The PARSE Programming Paradigm. Part I: Software Development Methodology. Part II: Software Development Support Tools

T. L. Casavant
Purdue University

Henry G. Dietz
Purdue University

P. C.-Y. Sheu
Purdue University

H. J. Siegel
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/ecetr

https://docs.lib.purdue.edu/ecetr/566

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
The PARSE Programming Paradigm

Part I: Software Development Methodology
Part II: Software Development Support Tools

The PARSE Group

TR-EE 87-22
June 1987

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
The programming methodology of PARSE (parallel software environment), a software environment being developed for reconfigurable non-shared memory parallel computers, is described. This environment will consist of an integrated collection of language interfaces, automatic and semi-automatic debugging and analysis tools, and operating system — all of which are made more flexible by the use of a knowledge-based implementation for the tools that make up PARSE.

The programming paradigm supports the user freely choosing among three basic approaches/abstractions for programming a parallel machine: logic-based descriptive, sequential-control procedural, and parallel-control procedural programming. All of these result in efficient parallel execution.

The current work discusses the methodology underlying PARSE, whereas the companion paper, "The PARSE Programming Paradigm — II: Software Development Support Tools," details each of the component tools.
1. Introduction

Production of high quality software is a critical bottleneck in the efficient utilization of computer systems today. Production of software for single processor systems presents the problems of:

- making efficient use of programmer time,
- producing software which is efficient, and
- producing software which is reliable.

In parallel computing systems, each of the above problems exists; in addition, however, there are problems concerning:

- partitioning problems for parallel execution,
- debugging/preventing deadlock and race conditions (debugging asynchronous execution in general), and
- production of efficient, "high performance" solutions (also predicting performance of solutions).

These problems will be addressed by providing, for the class of reconfigurable non-shared memory parallel machines, an integrated collection of language interfaces, debugging and analysis tools, and operating system\(^1\): PARSE (parallel software environment).

The PARSE model is described from the user's point of view. An implementation of PARSE is currently under development.

In PARSE, a choice of language interfaces provides a programmer with the ability to select the most appropriate or natural specification of a problem or a solution to a problem. The user may chose to employ any one or any combination of:

- Descriptive language.
- Sequential procedural language.
- Parallel procedural language.

\(^1\) The operating system is not discussed in the current work.
In any case, efficient, highly-reliable, parallel code results. This is achieved using PARSE's knowledge-based software tools to automatically or semi-automatically create, modify, or improve a problem's solution or to optimize its implementation. The knowledge base serves both to enhance the system's abilities and to make the implementation of PARSE more flexible and general.

Problems whose solutions are most naturally stated in descriptive form can be presented to PARSE in that form — without a detailed procedural specification. Using knowledge-based transformation, PARSE provides the potential for efficient parallel execution of such descriptions by compiling logic-based descriptions into re-uses of known procedural solutions to logic specifications of sub-problems. These procedures may be either sequentially control-structured, in which case other portions of the system automatically transform it into a parallel form, or they may be parallel control-structured.

The programmer may also choose to express an algorithm directly in a sequential procedural notation. In sequential code, data access rights represent the stores/fetches that a region of code might make, hence, they represent key constraints on parallel execution of the code. Automatic and semi-automatic tools collect needed information about data access rights, use this information to parallelize the code, and use knowledge of target machine characteristics to aid the programmer in improving the program.

Where a parallel procedural solution has been conceived, semi-automatic debugging and analysis tools may be employed to predict performance and to reproduce sequences of pseudo-random asynchronous events. Thus, the instrumentation of hardware for debugging parallel control is avoided by utilizing PARSE to simulate a virtual machine in software.

The current work is intended to present a user view of the PARSE paradigm and to motivate its structure. A companion paper, entitled "The PARSE
Programming Paradigm — II: Software Development Support Tools,” provides details of the operation of each of the component tools.

Section 2 relates previous research to the PARSE system. The relationships between the various tools that constitute PARSE are described in section 3. Section 4 provides an overview of the use of knowledge within the PARSE system. Brief descriptions of the individual tools, again from the point of view of how they interact, are given in sections 5 through 8. Section 9 describes the current status of PARSE.

2. Background & Overview of PARSE

A programming environment is designed to simplify the task of program development; what constitutes a programming environment depends on the designer’s view of the key difficulties in the software development process. One group of environments is tailored to minimize the effort in developing large software systems by extending concepts of modular program design. Examples of systems motivated by this principle include Cedar [Tie84], Mesa [Swe85], Jasmine [MaW86], and Starlite [CoA86]. Additional benefits have been gained by applying knowledge-based program transformation (e.g., PDS [Che84], CHI [SmK85]) or by providing intelligent programmer assistance as in Programmer's Apprentice [Wat81]. The primary targets for these environments are single processor systems and networked work-stations.

A second group of environments are intended to simplify development of parallel programs. The main objective of these systems is to enhance parallelism in solutions as opposed to improving modularity. PTOOL [AlB86] is an environment designed to aid the programmer in restructuring sequential programs into parallel equivalents. This is accomplished by providing the programmer with information distilled from sophisticated global flow and dependence
analyses of conventional FORTRAN code. Some systems, such as Poker [SnS86] and Pisces [Pra85], have been constructed based on a small set of programming language and operating system primitives designed to facilitate portability and experimentation with language design. More complex systems which augment a higher level parallel programming language with tools to aid the programmer in visualizing program behavior have also been proposed. Among these are PIE [SeR85] and an environment for CSP developed at Tektronix [DeS86]. Both environments support prototyping and graphical visualization through multiple windows which illustrate various attributes of program behavior.

PARSE is a very large environment; although most of PARSE belongs to the second group of environments described, some portions belong to the first group. Some of the tools in PARSE allow the user to ignore parallelism (via automatic parallelization) and use knowledge to simplify software development. Other tools within the system aid the user in improving the performance, efficiency, and reliability of parallel programs.

PARSE enables the user to express problem solutions in descriptive, sequential procedural, or parallel procedural form, thus allowing the user to choose the most appropriate abstraction for the problem under consideration. This flexibility is a distinguishing feature of PARSE.

Many of the individual components of PARSE have been previously investigated in prototype form. XPC (explicitly parallel C) follows the concepts and structure of Parallel C [KuS85], a language designed to efficiently express both SIMD- and MIMD-style parallel control. XPAT (explicitly parallel algorithm analysis tool), a tool for design and implementation of parallel programs, is a modification and extension of a tool for analysis of distributed scheduling algorithms — DSSAP (distributed scheduling simulation and analysis package) [Cas86], which is based on a CFA (communicating finite automata) model
RC (refined C) is sequential language, based on ANSI C, which allows the programmer to explicitly state data access rights, potentially resulting in perfect flow analysis for parallelization. Using recent advances in automatic parallelization technology, RC compilers have been developed to convert sequential procedural code into safe and efficient parallel procedural code for large and small grain shared memory MIMD machines [DiK86]; the RC compiler in PARSE represents an extension of this technology to non-shared memory machines. This technology will also be employed to produce CR (C Reflex), a tool which will accept RC code, analyze it, and help the user to modify it for enhanced parallelism. CP (C Prefine), which provides for automatic parallelization of conventional C code by converting it into RC code, has been prototyped using a simplified flow analysis technique.

In addition to providing programmers with support for procedural programming, PARSE provides KBLP (knowledge-based logic programming). KBLP is a logic programming system which associates efficient procedural implementations with logic descriptions and matches user logic programs to these descriptions to obtain the implementations. The KBLP environment also combines three complementary technologies to help programmers write logic programs: logic programming, object-oriented design, and program transformation. First, the logic programming language PROLOG [War77] is augmented by the concepts of objects and classes [Zan84, She86]. Second, programmers are provided with a rich collection of generic objects and generic procedures, and the reusability of these modules is in many ways automated [She86]. Third, in contrast to most existing approaches that explore the parallelism in logic programs through syntactic transformation (e.g., [TuM86, LiW86]), KBLP transforms a descriptive logic program semantically to its procedural form so that more
efficient parallel execution can be obtained.

A discussion of the various user interfaces to PARSE, and scenarios of their uses, appear in the following section. Later sections describe the principles underlying the individual components of the PARSE system.

3. User Interfaces

In this section, the interrelationships among the components of PARSE are presented through a description of the system from the user's point of view and several brief scenarios of its operation.

Figure 1 is a graph illustrating PARSE in which the nodes represent the forms taken in transforming a problem from its initial specification into a parallel implementation of its solution. The arcs, therefore, represent transformations between forms, whether the transformation is performed automatically, semi-automatically, or manually. To simplify the graph, we have shown PARSE's interrelations as a strict hierarchy, although many cycles are possible using manual transformations.

The dotted lines shown in Figure 1 partition the logical structure of PARSE into three categories of abstraction used to express problem solutions: descriptive, sequential procedural, and parallel procedural. Each of these categories is now addressed in order.
3.1. Descriptions

In solving any problem, it is first necessary for the programmer to form an intuitive, informal understanding of the problem. This description of the problem is the starting point marked as Problem. From this “specification” the user may manually derive an abstract Sequential algorithm (a procedure using
sequential control flow, possibly with parallelism expressed using data access rights) or a Parallel algorithm (a procedure using explicitly parallel control constructs); alternatively, the user may create a precise logic-based description of the solution to the problem, expressed as KBLP (Knowledge-Based Logic Program) code.

KBLP can be considered to be a Prolog-like logic-based language, however, its execution strategy is radically different. Instead of generating the classic "full-width search" for a solution to the constraints expressed in a logic program, portions of the logic program are matched to pre-defined pairs of logic specifications and efficient procedural solutions. These procedures may be implemented either as sequential-control code (RC) to be optimized and parallelized by the RC compiler or as parallel-control code (XPC code).

3.2. Sequential Procedures

If an abstract Sequential algorithm has been designed, it may be written as either C code or RC code. C is a well-known systems programming language which has gained wide acceptance in many application domains. Unfortunately, C is an extremely difficult language to analyze for automatic parallelization (due to extensive use of pointers, recursion, and separate compilation). This expensive (yet imperfect) analysis is performed by CP. However, the result is RC code.

RC code is sequentially control-structured code, but is annotated with data access information which enables rapid, accurate, and precise parallelization analysis. The annotations used are similar to those encouraged for software engineering reasons: writing, maintaining, and improving RC code is easy for a C programmer.
Further, because these data access rights clearly imply which operations may be executed in parallel, RC code may be viewed as a "fail-safe" notation for expressing parallel computations — a technique which does not allow the programmer to write code which would result in a race or deadlock. Many "parallel algorithms" are quite naturally expressed using data access constraints within "sequential" RC code.

The RC compiler generates efficient (parallel) XPC code. CR accepts RC code, analyzes it, and helps the user to modify it for enhanced parallelism.

3.3. Parallel Procedures

For solutions to problems which consist of abstract Parallel algorithms, XPC code may be used to directly express parallel programs. Thus, the arc in Figure 1 originating at Parallel algorithm and terminating at XPC code represents a manual transformation. In order to provide the user with information useful in refining and tuning parallel algorithms, an alternative path is provided from Parallel algorithm to XPC code. The form of the specification required for this path is based on the formal CFA model. An automated analysis tool — XPAT, represented by the self-loop edge on CFA model, operates on a CFA description of the algorithm to provide the user with information regarding the performance and efficiency characteristics of the solution. (Performance characteristics are those which are related to the objectives of the algorithm(s); efficiency is objective-independent.) In order to provide this analysis, certain additional information must be provided to XPAT which describes minimal system dependent characteristics and algorithm performance objectives. More information regarding the function and structure of XPAT is in section 5. The edge from CFA model to XPC code is implemented by an automatic transformation tool — PREPAR (prepare for parallel compilation),
which converts the algorithm specifications from a CFA model into XPC code.

Finally, the transformation of code from RC code or KBLP description to XPC Code is accomplished through automatic translation. These transformations have been previously described (i.e. knowledge-based compilation of KBLP code and RC compiler).

Table 1 summarizes the above section, giving brief descriptions of the problem/program representations used within PARSE (the nodes of Figure 1) and the software tools which constitute PARSE.
<table>
<thead>
<tr>
<th>Representation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
<td>unrestricted computational problem</td>
</tr>
<tr>
<td>KBLP description</td>
<td>a problem or solution specified as a knowledge-based logic program</td>
</tr>
<tr>
<td>Sequential algorithm</td>
<td>unrestricted algorithm using sequential control, possibly parallel in terms of data access constraints</td>
</tr>
<tr>
<td>Parallel algorithm</td>
<td>unrestricted algorithm using parallel control</td>
</tr>
<tr>
<td>CFA model</td>
<td>specification of a communicating finite automata model</td>
</tr>
<tr>
<td>C code</td>
<td>code written in the C programming language</td>
</tr>
<tr>
<td>RC code</td>
<td>code written in refined C; C extended to permit annotation with explicit data access rights</td>
</tr>
<tr>
<td>XPC code</td>
<td>code written in explicitly parallel C; C extended to permit explicit parallel control and data layout</td>
</tr>
<tr>
<td>Code module</td>
<td>a module of target-machine object code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBLP compiler</td>
<td>transforms logic programs to match the logic descriptions of reusable procedures, compiling the program into efficient RC or XPC code consisting mostly of procedure reuses</td>
</tr>
<tr>
<td>CP</td>
<td>C prefine (preprocessor for refinement); uses sophisticated analysis techniques to obtain data access information from C code, resulting in RC code directly stating the data access rights</td>
</tr>
<tr>
<td>RC compiler</td>
<td>uses both loop and irregular code parallelization techniques to convert RC code into efficient (parallel) XPC code</td>
</tr>
<tr>
<td>CR</td>
<td>C reflex (refined language expert); analyses RC code and helps the programmer to improve it</td>
</tr>
<tr>
<td>XPAT</td>
<td>explicitly parallel algorithm analysis tool; used to design and debug parallel algorithms expressed using CFA models</td>
</tr>
<tr>
<td>PREPAR</td>
<td>prepare for parallel compilation; converts a CFA model into an XPC program</td>
</tr>
<tr>
<td>XPC compiler</td>
<td>converts XPC code into Code modules</td>
</tr>
</tbody>
</table>

Table 1. Representations & Tools within PARSE.
3.4. Scenarios

The following scenarios illustrate some of the ways in which programs may be developed using the PARSE tools:

- **Applications involving complex basic operations.** Many algorithms (procedures) in support of applications like image understanding have been developed, further, the conceptual tasks are easily abstracted at a very high level. One might naturally choose, therefore, to specify an image understanding system using the logic notation of KBLP. KBLP would rearrange and match the logic structure to the logic descriptions associated with support algorithms, and would therefore generate very efficient code consisting mostly of invocations of these procedures. If RC coded procedures were used, the RC compiler would then convert them into XPC code. Finally, the XPC compiler would generate efficient executable code.

- **Parallelizing an existing application.** Many useful programs have been written in C, including virtually all software designed to run under UNIX and the tools of PARSE itself (that have been developed thus far). To develop an efficient parallel version of a C program, the programmer would first use CP to convert the program into RC code. More than likely, ambiguities would surface in the analysis of the C code. The programmer would then use CR to help him improve the RC code by removing ambiguities that would have caused loss of useful parallelism. The RC compiler, followed by the XPC compiler, would then convert the RC code into efficient executable code.

- **Expressing parallelism using data access specifications.** Most speedup-oriented “parallel” algorithms are easily expressed using sequential control constructs and data access right specifications in RC. Unlike other expressions of parallelism, however, the RC compiler insures that parallelism expressed in this way cannot result in a race or deadlock. Perhaps more importantly, it can be debugged using conventional techniques (such as insertion of debugging output statements) and debugging one scheduling of the program insures that all schedulings are debugged. It can also be improved using CR. The RC compiler, followed by the XPC compiler, would compile the RC code into efficient executable code.

- **Expressing parallelism using parallel control.** Algorithms which embody explicitly parallel control (particularly those involving real time or using non-deterministic synchronization/communication structures) may be expressed using either XPC code or a CFA model. For critical code, a CFA model would be used, so that XPAT could predict performance and aid in design improvement and debugging; the final version of the CFA model would then be converted into XPC code by PREPAR. Less critical code could be written in XPC directly. The XPC code would be compiled into efficient executable code by the XPC compiler.
4. Knowledge Use within PARSE

As has been indicated, PARSE is not limited to a single machine or application domain. To achieve this generality, the types of knowledge which are used by the tools within PARSE have been abstracted; therefore, this knowledge base can be isolated from, and shared among, the tools. PARSE's knowledge base incorporates knowledge of the following broad types:

- **User input.** This knowledge is in the form of interactive responses to queries for additional information. It is typically user-provided knowledge to be used only in processing the program currently under development.

- **Application dependent knowledge.** This is knowledge specific to a problem or application domain (e.g. circuit simulation) which can be used to aid in selection of an implementation or to improve the performance of a program at execution time.

- **Generic functions.** These are abstracted procedures that can be instantiated to operate on any objects of a particular class (e.g. graph algorithms).

- **Automatic parallelization.** This knowledge consists of information and procedures which allow problems and their high-level specifications to be parallelized or a parallel execution structure to be modified.

- **Program structure.** This is knowledge of program flow structure analysis techniques and information, which are needed in support of complex program restructuring (mostly in support of automatic parallelization).

- **Target machine.** This knowledge represents the machine characteristics which are significant in making implementation decisions (e.g. the maximum number of processors, reconfigurability, etc.).

Classifying knowledge along these lines, Table 2 shows which kinds of knowledge are applied by each automatic or semi-automatic transformation tool. Manual transformations, which correspond to the other arcs from Figure 1, are not described in the table.
<table>
<thead>
<tr>
<th>Transformation</th>
<th>Types of Knowledge Used in Transformation&lt;sup&gt;3&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>From → To</td>
<td>Input</td>
</tr>
<tr>
<td>(Name of Tool)</td>
<td></td>
</tr>
<tr>
<td>KBLP → RC (KBLP compiler)</td>
<td>—</td>
</tr>
<tr>
<td>KBLP → XPC (KBLP compiler)</td>
<td>—</td>
</tr>
<tr>
<td>C → RC (CP)</td>
<td>Some</td>
</tr>
<tr>
<td>RC → RC (CR)</td>
<td>Yes</td>
</tr>
<tr>
<td>RC → XPC (RC compiler)</td>
<td>—</td>
</tr>
<tr>
<td>CFA → CFA (XPAT)</td>
<td>Yes</td>
</tr>
<tr>
<td>CFA → XPC (PREPAR)</td>
<td>—</td>
</tr>
<tr>
<td>XPC → Code (XPC compiler)</td>
<td>—</td>
</tr>
<tr>
<td>Runtime Config. of Code modules (operating system)</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2. Use of Knowledge Types in PARSE.

---

<sup>3</sup> Refer to the text for an explanation of the abbreviations used for knowledge types appearing in this table. An entry marked "—" means that this type of knowledge is not employed, "Yes" means that this type of knowledge is used, and "Some" means that this type of knowledge may be applied only under special circumstances or that its use is otherwise limited.
5. XPC Language & Compiler

Although XPC and its compiler are both relatively simple, the XPC compiler is the one tool always used in program development under PARSE and every program is eventually expressed as XPC code. There are four primary reasons for XPC's importance as the basic support for explicitly parallel control programming:

- **Nature of an application problem.** The simulation of the interaction of a large number of loosely-coupled, non-homogeneous, communicating processes may be most naturally specified in an explicitly parallel style. For example, a model of decision-making in an economic system is easier to express using explicitly parallel control.

- **Nature of an application's run-time environment.** Current technology makes other programming techniques very natural for speedup-oriented computation-bounded tasks, but not for real-time constrained tasks. For example, the control of a real-time distributed process in a FMS (flexible manufacturing system) is most naturally expressed using explicit control parallelism.

- **Need to tune for performance.** The refined language approach results in very reliable parallelism recognition. Hence, it permits specification of parallel algorithms by specification of data access rights (rather than by parallel control constructs); but such a specification merely suggests a parallel form — the compiler determines the resulting parallelism. Like any other transformation mechanism, however, there are always some attributes of the problem being solved which are not considered by that mechanism, sometimes resulting in less efficient parallel code than a programmer might devise. To achieve this extra performance edge, the programmer must have access to the explicitly-parallel representation of the program; the programmer must be given the ability to specify the explicitly-parallel control structure which will be used.

- **To have a uniform interface.** Since PARSE is to be targeted to a wide class of machines, it is useful to provide a straightforward parallel control language to be used by the higher levels of PARSE in much the same way that a conventional compiler uses assembly language. By generating XPC code, all tools except the XPC compiler are insulated from machine-dependent quirks in the expression of parallel control constructs.

The last two reasons for XPC's importance are, in a very real sense, contradictory: improving XPC's ability to be used for tuning performance to a particular machine generally degrades the portability of the language. For the class of machines described earlier, the main conflict is between support of run-
time hardware reconfigurability and portability across various target architectures. These are critical requirements for parallel programming languages [ScS86].

To support reconfiguration, XPC code must be able to specify operating system functions which support selection of virtual machine sizes (numbers of processors), allocation of subtasks to processors within a virtual machine, and selection of the mode of parallelism for each virtual machine. For example, in a hypercube machine, operating system calls are provided which allow the user to dedicate specific sub-cubes (virtual machines) to particular sub-tasks within a problem. Partitioning of a system into virtual machines is also allowed in PASM [SiS87], however, with the additional capability to select either MIMD or SIMD mode for each virtual machine. In order to optimize performance when the user either has knowledge of intertask communication patterns or the operating system is capable of dynamic remapping of tasks due to observed patterns, XPC must be able to specify that re-assignment of processes to different nodes within a virtual machine take place. Such constructs are not easily implemented on all machines.

Portability of XPC code is closely related to the implicit synchronization and communication within an explicitly parallel control construct, which derives from the underlying virtual machine configuration. For example, a SIMD-oriented definition of a doall construct might specify that all fetches must occur before the store in each statement within the body. This definition would often result in excessive synchronization operations in MIMDs, since, although few bodies would actually require such synchronization, the compiler has no easy way of determining when they are unneeded. Hence, such a definition is not very portable. A more portable definition of doall would state that the order of operations across processes within the body is unspecified and not
necessarily consistent from one run to the next. (This definition would occasionally force the SIMD user to write a sequence of several *doalls* instead of a single SIMD-style *doall*.)

XPC will provide sufficiently rich explicit parallelism constructs so that most parallelism constructs supported by any given machine can be expressed in XPC code. In this sense, the *language* will be portable. However, in support of these machine features, an individual program may use a construct which is not naturally supported on all of the target machines: reusing an earlier example, the ability to select SIMD mode for a virtual machine is not directly supported on a hypercube machine. The design of XPC endeavors to achieve portability of individual programs written in XPC by:

- enabling the XPC compiler to generate code for these “unnatural” constructs, however inefficient it may be (for example, simulating SIMD synchronization on a MIMD) and
- wherever possible, using language constructs that express parallelism in the most portable way (as in the *doall* example).

6. **XPAT & PREPAR**

As stated in the previous section, an explicitly parallel programming interface is provided in PARSE for a number of reasons. Among these is the direct use of XPC as a programming language. PARSE provides the user with a tool to aid in using this interface directly - XPAT. In Figure 1, XPAT is a self-loop on the CFA model specification of a parallel procedural algorithm. XPAT permits a user to specify and analyze algorithms which are specified in a form which can be automatically transformed into XPC code. The analysis provided has three uses:

1. To allow the user to debug interprocess communication and synchronization aspects of asynchronous computations without requiring instrumentation of the target hardware environment.
(2) To support analysis of efficiency of algorithms to permit the user to make intelligent modifications to improve the use of system resources.

(3) To evaluate the performance of an algorithm in terms of the objective of the algorithm itself. (This is particularly useful for algorithms which rely on noisy state information and/or produce results by heuristic methods.)

A further goal of the research surrounding development of XPAT is the formal modeling and analysis of parallel algorithms. For this reason, and to enhance the production of a reliable tool, the design of XPAT should be based on a formal model of computation. Potential candidates include communicating finite automata [AhU79, BrZ83, CaK86a], petri-nets [Pet77], and modifications to petri-nets [Gar85, MaF84, MaL86, Ozs85].

The XPAT environment consists of an integrated environment of source- and object-level modules, textual processors/transformers, and user interfaces. The user specifies the semantics of parallel algorithms in terms of finite automata (FA). Communication between cooperating asynchronous processes is specified by a combination of the definition of the output component of the FA and graph theory (to specify topology).

A working prototype of XPAT is DSSAP [Cas87]. DSSAP is based on a modified communicating finite automata model which was originally created to specify and analyze distributed decision-making algorithms. DSSAP was developed for use in conducting experimental studies of a number of distributed algorithms from the class of computations known as distributed task scheduling based on the objective of load-balancing [CaK86b, CaK86c, ChA82, NiH85]. This prototype has been used for performance prediction and checking of semantics in more than 20 scheduling algorithms based on structures ranging from very simple load re-distribution techniques [Cas86] to bidding [Smi80] and Bayesian Decision or Team Theory [Lin71, Sta85].

XPAT represents a refinement and extension of the structure and functionality of DSSAP. The greatest difference between XPAT and DSSAP is in
the user interface. The goal of DSSAP was to provide experimental results relating to a particular model of computation; XPAT has a goal much larger in scope. Therefore, the user interface of DSSAP was given little consideration in comparison to XPAT. There are two interfaces to consider.

The first involves the specification of algorithm structure and semantics to XPAT. The second aspect of interface design involves the manner in which experiment results are reported to the user. The design of XPAT contains facilities for (among others) a graphical interface to allow users to glean performance and efficiency information from a visualization of algorithm behavior in a global sense.

XPAT also has greater flexibility in allowing users to specify algorithms with arbitrary objectives. This generality supports applications development in fields including image analysis and understanding, non-homogeneous control, and decentralized decision-making applications such as economic modeling and distributed fault-diagnosis.

The second aspect of the XPAT environment is PREPAR, which transforms algorithms specified as CFA into a form compatible with the XPC compiler. This involves some context-sensitive transformation and potentially some interaction with the user to resolve ambiguities. In addition to the basic transformational responsibilities of PREPAR, an integration function is also performed. Since not all parallel algorithms involved in the solution of a problem (at the parallel algorithm level of figure 1) will require refinement with the aid of XPAT, PREPAR must be able to link the output of PREPAR (which are XPC modules) to the XPC modules which constitute the remainder of the solution to the application problem.

In summary, the primary attributes of the XPAT environment are:
- flexibility with respect to target environments,
- compatibility between user code and target environment,
- capability to isolate cause and effect among many environmental conditions,
- reproducibility of observed behavior, and
- ability to produce rudimentary predictions of performance and efficiency.

7. RC Compiler, CR, & CP

In addition to supporting development of code using explicitly-parallel control, PARSE provides software tools which perform automatic detection of parallelism in code written using purely sequential control constructs. Such an approach offers many benefits:
• the possibility of migrating previously written (sequential) applications to parallel computers,
• the ability to insure that each program will be race-free and deadlock-free,
• the ability to write and debug new code for parallel algorithms in the same style used for sequential algorithms (expressing parallelism by data access rights rather than by explicitly parallel control constructs), and
• the ability to insulate the programmer from most of the machine details associated with choice of efficient parallel implementation.

However, these benefits are not easily gained.

One problem is that conventional sequential languages incorporate constructs which "block" efficient, precise, compile-time analysis of programs for the purpose of automatic parallelization. The other major difficulty is that our target machines employ non-shared memory MIMD (or SIMD/MIMD reconfigurable) architectures: automatic parallelization technology for such machines is very new and bears little resemblance to the vectorization-oriented automatic parallelization which has long prevailed. PARSE uses the Refined-Language Methodology [DiK86] to address these problems.

The Refined-Language Methodology is a complete approach to the programming of highly-parallel computers, based on automatic detection of parallelism in code written using sequential control. It includes both a technique for
modifying existing sequential languages to minimize the ambiguity in analysis of their constructs and a new parallelization technology which is primarily intended for MIMD-style parallelization.

Within PARSE, the refined-language methodology is applied to create:

- an RC compiler which reliably recognizes parallelism in RC code and generates XPC code embodying that parallelism in a form most appropriate for the target machine,
- a tool, called CR, which helps the programmer improve the RC specification of the parallel algorithms, and
- a tool, called CP, which aids in migrating existing C programs to parallel computers by converting them into RC equivalents.

Refined C differs from ANSI C only in that it extends the ANSI C function prototype and declaration syntax to explicitly state data access rights and it incorporates the concept of "partitioning" a data structure into mutually exclusive parts.

An RC compiler for PARSE is being designed for non-shared memory reconfigurable target machines — which are particularly difficult targets for conventional automatic parallelization techniques. Many of the traditional do-loop oriented parallelizations result in "pipelined" code, yet that structure is particularly inefficient on non-shared memory machines [Li85]. For PARSE’s RC compiler, a new transformation is being developed to provide a far less synchronization-intensive parallelization of arbitrary (while) loops. Research in irregular code parallelization (the parallelization of code containing arbitrary calls, conditional and looping control constructs) [Fis84, Nic85, SaH86, KIrS87] has also made significant advances toward efficient parallelization for this class of computers; many of these techniques will be employed within

---

4 The transformation involves splitting each while loop into three loops, one sequential and one parallel together inside a loop which may be pipelined. The effect is to place loop control synchronization within a single processor, hence eliminating the synchronization cost.
PARSE.

CR will follow much the same principles as Bulldog [Fis84] in locating ambiguities in parallelization analysis and helping the programmer to resolve these flaws.

Prototypes of CP have been built, but none of these prototypes perform sophisticated dependence analysis [All86, BuC86] to automatically generate partitionings of data structures. The PARSE version of CP will.

8. Knowledge-based Logic Programming

It has been recognized that a logic programming language such as PROLOG can serve as very-high-level specification language [RGP86]. Also, the specification language can itself be a programming language; consequently, the problem of efficiency can be simplified. In some cases the specification might already behave as a tolerably efficient program, although in other cases transformation may be needed to remove inefficiency. To provide users with the capability of logic programming, in PARSE we devise the knowledge-based logic programming environment (KBLP). There are two goals that KBLP is aimed at:

1. To facilitate logic programming with reusable software modules.
2. To compile logic programs into efficient procedural parallel code.

KBLP intends to achieve the first goal with the help of generic objects and generic procedures. The rationale behind this is that if we have sufficient generic objects (and of course their related operations) and generic procedures defined in a programming environment, it is very likely that a new problem can be solved with existing objects and procedures. On the other hand, the second goal is achieved through semantic logic program compilation, which compiles logic programs to efficient parallel code with the same semantics.
Briefly, in KBLP, the functionalities of generic procedures are described as logic programs, but they are implemented as procedural programs: the body of each procedure is implemented as either XPC or RC code. Similarly, objects are declared with logic statements. Following the object-oriented design paradigm [Boo86], a new problem is defined with the declaration of the objects involved. These objects are then matched against the existing objects in the system, and if a match can be found, the reusable operations associated with the existing objects can be used to solve the new problem. It should be noted that at the top level the resulting programs are still logic programs, except that a conjunct in a clause may be implemented with a procedural program. Consequently, the resulting program may not be efficient. To get around this, KBLP applies a set of rewriting rules either to merge pieces of the program or to reorder the conjuncts such that unnecessary backtracks can be reduced.

If the final result has only one clause, then the (very efficient) XPC implementation of that clause is used. Otherwise, a set of corresponding procedures which are coded in RC will be retrieved by KBLP and the RC compiler — with its ability to analyze, restructure, and parallelize across procedure boundaries — will be used to minimize the inefficiency due to the conjunction of multiple clauses.

9. Status

The overall structure of PARSE has been defined. The functional requirements of the components of PARSE have been defined and prototypes of some components have been constructed.

DSSAP, the forerunner of XPAT, is machine independent and currently runs under VAX/UNIX. The groundwork for XPC has been completed in the specification of Parallel C and implementation of a parallel assembler for the
PASM [SiS81] parallel processing system at Purdue University. In addition, all multiprogramming facilities of V7 UNIX were implemented for a distributed system of 68000 microprocessors [CaK84]. This combination of efforts provides the basis for the operating system and language interface prototype for the parallel procedural components of PARSE.

The preliminary definition of RC appeared in [DiK84] and has already undergone revisions to reflect the new ANSI C standard: prototypes of RC compilers have been built for several MIMD computers, including RP3 [Pfi85]. Several prototype versions of C Prefine have also been constructed. Although the flow and data dependence analysis techniques used in these tools have been relatively simple, they have already shown great promise in MIMD-style parallelization. For example, a precise technique for tracking pointer-reference aliases is incorporated, as well as the irregular code parallelization analysis which characterizes refined-language compilation. These methods often find substantial parallelism in code which other techniques consider to be essentially sequential — as in the quicksort example given in the companion paper — yet they permit these other parallelization techniques to be used on constructs for which they are most appropriate (e.g. vectorizable loops). In addition, a large number of conventional optimizations are performed. This technology not only makes sequential procedural programming efficient in parallel execution, it also forms the basis for logic description transformation and parallelization.

10. Summary

PARSE is designed to help programmers to more easily create high quality software for reconfigurable, non-shared memory parallel machines. It does this by conventional software engineering means, but also by aiding in the parallelism-related problems of partitioning problems for parallel execution,
debugging/preventing deadlock and race conditions (debugging asynchronous execution in general), and producing efficient, "high performance," solutions (also predicting performance of solutions).

Although PARSE is a complete environment, it does not restrict the programmer to a particular programming model or strategy. Descriptive, sequential procedural, and parallel procedural languages are all supported and integrated into PARSE — as are tools which provide debugging and analysis facilities.

The primary target for the first implementation of PARSE is PASM. However, a hypercube version and other implementations are planned; the abstraction of a knowledge-based system architecture makes PARSE more amenable to such changes.

Acknowledgments

The authors gratefully acknowledge many useful discussions with Thomas Schwederski, David Klappholz, and Ken Stein.

References

[BuC86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization,"


[Li85] Z. Li, “A Technique for Reducing Data Synchronization in Multiprocessed Loops,” MS


1986, pp. 628-635.


ABSTRACT

PARSE (parallel software environment) is a software environment that assists programmers in the development of parallel programs for reconfigurable non-shared memory computers. The environment supports programming in logic-based descriptive form, sequential-control procedural form, and parallel-control procedural form. Such a rich choice of language interfaces provides a programmer with the capability of selecting the most appropriate or natural specification of a solution to a problem. The major components of the PARSE environment are described: (a) KBLP, which compiles a logic program to procedural parallel code; (b) CP, which transforms C code into RC code; (c) the RC compiler, which generates efficient parallel code from sequential RC code; (d) CR, which analyzes and helps the programmer to improve RC code; (e) XPC, which provides a convenient notation for expression of explicitly-parallel control flow and data layout; (f) XPAT, which provides a programmer with information regarding the performance and efficiency characteristics of a parallel program; and (g) PREPAR, which converts an algorithm modeled for XPAT into XPC code.
1. Introduction

PARSE (parallel software environment) is a software environment for reconfigurable non-shared memory parallel machines; it is an integrated collection of language interfaces, debugging and analysis tools, and operating system which allows users to select from alternative abstraction mechanisms. A choice of language interfaces provides a programmer with the capability of selecting the most appropriate or natural specification of a solution to a problem, thus best utilizing the programmer's time. High performance, efficiency and reliability are enhanced primarily through the aid of intelligent tools. These are automatic or semi-automatic tools used to modify solutions to produce improved solutions or optimized implementations. Unlike most existing work, which is either tailored towards single-processor program development systems (e.g., Cedar [Tei84], Mesa [Swe85], Jasmine [MaW86], Starlite [CoA86], PDS [Che84], CHI [SmK85], Programmer's Apprentice [Wat85]) or tailored towards pure program parallelization (e.g., PTOOL [AlB86], Poker [SnS86], PIE [SeR85]), PARSE merges both objectives.

If a problem is best stated in a descriptive form, intentionally leaving out detailed procedural information, the descriptive language interface will be used. PARSE provides the potential for efficient execution of such programs by using the notion of a compiled logic based program which re-uses procedural solutions to logic specifications. A knowledge-based approach is used to support this compilation. A sequential procedural language is used to specify problems for which parallel execution is possible, but where the user can be isolated from the parallelism. This programming abstraction is supported by automatic and semi-automatic tools which either transform or aid in the transformation of sequential programs into parallel implementations. Where parallel procedural solutions (explicitly parallel programs) are to be used, semi-automatic debugging
and analysis tools may be employed to predict performance and to reproduce sequences of pseudo-random asynchronous events. Thus, the instrumentation of hardware for debugging is avoided by utilizing PARSE to simulate a virtual machine in software.

Briefly, PARSE is the integration of the following seven subsystems:

(a) **KBLP** (knowledge-based logic programming environment), which facilitates logic programming with software reusability and semantic logic program transformation.

(b) **RC** (refined C) compiler, which compiles a sequential algorithm written in RC code into explicitly parallel code (XPC) code. Unlike C code, RC code is annotated with *data access* information which enables rapid, nearly perfect, parallelization analysis.

(c) **CR (C Reflex)**, which analyzes and helps the programmer to improve RC code.

(d) **CP (C Prefine)**, which transforms a sequential algorithm written in C into RC code.

(e) **XPC** (explicitly parallel C), which provides a convenient notation for expression of explicitly-parallel control flow and data layout.

(f) **XPAT** (explicitly parallel algorithm analysis tool), which formally analyzes the performance of a parallel algorithm specified in terms of a CFA (communicating finite automata) model.

(g) **PREPAR** (prepare for parallel Compilation), which converts a CFA modeled algorithm into XPC code.

The purpose of this paper is to describe in detail the above building blocks of PARSE. This paper is organized as five sections, one discussing each major software tool in PARSE. XPC, XPAT & PREPAR, RC compiler & CR, CP, and KBLP. A companion paper, entitled “The PARSE Programming Paradigm — I: Software Development Methodology,” describes the interrelationships of these software tools.

2. **XPC**

Many problems, such as process control, require parallelism of a form which is well understood and which may benefit from explicit programming in a procedural language extended to allow specification of parallel control structure.
The definition of XPC provides both for SIMD/MIMD control parallelism and for explicit statement of data layout across the local memories of the processors. For example, the following syntax (c.f. Parallel C [KuS85]) specifies that for each of 100 processors, an array of 10 integers called a should be allocated:

```
parallel [100] int a[10];
```

Given the above declaration, it is possible to specify the parallel addition of one to a[5] in all processors by:

```
++a[*][5];
```

Using selectors, it is also possible to specify complex SIMD-style processor enable/disable patterns. The following increments only those a[5]'s in odd-numbered processors:

```
selector [100] odds = [n-1\{X\}\{1\}];
++a[odds][5];
```

MIMD-style parallel control will be supported using calls for explicit communication and synchronization.

In general, support for explicit parallelism must satisfy several criteria. Among them are the need to provide power and flexibility to the user of the system. A second aspect which influences the choices of abstractions, however, is the capability of the underlying system to efficiently support the abstract mechanisms chosen. For example, in a very loosely coupled collection of local-memory processing elements, as in a hypercube multiprocessor, shared variables may not be efficiently implementable. Despite this, shared variables may be desirable for applications such as decentralized control, where a shared-variable "distributed monitor" abstraction [Dij75, Hoa74] is often used. Since our goal is to support a class of machines, XPC may include primitives for explicit parallel control over a range of granularities.
A substantial research activity which must be undertaken as a part of this research then must be the decision as to the set of primitives to provide. The decision as to which set to provide will be largely determined by the needs of two groups.

1. The designers of the language processing system for automatic detection of parallelism in refined sequential programs. The set chosen for this group will constitute the virtual machine interface to the language designers and the implementation of this set is what will allow the integrated software development environment to be portable to any machine satisfying the properties of the class of machines under study.

2. The users of the explicitly parallel programming environment. In addition to the direct needs for supporting explicit parallelism for this group, an eye must be kept toward the design of tools for the support of this environment. This constraint, however, is not very stringent since prior modeling work has shown the ability to construct simulation tools and analysis packages for working with some of the least structured primitives.

In providing these primitives, a fixed set of areas must be addressed. These areas, along with some enumeration of sub-issues, are:

1. Data layout specification.

2. Process creation.
   - Parallelism in control flow. (e.g. spawn, fork)
   - Parallelism in data access.

3. Inter-process communication (IPC)
   - Blocking/Non-blocking.
   - Shared variable vs. Message passing abstractions.
   - Remote procedure call support.
   - Support for transaction abstraction.

   - Wait/signal.
   - Generalized message passing.

5. Machine reconfiguration
   - Partition size specification.
   - PE computation mode selection (e.g. MIMD vs. SIMD).

The decision as to which primitives to use for a particular configuration should be embodied in two different places depending on the mode of parallel programming desired by the user. In the case of explicit parallelism (XPC
interface), the user will employ XPAT to provide preliminary performance and efficiency characteristics about the application. This information may then be used to make decisions regarding the ultimate partitioning of the problem, and the choice of IPC mechanisms and synchronization techniques.

3. XPAT & PREPAR

A consequence of allowing explicit specification of asynchronous and parallel operations is that solutions are more prone to failure due to design flaws (software faults). Parallel code generated from sequential specifications of operation (as provided by the RC interface) can be guaranteed free of such phenomenon as races or deadlock during execution. Much can be done in the way of providing safer explicit mechanisms for parallel access to shared objects (e.g. monitors); guarantees of safety, however, are practically impossible. In addition to being extremely costly, instrumentation at the hardware level to support debugging may be impossible without altering actual behavior of the production system. Hence, software development must be supported by means of automated tools for verification of designs which allow controlled experimentation and modification of software for an arbitrary target environment. In addition to debugging capability, it is very useful to do performance evaluation before implementation in the target environment.

Within PARSE, XPAT is the tool to be used in software development employing prototyping as a mechanism for refining specifications. These prototypes are expressed as CFA models, which can be converted into XPC code through use of PREPAR. The analysis provided has three uses:

1. To allow the user to debug interprocess communication and synchronization aspects of asynchronous computations without requiring instrumentation of the target hardware environment.

2. To support analysis of efficiency of algorithms to permit the user to make intelligent modifications to improve the use of system resources.
(3) To evaluate the performance of an algorithm in terms of the objective of the algorithm itself. (This is particularly useful for algorithms which rely on noisy state information and/or produce results by heuristic methods.)

A further goal of the research surrounding development of XPAT is the formal modeling and analysis of parallel algorithms. For this reason, and to insure the production of a reliable tool, the design of XPAT should be based on a formal model of computation. Potential candidates include CFA [Aho79, BrZ83, CaK86a], petri-nets [Pet77], and modifications to petri-nets [Gar85, MaF84, MaL86, Ozs85].

3.1. DSSAP: an XPAT Prototype

A working prototype of XPAT is DSSAP [Cas87]. This tool was developed for use in conducting experimental studies of a number of distributed algorithms from the class of computations known as distributed task scheduling based on the objective of load-balancing [CaK86a,b,c, ChA82, NiH85]. DSSAP has been used for performance prediction and checking of semantics in more than 20 scheduling algorithms based on structures ranging from very simple load distribution techniques [Cas86] to bidding [Smi80] and Bayesian Decision or Team Theory [Lin71, Sta85]. DSSAP is based on a modified CFA model which was originally created to specify and analyze distributed decision-making algorithms. As a concrete example of the nature of XPAT, a brief description of DSSAP is presented here. This section will only provide a description of the tool and its structure; details of the model appear in [CaK86a].

The basic structure of this tool is that of a package of source-level routines and inputs to executable programs as depicted in Figure 1.
The user's algorithm is described in the form of a CFA (FA Definition in Figure 1). The notation used to specify the elements of the CFA is that of a set of reserved structure and procedure names in the C programming language. This component of DSSAP is all that is required for a user to specify the structure and semantics of a distributed scheduling algorithm. In order to accomplish its analysis, however, DSSAP requires additional information.

The functional goal of the algorithm is also specified in C in Performance Objective. This specification is necessary in order for DSSAP to report
information to the user regarding the behavior of the algorithm in terms which are relevant to the application. The FA Engine constitutes a static component of the system used to drive the state transitions in the CFA and to simulate asynchronous events under the conditions of a user specified experiment. User modification of FA engine is not required under normal circumstances.

The characteristics of an experiment are described in Workload Description and Statistics Generation Information. The workload description consists of a specification, for each asynchronous component of the algorithm, of a script of events to occur, or of a specification of simulated dynamic behavior. The latter is accomplished by providing the mean and variance for one of a collection of well-known probability distributions. Statistics generation information is simply a dynamic input to the executable simulation module which directs the Performance Objective module as to the format and quantity of output to provide concerning the results of the experiment to be conducted. The final component of DSSAP is a specification of the interconnection topology of the communicating entities in the algorithm. This is represented in Figure 1 as Structure Specification and is in the form of a dynamically-supplied adjacency list.

3.2. XPAT

XPAT represents an extension of structure and function of DSSAP. XPAT consists of an integrated environment of source- and object-level modules, textual processors/transformers, and user interfaces. The three goals in the design of XPAT are: 1) to extend the functional attributes of DSSAP, 2) generalize the user-interface related aspects of DSSAP, and 3) to provide greater flexibility in allowing users to specify algorithms with arbitrary performance objectives. Examples of the generality sought include image analysis and understanding, non-homogeneous control, and decentralized decision-making applications such
as economic modeling and distributed fault-diagnosis.

The most significant enhancement of XPAT over DSSAP is in the user interface. While the primary goal of DSSAP was to provide experimental results relating to a particular model of computation (i.e. CFA), XPAT has a goal much larger in scope. There are two interfaces to consider. The first involves the specification of *algorithm structure and semantics* to XPAT. The main change to this interface involves removing the use of *reserved names* for specifying CFA components. In order to facilitate this removal, a textual preprocessor is invoked to transform the user-provided CFA model specification of the algorithm to a compilable form. The existence of this preprocessor will be transparent to the user.

The second aspect of interface design involves the manner in which experiment results are reported to the user. The format employed in DSSAP consisted of a tabular listing of evaluations of the user-supplied performance objective function at specified points in time. There was no explicit restriction to this format, but the DSSAP environment did not easily facilitate any other form of output. Therefore, the design of XPAT contains facilities for (among others) a graphical interface to allow users to glean performance and efficiency information from a visualization of algorithm behavior in a global sense.

The other distinguishing features of XPAT (i.e. extended functionality and greater flexibility in specifying algorithm objectives) are accomplished in a number of ways. First, we expand the functionality of the Workload Description component to allow arbitrary invocation of user-supplied, procedurally-specified events. Second, the current restrictions on state transitions as imposed by the notion of algorithm *phase* as described in [CaK86a] are relaxed allowing analysis of algorithms with less periodic behavior. Finally, XPAT will permit easier specification of algorithms with non-homogeneous structure with
respect to nodal computation.

3.3. PREPAR

PREPAR transforms algorithms specified as CFA into a form (Standard C) compatible with the XPC compiler. This involves some context-sensitive transformation and potentially some interaction with the user to resolve ambiguities. In addition to the basic transformational responsibilities of PREPAR, an integration function is also performed. Since not all parallel algorithms involved in the solution of a problem at the highest level may require analysis by XPAT, PREPAR must be able to link the transformed CFA specifications to the XPC components which solve the rest of the problem.

4. RC Compiler & CR

PARSE also provides software tools which perform automatic detection of parallelism in code written using purely sequential control constructs. This approach offers a way to migrate previously written (sequential) applications to parallel computers, guarantees freedom from race and deadlock conditions, and insulates the programmer from most machine dependencies. However, automatic parallelization of code written in conventional languages is somewhat unreliable. Ambiguities which block discovery of precise data access rights, which represent the stores/fetches that a region might make, result in poor automatic parallelization.

PARSE uses the Refined-Language Methodology to minimize this ambiguity. The Refined-Language Methodology is a complete approach to the programming of highly-parallel computers, based on automatic detection of parallelism in code written using sequential control. It includes both a technique for modifying existing sequential languages to minimize the ambiguity in
analysis of their constructs and a new parallelization technology which is primarily intended for MIMD-style parallelization.

Within PARSE, the refined-language methodology is applied to create a RC compiler and a tool which helps the programmer improve the RC specification of his parallel algorithms. (CP, a tool which aids in migrating existing C programs to parallel computers by converting them into RC equivalents, is discussed in Section 5.)

4.1. Refining a Language (C)

The Refined-Language Methodology begins with any conventional high-level language (HLL) base: C, FORTRAN, PASCAL, LISP, ADA, etc. For PARSE, we have chosen to use C.

C was chosen primarily because a refined C compiler, generating code for a shared-memory MIMD, had already been constructed and substantial experience gained in the process. However, C also is the language in which virtually all the PARSE tools are written, hence basing the refined language support on C leaves open the possibility of self-bootstrapping the system to run native on various parallel computers.

The first step in refining a language is, if the base language incorporates explicitly-parallel control constructs which do not have sequential-equivalent semantics, then these constructs are disallowed in the corresponding refined language (for C, there are no such constructs). Since the resulting language is purely sequential, it is impossible that a program written in this language would harbor a race or deadlock condition; further, a flow-analyzing compiler, restructuring the program into parallel code by using only known correctness-preserving transformations, cannot possibly introduce a race or deadlock condition. Therefore, the sequential and all such parallelized versions of a program
must produce the same result — debugging any one implies that all are
debugged. Further, since the compiler decides what kind of parallelism to gen-
erate and how that structure should be implemented on the target machine, the
applications programmer can be insulated from machine dependent considera-
tions (although the compiler for each target machine cannot be).

Unfortunately, the amount of useful parallelism found by a flow-analyzing
compiler examining a program written in such a language is not necessarily all
(or even a large fraction of all) that is present in the program. It is also
difficult to express a parallel algorithm in most such languages. Both of these
difficulties are caused by certain language constructs obscuring the fact that
some operations can be parallelized.

Hence, the second and final step in creating a refined language is to
"refine" the language constructs so that (parallel) data access rights can be
directly stated, providing the compiler with easy access to exactly the informa-
tion it needs and providing the programmer with constructs to express the
parallelism envisioned\(^1\). These refinements are made to blend-in with the syn-
tax and semantics of the base language, and generally constitute only minor
dialectical differences.

In refined ANSI C (RC), the only refinements are:

- a set of extensions to the ANSI C function prototype / declaration syntax
so that permissions for functions to access variables can be directly stated
and
- the new concept called "partitioning" — a way of independently specifying
access rights to arbitrary mutually-exclusive portions of a data structure
(typically an array).

\(^1\) Although the programmer can easily express the parallelism envisioned, the
compiler makes the final decision as to whether that parallelism should be used and, if
so, by what implementation. Further, a programmer mistake cannot result in a race or
deadlock; such a mistake would simply cause the compiler to detect less parallelism.
These "minor" refinements permit a parallelizing compiler to find substantial parallelism in most RC programs without requiring the compiler to perform extensive inter-module flow-analysis or theorem proving, whereas typical C code defies even these (very expensive) analysis techniques. By stating data access rights, the user is also able to write new code for parallel algorithms (and to debug them) in a familiar style.

A simple, yet dramatic, example of the improvement in reliability of automatic parallelism recognition is seen in the following refined C (RC) version of quicksort:

```c
/* Function prototype */
void sort(int *a);

/* Function definition */
void
sort(a)
{
    register int i, j, x, w;
    register int *below, *mid, *above;

    i = 0; j = count(a)-1;
    x = a[count(a) / 2];
    do {
        while (a[i] < x) ++i;
        while (x < a[j]) --j;
        if (i <= j) {
            w = a[i]; a[i] = a[j]; a[j] = w;
            ++i; --j;
        }
    } while (i <= j);

    part(a[w], below,(w<=j), mid,(w<i), above);
    if (count(below) > 1) sort(below);
    if (count(above) > 1) sort(above);
}

2 This code reflects the current definition of RC, which is based on the new ANSI C definition. The original definition of RC differs somewhat in syntax [DiK85].
The fact that the two recursive calls to sort may be executed in parallel is obvious in this version — because of the parallel structure of the data access rights defined by the partition statement — but even theoretically might not be able to be determined using the best compile-time analysis on the conventional ANSI C version of the program.

Therefore, refined languages not only provide an efficient means for obtaining parallel execution from software developed as conventional sequential code — they also provide a "fail safe," very machine-independent, way to specify parallel algorithms: parallel data access rights.

4.2. Compiling RC for Non-Shared Memory

Since a refined language is, in essence, a conventional sequential language, any of the sophisticated techniques applied to parallelizing "dusty deck" FORTRAN [PaK80, AlK82, Fis84, Kuc84, Nic85, VeI85, BuC86, SaH86] can be used for compiling RC. Although these techniques perform poorly for languages like C, RC code provides sufficiently precise flow information that these techniques might work better on RC code than on FORTRAN code.

In fact, information is so readily available that many of the more complex analysis technologies (such as dependence analysis [Al86]) are typically unnecessary when operating on RC code. Hence, an RC compiler can use simpler and faster analysis techniques. In implementing the PARSE RC compiler, we will not employ compiler technologies which require symbolic execution or theorem proving to produce efficient parallel code.

However, many sequential language approaches have, until very recently, been targeted to vector-oriented computers — a very different kind of parallelization from that needed for non-shared memory partitionable/reconfigurable machines. To the best of our knowledge, no automatic parallelization technique
has ever before been successfully used to generate code targeted to this class of machines.

For example, in the RC code for quicksort given above, there are no loops that can be parallelized using vectorization-type transformation. The parallelization of the two recursive calls, and also the parallelization of the two while loops, can only be accomplished by attempting to parallelize regions containing irregular code — code which contains arbitrary control constructs. Whereas most automatic parallelization techniques are based on local parallelizations of do-loop bodies, and often result in synchronization-intensive code (pipelines), the refined language technique is based on parallelization of irregular code, making heavy use of global flow information which is directly available in an RC program.

Recent work in refined language parallelization analysis has resulted in the development of a formal notation and algorithms for finding and describing MIMD-style parallelism in irregular code [KIS87]. Further, a generalized technique for low synchronization (non-pipelined) parallelization of bodies of loops containing control and other dependencies (typically, while loops) has been developed.

Planned extensions to the refined-language transformation technology for the RC compiler within PARSE include a modification of the process-packaging scheme [DiK84] designed to ease communication/reconfiguration costs by selective duplication of computations [Fis84] and more sophisticated management of local memory/variable allocation [All86].

Since we are constructing RC and tools to be relatively "generic" with respect to non-shared memory partitionable/reconfigurable computers, RC will generate XPC code as its output (rather than a particular machine language). Choice of XPC constructs will be guided by a machine description accessed from
the PARSE knowledge base.

4.3. CR

While RC provides constructs which the programmer may use to specify very precise data access rights, the stated rights are only required to be accurate, not to be precise. In other words, a valid RC program might grant more generous access rights to various data than are actually required. In general, it is impossible for the compiler to determine if the programmer has been excessive in this respect: if it could determine that, the analysis could be carried-out perfectly on ordinary C code.

However, the programmer does not care about being precise unless imprecision has some negative effect — typically loss of execution-time speedup by failing to parallelize some region of code. Not all imprecise data access references have such an effect. Those that do can be identified by observing "parallelization failures" caused by particular data access constraints.

CR will be an expert in understanding the way in which the RC compiler parallelizes code for a particular machine. Using this expertise, it interactively guides the programmer to make small improvements in parallel algorithms specified as refined-language code: usually by asking the programmer if a piece of code can be rewritten to remove particular data access constraints associated with the most costly parallelization failures. These improvements are therefore based on maximizing useful parallelism for the target machine, with the pleasant side-effect of making the RC code easier to understand (more precisely specified) as it becomes more efficient.

For example, again consider the RC quicksort given above, but imagine that access rights for the recursive calls had been over-generously stated as they would be in a conventional C quicksort. In other words, the portions of the
array used in both calls would use the same name (the array's name) and hence probably would not be distinguishable. The sort routine modifies values within the array it is passed; hence, the execution of the two recursive calls in parallel would appear to generate a race condition. To prevent the race, the RC compiler would generate sequential code for the two calls.

However, CR's analysis would discover that the expected benefit in parallelizing the recursive calls is very high and that only the single constraint involving the array name caused the failure to parallelize the calls. It would therefore ask the programmer to be more precise about which portions of the array were referenced in each call, indicating that both calls would have to reference different portions in order to make parallelization safe. Hopefully, the programmer would answer this question by inserting a partition statement (part) as seen in the code above.

Not only would this change improve the parallelism, but it increases the information content of the program — it makes the fact that the calls operate of different portions of the array more obvious to a human reader as well as to the RC compiler.

5. CP

In Section 4, we noted that a major benefit of automatic parallelization technology is that previously written (sequential) applications can be migrated to parallel computers. The portion of PARSE which supports this is called CP.

While it is possible to perform the transformation directly from a C program into (parallel) XPC code, this is not how CP operates. Analysis of ordinary C code is expensive; it is also often fruitless. If C code were transformed directly into XPC code, we would find that:
There would be no way of improving the C code's parallelization except to
hand-improve the XPC version of it — a difficult and error-prone task.

Every time the C code must be recompiled, even for a minor change, the
entire C program must be analyzed, including any separately-compiled
modules which are used. This implies very long compile times, moderated
only by using incremental interprocedural analysis techniques [CoK86] by
maintaining information hidden from the programmer (which makes it
difficult for the programmer to improve the precision of the data access
constraints). Such compilation delays are unacceptable in a software
development environment.

Instead, CP attempts to transform C code into RC code. This greatly
increases the maintainability of the code as a parallel program, for the reasons
outlined in the previous section.

The transformation of C code into RC code employs the same analysis
techniques used by "dusty deck" parallelizers [PaK80, AlK82, Fis84, Kuc84,
Nic85, Vei85, BuC86, SaH86]. Like them, CP may take a long time to complete
the analysis across all modules — once it has analyzed the program, however, it
simply embeds the pertinent results in the RC code it generates. At this point,
the C version is essentially discarded, and maintenance and further develop­
ment act upon the RC version. The RC compiler and CR can both be used to
full advantage on the transformed program.

It is worth noting, however, that the conversion of C code into equivalent
RC code is not even conceptually a simple task. Certain (obscure) uses of C
constructs cannot be mechanically translated using current technology and CP
will merely flag these for the user to translate by hand.

There is also the complex issue of what information should be collected.
Unlike many other interprocedural analysis techniques, the refined language
tools accumulate and operate on summary information which understands the
difference between a region of code which uses a variable's value and later
defines a new value for the variable and a region which defines a new value for
the variable and then uses that value, but never uses the value the variable had
at entry to the region\(^3\). Although this distinction is of relatively little use in performing conventional (sequential) optimization, it is useful in automatic parallelization.

For example, if we consider each line of code to be a region, then:

```plaintext
one:   a = b * c; d = a + e;
two:   a = f * g; h = a + i;
```

specifies that regions one and two can be executed in parallel, despite the fact that one defines \(a\) and two uses it. The parallelization would simply create a new name (variable) for two's variable \(a\). The refined language tools would all understand this even if two were the body of a function in a different file from region one.

In CP, information is collected \textit{iff} it is relevant to parallelization. For example, the above distinction is made by the prototype CP (although it is of little help in conventional flow analysis). Other information, such as interprocedural constant propagation, is not maintained by CP because that information does not directly aid parallelization (although it helps in conventional optimization).

The PARSE version of CP will collect the same information, but will employ dependence analysis techniques [All86] to construct partitions of arrays automatically. Prototype CP did not separately track references to portions of data structures, hence it could not automatically generate a partition along those lines.

\(^3\) The first requires that the definition of the variable before the region be computed before the region is entered, whereas the second region could be executed in parallel with the code defining the variable before the region. These regions can be parallelized by the compiler allocating a separate storage location for the variable's definition within the second region.
6. Knowledge-based Logic Programming

The automatic programming paradigm assumes that one can write specifications in a very high level language and then automatically transform the specifications to code. Unfortunately, due to the conceptual gap between the specification language and the implementation language, it has been recognized that automatic programming is difficult to achieve [RaG86]. To achieve the goal of automatic programming to a reasonable degree, in PARSE we have made several decisions: first, use a logic programming language as the high-level specification language; second, with the help of generic objects and generic procedures, provide programmers with reusable problem solving procedures; third, perform semantic transformation to transform a logic program to a procedural parallel program. In what follows we shall briefly discuss these issues:

6.1. Object-based Logic Programming

Although natural language should be the ideal specification format, it has been realized that this type of problem specification may be incomplete, ambiguous, and possibly contradictory. We believe that symbolic logic can reconcile the requirement that the specification language be natural and easy to use with the advantage of its being machine-intelligible. Furthermore, due to the recent advances in logic programming languages, the specification language can itself be a programming language; consequently, the problem of efficiency can be simplified. In some cases the specification might already behave as a tolerably efficient program, although in other cases transformation may be needed to remove inefficiency.

In KBLP, logic programming is coupled with the object-oriented system design paradigm in order to provide modularity and software reusability. This coupling is originated from [Zan84]. Briefly, the notion of objects is
implemented by a new infix operator "with", that takes as left operand an object and as right operand a list of methods (each of which is an arbitrary PROLOG clause). The class hierarchy is implemented by a special predicate "isa". The application of methods to objects is specified by messages using the infix operator ":", and a message cannot succeed until the following three steps succeed: 1) unification of the object, 2) unification of the method, and 3) proof of the method. When a message specifies the passing of a method M to an object O, the message interpreter first attempts to unify M with the methods associated with O, and if successful, attempts to prove the method. Otherwise, it attempts to unify M with the methods associated with the ancestors of O in the class hierarchy, moving upward until either the unification succeeds or no ancestor remains.

In KBLP, the above framework is further augmented by the following predicates:
1) \( \text{class}(a) \) is true if \( a \) is an object class.
2) \( \text{instance_of}(a,b) \) is true if object \( a \) is an instance of class \( b \).
3) \( \text{subclass_of}(a,b) \) is true if class \( a \) is a subclass of class \( b \).
4) \( \text{attribute}(a,b) \) is true if object class \( a \) has \( b \) as one of its attributes.
5) \( \text{attribute_value}(a,b,c) \) is true if object \( a \) has \( c \) as the value of its attribute \( b \).

6.2. Solving New Problems with Reusable Software

Studies [BiP85] have shown that reusability is one of the most significant factors in improving software development productivity and quality. Knowledge-based systems can be of significant help in increasing reusability. An intelligent library assistant, for example, can help in retrieving from the library a module that most closely matches a specification or a segment of specification, where the module has been efficiently implemented by an expert. Although the subject of semantic information retrieval has not been completely resolved, we believe that semantic retrieval can be significantly facilitated with the help of
generic objects and generic procedures. The important idea is that if we have sufficient generic objects (and of course their related operations) and generic procedures defined in a programming environment, it is very likely that a new problem can be solved with existing objects and procedures. In KBLP, the functionalities of generic procedures are described as logic programs, but they are implemented as procedural programs (written in XPC or RC). Similarly, objects are declared with logic statements. Following the object-oriented design paradigm, a new problem is defined with the declaration of the objects involved. These objects are then matched against the existing objects in the system, and if a match can be found, the reusable operations associated with the existing objects can be used to solve the new problem. It should be noted that at the top level the resulting programs are still logic programs, except that a conjunct in a clause may be implemented with a procedural program. Consequently, the resulting program may not be efficient.

The above capability is made possible by automatically relating two entities A and B, where A and B can be an object or a class according to the following two possible relationships:

(a) object A is an instance of class B, or
(b) class A is a subclass of class B, where

\[ \text{case (a) holds if} \]
\[ (1) \quad \text{instance}_{-}o f(X,A) \rightarrow f_A(X) \]
\[ (2) \quad f_B(Y) \rightarrow \text{instance}_{-}o f(Y,B), \text{ and} \]
\[ (3) \quad f_B(X) \text{ where } \text{instance}_{-}o f(X,A); \]

\[ \text{case (b) holds if} \]
\[ (1) \quad \text{instance}_{-}o f(X,A) \rightarrow f_A(X) \]
\[ (2) \quad f_B(Y) \rightarrow \text{instance}_{-}o f(Y,B), \text{ and} \]
\[ (3) \quad f_A(X) \rightarrow f_B(X). \]

As an example, consider an airline scheduling and routing application, let the following formulas hold:
(a) \text{class(flight)}

(b) \text{setof(<X_1,X_2>),instance_of(X,flight) } \land \text{ attribute_value(X,source,X_1) } \land \
\text{ attribute_value(X,destination,X_2), f)}

(c) \text{instance_of(C,f)} \land \text{ attribute_value(C,source,C_1) } \land \
\text{ attribute_value(C,destination,C_2) } \rightarrow \text{ instance_of(C_1,city)} \land \
\text{ instance_of(C_2,city)}

(d) \text{instance_of(V,set) } \land \text{ instance_of(E,relation) } \land \text{ domain(E,<V,V>) } \rightarrow \
\text{ instance_of(<V,E>,graph) } \land \text{ attribute_value(<V,E>,vertex_set,V) } \land \
\text{ attribute_value(<V,E>,edge_set,E)}.

By (d) we can obtain
1. \text{instance_of(<city,f>,graph), since}
2. \text{instance_of(city,set)} (a class is a set), and
3. \text{instance_of(f,relation) } \land \text{ domain(f,<city,city>)}.

6.3. Problem Description and Semantic Optimization

A very important characteristic of KBLP is that operations are described
with logic programs but implemented with procedural programs. For simplicity,
we shall concentrate on descriptions that are in conjunctive form:

\[ f_1 \land \ldots \land f_n. \]

For instances, consider the object class \text{graph}. Provided that associated to
\text{graph} we have a method \text{path(A,B,P)}, which asserts a path P between two ver­
tices A and B in a \text{graph}. Also assume that associated to the class \text{graph} there
are two other predicates: \text{member(V,P)} and \text{length(P,L)}, where \text{member(V,P)} is
true if vertex V is included in path P and \text{length(P,L)} asserts that L is the
length of the path P. The following descriptions can be provided by a user:

1. \[ f_1 = g:\text{path(a,b,P) } \land \text{g:length(P,L) } \land \text{lesseq(L,r)} \]
2. \[ f_2 = g:\text{path(a,b,P) } \land \text{g:member(c,P) } \land \text{g:length(P,L) } \land \text{lesseq(L,r)}. \]
3. \[ f_3 = g:\text{path(a,b,P) } \land \text{g:~member(c,P) } \land \text{g:member(d,P) } \land \text{g:length(P,L) } \land \text{lesseq(L,r)}. \]

\[ ^4 \text{Here we assume that arithmetic operations are done automatically.} \]
In order to translate the above declarative descriptions to procedural forms, we take the following expert system approach. The expert system uses the intensional axioms (knowledge) which are represented as rewriting rules [Bun83] to translate a declarative description \( f \) into an equivalent procedural description \( f' \).

The main ideas behind the expert system are goal normalization and goal reduction. Specifically, a piece of a conjunctive description is normalized if no predicate in that piece whose variable(s) is dependent upon the variable(s) of any method in the same piece that searches a large space. A piece of a conjunctive description can be reduced if there exists an equivalent predicate for that piece. The purpose of the intensional processor is to successively apply the rewriting rules, without changing the semantics of the description, to transform a description into a more normalized as well as more reduced form.

The success of such a transformation process will depend heavily on the knowledge of combinatorial problems and specific domains. For instance, assume that there also exists an algorithm called \( \text{shortpath}(A,B,P,R) \), which asserts a path \( P \), of length less than or equal to \( R \) between vertices \( A \) and \( B \), associated with graph in the object base. The above descriptions can be processed more efficiently if we know that: (a) description (1) can be reduced, and (b) finding a path which does not pass through a specific vertex is equivalent to finding a path in a modified graph which excludes the undesirable vertex.

In general, we need three types of knowledge to support the expert system:

(a) *Efficient Algorithms for Combinatorial Problems:* These algorithms should be coded as methods and reside in the object base.

(b) *Goal Reduction Knowledge:* For an algorithm \( a(T) \), we should specify the function achieved by the algorithm. This can be done by asserting the rewriting rule:

\[
\text{w}_i(T_i) \land \text{w}_o(T_o) \land \text{w}_{i_0}(T_{i_0}) \land (T_i T_o) \leftarrow a(T_i T_o),
\]
where \( T_i \) are input variables, \( T_o \) are output variables, and \( w_i(T_i), w_o(T_o), w_o(T_i, T_o) \) are formulas specifying the desired relationship between input and output variables.

(c) **Goal Normalization Knowledge:** This type of knowledge is represented as rewriting rules of the form \( L(t) \rightarrow R(s) \), where both \( L(t) \) and \( R(s) \) are conjunctions of predicates, to assert the fact that \( L(t) \) and \( R(s) \) are equivalent.

Thus, for instance, if we have the knowledge:

(4) \( G:shortpath(A,B,P,R) \leftarrow G:path(A,B,P) \land P:length(L) \land lesseq(L,R), \) and

(5) \( G':path(A,B,P) \land G:remove(C,G') \leftarrow G:path(A,B,G) \land P:\lnot \text{member}(C), \) where \( G:remove(C,G') \) is a method introduced to remove a vertex \( C \) from a graph \( G \) and the resulting graph is \( G' \),

then (1),(2), and (3) can be transformed to:

(6) \( f_3 = g:shortpath(a,b,r,P). \)

(7) \( f_2' = G':path(a,b,P) \land g:remove(c,G') \land P:length(L) \land lesseq (L,r) \) (after applying (5)).

(8) \( f_2'' = G':shortpath(a,b,r,P) \land g:remove(c,G') \) (after applying (3)).

(9) \( f_3' = G':path(a,b,P) \land g:remove(c,G') \land P:length(L) \land lesseq (L,r) \land P:\lnot \text{member}(c). \) (after applying (5)).

(10) \( f_3'' = G':path(a,b,P) \land g:remove(c,G') \land G':remove(d,G'') \land P:length(P,L) \land lesseq (L,r) \land P:\lnot \text{member}(c) \) (after applying (5)).

(11) \( f_3''' = G':shortpath(a,b,r,P) \land g:remove(c,G') \land G':remove(d,G'') \) (after applying (4)).

7. **Conclusion**

In this paper we have described the essence of the PARSE programming environment for reconfigurable non-shared memory parallel computers. We have described in detail the major building blocks of the environment: XPC, XPAT & PREPAR, RC compiler & CR, CP, and KBLP. The combination of these subsystems makes PARSE a flexible programming environment in which a programmer can develop parallel programs with logic descriptions, sequential descriptions, or explicitly parallel descriptions of a solution to a problem. Furthermore, the intelligence embedded in the environment makes the final parallel programs efficient and reliable.
The overall structure of PARSE has been defined. The functional requirements of the components of PARSE have been defined and prototypes of some components have been constructed. The primary target for the first implementation of PARSE is PASM. Future plans include implementation on a hypercube machine among others.

Acknowledgements

The authors gratefully acknowledge many useful discussions with Thomas Schwederski, David Klappholz, Ken Stein, and Jon Kuhl.

References

[AIK82] J.R. Allen, K. Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," Report MASC TR 82-6, Department of Mathematical Sciences, Rice University, Houston, Mar. 1982.


[MaL86] D.C Marinescu, C. Lin, "Preliminary results on multiprocessor modeling and analysis using stochastic Petri nets," Twenty-Fourth Allerton Conference Communication, Con-


