability that no measurement originated from target t. This algorithm uses the same input data, the a priori matrix P, as in the CR-JPDA. The computation for β, given p_{jt}, $A = \{1..n\}$ (all targets), $B = \{0..m\}$ (all measurements) is

$$\beta_{jt}(A,B) = \begin{cases} p_{jt} \cdot F(A \setminus t, B \setminus j) & j \in \{1..m\},\ t \in \{1..n\} \\ p_{0t} \cdot F(A \setminus t, B) & j = 0, t \in \{1..n\} \end{cases}$$

The β values must be normalized across measurements

$$\overline{\beta}_{jt} = \frac{\beta_{jt}}{\sum_{k=0}^{m} \beta_{kt}(A,B)}$$

The definition of F is

$$F(A,B) = p_{0t} \cdot F(A \setminus t, B) + \sum_{j \in B \setminus 0} p_{jt} \cdot F(A \setminus t, B \setminus j) \quad \text{for some} \quad t \in A$$

where the base cases are

$$F(A,B) = \begin{cases} \sum\limits_{j \in B, t \in A} p_{jt} & \text{if } |\mathrm{A}| = 1 \\ (p_0)^{|A|} & \text{if } B = \{0\} \end{cases}$$

Using these definitions, it is straightforward to develop a Proteus implementation, which is shown in figure 8 (The *Z*, *members*, and *without* functions are identical to previous versions).

4.8.2 Commentary on ZB-JPDAF

The description of the ZB-JPDAF algorithm is obviously the result of years of algorithm refinement. The equations presented are clear, recursive equations. All the base cases appear to be described equally well. From this confident presentation, the ZB-JPDAF was quickly implemented in Proteus, using the exact structure as presented in the algorithmic description.

This algorithm's execution is a tree exploration, and 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-density of measurements, then the work complexity is large, and as the density of measurements goes down, so does the work. The work is $O(\max(m,n)) \leq T(m,n) \leq O(m!n)$. The authors claim that there are typically no more than three probable measurements for each target, thus the average case has a complexity of $O(n3^m)$.

The 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 sub-problem to solve will be of different size, so allocation of sub-problems to processors will result in all solutions running as slow as solving the slowest problem. Unbalanced, irregular computation such as this is exactly the style of computation where conventional programming languages offer little or no 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.

```
-- An implementation based on the Zhou-Bose95 paper about an efficient
-- implementation for data association in multitarget tracking.
-- targets : [0..n-1]
-- measurements : [0..m]
-- probability matrix: p[0..m][0..n-1]

-- functions Z, without, lg, and members are the same as other versions

function F(A, B) {
 return
     if #members(A) == 1
     then +/[t in members(A) : +/[j in members(B) : p[j][t] ] ]
     else if B == Z(1)
          then p[0][0] ** size(A)
          else
               let t = max/members(A)
               in p[0][t] * F(A 'without' t, B) +
                    +/[j in members(B) | j != 0 && p[j][t] > 0.0 :
                         p[j][t] * F(A 'without' t, B 'without' j) ];
}

function compute_beta(p)
{
 return let
          m = #p - 1,
          n = #p[0],
          A = Z(n),
          B = Z(m+1),
          beta = [j in [0..m] : [t in [0..n-1] :
                    if p[j][t] > 0.0
                    then if j > 0
                         then p[j][t] * F(A 'without' t, B 'without' j)
                         else p[j][t] * F(A 'without' t, B)
                    else 0.0
               ] ],
          c = [t in [0..n-1] : +/[j in [0..m] : beta[j][t]]]
        in
          [j in [0..m] : [t in [0..n-1] : beta[j][t] / c[t] ] ];
}
```
Figure 8. A functional, recursive implementation of the Zhou-Bose algorithm.

Nested data-parallel programming languages automatically generate functions that apply to all nodes at a single level of the tree, based on the function that applies to a single node (which is all the developer writes). Programming languages such as Proteus and Nesl provide substantial leverage by providing inter-function parallelism, especially in the case of irregular decomposition. In this algorithm, the step complexity is related to the depth of the tree, since all nodes at each level will be executed in one step. There is no doubt that the code could be transformed to an iterative rather than recursive style, and be parallelized similar to the methods used by the Proteus and Nesl translation tools, but the resulting code would be large, complex, difficult to understand and difficult to modify.

Templates are an alternate solution. Tree-search algorithms have been studied for efficient parallel implementation, and the resulting algorithms are complex ([Karp and Zhang 1988] is an example of the complexity required to minimize communication in parallel branch-and-bound algorithms). If templates existed for parallel tree-search, then all that would be required of a developer is to write how a node is expanded, and what its return value is. The remaining code would be included in the template, and provide efficient execution for a variety of parallel architectures. Unfortunately, at this time, templates for parallel tree-search do not exist.

The conclusion is that the computation is irregular enough that if left unconsidered (treated identically), the solution would make poor use of any parallel computer. The solution is to explore the irregularity in an environment that supports it, the three suggested are programming languages with support for irregular data (our choice for this problem), robust multi-programming concurrent environments, or with templates. It is too complex a task to solve in other manners.

4.8.3 Analysis of the ZB-JPDAF Implementation

The equational representation of this algorithm is not only straightforward to implement, it also provides an intuitive explanation of the computation, by essentially saying “assign one measurement to one target, and repeat the calculation with the remaining data.” At each level of the search, all possible assignments of measurements are associated with a single target, with the probabilities appropriately calculated, which yields the overall joint probabilities.

The equational description provides a firm foundation for developing a baseline prototype of the ZB-JPDAF algorithm in Proteus. It 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 equations ignores many 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 create an identical sub-problem. If the results are *memoized*, then the worst-case work is reduced by $m!/2^m$. 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.

Any further implementation of the ZB-JPDAF would require a parallel memo-table implementation, either direct or hashed. Thus, the difference between the description of the ZB-JPDAF and an efficient parallel implementation is quite large. Some programming languages provide support to handle the irregularity of the tree-search, but no support (other than hand-coding) exists for parallel memoization, which is crucial to the work-efficiency of this algorithm. The development is exacerbated when a language that does not support nested data-parallelism is used. The steps required to develop an efficient implementation in an efficient, low-level programming language are:

- build a function that evaluates all nodes at one level of the tree in parallel, including setting up the data for each level.
- build code for a parallel memo-table, perform a lookup prior to evaluation eliminating further redundant evaluations, and insert the results of a parallel evaluation into the memo-table in parallel.

While this is no small task, it is certainly an achievable result. It gives pause, long enough to ask whether or not this is the most efficient algorithm to solve the problem at hand.

4.9 A Message-passing Study of the JPDA

Along the with the high-level implementations in Proteus, a paper study analyzing the hypothetical computation and communication costs of the JPDA for a specific architecture was performed by Professor Robert Wagner (Duke University) [Wagner 1995]. This section is a synopsis of the study.

4.9.1 Foundation of Study

The goal of the MP task parallel study is efficient execution of the JPDA, by looking at alternatives to minimize the most costly parts of the program. To determine the costs, a model is developed based on the IBM SP1 and SP2 computers, which have between 64 and 2048 RS/6000 processors running at 62.5MHz. Each processor can execute one floating-point multiply and add instruction (FMA) every cycle. The communications hardware supports point-to-point messages on a network with a 30usec latency and 8Mbps transfer rate. The model assumes no contention in the network, based on the manufacturer's claims.

To achieve a baseline against which further versions can be compared, a sequential version was developed. In addition to measuring the output of the XLF compiler, two tools were developed to improve and measure performance. The first tool performed some partial evaluation of the program, eliminating conditionals and unrolling loops. The second tool simulates a cache similar to that on the RS/6000, measuring cache hits and misses. The combination of these two tools predicted that the JPDA code could run at 71% of peak performance of the RS/6000 on input data with 19 tracks and 20 measurements.

4.9.2 Parallel Algorithms

The first attempts at parallelism yielded two versions of the program. The key computational statement in the JPDA is:

```
f[el+1][j][a] += p[j][t] * f[el][j][a less t];
```

It is responsible for virtually all the work performed. The index *j* ranges from 0..*el*, and *el* ranges from 0..*m*. If there are only *m* processors, then it makes sense to assign all elements of *f* with equal values of *j* to the same processor, since it eliminates communication (each processor has a copy of *P* and can compute *f*[0]). This parallelization scheme is no longer efficient when the number of processors, *P*, exceeds *m*. This version is called P0. When this decomposition is used, the computation time (TP0) and communication time (CP0) are calculated by:

$$TP0(n,m,P) = \sum_{el=0}^{m-1} \frac{n \cdot \left\lceil \frac{el}{P} \right\rceil \cdot 2^{n-1}}{0.71 \cdot 62.5} \ \mu\,\mathrm{sec}$$
$$CP0(n,m,P) = m \cdot \left(2^{n-1} + 30\right) \ \mu\,\mathrm{sec}$$

The communication cost can be ignored, since there are idle processors that can redundantly compute the initial values of *f* required for subsequent calculations. For *n*, *m*, and *P* set to the values of 19, 20, and 20, the value of TP0(19, 20, 20) is 2.25 sec.

A simple scheme that uses more than *m* processors is to let *np* bits of *a* ($0 \le a < 2^n$) represent the processor number on which f[el][j][a] resides ($np = \lg P$). When *t* represents a bit in this range, communication is required to obtain the remote value. Version P1 of the JPDA uses this distribution when the set of bits are the *np* highest bits of an *n*-bit number. When this decomposition is used, the computation time (TP1) and communication time (CP1) are calculated by:

$$TP1(n,m,np) = \frac{2^{n-np-2}\, m(m-1)(n+np)}{0.71 \cdot 62.5} \mu\,\mathrm{sec}$$
$$CP1(n,m,np) = m(m-1) \cdot np \cdot \left(2^{n-np} + 30\right) / \, 2\mu\,\mathrm{sec}$$

Substituting actual values for *n*, *m*, and *np* of 19, 20, and 6 (64 processors), the computed values are TP1(19, 20, 6) = 0.438sec and CP1(19, 20, 6) = 9.37sec. This reveals that the computational component is minimal compared to the communication. One quick way to reduce the communication is to transmit single-word values rather than the double-precision values used. This simple modification would reduce the communication component to 4.7sec.

An improved parallel scheme was developed where data is permuted at each <el, j> step. During the first phase, each processor calculates as much as possible with local data as described in P1, where the *np* highest bits of *a* define the processor assignment. The next phase shuttles the data based on the next *np* lower bits, where a similar computation occurs. This shuttling of data continues until all the values are computed.

On the architecture under study, the claim is that data can be directly transmitted between many pairs of processors without contention. On a hypercube architecture, it would perform communication than P1, since each processor sends data to and receives data from all other processors, requiring multiple routing steps. In the non-congested model used here, processor p_i simply sends data to the destination processor p_k. The time to send *W* words using *P* processors as individual messages is:

$$CP2(W, P) = (60P + W)\mu\,\mathbf{sec}$$

This states that latency grows with the number of processors, but the transmission time is independent of the number of processors. The latency can be reduced by packing all data destined for the same processor into one packet, incurring the latency only once (instead of for each value transmitted). The size of the packet contains 2^{n-2*np} values.

The next version, P3, was developed to have both the computation and communication complexity scale as $O(m^2 n 2^{n-1}/P)$, where there are $P = 2^{np}$ processors, *n* tracks and *m* measurements. This algorithm scales with the number of processors, but only if $n > 3np$.

The implementation of P3 relies on the analysis of communication in P2, and manages to place all data sent from p_i to p_k into one message, incurring the message latency only once. The message size grows as the algorithm progresses, since *f*-values with index sets [0..*el*] by $[2^x..2^{x+np}-1]$ must be transmitted, and *el* is increasing.

The computation and communication times for P3 are calculated by the following equations:

$$TP3(n,m,np) = \frac{m(m-1)n2^{n-np-2}}{0.71 \cdot 62.5}\mu\,\text{sec}$$
$$CP3(n,m,np) = 120m \cdot 2^{np} + m(m-1) \cdot 2^{n-np}\mu\,\text{sec}$$

These expressions show that both the computation and communication costs are reduced as *P* increases. Substituting similar values, the computed values are $TP3(19, 20, 6) = 0.333$ sec and $CP3(19, 20, 6) = 3.267$ sec, which is better than any of the previous versions that perform communication. Doubling the number of processors reduces both costs, by as much as a factor of two.

The principle behind algorithm P3 relies on collecting *f*-values that match in *np* bits of the *a*-index and then combining them appropriately. This step is followed by another step, in which *f*-values that match in a different set of bits of their *a*-indices are brought together. This general principle can be extended, by reducing the time needed for the sorting operations that bring *f*-values together.

Two approaches to reducing this sort time have been investigated. The first involves splitting the available 2^{np} processors into two disjoint sets of processors, the "A" processors, and the "B" processors. "A" processors send data to appropriate "B" processors, and vice versa. This avoids having the same processors both send and receive data during one communication step. Analysis shows that this reduces the message latency term in the performance equation by a factor of 4, but leaves the transmission-time term unchanged. The second approach uses a two-digit radix sorting approach for the sort operation, accompanied by splitting the set of processors in two. This radix sort scheme improves performance when $np \geq n/2$, although processor efficiency is reduced, compared to that of algorithm P3. These two variations are P4 and P5.

In P4, each processor holds twice as much data, and participates in only half as many send and receive operations. Each processor sends data to only half as many processors as P3, yielding a reduction of a factor of 4 in latency over P3. The communication cost is updated to:

$$CP4(n,m,np) = 30m \cdot 2^{np} + m(m-1) \cdot 2^{n-np}\ \mu\,\mathbf{sec}$$

Version	TP(19, 20, np)			CP(19, 20, np)					
	np=6	np=7	np=8	np=6	np=7	np=8	np=9	np=10	np=11
0	2.25 (20 processors)			0					
1	0.438			9.37					
1.5, send floats	0.438			4.70					
3	0.333	0.164	0.083	3.267	1.863	1.393			
4				3.15	1.633	0.932	0.696	0.809	
5						1.585	0.817	0.447	0.271

Table 3. Theorized computation and communication times for the various parallelization schemes of the CR-JPDA algorithm and varying number of processors in a message-passing environment.

This gives $CP4(19, 20, 6) = 3.15$ sec, $CP4(19, 20, 7) = 1.633$ sec, and $CP4(19, 20, 8) = 0.932$ sec, which shows an almost linear speedup in communication time. Speedup falls off when

$$2^{2\cdot np} > \frac{(m-1)\cdot 2^n}{30}$$

so the improvement degrades when $np \geq 9$.

Algorithm P5 was developed to effectively use more than $2^{n/2}$ processors (for the problem size investigated here, this indicates performance gains using more than 1024 processors). This algorithm works by further breakdown of the a index used in accessing f, and interspersing parts of a around the j index, as f{A}{q1}{q2}[c1][j][c2][b] where $a = q1|q2|c1|c2|b$ (for complete details, see the full report). This change allows the proper elements of f to occupy contiguous memory locations, and thus be sent with one call to send(). The communication performance increases on machines with 2048 processors, and is described by:

$$CP5(n,m,p) = 60m\left[2^{\lfloor (np-1)/2 \rfloor} + 2^{\lceil (np-1)/2 \rceil}\right] + m(m-1)\cdot 2^{n-np+1}\ \mu\,\mathbf{sec}$$

This yields communication performance times of CP5(19, 20, 8) = 1.585 sec, CP5(19, 20, 9) = 0.817 sec, CP5(19, 20, 10) = 0.447 sec, CP5(19, 20, 11) = 0.271 sec, thus the runtime continues to decrease even with 2048 processors. Table 3 shows a comparison of all values computed.

4.9.3 Algorithm development methodology

The various algorithm variations explored, ultimately leading to algorithm P3, were all obtained by performing various transformations on the original algorithm. In principle, these transformations could be done using an automated system, as long as each transformation used is checked to see that it preserves the original meaning of the algorithm. The performance of each algorithm variant produced was evaluated, using the algorithm's statement to obtain estimates of the arithmetic computation, and communication cost (size and number of messages sent and received) at each parallel step, for the slowest processor.

The performance evaluations resulted in the conclusion that the communication bandwidth limited the parallel efficiency of each algorithm quite seriously. This conclusion suggested searching for algorithm variants that would reduce the total number of words transmitted by all processors during the computation. Communication latency proved fairly easy to manage, even when the number of messages increased linearly with the number of processors.

During the performance evaluation phases, techniques for improving the computation time component of total run-time were also explored. These techniques, which involve "blocking" inner loops, to remove unneeded tests, and to hold values that are being updated in registers until all updating operations are complete, were useful in assessing the speed of a purely sequential version of the JPDA algorithm, and would have some value in a parallel version of the algorithm. Assuming these techniques are used, a modest number of processors can divide the necessary work so that each processor's computation time is quite small (0.333 sec, for 64 processors on the running example). However, communication time is still dominant (3.267 sec for the same example).

It appears that a flexible approach to algorithm refinement should be adopted, one that allows the results of early estimates of performance or other algorithm characteristics to guide later choices of possible algorithm variant construction. It should be noted that a great deal of effort can be saved by performing all such explorations of algorithm variants at a very undetailed, possibly unexecutable level. Much effort is expended whenever an algorithm sketch is refined to a level of detail where it can actually execute, and produce useful results. If simulation of partial algorithms, and theoretical calculations of algorithm characteristics such as performance and correctness can be used, a small amount of programming effort can produce large increases in algorithm quality.

In the study herein reported, simulation of some parts of the memory hierarchy were used to predict cache performance. At the same time, the simulator was able to demonstrate that certain indexing transformations adopted to avoid cache conflict misses indeed preserved the intent of the original algorithm, for the cases tested. All the performance equations reported were derived theoretically, not measured. Furthermore, those equations that represent communication performance should be experimentally verified, to ensure that the hardware interconnection network conforms to the assumption that the time for a message to traverse the network is independent of other messages present in the network at the same time. This prediction could be verified by implementing and executing the current algorithm, or could be tested, using a careful experimental protocol. The assumption is critical to the choice of algorithm for the parallelization of the JPDA algorithm, and probably for others as well.

4.9.4 Conclusions about a message-passing CR-JPDA

Several task-parallel versions of the CR-JPDA algorithm have been derived which are suitable for use on massively parallel message-passing architectures, such as the IBM SP1 and SP2. These machines have many fast processing nodes, interconnected by a switch that exhibits no network contention, in theory, but has message-passing performance characterized by a several microsecond latency per message, and a time to receive one floating-point number that is some 30 times the time needed for one floating-point operation. These hardware characteristics produce parallel versions of the JPDA algorithm that are limited by the communication bandwidth, if the number of processing nodes to be used is much greater than m, the number of measurements.

The best algorithm among those investigated here depends on the problem size, and the number of processors used. When the number of processors $P = 2^{np}$ is less than m, a simple parallel algorithm, in which each j-task is performed by a separate processor, and idle processors compute task $j = 0$ redundantly, seems to be best. It yields run-time for the running example ($n = 19$, $m = 20$) of 2.25 secs, using 20 processors. For larger number of processors, this run-time is not improved until 128 processors are used, and algorithm P4 is employed. CP4(19,20,7) = 1.633 secs, CP4(19,20,8) = 0.932 secs, CP4(19,20,9) = 0.696 secs, so that algorithm P4 is best, when $np \leq n/2$. For larger values of np, algorithm P5 improves performance most: CP5(19,20,10) = 0.447 secs, CP5(19,20,11) = 0.271 secs. Algorithm P5 has only been outlined, so it is possible that its performance is not in fact as good as predicted here. Also, computation time for algorithms P4 and P5 have not been included in the comparison.

4.10 Conclusion of MTT Case Studies

While there have been plenty of artifacts produced in the case studies of multi-target tracking algorithms, the more import product is the methodologies that were used to choose what steps to take in developing the MTT artifacts. In the case of the Proteus development, the tactic defined earlier in this report was followed, and in the "Task Parallel" study, Wagner demonstrates how to predict parallel performance of an algorithm without necessarily going through the task of full development (with all the difficulties required).

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 uncovered problems where solutions were found, and often, the solutions were useful in improving their ancestral codes.

The work also showed 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. Even though the memoized ZB-JPDAF was not implemented as a parallel algorithm, clearly it is possible, and that the gains would provide an even larger gain over the CR-JPDAF. Another point to note is the relation between problem size and machine size. This was pointed out specifically in the MP task-parallel study, where if $P \leq m$, one algorithm should be used, but for larger values of P, alternate algorithms will perform better.

This section describes what is mostly now a manual process. In examining multiple development paths, better understanding leads to better tool support for future development of parallel applications.

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 outside efforts, most notably the Sisal implementation of the JPDA in the MITRE report. It is the belief that the strength and longevity of the Sisal JPDA implementation were unexpected to the authors, and even so, their implementation will still perform well on today's fastest computers. Their other implementations have limited applicability, and must be actively modified to remain useful.

More powerful optimization models must be developed. Current high-level parallel programming languages rely on functional (side-effect free) descriptions, limiting the developer to explore 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

Several activities occur in developing a tree of program versions. It 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 this particular example might be solved by consolidating routines in an additional file that both versions share, this is a primitive solution that is bound to be inadequate. 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 were 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. More on this point will be addressed in portable targets. Also, 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 Multi-lingual 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 multi-lingual 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 non-bonded 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.

5.5 Information Gathering Tools

Of course, a key result 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 support, so tools would be useful.

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, that is, which 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 semi-automatically 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.

Moreover, certain aspects of the functional style may be used to aid in the synthesis of programs from yet higher-level specifications (e.g., in the tensor-product formulation work in [Gupta, et al. 1995, Li, et al. 1996]). There is an intention to investigate, as part of Task III, the use of such synthesis techniques targeting MIMD machines which in addition take into account multilevel memory hierarchy (i.e., I/O and cache).

5.7 Portable Compilation Targets

As stated earlier, complex translation and optimization schemes take significant time to develop. This time is lengthened by having different output models (languages, or libraries) for different architectures. Blelloch has demonstrated, with the C+CVL target, that an architecture-independent compilation target ensures the longevity of the Nesl project [Blelloch, et al. 1993, Blelloch 1992]. 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

The development of high-performance parallel software depends upon systematic version development using prototyping. This style of development promotes the exploration of multiple solutions to a problem, where answers to key questions about the software can be found. This method of tree-based refinement works well with existing parallel programming paradigms, by allowing developers to either translate the high-level code into efficient programming languages, or by directly calling pieces of the high-level prototype. It also will work in conjunction with advances in parallel program design, such as templates and automatic code generators (from specifications), by shortening the path from high-level specification to the low-level executables.

Multiple related versions all derived from a single specification aids in targeting software to a variety of parallel architectures. Characteristics of the solutions can be tabulated and applied to new architectures as they appear, or new characteristics can be determined when the former no longer apply. By writing solutions in a high-level parallel programming language with clear concurrency semantics, automated performance tools, such as PAWS, can aid in quickly determining performance characteristics of a variety of solutions.

Experience with high-level prototypes gives developers essential insight to the algorithms to answer critical questions that affect the parallel implementation. By exploring solutions at a high-level, the developer is able to explore *what* the solution does separate from *how* it executes. The separation of these two unknowns not only gives the developer better insight into the high-level solution, but leads to better implementations as well.

High level program transformations and optimizations are critical to automatically achieving an efficient implementation. With programs written in high-level languages, the semantics of the language allow for powerful transformation operations that aid in achieving sophisticated parallel programs. Lower level languages have semantics that are both too constrained for efficient parallel execution and too detailed so that it is difficult to apply (and develop) sophisticated transformation techniques. The transformations in the Proteus translator [Prins and Palmer 1993] and those in the Nesl compiler [Blelloch and Sabot 1990] are excellent examples of sophisticated code generation that is difficult to duplicate in a lower level language.

Large benefits in software management and performance analysis could be gained with additional software development tools. It is evident from earlier reports that there were unforeseen problems in implementing MTT algorithms. High-level languages are just one facet of better development methods for parallel software. The ability to develop sophisticated automatic transformation and optimization tools allows developers to quickly write and run highly parallel code. This is aided by the use of tools such as PAWS to help developers understand characteristics of their code without having to run it directly on the parallel hardware (often hampered by limited access, job queues, security). In addition, tools to aid in managing a version-tree would help to maintain the multiple solutions developed. And, of course, any tools that help determine problematic areas in parallel software will be very useful. We maintain that any of these tasks will be simpler when development occurs in a high level language such as Proteus.

Continued development aiding low-level software development (such as templates) narrows the gap between prototyping and efficient parallel programs. Currently, a wide gap between Proteus and data-parallel execution must be bridged. This gap has been narrowed somewhat by CVL, which supports data-parallel execution. Templates will support additional styles of execution, eliminating the need for time-consuming development of low-level support libraries.

7. References

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

[Antonio 1994] John 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.

[Backus 1978] J. Backus, "Can Programming Be Liberated From The Von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. of the ACM*, 21(8):613-641, 1978.

[Bar-Shalom and Fortmann 1988] Y. Bar-Shalom and T.E. Fortmann, *Tracking and Data Association*, Academic Press, 1988.

[Bar-Shalom and Li 1993] Yaakov Bar-Shalom and Xiao-Rong Li, *Estimation and tracking : principles, techniques, and software*, Artech House, 1993.

[Barrett, et al. 1994] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine and Henk van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994.

[Blelloch, et al. 1993] 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 University, Feb., 1993.

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

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

[Blelloch 1996] Guy E. Blelloch, "Programming Parallel Algorithms," *CACM*, 39(3) 1996.

[Cann 1993] David C. Cann, "SISAL 1.2: A Brief Introduction and Tutorial," Technical Report, Lawrence Livermore National Laboratory, 1993. Available via ftp.

[Chandy 1995] K. Mani Chandy, "Concurrent Program Archetypes," Electronic Report http://www.etext.caltech.edu/Papers2/ArchetypeOverview/ArchPaper.html, Computer Science, 256-80, California Institute of Technology, Pasadena, CA 91125, 1995.

[Culler, et al. 1993] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[Fox, et al. 1993] G. C. Fox, S. Ranka and Parallel Compiler Runtime Consortium, "Common Runtime Support for High-Performance Parallel Languages," Draft Technical Report PCRC-001, Northeast Parallel Architectures Center, Syracuse University, New York, July, 1993.

[Games, et al. 1994] Richard A. Games, John D. Ramsdell and Joseph J. Rushanan, "Techniques for Real-Time Parallel Processing: Sensor Processing Case Studies," Technical Report MTR 93B0000186, MITRE, April, 1994.

[Geguelin, et al. 1991] A. Geguelin, J. Dongarra, A. Geist, R. Manchek and V. Sunderam, "A User's Guide to PVM: Parallel Virtual Machine," Technical Report, Oak Ridge National Laboratory, July, 1991.

[Gupta, et al. 1995] S. Gupta, Z. Li and J.H. Reif, "Generating Efficient Programs for Two-Level Memories from Tensor-Products," *Proceedings of the 7th IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, Washington DC, p. 510-513, 1995.

[Hull, et al. 1995] D. Hull, W. Feng and J. W. S. Liu, "Enhancing the performance and dependability of hard real-time systems," *IEEE Computer Performance and Dependability Symposium*, Erlangen, Germany, p. 174--182, 1995.

[Karp and Zhang 1988] Richard M. Karp and Yanjun Zhang, "A Randomized Parallel Branch-and-Bound Procedure," *Proc of the 20th Annual Symposium on Theory of Computing*, p. 290-300, 1988.

[Levin and Nyland 1993] Gary Levin and Lars Nyland, "An Introduction to Proteus, Version 0.9," Technical Report TR95-025, UNC-CH, August, 1993. 63 pages.

[Li, et al. 1996] Z. Li, J. H. Reif and S. K. S. Gupta, "Synthesizing Efficient Out-of-Core Programs for Block Recursive Algorithms using Block-Cyclic Data Distributions," *International Conference on Parallel Processing*, 1996.

[Mills 1995] Peter Mills, "An Efficient Data-Parallel Algorithm for Data Association in Multitarget Tracking," Draft Technical Report, Duke University, June, 1995.

[MPI Forum 1993a] MPI Forum, "Document for a Standard Message-Passing Interface," Technical Report, U. of Tennessee, 1993a.

[MPI Forum 1993b] MPI Forum, "MPI: A Message Passing Interface," *Supercomputing '93*, 1993b.

[Pease, et al. 1991] Daniel Pease, Arif Ghafoor, Ishfaq Ahmad, David L. Andrews, Kamal Foudil-Bey, Thomas E. Karpinski, Mohammad A. Mikki and Mohamed Zerrouki, "PAWS: A performance evaluation tool for parallel systems," *IEEE Computer*, 24:18-29, 1991.

[Prins and Palmer 1993] Jan Prins and Daniel Palmer, "Transforming High-Level Data-Parallel Programs into Vector Operations," *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, p. 119-128, 1993.

[Prins and Goldberg 1995] Jan F. Prins and Allen T. Goldberg, "Parallel Processing and Software Development," Deliverable A004 Report, UNC and Kestrel Institute, May 19, 1995.

[Roecker 1994] J. A. Roecker, "A Class of Near Optimal JPDA Algorithms," *IEEE Trans. on Aerospace and Electronic Systems*, 30(2):504-510, 1994.

[Roecker and Phillis 1993] J.A. Roecker and G. L. Phillis, "Suboptimal joint probabilistic data association," *IEEE Trans on Aerospace and Electronic systems*, April:510-517, 1993.

[Rogers and Pingali 1989] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference," *Conference on Programming Language Design and Implementation*, p. 69--80, 1989.

[Tetewsky 1994] Avram K. Tetewsky, GUI Development for Real-Time Applications, Dr. Dobbs Journal, June, 1994.

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

[Wagner 1995] Robert A. Wagner, "Task Parallel Implementation of the JPDA Algorithm," Technical Report, Department of Computer Science, Duke University, Durham, NC 27708-0129, June, 1995.

[Zhou and Bose 1993] B. Zhou and N. K. Bose, "Multitarget Tracking in Clutter: Fast Algorithms for Data Association," *IEEE Trans. on Aerospace and Electronic Systems*, 29(2):352--363, 1993.

[Zhou and Bose 1995] B. Zhou and N. K. Bose, "An Efficient Algorithm for Data Association in Multitarget Tracking," *IEEE Trans. on Aerospace and Electronic Systems*, 31(1):458--468, 1995. [Alpern, et al. 1993] B. Alpern, L. Carter and J. Ferrante, "Modeling Parallel Computers as Memory Hierarchies," *Workshop for Portability and Performance for Parallel Processing*, Southampton, England, 1993.