A Refinement Methodology for Developing Data-Parallel Applications*

Lars Nyland,[1] Jan Prins,[1] Allen Goldberg,[2] Peter Mills,[3] John Reif[3] and Robert Wagner[3]

[1] Dept. of Computer Science, University of North Carolina
Chapel Hill, NC 27599-3175 USA
[2] Kestrel Institute, 3260 Hillview Ave., Palo Alto, CA USA
[3] Dept. of Computer Science, Duke University, Durham, NC 27708 USA

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.

We consider a refinement methodology 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.

* This work supported by Rome Laboratory under contract #F30602-94-C-0037.

- **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.

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 [3, 5, 4, 9]. 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.

The target architecture will have an influence on many aspects of the design, and here we classify parallel architectures into four classes – two shared memory classes and two distributed memory classes. The effects of each of the architectural classes is considered with regard to its impact on application development.

- **UMA shared memory.** In this class, global memory has uniform memory access (UMA) time. This challenging requirement is currently met only by cacheless machines with memory bandwidth matched to processor speed. Examples: Cray T90, NEC SX-4, and the Tera Computer.
- **NUMA shared memory.** A non-uniform memory access architecture has hierarchically organized memory, often viewed as multiple levels of cache. Examples are SGI Challenge machines and the Convex Exemplar.
- **Small-grain distributed memory.** No shared memory abstraction is provided in hardware, but fine-grain message passing and low-latency synchronization are supported. We've labeled this class SIMD, since it describes machines such as the MasPar MP-2, but in also describes MIMD machines with fine-grain support, such as the Cray T3D.
- **Large-grain distributed memory.** Message-passing (MP) machines where large messages must be used to achieve reasonable performance of processor interconnect. Examples are the Intel Paragon and IBM SP/2.

2 Design Refinement Methodology

Effective software development involves prototyping at several different design levels to discover the characteristics of a problem. Figure 1 describes the stages along the refinement paths from high-level designs to architecture-specific implementations. A description of each of the stages follows.

2.1 Problem Definition and Validation.

At the *specification* stage (top of figure 1), we seek an initial description of the problem and validation of the problem against functional requirements. The focus is on determining the feasibility and practicality of obtaining a high-performance algorithmic solution for a succinctly-described problem.

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 both necessary, because the algorithms are often computationally intensive, and feasible, because of their succinct description. Asymptotic analysis, prototyping, and exploration of algorithm variants can all be performed quickly at this level.

One set of measures for selecting parallel algorithms are *work* and *step* complexities. The *work* complexity is a measure of all operations performed, while the *step* complexity measures the number of parallel steps (minimum number of sequential steps). While these measures are not completely realistic, we seek to separate concerns and formulate a tractable, staged analysis methodology. Finding a work-efficient algorithms is the goal, even though they may lead to irregular solutions with data-dependent behavior. For large problems on fixed size machines, work-efficient algorithms give the best performance.

Analysis for parallel architectures. The analysis techniques at this stage are based on medium-level parallel computing cost models such as LogP [6], BSP [12], and HMM [1] that can be used to estimate communication and memory performance. The model parameters of the algorithm are obtained analytically (when possible, or by instrumenting a prototype of the algorithm). One of the 4 architectural classes must be supplied to complete the analysis.

2.2 Design Refinement

The refinement of programs, shown in figure 1 between the dashed lines, occurs in a high-level design notation. We've chosen a data-parallel subset of Proteus [9] as our design notation. An initial implementation of specification provides a starting point that provides a foundation for correctness, analysis, information-conveyance and measurement purposes. At a high level, programming for data-parallel execution is only slightly more complicated than programming for serial execution in a language where collections of data are fully supported.

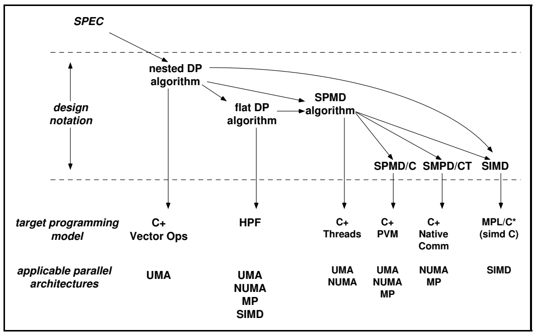

Fig. 1. Refinement and translation steps in the development of data-parallel applications

The different refinement paths are:

- Nested data-parallel design refined to flat data-parallel design by converting nested iterators and nested sequences to loops with integer iterators and rectangular arrays. Flat data parallel programs are suitably expressed using languages such as HPF, Fortran90, C* and MPL.
- Nested (or flat) data-parallel refined to SPMD. SPMD programs have multiple threads of control using a single shared-memory, where decomposition is under program control, varying from simple memory-decomposition models to complex, adaptive, load-balancing methods.
- SPMD refined to SPMD/C. This model adds explicit communications to an SPMD program, eliminating the shared-memory constructs.
- Refining to an SIMD Model. SIMD machines can be viewed as fine-grain message-passing machines, or as vector architectures. Refinement to an SIMD model can begin with either the nested or flat data-parallel versions, or it can be derived as similar to other message-passing versions.

2.3 Translation to Target Programming Models

Translation of high-level data-parallel designs can target a variety languages as shown by the vertical arrows in figure 1. Different designs are matched to different languages and different parallel architectures. Translation from the design notation may be manual or automatic; the automatic translation of nested data-parallel programs to vector models is an active area of research [11, 4].

3 Case Studies: Multitarget Tracking

In this section, we turn our attention to an example, demonstrating the 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, we will develop a tree of refined algorithms and show the need for additional automated support.

Multitarget tracking (MTT) can be described simply: a set of N targets are being tracked when new location data arrives as a set of M positions. Prediction models compute the expected locations of the tracked targets, but the new data may not coincide with the estimated positions. The problem is to find the joint probability that a target t, $1 \leq t \leq N$, is represented by a measurement j, $1 \leq j \leq M$. Any algorithm that computes this result falls in the category of multitarget tracking algorithms that we are studying.

3.1 Solutions to the Multitarget Tracking Problem

We have explored two published algorithms that solve the multitarget tracking problem. The first is the *column-recursive joint probability data association* (CR-JPDA) algorithm, since it has been the subject of parallel implementation studies [8, 2]. Zhou and Bose also present a parallel MTT algorithm, the *tree-search joint probability data association filter* (ZB-JPDAF) algorithm, that performs much worse than the CR-JPDA in the worst case, but has claims of better performance in average and highly likely cases [14].

The CR-JPDA is a specific association strategy that uses a weighted average of returns. The *column-recursive* algorithm has work complexity of $O(NM^2 2^N)$, but this is improved by a factor of $N(M+1)$ over the computation of all permanents of a matrix, the direct method of multitarget tracking. The algorithm is a dynamic programming strategy, relying on solutions for sub-problems to compute the answers to larger problems.

3.2 Development and Refinement

Figure 2 shows our development hierarchy of MTT solutions. The development of our multitarget tracking algorithms begins at the root with a specification of the problem to be solved.

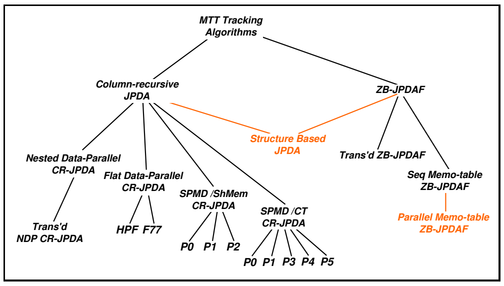

Fig. 2. Our development tree of the MTT algorithms. The grayed instances have yet to be developed.

Below the specification are the two algorithms under consideration, the CR-JPDA and the ZB-JPDAF. These two implementations are written in Proteus, and serve the purpose of achieving a baseline implementation with no initial concern for parallelism. The two implementations are concise, requiring about 40 lines of Proteus each. They were validated against one another prior to further development (details in [10]).

The first descendent of the CR-JPDA targets parallelism, it is a nested data-parallel implementation in Proteus, which is suitable for automatic 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.

Additional studies were performed, as outlined by the design refinements described in section 2. A flat data-parallel version of the CR-JPDA was developed and translated manually 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 alternate memory decompositions that improved performance. And finally, a paper study was performed to estimate the performance of a variety of message-passing implementations of the CR-JPDA [13].

3.3 Results of the Implementation Study

A brief description of the variants of multitarget tracking programs follows. For a complete description, including code and analysis, see [10].

Nested Data-parallel Implementations. Prototyping the CR-JPDA and the ZB-JPDAF with a high-level language yielded quick development and concise descriptions on which further implementations can be based. Not only were the two prototypes validated against one another, but their work complexity was examined by running both on a variety of input data. As suggested by the authors, a *memoized* implementation of the ZB-JPDAF drastically improved the performance. At this high level, it is possible to recognize that the CR-JPDA is a dynamic programming variation of the ZB-JPDAF, where the tabular nature of the implementation removes all variance from the execution time. The CR-JDPA always has the same performance, where the ZB-JPDAF is highly sensitive due to data-dependent branching factors.

Flat Data-parallel Implementations. The next results were from conversion to flat data-parallel implementations. The (manual) transformation removes nested structures, such as nested iterators, nested function calls, and nested summations, turning them into loops populated with assignment statements. The nested data structures are also made regular (same length at in each dimension), where space is wasted to provide regularity. These versions, written in HPF and Fortran77, gave us the opportunity to explore regular data decompositions for parallel execution. Unfortunately, the work in the CR-JPDA is unevenly distributed over the data, so regular data decompositions will never balance the work evenly, necessitating further refinement for higher performance.
SPMD Implementations. In an SPMD model, memory decomposition is under program control, as is work decomposition. Three versions were explored at this level: one similar to the flat data-parallel version (as a benchmark); an attempt to balance memory requests with a modified block-cyclic layout; and uneven data decomposition to balance the work. Both of the attempts to improve performance were successful, with good load-balancing leading to good scaling behavior.
Message-passing Studies. Our final study looked at the CR-JPDA as it might be implemented on a message-passing architecture (described fully in [13]). A simple model was developed to describe the computation and communication costs of different decompositions of the CR-JPDA. The initial implementation was based on the simple observation that M processors could be used to compute results with almost no communication. Since this provides only limited parallelism, several more implementations were studied in an attempt to use more processors (64-2048). It took substantial algorithm enhancement and complex message coordination to achieve any solutions that could outperform the initial solution.
Summary of Implementations. The refinement strategy used to develop the MTT algorithms cleanly separates development issues. In the prototypes, the primary concern was one of correctness and agreement between the two algorithms. The flat data-parallel implementations explored regular memory decompositions. The SPMD version focused on load-balancing the work and data with irregular decompositions. Finally, the message-passing study looked at several variants of the algorithm in an effort to achieve a scalable program on a given architecture. Developing correct implementations early and quickly allowed the later developments to focus mainly on performance issues.

4 Tools to support parallel software development

The work performed in the case study used a collection of software tools that only minimally support the task. What follows is a suggested set of tools to support the design methodology described here.

– **Parallel programming language for prototyping.** The brevity, clarity, and analyzability of our Proteus implementations along with other implementations (Sisal implementation in [8]) make a strong argument for using high-level languages for prototyping. High-level programming languages help the developer gain intuition and insight about the inner workings of complex algorithms prior to exploring optimizations for high-performance.
– **Repository version manager.** Developing a tree of programs is not a one-way process, rather there is constant flow of information among program instances up and down the version tree. 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 the versions in this report were managed manually.
– **Powerful transformation tools.** The translation and optimization strategies developed by us, the Scandal group at CMU and the Sisal group at LLNL permit the developer to express an parallel solution in a concise, clear notation that can be translated to code that runs competitively with hand-written, hard-to-understand, low-level implementations.
– **Multi-lingual programs.** If prototypes are refined into reliable applications, often performance can be increased with multi-lingual capabilities. In one instance, a small part might be much more efficient if rewritten in a low-level language. Rather than commit the entire prototype to the lower-level language, support for multi-lingual programs could drastically reduce development time. As an alternative, consider the integration of a prototype into a much larger existing system,

- **Information gathering tools.** The main goal of developing parallel programs is high performance. As such, it is important to know where the program is performing and consuming resources as expected and where it is not. Performance analysis tools that provide information about highly optimized programs are key to achieving this goal.
- **Portable compilation targets.** Architecture-independent compilation targets ensure the portability and longevity of parallel applications, as is demonstrated by the Nesl project [4], the Sisal project [5], and the Fortran-M project [7].

5 Conclusions

We have proposed a tree-based refinement strategy for developing data-parallel applications. As development progresses down the tree, algorithms are specialized, perhaps to meet architectural and performance considerations. Branches represent different algorithms or different specializations. To accommodate new target architectures, we add new branches; to accommodate specification changes, we retrace the refinements steps through the tree. The application is represented by the tree of refined programs, not one particular refined version in the tree.

We have used the refinement methodology to develop parallel multitarget tracking algorithms. The separation of concerns allowed us first to concentrate on the correctness and high-level performance of 2 competing algorithms (the CR-JPDA and the ZB-JPDAF). Further refinement allowed the explorations of regular and irregular decomposition to achieve a balanced work-load. Additional models were used to determine the costs of a family of message-passing implementations, and what was required to achieve scalable performance over a wide range of parallelism.

References

1. B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proc. Foundations of Computer Science*, 1990.
2. 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.
3. 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.
4. Guy E. Blelloch. Programming parallel algorithms. *CACM*, 39(3), Mar. 1996.
5. David C. Cann. SISAL 1.2: A brief introduction and tutorial. Technical report, Lawrence Livermore National Laboratory, 1993.
6. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. Symposium on Principles and Practice of Parallel Programming*, 1993.
7. Ian Foster. *Designing and building parallel programs*. Addison Wesley, 1995.
8. 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.
9. Allen Goldberg, Peter Mills, Lars Nyland, Jan Prins, John Reif, and James Riely. Specification and development of parallel algorithms with the proteus system. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms*. American Mathematical Society, 1994.
10. Lars S. Nyland, Jan F. Prins, Allen T. Goldberg, Peter H. Mills, John H. Reif, and Robert A. Wagner. A design methodology for data-parallel applications. Technical report, Univ. of N. Carolina, 1995. Available as http://www.cs.unc.edu/Research/aipdesign.
11. Jan Prins and Daniel Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of Principles and Practice of Parallel Programming*, pages 119–128, San Diego, CA, 1993.
12. Leslie G Valiant. A bridging model for parallel computation. *CACM*, 33(8):103, August 1990.
13. Robert A. Wagner. Task parallel implementation of the jpda algorithm. Technical report, Department of Computer Science, Duke University, Durham, NC 27708-0129, June 1995.
14. 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.

This article was processed using the LaTeX macro package with LLNCS style