RL-TR-93-114
Final Technical Report

AD-A269 193

CASE STUDIES OF SOFTWARE DEVELOPMENT TOOLS FOR PARALLEL ARCHITECTURES

Intermetrics Inc.

Chris Garrity, Greg Allen, Ed Chianese, John Reardon,
Larry Shafer

DTIC
ELECTE
SEP 13 1993
S E D

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

93 9 13 021

93-21225

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations. RL-TR-93-114 has been reviewed and is approved for publication.

APPROVED: Joseph P. Cavano

JOSEPH P. CAVANO
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control and Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1993	Final Aug 90 - Dec 92

4. TITLE AND SUBTITLE
CASE STUDIES OF SOFTWARE DEVELOPMENT TOOLS FOR PARALLEL ARCHITECTURES

5. FUNDING NUMBERS
C - F30602-90-C-0023
PE - 62702F
PR - 5581
TA - 20
WU - 74

6. AUTHOR(S)
Chris Garrity, Greg Allen, Ed Chianese, John Reardon, Larry Shafer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Intermetrics Inc.
733 Concord Ave
Cambridge MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CB)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-93-114

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Joseph P. Cavano/C3CB/(315)330-4063

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Parallel Evaluation and Experimentation Platform (PEEP) is the result of an effort at Rome Laboratory to identify the most promising general-purpose software development tools, techniques and approaches from industry and academia for programming high performance parallel computers to meet the needs of Command and Control (C2) applications. The PEEP is a prototype platform for evaluating the applicability of results from parallel programming research efforts to improve the productivity of designers and developers. Intermetrics conducted a study of available innovative tools and techniques, beginning in early 1990. From the survey, Intermetrics chose candidates for inclusion. on a prototype platform, and began to install and evaluate the chosen components. With the prototype PEEP, a number of case studies were conducted to develop small parallel programs using the selected tools. The purpose of these case studies was not to advance the state of the art in parallel algorithms, but to exercise the tools collected for the prototype PEEP. This work identified requirements on architectures, life cycle activities and technologies to support parallel development and developed a long range plan for the PEEP. The conclusions from these case studies also suggest useful methodologies for developing parallel software, and have led to recommendations based on the performance of the current tools and on the projected needs of parallel software development.

14. SUBJECT TERMS Parallel Software Engineering, Parallel Processing, Parallel Software Development

15. NUMBER OF PAGES 106

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

Preface

This report was prepared for Rome Laboratory under contract number F30602-90-C-0023. Joseph Cavano of Rome Laboratory provided direction for the project. Chris Garrity was the Project Manager. Contributions to the project were made by Greg Allen, Ed Chianese, John Reardon and Larry Shafer of Intermetrics.

Accesion For	
NTIS CRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Abstract

The Parallel Evaluation and Experimentation Platform (PEEP) is the result of an effort at Rome Laboratory to identify the most promising software development tools, techniques and approaches from industry and academia for programming high performance parallel computers to meet the needs of Command and Control (C^2) applications. The PEEP is a prototype platform for evaluating the applicability of results from parallel programming research efforts to improve the productivity of designers and developers. Intermetrics conducted a study of available innovative tools and techniques, beginning in early 1990. From the survey, we chose candidates for inclusion on a prototype platform, and began to install and evaluate the chosen components. With the prototype PEEP, we conducted a number of experiments developing small parallel programs using the selected tools. The purpose of these experiments was not to advance the state of the art in parallel algorithms, but to exercise the tools collected for the prototype PEEP. Based on this work, we identified requirements on architectures, life cycle activities and technologies to support parallel development and developed a long range plan for the PEEP. Our conclusions from these experiments also suggest useful methodologies for developing parallel software, and have led to recommendations based on the performance of the current tools and on the projected needs of parallel software development.

Table of Contents

Preface i
Abstract iii
Table of Contents v
1 Introduction 1
1.1 The Challenge 1
1.2 Background 1
1.3 Overview of Approach 3
2 Survey of Parallel Processing Tools and Techniques 4
2.1 Introduction 4
2.2 Tool/Problem Matrix 6
2.3 Discussion 9
2.3.1 Revised Matrix Structure 9
2.3.2 Revised Tool/Problem Solution Matrix 12
2.4 Further Development 13
3 PEEP Configuration 14
3.1 Architecture 14
3.2 Design 15
3.3 Prototype Implementation 16
3.3.1 Integration 18
3.4 Evolution 19
4 Experimentation and Evaluation 20
4.1 Problem Characteristics 20
4.2 Approach 21
4.3 Requirements Analysis and Definition 22
4.4 Design 23
4.5 Implementation 26
4.6 Testing and Integration 29
4.7 Supporting Technologies 29
4.7.1 Prototyping 29
4.7.2 Modeling, Analysis and Simulation 29
4.7.3 Visualization 30
4.7.4 Configuration Management 30
4.8 Discussion 31
4.8.1 Tool Evaluation Summaries 31
CODE 31
CSN_Illustrate 32
CSTools 32
Id 32
*Lisp 32
PProto 33
Crystal 33
PAWS 33
PCN 34
POKER 34
TANGO 34
Other Tools 34
4.9 Methodology 35
4.9.1 Problem Partitioning 35
4.9.2 Combining Id, Crystal and CODE 35
4.9.3 Surveyed Projects 36

5 Conclusions 38
5.1 Requirements for the Software Engineering of Parallel Systems 38
5.1.1. Life Cycle Requirements 38
5.1.2 Integration Requirements 41
5.2 SEPA Long-Term Plan 42
5.2.1 Tool Enhancements 42
5.2.2 PEEP Evolution 43
5.2.3 Sustain PEEP Technology Base 44
5.2.4 Next Generation PEEP 46
References 48
Special References 55
A. Experiment Descriptions 56
A.1 Experiment 1: Exploiting Inherent Parallelism 56
A.2 Cooperative Concurrency 56
A.3 Target-Independent SIMD 58
B. Tool Summaries "Data Sheets" 63
Parallel Environments 63
CODE 63
Express 64
Faust 65
ISSOS 65
Omega/PegaSys 66
PARET 67
PIE 67
POKER 68
Prometheus 69
PSG 69
RPDE 69
TOPSYS 70
FORTRAN Parallelism Support 70
FORCE 70
PAT 71
Pisces 72
Rn 72
Schedule 73
Special Purpose Tools 73
BALSA-II 73
Dalek 74
E-L 74
FIELD 74
GARDEN 75
Hypertasking 75
IPS-2 76
MPD 76
Olympus 76
PADWB 77
PAWS 77
PProto 79
TANGO 80
VMMP 80
Parallel Languages 81
C* 81
Crystal 82
DINO 83

Id 83
Linda 85
*Lisp 86
Molecule 86
MultiLisp 87
Occam 88
Paralation Model 88
PCN 89
Proteus 90
RAPIDE 90
Strand-88 91
UC 91
UNITY 92

1 Introduction

1.1 The Challenge

The technology for parallel processing is evolving rapidly. New hardware architectures are being introduced each year in both the research and commercial sectors. Following these developments, and sometimes in advance of them, is a multitude of approaches to the development of applications capable of exploiting these "high performance" parallel processing architectures. However, there is neither consensus on the "best" processors nor the "best" methods for developing applications to exploit these architectures—nor is there likely to be. Rather, a number of diverse architectures will emerge and coexist, as will a multitude of methods for software development. Therefore, to exploit the potential of parallelism, the challenge is to provide a means to match the "best" architecture(s) with the "best" method(s) for a given application problem.

While high-performance, parallel architectures have received much attention for Scientific Computing, their suitability to embedded Command and Control (C^2) applications has been less studied. The potential in C^2 is somewhat different from Scientific Computing, as a result of the different characteristics of the problem space. For example, many scientific applications consist of fairly simple calculations carried out on a massive scale—for such applications a Single Instruction Multiple Data model is appropriate. However, C^2 applications often deal with large numbers of distinct, autonomous entities, each with its own state and set of behaviors, as in simulation, tracking, or Battle Management. Because C^2 applications are often embedded systems, much of their code must deal with interactions with other systems (e.g., timing, event handling), in contrast to scientific applications which are purely transformational (input/output-oriented). Furthermore, as embedded systems, these applications may have much higher Reliability, Maintainability and Availability requirements than laboratory applications. Consequently, the complexity of these applications results in software which dwarfs the "codes" developed for scientific computing. Unfortunately, there is little experience in parallel programming-in-the-large—meeting this challenge is the goal of Software Engineering for Parallel Architectures (SEPA) project.

1.2 Background

The emerging generation of parallel processors offer unprecedented opportunities for high performance computing, in terms of pushing the limits of what may be computed, and doing so with greater accuracy, efficiency, timeliness, and reliability.

Hardware architectures for parallel processing may be usefully characterized as falling into three classes: Multiple Instruction Multiple Data (MIMD), Single Instruction Multiple Data (SIMD), and mixed systems. Within the class of MIMD architectures, the principal parameter of variation is the interconnect strategy—the topology of connections among processors. The range of interconnection strategies varies widely, from pair-wise connections intended to make arbitrary communication sufficiently fast for typical computations, to specialized busses, grids, and hierarchical structures, usually intended to optimize communications under some set of assumptions. MIMD architectures may utilized

shared memory between processors, or message passing, or both. This mix is another important parameter of MIMD systems.

There are several commercially available MIMD processors including the BBN Butterfly, the Sequent machine, "hyper cubes" (so named for their interconnection scheme) from a number of vendors, and others. MIMD machines are typically constructed from "stock" processors such as the MC680X0.

In the class of SIMD architectures there are fewer commercially available machines. The Connection Machine, is perhaps the best known; array processors and vector processors also fall into this class. As with MIMD architectures, the principal parameter of variation is the interconnection scheme.

There are a variety of mixed architectures. For example, Cray computers are built from one to four Single Instruction Single Data (SISD) processors, integrated with vector processing elements. Signal processing computers are also mixed architectures, typically containing sequential processors, application-specific elements, and array processors.

Dataflow machines present a highly parallel "virtual machine" but may in fact be constructed of any types of underlying hardware.

Neural network architectures are another mixed system—constructed from a variety of sequential, array, and specialized processors—but giving the appearance of SIMD; each neuronal processing element has its own data but follows a single processing rule: to produce its output by firing when the weighted sum of its inputs exceeds a certain threshold.

Of course to achieve the goals of high performance computing, parallel hardware isn't enough—we need software capable of exploiting its capabilities. Thus, to exploit parallel architectures we must ask, *Where will this Software come from?* There are several alternatives: (1) existing, sequential software may be reused as-is; (2) sequential software may be re-engineered to exploit parallelism; and (3) new software may be developed which exploits the parallel capabilities of hardware through its application of new (parallel) algorithms and data structures. Alternatives (2) and (3) require development; unfortunately, software development is hard, even for today's well-understood sequential machines. The discipline of Software Engineering has emerged over the past 30 years out of the field of Computer Science to confront the issues of developing large, critical software applications. Software Engineering contributes models, languages, tactical methods, and strategic processes for organizing software development.

Although we now are beginning to have adequate models of sequential computing, we lack comparably powerful formal models of parallel computation. Although it is easy to generalize a sequential model to a parallel collection of sequential "threads," there are two further considerations: coordination between threads, and the issues of parallel data. That must be factored into a process for engineering parallel software. We lack adequate theories in these areas which are suitably rigorous and sufficiently expressive to be generally useful. This seems to be because our concepts in these areas are still highly architecture-specific, or so low-level as to be ineffectual for large systems (in the same way that attempting to

program an automated teller system using a Turing machine formalism would be ineffectual).

As we attempt to raise the level of software engineering for parallel systems, we face issues such as:

- What impact do parallel architectures have on the design process? And, what tools can help to manage this impact?
- How does one understand and isolate target machine dependencies?
- What are the trade-offs between portability and efficiency?
- How does language choice influence design?

To address questions such as these, the SEPA program built a prototype framework within which to evaluate current parallel processing technologies: hardware, software, and methods. This framework is the Parallel Evaluation and Experimentation Platform (PEEP).

1.3 Overview of Approach

In this section, we outline our approach to developing the PEEP.

Based on an initial understanding of the problem domain, we began by surveying currently available capabilities to support parallel development. These capabilities include: parallel programming languages, parallel programming tools and environments, software design and analysis tools and methods, and the underlying hardware needed to host these capabilities. The survey led to an identification of a number of promising, publicly available tools and techniques which we characterized for their applicability to various parallel programming problems. The resulting preliminary "tool/problem matrix", documented in the survey report, led to the initial population of the PEEP. We have subsequently refined the initial matrix based on further experimentation and analysis. The original and revised results are discussed in section 2.

The most visible result of the SEPA program is of course the PEEP itself, which has been defined, configured, operated and delivered within the course of this effort. Section 3 provides an overview of the PEEP, its design and prototype implementation, and a discussion of its continued evolution.

The PEEP served as a base upon which to evaluate parallel development methods. Three experiments were defined and carried out, reflecting three different "problems" in software development for parallel systems. In section 4, we describe this series of experiments, and our findings with regard to methodology and the usability of tools and techniques in support of that methodology.

Section 5 summarizes our overall conclusions and recommendations for the future in terms of requirements for the software engineering of parallel systems, and a SEPA long-term plan.

2 Survey of Parallel Processing Tools and Techniques

2.1 Introduction

As a part of the SEPA program, we surveyed existing, available technology for parallel architectures. We began with a literature search, using the facilities of the MIT libraries, and a number of computerized searching systems. In many cases we followed up by contacting the researchers, finding references to current and upcoming projects as well. As a foundation, we had access to two recent surveys: Survey of Parallel Computing [Miller, 1989] and Software Techniques for Non-Von Neumann Architectures [Lightfoot, et al., 1990]. Our work was meant to supplement, not replace, these existing surveys. The focus of this survey was parallel software development tools. For this reason operating systems such as Mach or Cronus were not considered as part of the survey since they do not fit in the tools category. It is important to note that the survey was not exhaustive, and that it has a bias towards University projects rather than commercial products. Papers in the literature tend to be the results of academic research rather than descriptions of commercial tools.

The technologies we surveyed fell into several categories:

- programming environments and tools relevant to parallel programming
- program visualization (including program animation and modeling)
- parallel languages

The purpose of surveying the first category was two-fold: to identify promising approaches to integrated frameworks (i.e., environments) in support of parallel program development; and to identify promising tools supporting some aspect of parallel program development.

Under this category, the following tools and environments were surveyed (descriptions of these, and all other tools mentioned in this report are provided in appendix B):

GARDEN
FIELD
PIE
Prometheus
Faust
CODE
Pisces
Rn
BALSA II
TANGO
PARET
VMMP
Omega/PegaSys
PSG
POKER
ISSOS
Unity

PADWB
Schedule

Tool	Design	Graphics /Anim.	Alg. Select	Integra-tion	Sim/ Proto	Portabil-ity	Perform.	Partition	Debug/ Test
GARDEN	X	X	X		X			X	
FIELD				X					
PIE	X		X				X		X
Prometheus	X		X				X		X
Faust	X		X	X		X	X	X	
CODE	X					X		X	
Pisces						X			
Rn	X			X	X		X		X
BALSA II		X							
TANGO		X							
PARET	X	X			X		X	X	
VMMP						X			
Omega/ PegaSys	X	X							
PSG	X						X		X
POKER	X	X	X		X		X		
ISSOS			X		X		X		
UNITY	X		X						
PADWB			X		X				
Schedule						X			

The following systems were examined as visualization systems:
BALSA-II
TANGO

We initially looked at the following parallel languages:
MultiLisp
Occam
Linda
Molecule
Crystal
Paralation model
C*
Id
Unity
UC
Joyce

Language	Technology	Developer	Targets*	New/Extension
MultiLisp	Large/medium grain Dataflow style Primarily MIMD	Halstead/MIT	Encore Multimax BBN Butterfly Alliant FX/8	Extension
Occam	Large/medium grain Communicating sequential processes Message based	INMOS	Transputer	New Language
Linda Model	Large/medium grain Broadcast messages MIMD	Gelernter/Yale	Intel iPSC/2 Encore Multimax Sequent Symmetry Alliant FX/8	Extension
Molecule	Layered software development Dataflow MIMD	Hwang/USC Xu/Rutgers	Intel iPSC/2	Extension
Crystal	Functional language Data parallel Large/medium grain	Chen/Yale	Intel iPSC/2 Connection Machine VAX (interpreter) Sun (interpreter	New Language
Paralation Model	Data parallel Fine/medium grain SIMD/MIMD	Sabot/Thinking Machines	Connection Machine	Extension
C*	Data parallel for massively parallel H/W primarily SIMD	Frankel/ Thinking Machines	Connection Machine	Extension
Id	Functional language Dataflow Fine grain	Arvind/MIT Nikhil/MIT	MIT tagged-token dataflow architecture	New language
UNITY model	Parallel program specification	Chandy/UTexas Misra/UTexas	Not applicable	New language
UC	separation of programming and efficiency issues parallel program maintenance ease	Bagrodia/UCLA Chandy/CalTech Kwan/UCLA	Connection Machine	Extension
Joyce	Distributed processes Remote procedure call across network Large grain	Brinch-Hansen/USC	Interpreter available	New language

2.2 Tool/Problem Matrix

Based on the information gathered in the survey we initially developed a tool/problem solution matrix covering the software development problems of parallel software, and some additional problems relating to quality and management concerns of large software systems.

It was interesting to note that most of the tools surveyed do not address the quality and management aspects of developing software. The primary reason these issues are not addressed in the tools surveyed is probably that these tools are almost entirely university research projects. As such, they are not concerned with risk assessment or measuring

* Targets represents known implementations of the languages at the time of the survey. It is possible that there are other implementations, either not known at the time, or implemented since then.

productivity, or with the engineering of truly large and complex systems. Further, for many software engineering activities, whether the problem domain or target hardware is inherently parallel is irrelevant to the activity; thus technical or management tools for parallel software development need not be different from those for ordinary software engineering.

The initial matrix was split into 3 tables. The first mapped tools to parallel software engineering problems. The second mapped tools to quality issues, and the last mapped tools to management issues.

PARALLEL SOFTWARE ENGINEERING PROBLEMS

Tool	Spec	Design	Code/ Test	Alg. Sel	Perf/ Eval	Data Dist	Part	Load Bal	Comp Repl	Comm	Debug /Test	Reuse	Num Procs
PIE		X	X	X	X						X		
Prometheus	X	X	X	X	X						X		
Faust	X	X	X	X	X	X	X			X			
CODE	X	X	X				X					X	
Pisces			X										
Rn		X	X		X						X		
BALSA II				X							X		
TANGO				X							X		
PARET	X	X			X		X	X		X			X
VMMP				X									
Omega/ PegaSys		X											
PSG		X			X						X		
POKER	X	X		X	X					X			
ISSOS			X	X									
Schedule													
PAWS				X									X
IPS-2					X		X	X		X	X		

PARALLEL SOFTWARE QUALITY ISSUES

Tool	Reliability	Maintainability	Portability	Efficiency
PIE				
Prometheus				X
Faust		X	X	X
CODE			X	X
Pisces			X	
Rn				X
BALSA II				
TANGO				
PARET				X
VMMP			X	
Omega/ PegaSys	X			
PSG				
POKER				X
ISSOS				
Schedule			X	
PAWS			X	
IPS-2				X

PARALLEL SOFTWARE MANAGEMENT PROBLEMS

Tool	Productivity Measurement	Risk Assessment	Resource Allocation	Architecture Determination
PIE				
Prometheus				
Faust				
CODE				
Pisces				
Rn				
BALSA II				
TANGO				
PARET				X
VMMP				
Omega/ PegaSys				
PSG				
POKER				
ISSOS				
Schedule				
PAWS				X
IPS-2				

2.3 Discussion

One result of our study was a reformulation of the matrix developed earlier in this project. We expected to update the matrix as we used tools to reflect additional capabilities not mentioned in the papers, or to downgrade capabilities that were not supported to the extent we expected. The original matrix also needed revision in structure for three main reasons. First, some of the "problems" in the original matrix are simply phases of the software development cycle, such as design, and are not specific to parallel software development even though there may be some unique problems. For some of the "problems" which are unique to parallel software development, it seemed important to identify where in the development process these problems need to be addressed. Finally, some of the "problems" are usually determined by external factors, and are not directly addressable by themselves. We also classified the tools in terms of supporting some particular technique, such as simulation, that is used to help the developer solve the problems. The new tool/problem solution matrix is structured in terms of the software development life cycle, and the unique problems added by parallel architectures to each phase of the process. The following sections describe the new matrix structure, and present an updated matrix based on the tools actually evaluated.

2.3.1 Revised Matrix Structure

Of the original "problems", Specification, Design, Code/Test, and Debug/Test become life cycle activities. In the new matrix they are broken down into Requirements Analysis and Definition, High Level Design, Low Level Design, Implementation, and Testing and Integration. This life-cycle and the parallel programming "problems" addressed during each phase are discussed below.

Requirements Analysis and Definition is concerned with defining the application and identifying interfaces. The activities associated with this phase have to do with analyzing the requirements for completeness, consistency, feasibility, and customer acceptability. These activities are not necessarily changed if the architecture is parallel rather than conventional. For example, user interface requirements having to do with screen layout would not influence the architecture of the machine, but user interface requirements having to do with response time could influence the architecture if it became clear that they could only be met with a parallel implementation. So it is possible that all requirements analysis could be affected by having a parallel architecture, or it could be completely independent. Ideally, this phase of the software development process would be completely architecture independent unless a specific architecture was listed as a requirement.

Design immediately follows requirements analysis and definition, and in some cases the division can be blurred. Consider the case where an application has been partitioned into it's major processes during requirements analysis to determine feasibility. Is this still requirements analysis, or is it the beginning of design? Even the line between high level and low level design can be blurred. For the purpose of this study, high level design is classified as identifying the main components of the software and the interfaces between them, while low level design is more concerned with how the previously identified components will be implemented. In addition to the normal design processes of decomposing the problem and

defining interfaces, high level parallel software design also involves Process Partitioning, Data Distribution, and Data Coordination. These are defined as follows:

- Process Partitioning is the process of breaking the application into separate processes that could be performed in parallel.
- Data Distribution refers to the process of determining how data should be divided among processors. Some aspects of this problem vary depending on the architecture, and may be influenced by the process partitioning. For example, it is more important for data to be local to the process that uses it in a message passing MIMD architecture than on a shared memory machine.
- Data Coordination is the process of identifying the communication or synchronization of data between processes. This previously came under the heading of Communication.

During low level design the high level design problems are investigated in more detail, and the following additional problems must be addressed:

- Algorithm Selection may be determined by the architecture if it is already known, or it may determine the appropriate architecture. Ideally the algorithm would still be architecture independent, but given the current state of the practice, this is usually not the case.
- Data Type Determination is the process of specifying the important abstract data types.
- Control Synchronization refers to the process of determining necessary synchronization between processes. Such as points where one function must complete before another can proceed. This also came under the Communication heading in the old matrix.

It should also be noted that language selection must be done during the design phase. This is also true of sequential software development, but for parallel software development it is more likely to be determined by external factors such as availability on the target, or availability of existing source code. One problem is not unique to parallel software design, but should be mentioned because none of the parallel design tools support it. That is the production of review materials. Part of any design process is to hold a design review where experts read the design and comment on it. It is necessary, therefore, to produce some sort of review materials for the reviewers to read. This is lacking in all of the tools surveyed.

Implementation is probably the best defined portion of the development process. In the new matrix it is broken down into coding, linking/loading, and debugging.

- Coding includes both writing and compiling the application.
- Linking/Loading refers to the process of building the complete application from the compiled code and loading it on the parallel target.
- Debugging covers the testing of independent pieces of code; finding and fixing bugs as they arise.

In general purpose computers, linking/loading is usually considered to be the last step of coding. We have split it out for parallel machines because the linking and loading of a

parallel application usually involves some mapping of processes to processors. All of the implementation problems apply equally to sequential programs, however, special tools, e.g. parallel debuggers, are needed in the case of parallel software.

Testing and Integration is the part of the process where all the pieces of the application are brought together and tested as a whole to determine whether all the requirements are met. The testing and integration problems are Validation Testing, and Performance Measurement and Tuning.

- Validation Testing verifies the application meets requirements. In general, validation testing is the processor of measuring the reliability and correctness of the software. Testing parallel software involves finding additional problems, such as non-deterministic bugs or deadlock situations.
- Performance Measurement and Tuning involves finding the bottlenecks in the application and making adjustments to the application to alleviate them without creating other bottlenecks.

The remaining problems in the old matrix, Load Balancing, Computation Replication, Number of Processors and Reuse were removed for the following reasons.

Load Balancing divides into static load balancing and dynamic load balancing. Dynamic load balancing has to be implemented at the operating system level. It is not something that can be added after the fact by a tool, and it is not addressed directly during the development process. Static load balancing is addressed during the development process, but is covered by a combination of the other development problems: performance evaluation, process partitioning, and data distribution.

Computation replication, that is the duplication of a computational unit on more than one processor, tends to be language dependent, for example, via task types in Ada. In languages that are extensions of sequential languages it is sometimes handled by an additional tool at link time which maps processes to processors, but again it is language dependent.

Number of processors represented the problem of determining the ideal number of processors for a given application. This was generally not a concern during the development process. Either the number of processors was a known, limited number, in which case the challenge was to design the most efficient algorithm for that number, or the number of processors was variable, and the solution was designed in terms of an unlimited number of "virtual processors" which could then be mapped to the actual number of processors when it was known. The latter was also the best approach to take when concerned about scalability as it is easier to map a greater number of virtual processors to fewer actual processors, than it is to break up an algorithm that was designed with a given number of processors in mind.

Reuse could be classified in two of ways. First is the ability to reuse existing code fragments, "dusty decks", as is, in a parallel framework. This is usually dependent on the language and whether it is based on an existing sequential language. Second is the ability to write code that can be reused on different parallel architectures. This is more of a quality

issue, and it depends on whether this is an important consideration for the application in question. Quality issues are discussed below in section 2.4.

In addition to identifying the parallel software development problems addressed by a particular tool, we identify whether a particular problem solving technique (such as simulation) is employed. In some cases a tool does not really focus on any one particular problem, but supports a particular technique which could be used by the developer to analyze a problem.

2.3.2 Revised Tool/Problem Solution Matrix

The revised matrix includes only tools which were evaluated in more depth during this project. This includes tools which were used during the development of the sample problems, or tools for which we had the user documentation in addition to the published paper.

	Requirements Analysis			Design					
Tool/ Language	Complete	Consistent	Feasible	Proc. Part.	Data Dist.	Data Coord.	Alg. Sel.	Data Type	Control Synch.
*Lisp									
Ada 9X									
BALSA II							*		
CODE/ROPE				†		†			
Crystal									
CSN_Illustrate									
CSTools									
Id				†		†	†		
PAWS							*		
PCN									
POKER						*			*
PProto	*	*	*	†		†			
TANGO							*		

	Implementation			Test & Integration		Support Technology		
Tool/ Language	Coding	Link/ Load	Debug	Validation	Perf. Eval	Prototype	Simulation	Visualize
*Lisp	*	*					*	
Ada 9X	*	*						
BALSA II								*
CODE/ROPE	*	*						
Crystal	*							
CSN_Illustrate		*						*
CSTools	*	*	*					
Id	*				*	*	*	*
PAWS					*		*	*
PCN	*	*			*			*
POKER							*	
PProto						*	*	
TANGO								*

In the matrix '*' indicates a problem that is intended to be solved, '†' indicates a problem which the tools does not claim to solve, but to which we tried to apply it because there seemed to be a possibility the tool would help.

Note: Languages are now included on the list to the extent that a particular compiler or interpreter was available on the PEEP prototype. See section 4 for more detailed evaluations of the tools.

2.4 Further Development

Based on the updated Tool/Problem Solution Matrix it appears that future matrices should also try to capture quality issues. That is, how the tool or technique improves the quality of the resulting application. To some extent the quality of the application depends on the requirements. For example, architecture portability may not be an important consideration is the application is required to run on only a single target architecture. In this case a non-portable implementation is not necessarily a low quality result.

Therefore further matrices should try to capture the support of the following system "ilities":

Maintainability
Evolvability
Scalability
Architecture Portability
Reusability
Productivity

Note, efficiency is not on this list, as presumably the main reason for using a parallel architecture is to improve efficiency or throughput. As a result, all tools that aid in parallel software development could be said to improve efficiency.

3 PEEP Configuration

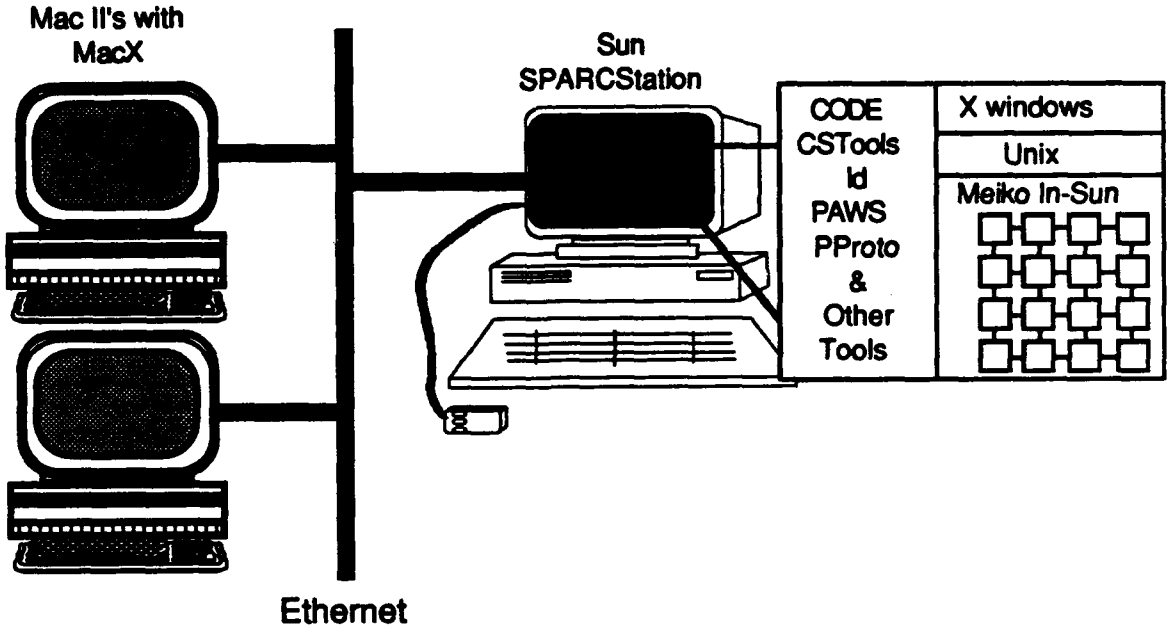
In this section we provide an overview of the PEEP configuration and the criteria that led to this configuration.

3.1 Architecture

The PEEP is designed to be an evolvable configuration of hardware and software, such that it is both immediately usable, relatively robust, and yet open. The hardware base which has been selected consists of well-established, off-the-shelf, mature "stock" hardware.

Sun workstations form the main platform of the PEEP. Two Suns were procured for the project, a Sun4/330 and a Sun4/470. These workstations include color monitors and a graphics accelerator in order to provide full color graphics. The intent of selecting these machines is to support the bulk of software development. In addition to the Suns, the PEEP will incorporate an Apple Macintosh computer, to utilize that system's sophisticated user interface, while integrating it with the Sun "backbone" which will act as file server. At Rome Laboratory, the Sun4/470 will be a back-end server to a network of Apple Macintoshes connected via Ethernet. The Macintoshes serve as multi-user front ends for the PEEP and also be used to perform data analysis on research results and maintain the PEEP documentation. The architecture is shown in the figure below.

The Meiko In-Sun Computing Surface was installed in both Sun workstations. It provides a scalable, multi-processor distributed memory architecture. Each individual processor executes sequential code using its own dedicated memory. Processors communicate via efficient, high speed inter-process message links. Each computing surface contains 16 processors and is configurable in a number of different ways. This flexibility in the configuration will allow the Suns to be used to help determine the most efficient processor configuration for a given parallel application. Computing surfaces may be combined using the expansion slots on the Suns, so more boards may be easily added in the future to further expand the parallel capabilities of the Suns.



3.2 Design

As the tools being evaluated on the PEEP will change over time, it is important to have a solid foundation on which to build. A primary goal of the PEEP design is openness. The PEEP should be open to hosting new tools, languages, and methods. Also, given its primary role as a platform for experimentation with software tools from so many diverse sources, it must be easy to host as many of these as possible. For this reason, Suns, running UNIX and X, and Macintoshes, running X, were an ideal choice for hosting the diverse collection of university and commercial tools identified by our earlier study.

There will be a core set of tools on the PEEP upon which the rest of the system will be built. Sun workstations are equipped with a number of tools that naturally become a part of the core PEEP. The Suns run the UNIX operating system, which has rapidly become accepted as a de facto standard operating system. Other tools which are packaged with the workstation include sequential language compilers and debuggers, a program building facility (make), source configuration management tools, and a number of editors and other text processing tools.

Early in the design process, we considered the possibility of recommending a multi-processing operating system like Cronus or Mach. In the long-run, this may be an evolutionary goal, however, at present, most of the tools with which we have populated the PEEP run under UNIX (Sun OS), and it remains the best choice for this time.

Layered on top of Sun OS is Sun's Open Windows graphical user environment. This is based on the X Window System from MIT. Tools which require basic X facilities run without modification under Open Windows, as will tools developed specifically to exploit Open Windows.

Open Windows provides the most basic level of tool integration from the user's viewpoint. Tools resident on the PEEP are accessible to users via a special PEEP menu that has been added to the default workspace menu provided by Open Windows.

Further integration is provided by FIELD from Brown University. FIELD was developed as a framework for integrating software development tools. Using FIELD's message passing system, independent parallel software development tools could share information. Another tool from Brown University, GARDEN, provides a mechanism for supporting multiple parallel languages for prototyping and language evaluation.

3.3 Prototype Implementation

To implement the PEEP prototype the various hardware components described in section 3.1 were purchased. The main task then was to populate the PEEP with promising parallel tools and languages identified in the survey. This was accomplished in stages and tool acquisition continued throughout the project.

Sun workstations were chosen as the platform because many of the surveyed tools would run on them, however, there were some tools that were not implemented for the Sun. These tools were immediately excluded from the list of tools to be considered for the PEEP as porting any tool to a new target was considered outside the scope of this project.

Of the remaining tools we focused on the ones that were the most language or target independent, or that at least supported multiple targets or languages. Acquisition of a tool usually involved the following steps:

1. Contact the main developer to determine current status of the project.
2. Determine whether Sun workstation implementation of the tool is available.
3. Get pricing information and licensing agreement.
4. Negotiate license agreement.
5. Complete purchase and receive delivery.

In some cases the software was freely available and accessible via the internet. Step 3 in that case becomes a simple case of downloading the software over the internet, and steps 4 and 5 are not necessary. The results of our attempts to get each tool are documented in the table below.

Tool	Contact	Install	Problems	Comments
*Lisp Sim	Thinking Machines	Yes		*Lisp Subset
BALSA II	Mark H. Brown	Yes	Macintosh Host only	Demo disk installed
CODE/ROPE	James Browne	Yes	Took over 6 months to get software	New version, CODE 2.0, will provide better support
Crystal	Marina Chen	Yes	Compiler for iPSC/2 target never available	Interpreter for Sun host used

Tool	Contact	Install	Problems	Comments
CSTools	Meiko	Yes		Compiler and tools supplied with Meiko transputer board
DINO	Robert B. Schnabel	No	Unable to contact developer	
EXPRESS	Parasoft	No		Commercial product similar to CSTools
Faust	Hammerslag	No	Tools no longer supported	Did receive tape, but unable to install software
FORCE	Harry F. Jordan	No	Product not supported	
Hypertasking	Marc Baker	No	Unsupported Intel software	iPSC/2 host only
Id	Arvind	Yes	Took over 2 months to negotiate license	Initial version required Common Lisp to be installed
IPS-2	Barton Miller	No	VAX hosted only	Started a Sun port, but never finished
ISSOS	Karsten/Schwan	No	Unable to contact developers	Project ended
Linda (C)	David Gelertner Scientific Computing Associates	No		Research became commercial product
Molecule	Kai Hwang	No	Unable to contact developer	
Olympus	Gary Nutt	No	Incompatible with Xwindows	Received license agreement
PADWB	Gould	No	Unable to contact developers	Appears published paper may have been proposal rather than results
PARET	Nichols/Edmark	No	Unable to contact developers	
PAT	Kevin Smith	No	FORTRAN specific	Freely available via ftp
PAWS	Dan Pease	Yes	Not available until end of project	Only beta version
PCN		Yes	Not available until end of project	Available via ftp over internet
PIE	Zary Segall	No	Unable to contact developer	

Tool	Contact	Install	Problems	Comments
PISCES	Terry Pratt	No	FLEX/32 host only	Project on hold not likely to restart
POKER	Larry Snyder	Yes	Took over 6 months to negotiate license agreement	Sun3 version only
POSYBL		No	Required network of Sun workstations	Free implementation of Linda for network of workstations available via ftp
PProto	ISSI	No	Required ONTOS DBMS	PProto available from Rome Laboratory
Prometheus	Sean Arthur	No	Project abandoned	New project TaskMaster
STRAND-88	Tim Mattson	No		Commercial parallel language
TANGO	Steve Reiss	Yes		Available via ftp
TaskMaster	James D. Arthur	No	VAX hosted only	
TOPSYS	Thomas Bemmerl	No	iPSC/2 host specific	

A number of problems arose during this process:

- Many tools were lost because the developer had left the company or graduated. In at least one case the current employees had never heard of the tool. In another case the tool was available, but without documentation, and in an incomplete state.
- Universities are often not equipped to distribute the results of their research outside the academic community. The best example of this is the licensing agreements supplied by the Universities. It was usually necessary to make substantial changes to the agreement to make it acceptable for a commercial company, and in particular to make it available to the government. It often took months to negotiate the license agreements.
- The tool might not exist because the published paper represented future plans that were not implemented due to never receiving the funding support.
- Software support varies a great deal. Some ongoing projects are only too happy to provide support when they learn that a third party is using their tool. In other cases the software is delivered "as is" with no documentation, no installation instructions, incomplete sources and no support.
- In rare cases we simply got no response at all, even after several phone calls or e-mail messages.

3.3.1 Integration

There were also problems with the planned integration approach of using FIELD and GARDEN.

First, the tools we acquired were too independent or self-contained to be integrated using FIELD. Often the tools were based on completely different assumptions making the sharing of information meaningless. Also, the type of integration that would have been of most use was "sequential." That is, the output of one tool could have been useful as the input to another. FIELD is designed for more "parallel" integration, where two tools are running at the same time and communicate when changes are made. For example, imagine a syntax directed editor for a parallel language, and a tool which does some sort of static analysis. If the editor broadcasts when a change is made, the analyzer can update its analysis based on the change.

Second, GARDEN was not robust enough for heavy use, and restrictions on input and output of the source form of a program made it difficult to use in the way we had intended. It is also the case that research on GARDEN has stopped for the moment while the developers concentrate on FIELD. As a result it is not likely to improve in the near term

However, we did integrate one tool using the FIELD mechanism. TANGO is an algorithm animation tool that was also developed at Brown University. It was already integrated with FIELD in its sequential form. In order to use TANGO with the transputer we ported the target specific portions of TANGO to the transputer. Once that was complete it was possible to animate a program running on the transputer with the display on the Sun workstation. For further evaluation of TANGO see section 4.

3.4 Evolution

Specific recommendations for the evolution of the PEEP are given in section 5.

Above the level of individual tools, it would be desirable that the degree of integration evolve toward greater tool integration and interoperability. This could be achieved via:

- a common intermediate language at the virtual machine interface
- common program database facilities
- common information capture capabilities

FIELD could still be used as the integration mechanism in some cases, but a common data format is needed to provide the "sequential" integration.

4 Experimentation and Evaluation

We conducted a series of experiments in parallel development. The purpose of these experiments was two-fold: (1) to evaluate the functionality and usability of the tools and techniques which had been selected for the PEEP, and (2) to investigate the larger issues of "parallel-programming-in-the-large": development methodology, tool and technique combinations, and life cycle concerns.

Based on the survey, we determined that not all life cycle areas were equally well supported. Current tools supporting such activities as requirements analysis and functional testing, for example, are somewhat limited in scope. In many cases, this reflects the fact that there has been limited success in these activities for all kinds of software development—not just for parallel architectures. Recognizing the practical limitations of available resources, we chose to concentrate our experimentation on those areas where there is the most activity and potentially the most leverage to be gained by applying the best available technology. Active areas include parallel languages, parallel development environments, and overall analysis techniques from early system level performance simulation to monitored execution.

This chapter is organized to highlight these methodological and life cycle issues. In the next section, we describe the three parallel programming problems we investigated. Subsequent sections address the full sequence of Requirements, Design, Coding, etc., following the SEPA Problem Space matrix we described in Chapter 2.

4.1 Problem Characteristics

We identified three parallel programming problems for experimentation on the PEEP. Each was selected to investigate certain issues in parallel programming. Our emphasis was *not* on creating new parallel algorithms—there is already plenty of work on this in the literature. Rather, our emphasis was on the *engineering* of relatively well-understood algorithms within a parallel programming context. This allowed us to focus on methodology, life cycle concerns, and tool/technique usability.

The first problem chosen for study was parallel sorting. Sorting is a fundamental computing problem invariably present in any large C^2, or information systems application. In a previous study done for Rome Laboratory, Computer Sciences Corporation identified such C^2 functions as pattern analysis and database maintenance where sorting is a potential component [Lightfoot, *et al.*, 1990]. Since sorting in the sequential case is so well-studied, we were interested in moving existing, sequential, algorithms to a parallel architecture such that any inherent parallelism could be exploited. The "porting" of sequential algorithms to parallel architectures is a form of parallelization which has been , and is likely to continue to be quite common.

The second problem chosen was parallel searching. Searching arises in many database retrieval and real-time pattern analysis functions. It is critical in large databases, in target and threat analysis, and in weapons management and assessment activities. [Lightfoot, *et al.*, 1990] document its use in more than half of C^2 functions. In a realistic environment, searching must take place where there are possibly multiple readers while a writer is

updating the search space with new data. We chose this problem to investigate "cooperative concurrency" wherein there is a degree of explicit coordination — a style quite different from that implied by the inherent parallelism of the previous problem. Furthermore, the problem may be tackled at various granularities from single element locking to large-grain tasking.

Image warping enables two-dimensional data to be presented from a three-dimensional perspective determined by a possibly varying angle of visibility. It is useful in analyzing topographical data and managing the display of terrain data in modern C^3I battlefield support; it could also be employed for realistic simulations. Image warping is an example of a more compute-intensive applications than sorting and searching, and involves enough numeric and graphics issues to raise an orthogonal set of parallel programming issues. Image warping is naturally modeled on a SIMD machine—we implemented it on the Transputer, a MIMD architecture. Because SIMD architectures are highly machine specific, we were interested in seeing whether we could achieve a "target-independent" formulation of the problem. This experiment also enabled us to investigate communication and data granularity trade-offs. Lastly, the system was connected to an X display, allowing us to consider the requirements for synchronization necessary to yield a coherent image.

Appendix A provides additional descriptions of the three experiments.

4.2 Approach

Our overall approach was to carry out our experimental development activities on the prototype PEEP—demonstration of the efficacy of coordinated use of tools on a powerful workstation with access to various parallel hardware architectures was a primary goal of the prototype PEEP. Tools were selected to each cover a relatively specific part of the life cycle, or support function. Since this selection did not address the question of common functionality, a remaining area of consideration was common services that can be shared to enhance overall platform effectiveness.

Initial versions of all experiments were programmed in C and targeted to a 16 node in-SUN Transputer board attached to the PEEP SPARCStation. The Meiko CSTools were used for the first phases of work on all the problems. We were also able to target one of the problems to the Intel iPSC/2, but were not able to get any timing results due to space and memory limitations.

The tools acquired for the PEEP to be evaluated as a part of our experiments are listed below. Summaries of these and all the tools mentioned in this report are provided in Appendix B. The tools evaluated are divided into two groups: tools which we attempted to use for the experiments; and tools which we installed, or tried to install, but were not able to use for the experiments for various reasons.

Used for experiments:

CODE– parallel CASE tool (University of Texas, Austin)
CSN_Illustrate – visualizing Transputer configurations (Intermetrics)
CSTools – C and FORTRAN compiler system for transputer (Meiko)
Id – dataflow language (MIT)

*Lisp – SIMD version of Lisp (Thinking Machines)
PProto – parallel requirements tool (RL/ISSI)

Not used for experiments:
Crystal – language with special index set notation (Yale)
PAWS – performance assessment tool (RL/Syracuse University)
PCN – hierarchical graphical language (Argonne Labs)
POKER – parallel environment (University of Washington)
TANGO – support of algorithm animation (Brown University)

Recommendations regarding these tools are included in section 5.

4.3 Requirements Analysis and Definition

Within Requirement Analysis and Definition, one is concerned with defining the problem to be solved, and the constraints on the solution. The result of this activity is a requirements definition which may be analyzed for completeness, consistency, realizability and customer acceptability. The requirements definition determines the functionality of the system to be built. Insofar as possible, the requirements specification should be as free from design and implementation considerations as possible. So, in one sense, the issue of whether a problem is to be solved using a parallel architecture is irrelevant in Requirements. This argues *against* the need for particular, parallel architecture-oriented requirements tools. While the state of the practice in requirements tools and techniques is somewhat immature, most decent requirements techniques do not presume or enforce a sequential model of computation (e.g., SADT, IDEF, SREM).

Due to the limited scope of our investigations—three problems relatively well-defined by existing algorithms—we did not undertake anything which approached a formal requirements activity for the three problems.

However, we did use Id as a high-level, architecture-independent executable specification language. In the experiments reported, these specifications were only at the algorithmic level, but the same style of specification could be used more abstractly in the specification of a larger system. While we found the dataflow model embodied by Id to be a very architecture-independent form of expression, there are some drawbacks to the dataflow model which could be a problem for large systems. In particular, the lack of a notion of state requires operations to be parameterized by each object they operate on (see Id specification of Search in A.2). This defeats information hiding—readability—in requirements specifications, and (as discussed below) introduces communication problems in implementations. Id's dataflow model also does not permit the easy expression of communication requirements.

We had expected that *Lisp would play a similar high-level, specification role, but we had limited success using the *Lisp simulator provided by Thinking Machines as only a subset of *Lisp is supported. The *Lisp simulator is intended to be a learning tool, not as a serious development tool.

PProto is intended to support requirements analysis, but it was not used for the experiments as it was not available at the time.

4.4 Design

Design is generally broken into High-Level and Detailed (or, Low-Level) Design. In High-Level Design, the problems pertain to:

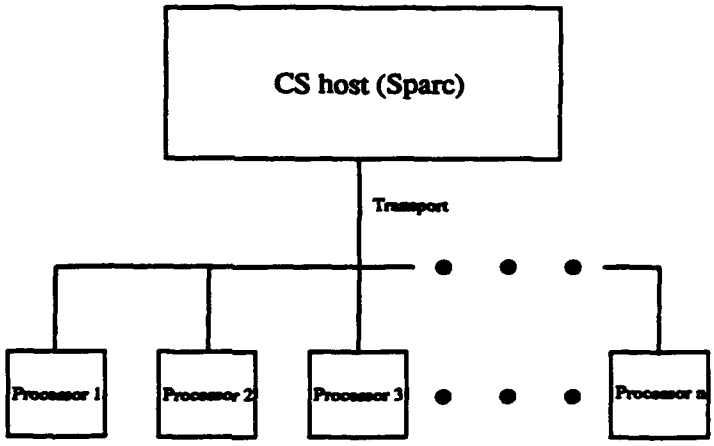
- Process Partitioning
- Data Distribution
- Data Coordination

In Detailed Design, the issues are:

- Algorithm Selection
- Data Type Determination
- Control Synchronization

The tools that were used during design are Id and PProto.

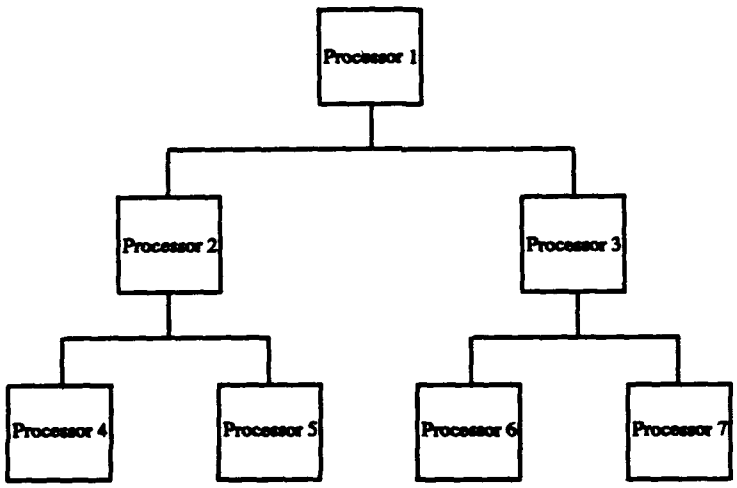
Our first experiment did not use any parallel design tools, as none were available on the system at the time. Instead we used what turned out to be a trial and error approach. The initial design for the sorting algorithm was to divide the data into as many chunks as there were processors available on the transputer, sort each chunk using a fast sequential algorithm on each transputer, and merge the sorted chunks. The communication between processors is shown in the diagram below.



The rationale behind such a design was that this would have the maximum number of processors working on the problem at the same time. The results of this sorting algorithm were quite poor as the partitioning of the data into chunks, and passing it out to the individual processors created communication bottlenecks.

We later turned to a dataflow approach to design. This was supported by the Id language and analysis tools. This involved demonstrating in Id (see A.1) that the quicksort algorithm itself had some inherent parallelism. We wrote the quicksort algorithm in Id, and ran the Id interpreter displaying a graph of parallel arithmetic operations. It was difficult to interpret the graph as there was no way to tell what was going on at any given point in time,

but it did seem to indicate that the quicksort algorithm itself could be exploited for some parallelism. This was implemented on the transputer, and the results were substantially better than the first sort implementation. The basic algorithm was to take the data and divide it into two lists, one greater than, one less than, a given component. The two lists are passed to two processors, while the first processor waits for their results. Their results are two sorted lists which can simply be concatenated to form the final list. This algorithm is repeated at each node until there are no more nodes, at which time the remaining elements are sorted using a sequential sort. This approach, shown in the diagram below, was not considered when designing the quicksort by hand because it appears to make poor use of the available processors.



At the point where the sequential sort takes place (the bottow row of processors) half of the processors are simply waiting for results, and only half of the processors are working on the problem. However, the use of Id as a design tool led us to this counter-intuitive solution which turned out to be quite effective as the results shown below demonstrate.

#of Item	2 processors		4 processors		8 processors		16 processors	
	1st Impl	2nd Impl	1st Impl	2nd Impl	1st Impl	2nd Impl	1st Impl	2nd Impl
80,000	172.6	9.0	149.8	4.7	143.7	3.0	157.3	2.4
160,000	342.8	18.9	304.3	10.0	276.3	6.0	296.0	4.9
320,000	689.8	42.5	611.2	21.6	548.9	12.6	562.7	10.0

We also tried to use the Id dataflow design approach to the search experiment. We were able to try out a number of approaches. At this time we received a new version of Id that allowed the performance graphs to use color in various ways. We were able to assign a color to each Id function in our program, and when the arithmetic operations graph was displayed, the graph was colored according to how may operations a given function was running in that cycle. This color coding of the graph was very helpful, and also served to show us that often our interpretation of the graphs had been wrong. In the black and white graphs we would try to associate a peak in the graph with something we knew was happening in the algorithm. When we used the color graphs we would discover that a

function quite different from the one we were expecting was contributing to the peak in the graph. Thus, the color version of Id is much more useful than the older version.

A drawback of the Id design approach with respect to the second experiment was the fact that as a very fine grained approach to parallelism, it did not really help with our goal of investigating synchronization issues. Id performs its locking at a very low level, so the type of explicit synchronization that one might need in an Ada implementation, for example, are missing in the Id implementation. Note that this does not necessarily mean that an Id specification is not a good way to start, but just that it did not support our goal of investigating synchronization issues. In general, Id provided a good way to investigate algorithms, but it is not designed to provide any kind of feel for what the communication costs would be once it was implemented in a non-dataflow language on actual hardware. Unfortunately, in all three experiments the communication costs were a large factor in the poor performance of early versions of each program. It appears that communication costs are something that software engineers inexperienced with parallel programming have difficulty predicting.

One approach we tried for investigating communication costs was to use PProto. PProto is not intended for use as a design tool, but since it should help with analyzing the feasibility of requirements, it appeared that it could help to predict communication bottlenecks. This analysis actually took place after having implemented the image warping algorithm a number of ways for the transputer. However, our premise was that if PProto had been available earlier it would have saved us from implementing some of the image warping versions by predicting the communication bottlenecks, and prompting us to investigate other approaches. In prototyping the first version of the image warping algorithm, which suffered from a severe I/O bottleneck, PProto did predict some of the communication costs. We were also able to analyze what would have happened if more processors were available.

While PProto was generally useful, there were two places where PProto could have provided more functionality. In order to prototype the image warping algorithm we first had to model the transputer board. This was quite time consuming, and PProto would be greatly enhanced if it had more architectures built in than it does now. This would probably help during requirements analysis too; particularly if a specific architecture is a requirement. Additionally, the level of hardware characterization currently supported is not detailed enough to base a hardware decision on. Possibly, if PProto had better hardware characterization facilities, it could be used for deciding which machine to buy.

The other place where we ran into problems with PProto was when we wanted to create a variation on an existing design. This was a problem at two levels. First, no way to create a copy of a whole existing prototype was identified in the PProto documentation. As a result, the only way we could create a variation was to either change the existing prototype, or create a new one, and duplicate all the work of creating the original. Secondly, creating the new variant was harder because the facilities for copying existing components is limited. For example, when prototyping the transputer architecture most of the nodes are almost identical. It would have been easier to create different variants of the transputer with different numbers of processors if it were possible to duplicate the component representing the node.

Overall, however, PProto was successful at helping us identify the communication bottlenecks, and in analyzing performance with different numbers of processors. It would also have saved time in the implementation phase, since it would have been easier to try the different communication strategies, which were actually implemented, in PProto, and then to implement the best of them.

We had originally felt that selecting a parallel programming language would be an important part of the design process. With the current state of parallel programming languages this turned out not to be the case. In general, the choice of actual implementation language would be limited to one of a few conventional languages, probably C or FORTRAN with parallel programming extensions (such as Linda), or possibly Ada. The languages designed for parallel programming, such as Crystal or Id, were not available at all on actual parallel machines, or only on a limited number. While due to logistics, this was not a concern for this project, in the future, the choice of implementation language may well be an important design decision. However, there are no tools currently that support making this choice.

We had originally identified a number of other tools as supporting design including, among others, POKER, CODE, and Faust. Upon closer investigation, these tools appear not to support the design phase so much as the implementation phase. They are concerned mainly with making it easier to compose parallel programs. Both CODE and POKER require that some amount of real code exist in order to perform as intended. It is possible that these tools could be used for some aspects of design, such as identifying shared variables in CODE, but code would have to exist for the rest of the program.

One last point should be made when discussing the tools that support the design phase. None of the tools produce anything that could be used as the basis of a design review. One drawback of this is that while the final design may be captured as a prototype or executable model, the decisions leading to that design are not captured anywhere.

4.5 Implementation

During implementation, target-specific programs are developed, based on the detailed design as input. Most module, algorithm, and data structure decisions should have been made in Detailed Design, so what remains to be done in this phase is the transformation of these structures to a programming language appropriate to the target architecture. Depending on the facilities of that programming language, this transformation may involve more or less effort, and considerations such as portability, scalability, etc., may be more or less supported.

Most of the tools surveyed fall into this category. Unfortunately, a lot of them are also specific to a particular language and target machine. It is much easier to design and build a tool or suite of tools with a particular language and target machine in mind. However, the intent of the PEEP is to provide a platform where a programmer can experiment with multiple languages and multiple target architectures, so we concentrated on tools that purported to support multiple languages or to be architecture independent. Debuggers are almost always very target dependent, so no debuggers were evaluated.

The tools used for implementation are CODE and CSTools, although with limited success in the case of CODE. Other tools discussed, but which were not used for the experiments are Crystal, PAWS, PCN and POKER.

CODE (version 1.2) supports three languages, Ada, C and FORTRAN, and a number of target machines. We tried to implement the first experiment in C with CODE. This should have been a good match because CODE supports a graphical dataflow representation, and the quicksort specification was in Id, a dataflow language. One could even imagine being able to automate the translation between Id and CODE to some extent. Unfortunately we ran into a few problems with the CODE implementation. First, the support in CODE for replicating code sequences did not work, so we could not implement the quicksort as specified. We continued experimenting with CODE, but using the examples that were provided as tutorials instead of our own sample programs. In this way, we were able to generate actual code that could have been compiled on a parallel machine. In order to produce code for a particular machine using CODE, a TOAD (Translators of a Declaration) for that machine is needed. We were not able to actually compile and run any of the generated programs because the set of TOADs provided with CODE did not include any of the parallel machines to which we had access (iPSC/2, warp systolic array, CM-2, Multimax or Alliant). We also tried generating an Ada version of the tutorial. The code produced looked correct, but relied on rendezvous very heavily which could result in poor performance, depending on the particular Ada compiler used. On the other hand, this was the most portable approach and should work with any Ada compiler. On the positive side, we were able to get support from CODE's developers, and in fact these problems should be fixed in CODE 2.0 which is about to be released. CODE 2.0 will support the replication of nodes and subgraphs, and has improved its Ada generation capabilities. It should also be noted that the type of mutual exclusion being implemented with Ada rendezvous could be enormously simplified if implemented using the Protected Type introduced in Ada 9X. Obviously, this will not be available in the next version of CODE, but is likely in future versions of CODE after Ada 9X is approved.

We also tried to use POKER as a coding tool. POKER supports simulation of a variety of non-shared memory MIMD machines. The user enters the parameters for the machine indicating the number of processors and the connections between them. We did not actually use POKER because it could not be installed on the SPARCStation. Based on reviewing the documentation POKER appears to have some capabilities similar to CODE. Like CODE, it can produce output that can be compiled on a limited number of target machines. It also supports a tracing facility for debugging, and a profiler for simulating execution times. In terms of languages, POKER only supports it's own dialect of C, and a simple sequential language called XX.

PCN was received too late in the project to attempt to implement any of the experiments. However, it was installed successfully. It includes a number of interesting features that can be described. PCN is a language and a set of tools. One interesting aspect of PCN is that it provides interfaces to FORTRAN and C that allow existing code to be reused. It also provides a debugger (PDB), an execution profiler (Gauge), and a trace analyzer (Upshot). From reviewing the documents it appears that PCN would support initial debugging and analysis of the program on the SPARCStation prior to moving the program

to a parallel target, or a network of workstations. Of particular interest is the fact that PCN supports a program running on a heterogeneous network of machines.

Crystal was also investigated as an implementation language. Our experience with this language indicates that it has a very steep learning curve. Since only a SPARCStation based interpreter is available (a compiler targeted to the iPSC/2 is being developed), we concentrated on other tools and languages.

While PAWS is not a tool that produces an executable that will run on a particular parallel target, it does provide the ability to compare how the same code would run on three different machines representing three very different classes of architectures: the CM-2, the Multimax and the iPSC/2. We did try to run an Ada implementation of the second experiment through PAWS to see how it would run on the different architectures. Given the nature of that experiment, we would have expected the program to map more naturally to the Multimax than to the Connection Machine. Unfortunately, PAWS was not able to process all the Ada constructs used in the program, so we were not able to get complete results. In general, it seemed as though the ability to compare different architectures would be more useful earlier in the development process, such as during design. One place where PAWS would be useful is in determining an appropriate parallel target for an existing sequential program that is about to be parallelized.

Faust was another development environment for parallel programs like CODE and POKER. It supported C and FORTRAN, and a number of parallel targets. Faust initially appeared very promising as it included an integrated editor, a parallelizing compiler and a program monitoring tool for performance analysis. If it had one drawback, it was that while it had advanced support for vectorizing optimization, it had little capability to deal with programs that are parallel at a higher level. In the end no experimentation was done with Faust because the project had been dropped, and the unsupported system that we received was incomplete, and impossible to run.

Finally, the most successful of our coding tools was CSTools from Meiko. It should be noted, however, that this is a commercial tool, and most of the preceding ones are not; it would not be fair to expect the University supplied tools to meet the same standards of quality as a commercial product. CSTools supports a number of languages, of which we had C and FORTRAN. It is limited to a single target, the Meiko transputer board. We were able to implement versions of all three of the experiments using CSTools. We did run into a few minor problems. The Quicksort algorithm is normally expressed recursively, and the first implementation followed this design. As a result, the transputer processors implementing the final sequential quicksort overflowed their stacks when the recursion levels got too deep. This was not handled very well by the tools, and it took some time to determine the actual cause of the failure. In general, the error reporting on the transputers could have been more helpful.

We were also able to run a C version of the quicksort on an iPSC/2 at Cornell. This was based on the code developed using CSTools for the transputer. To retarget the program to the iPSC/2 the calls to transputer specific library routines needed to be replaced with iPSC/2 specific library routines. Fortunately, the C libraries for the two targets are similar.

It was also the case that most of the coding effort went into developing the first version of each experiment. Subsequent versions of the program were developed in much less time. Usually, variations on the original program involved simply making a few changes, and then rebuilding and running the program to measure the new results.

4.6 Testing and Integration

There were no tools that supported testing. Testing for all three experiments consisted of hand-writing tests and simply running them. Also, the nature of the experiments did not allow any real evaluation of the tools' integration capabilities. In general we would assume that the engineer would want a tool that made it easy to bring together independently developed and tested pieces of a system. Each of the experiments was designed to be stand-alone so there was no integration work to be done after it was tested.

No tools for performance analysis were received in time to use them for analyzing the experiments. However, there are two tools in this area that are still worth looking at: PCN and IPS-2. Of these, PCN is probably the most promising as it does run on the Sun4 workstation. It includes both an execution profiler, and a trace analyzer. IPS-2 was not available for the SPARCStation, but it would appear to be a worthwhile addition to the system. The key feature of this performance measurement tool is the ability to display measurements at different levels of abstraction. If running a program on a large number of parallel processors, the user could easily be overwhelmed with the amount of performance data if there isn't some way to filter it. If it were ever ported from the VAX to a Sun host, it would be worth investigating on the PEEP.

4.7 Supporting Technologies

This section introduces a number of supporting technologies which are useful during more that one phase of the development process. Some of the tools mentioned above support some aspect of these technologies. We also mention a few additional tools here that support a specific technology without being specific to one phase of the development process.

4.7.1 Prototyping

Prototyping can be used through out the development process. During the requirements phase a prototype user interface could be mocked up to see if it meets requirements; during design, prototypes help the designer make trade-offs between different approaches. In particular, PProto lets the systems engineer experiment with different process decomposition and data partitioning strategies to see whether the requirements can be met, and if a particular strategy is better than the others. In some ways, the Id specifications of the problems could also be looked at as prototypes of the actual program.

4.7.2 Modeling, Analysis and Simulation

Most of the tools support simulation is some form. PProto simulates running the program as specified. The Id interpreter simulates running the program on an idealized

dataflow machine. POKER supports interfaces with a number of machine emulators. The most important aspect of being able to simulate a program is the ability to make comparisons. A particular simulation may not be able to predict the performance of the program on a specific machine, but it should be able to predict the relative performance of two simulations run using the same simulator. It is also important to have a simulator (or simulators) that simulate all the aspects of parallel programs. For example, the simulation capabilities of Id alone are not useful for predicting communication costs on a message passing architecture. It is useful to have both types of simulators, that is, both actual performance predictors like PAWS, and relative performance predictors like PProto.

4.7.3 Visualization

CSN_Illustrate supports a limited aspect of Hardware and Software Integration well, and shows how simple tools can improve development in this area. CSN_Illustrate simply shows the layout of the transputer network, and the mapping of processes to nodes. This is useful for determining the topology of the network, and determining if a particularly long communication path exists between two nodes that communicate extensively. Intermetrics implemented CSN_Illustrate because the report generated by the CSTools program builder contains so much information it was difficult to analyze. CSN_Illustrate presents the same information graphically. However, this is also the drawback of CSN_Illustrate, in that it is dependent on the CSTools report.

TANGO is a tool that supports algorithm animation. For this project it was ported to the transputer. TANGO had been used to show graphically how a particular algorithm, like a binary search, worked. Since all TANGO does is display events as it receives notification from the application, it appeared that it would work for parallel algorithms too. Unfortunately, the nature of the program tends to sequentialize the algorithm when it is animated. For example, two nodes are performing a process which in the animation is represented as moving a square to the right. In the display first one square, then the other will move, rather than moving both at once. As a result TANGO is not very useful for visualizing parallel algorithms.

4.7.4 Configuration Management

One aspect of software engineering that is missing from all of the tools is configuration management. In some cases this is not a problem because configuration management can be provided outside the tool. For example, a C source program for the transputer board can be kept under RCS or SCCS (UNIX revision control systems). Some tools would be much more helpful if there was an integrated configuration management system. In particular, it would be useful to be able to track variations of a program, and to be able to correlate a specific change to performance improvements in a specific area. It is also possible that while recording the textual changes between two files, an external configuration management system would miss a higher level change, such as recording what standard transformation caused the textual changes.

4.8 Discussion

No software methodology can be evaluated without usable tools to make experiments in semi-automatic processing possible. The quality of tools might to a first approximation be expected to depend on the amount of research effort expended. The highest usability was found in parallel languages, which are traditionally a prolific research area. Use of multiple languages is likely to produce meaningful results by testing experimental hypotheses that programming an algorithm in the right language or class of languages would enable the algorithm to meet some requirements or system objectives more effectively than others. In practice this has not been a factor. Genuinely new parallel programming languages are being developed, but are only available as interpreters or on one particular specialized hardware. A particular language like Id or Crystal can be used to prototype the problem, but in order to produce a system that will run on a useful target hardware, a dialect of one of the standard high level languages (Ada, C, FORTRAN) is most likely to be used.

Another active research area that has similar promise is compiler technology adapted to allow the detection and use of intrinsic parallelism in sequential languages. The kind of experimental hypothesis to be tested on these kinds of tools would be that some class of target environments could be successfully and effectively utilized by programs in some set of source languages compiled with each tool. Unfortunately, no such environments have been successfully ported to the PEEP so far.

CASE tools and performance analysis tools are slightly less active research areas. Some tools have been found partially useful, but incomplete due to less research activity. CODE from the University of Texas at Austin supports a few "ordinary high level" languages (Ada, C, FORTRAN) and compiles to several architectures via a special translator (called a Translator Of A Declaration, or TOAD) for each language/target pair. It provides an important capability, target architecture independence, that is essential for any parallel software development capability. CODE also supports a well engineered graphical user interface that allows the sequential segments of a problem to be stated in a way independent of communication and synchronization mechanisms. It thus supports parallel programming with MIMD flexibility and SIMD simplicity. CODE has a number of limitations, particularly in the ability to control synchronization code effectively, but there is a new version soon to be available that promises to improve this situation. It is an example of a promising tool with enough limitations currently to be usable only for demonstration problems.

4.8.1 Tool Evaluation Summaries

CODE

Only partial results could be obtained with CODE. Small programs similar to the supplied CODE demos could generate complete programs, but none were executed due to not having the right target hardware. Sort and search code could be produced (but without automatic replication, due to a bug already mentioned), but again could not be executed. The new version of CODE, CODE 2.0, should fix most of the problems. Another favorable aspect of CODE was that we were able to get some support from the developers, although

they were not able to send us any additional TOADs. We did not evaluate ROPE, the reusable component library associated with CODE. If it works as documented it does appear to be a useful facility for finding reusable components by searching for keywords in the description of the component.

CSN_Illustrate

CSN_Illustrate is a very simple tool for graphically displaying the actual configuration of the Transputer for a particular application. It is dependent on the format of the Meiko tools configuration report; and had to be modified when the report changed in a new release.

CSTools

CSTools are a commercial parallel programming toolset for the transputer. It includes a compiler, program builder, and debugger. These tools were quite effective for developing programs for the transputer. Other commercial environments, such as EXPRESS, were not available for comparison.

Id

Id capabilities improved over the course of our evaluation. Initially, it was difficult to interpret the parallelism graphs because separate activities (such as insertion and lookup in searching) could only be distinguished by reprogramming. The addition of a capability for associating colors with Id functions in the first Monsoon interpreter release in mid-1992 solved this dilemma. Now the interpretation of sources of parallel activity in parallelism profiles is done easily.

An issue with the use of Id is its user interface, which can be using the shell, but for debugging has to be under EMACS using Lisp commands. Both debugging and visualization were a problem because any execution failures produced a Lisp interrupt. For anyone familiar with functional languages who has time to learn a few Lisp library routines and is willing to read interpreter code, this is not too much of a problem. The Id development team at MIT was quite supportive in helping us install the system and get over initial learning hurdles. The documentation that accompanies the language also improved with the new release.

**Lisp*

Thinking Machines has a *Lisp simulator freely available. We had hoped that the simulator could be used to develop a *Lisp program on the SPARCStation that would then run with little or no modification on a Connection Machine. Our experience with the simulator was that along with being undocumented, it only implements a subset of the language and provides only a subset of the standard libraries. As such it is not possible to use it to develop programs of a realistic size.

PProto

With PProto, modeling of the image warping algorithm verified the communication issues observed in the code-level Transputer experiments. The graphic display of communication paths showed the bottlenecks that led to serialization of the graphic communications in the series of experiments. There were some difficulties with modeling the Transputer architecture due to the lack of enough database and library facilities to develop the model incrementally. The model had to be read in from source every time, and there was no way to distinguish additions related to prototyping different variants. This is a serious problem for a tool that is supposed to support rapid prototyping, since it is easy to lose information and even confuse the meaning of results without clear derivation mechanisms. This is partially related to a technique such as Meta-Crystal for expressing specific transformations, but shows that more general transformational derivation approaches are important for really doing rapid prototyping effectively. This problem is not specific to parallelism, and projects supporting this kind of refinement such as E-L at Harvard are very important for parallel software development. It is possible that the difficulties were related to the fact that the tool was only available for two weeks, and we were not familiar with its capabilities.

Issues of human interface and capacity were enough of a problem that the utility of the tool might be compromised for more than demonstration problems. With PProto, the slowness of rebuilding and executing successive experiments was aggravated by poor memory usage, and no way to clean up between partial builds means that the interpretation of successive experiments is more difficult. Whether this would affect the validity of results might require some expertise in the structure of the PProto interpreter, at least in the worst case or using large models. Another drawback to PProto is the fact that it relies the ONTOS object-oriented database system to run. This does not come with PProto and is not generally already installed on a user's system.

Crystal

This language was not fully evaluated. We have a working Crystal interpreter on the SPARCStation, but the compiler that is being developed was not available during this project.

PAWS

For PAWS, only the sample programs supplied with the tool successfully exercised the tool. However, this was to be expected as only the alpha version of the tool was available at the time. The user documentation for PAWS could also use improvement. At least one environment variable necessary to run PAWS is not documented. A beta version of the tool is now available, and work continues to add functionality and make it more robust.

PCN

The PCN system did not become available until the end of the project, so no experiments have been tried. However, its documentation is quite complete and indicates that it is available for a number of platforms. It is definitely worth further investigation.

POKER

POKER was not fully evaluated as it did not run on the Sun4 workstation. It also appears that no further work is being done on it, so it is not worth further investigation.

TANGO

Attempts to use TANGO calls in Transputer programs resulted in output that was not capable of being used to display parallel activities on the target hardware. TANGO was designed for animation of sequential programs, and would need additional support for identifying parallel activities at the lowest level of data collection.

Other Tools

Interesting FORTRAN tools, such as PAT and FORCE, were tracked down and determined to be available, but not pursued for installation. Free distributions of these are available, and instructions for getting them are included with the PEEP.

A demonstration version of BALSA II, another animation tool, was also tried. BALSA II is even more restrictive than TANGO in that it only runs and animates programs running on the Macintosh.

The experiences with Faust demonstrate a problem of interest for PEEP planning purposes. The problem is acquiring and installing tools, where the issues are first getting access and then verifying completeness of the delivery. Acquiring Faust itself was delayed for licensing reasons and was finally acquired almost two years after the start of the PEEP effort. When Faust did arrive, there were no installation instructions (or documentation of any kind) and support only for reading the tape. PEEP installers tried to understand how to get it running by experimenting with it and looking at source code. However, the source code did not seem to be complete so the effort was abandoned.

Inquiries were made on availability of other potentially interesting tools at the beginning of the prototype PEEP project. These included SISAL, STRAND-88, DINO, RAPIDE and Proteus, which are parallel research languages that may be useful in the future. Other tools with good concepts such as Hypertasking, IPS-2, TOPSYS and Pisces are not available on Suns, though all might be useful if they were eventually ported. TOPSYS is a parallel environment with special support for monitoring that would be likely to be immediately useful if it could be integrated successfully.

An important aspect of tool usability for tools having some partially successful results is the quality of description, documentation and support. Good support is essential to any useful software tool, and includes many kinds of documents: installation manuals, user

guides, tutorials and reference manuals, as well as consulting for problems encountered in initial use and longer term exercise. This is not a level of support normally regarded as feasible for university projects, but has been attempted by a number of the tool projects we dealt with. The good news is that many tried to include some effort in this direction. The bad news is that the results were mostly unsatisfactory.

Many tools have been discontinued. This includes some of the pioneering efforts in parallel environments, such as Prometheus and PIE, which were always thought to be important primarily for their influence on later work. Simulators, for example PARET from Bell Labs, are very important in rapid prototyping and studying relative costs of processing and communication elements, but tend to be used for the project that develops them and not to get enough support to last beyond that project. Molecule was an important project to be discussed in more detail in the methodology section. It was a classic research project, in that, it successfully demonstrated a useful methodology and was completed with no perceived need for any continued existence.

4.9 Methodology

If we look at the problem solving process for each of sample problems, the experience reveals as much about the effect of traditional software engineering techniques as any theoretical analysis. The important practical problems were problem partitioning, understanding variants of sample algorithms, and combining compatible tools to use the capabilities of each in describing a sensible development path.

4.9.1 Problem Partitioning

Both the sorting and searching experiments had the same problems meaningfully partitioning the problem. For sorting, one issue was how to relate processors to parts of the algorithm, as solved by the binary tree arrangement of the processors. For both problems, the question of how to partition the input effectively was solved only after some initial work was done to understand the communication costs. This is typically a goal of simulation studies. Even had a simulator been available the modeling could easily have taken as much time as the communications coding; still once code was available, additional experiments could be performed more quickly by local modifications of the existing programs. It is worth noting that the second implementations succeeded quickly partly due to this kind of reusability. The methodology this reuse seems to relate best to is that of CODE, involving a database of sequential program segments that can be connected under programmer control by data flow requirements and synchronization constraints.

4.9.2 Combining Id, Crystal and CODE

All three problems had some of the same methodological difficulties. While useful in verifying ideas about intrinsic parallelism, the Id programs provided no insight into the "real problems" of organizing the computation to control communication and synchronization

costs. This is where a combination of Id, Crystal and CODE will be the most help. Once an algorithm is demonstrated in Id, it can be transliterated to Crystal and decomposed using domain mappings into "agent" subproblems. This mapped program can then be input to CODE to separate sequential activities from synchronization methods. Then TOAD's will translate the code to the appropriate target architecture.

Actually generating the code for any of the applications by this process is still only possible in principle. While the Id programs can usually be demonstrated to work essentially as expected, the process of functionally compiling them by Meta-Crystal like transformations is going to have to be done by hand for the foreseeable future. Meta-Crystal, still under development, is not available for even initial evaluation, and cannot therefore be expected to be of production quality, as the experience with other tools demonstrates. Transliterating the iterative code to concrete languages such as Ada, C and FORTRAN is not especially difficult. Finally, CODE itself is still in a process of evolution and may be approaching the level of code development capabilities necessary for continued use. CODE 2.0 will be used for project work in higher level classes at the University of Texas this fall, and should be available shortly afterwards. Also, TOAD's are not completely trivial levels of translation, but can generally be developed by graduate level people in a few months for a particular target and run-time system.

This approach is basically the same as the Molecule paradigm. Molecule was a language project where the key feature of the language was the ability to specify the program at different levels of abstraction. In the highest level of abstraction, the program in expressed in target-independent dataflow semantics. In the next level of abstraction the program is expressed in terms of a set of primitive parallel operations that are appropriate for a particular type of architecture. Finally, the program is translated to a specific source language for which there is a compiler on the target hardware, such as C for the iPSC/2.

4.9.3 Surveyed Projects

The original expectations about these projects were actually quite limited. Typically university researchers are very happy about results that demonstrate the validity of an approach or the utility of a tool concept and organization and do not regard it as within their province to worry about the viability of particular implementations. A large number of projects included in the original survey of available tools and techniques were exactly of this nature.

A typical one was the Molecule project, which demonstrated the feasibility of beginning with dataflow level notations and using fairly well understood transformational and compilation techniques to produce executable code. This was the paradigm that seemed most like existing software tool and support methodology in the sequential world. What a casual reading of the literature of work on traditional languages and development environments suggested was that the Molecule paradigm was a practical possibility, using mostly available software technology. Evolutionary language and compiler approaches that supported aspects of this model were the FORTRAN PICSES and FORCE methodologies.

Other examples were the evolutionary variants of traditional sequential languages and run-time support techniques from Ada and Concurrent Pascal (and various Concurrent C's) to FORTRAN 90 and Schedule. From the environment world, R^n and Faust initially seemed to demonstrate that enough parallelism could be detected and incrementally improved upon by combinations of programmer supplied, static, and dynamic analysis to make the gradual improvement of essentially sequential techniques a practical alternative. Again, it turns out R^n and Faust have been primarily prototyping techniques and the engineering of real development environments on this basis has been essentially discontinued.

5 Conclusions

In this section, we summarize our findings and conclusions. These are organized as PEEP requirements, recommendations for the evolution of the PEEP, and a vision of a next generation PEEP system.

5.1 Requirements for the Software Engineering of Parallel Systems

This section is organized into life cycle requirements and requirements on supporting technologies, following the structure of the revised tool/problem solution matrix (section 2).

5.1.1. Life Cycle Requirements

Our survey and experimentation have led us to identify the following requirements for any system supporting the software engineering of parallel systems. Capabilities needed for parallel software engineering, based on our Survey and Experimentation include these:

- architecture-independent program notation for problem specification (e.g., using a dataflow language such as Id),
- ability to interpret programs to determine processing and communication costs, as well as to verify correctness of algorithms and adequacy of synchronization methods (e.g., using a simulator such as PProto),
- architecture-dependent evaluation methods to determine behavior on different architecture classes; this can include languages supporting various classes of processing and communication operations, as well as support for synchronization at various granularities (e.g., using appropriate software development techniques to develop and execute various algorithms on appropriate hardware, as partially supported by CODE/ROPE, or using architecture simulation as supported by PAWS),
- ability to relate programs in different notations to allow all kinds of architecture and target machine dependencies to be tried for programs implementing the same algorithm (e.g., as done by Meta-Crystal operating on Crystal, or as done by human beings using more conventional tools),
- determination of algorithm speedup as a function of number of processors (e.g., doing evaluation on multiple targets directly),
- ability to evaluate performance by measuring execution without stopping or interfering (e.g., as attempted with TANGO, and as supported by event-based debugging techniques such as MDP and Dalek).

These capabilities directly address the various phases of software development as follows:

- Requirements Analysis using iterative specification and simulation techniques such as Id, and rapid prototyping for requirements verification,
- Architectural Design using refinement and transformation methods, as well as modeling and interpretation techniques that compare multiple execution such as PAWS and PProto,

- Low-Level Design and Coding using multiple languages and capabilities such as CODE/ROPE; and target support software for organizing, compiling, building and testing architecture- and target- dependent code and modules,
- Visualization and Performance Analysis using tracing and monitoring techniques such as those supported in modern parallel debuggers, and in PCN's monitoring facilities.

The remaining areas of the Revised Matrix are covered by tools that are not really unique to parallel systems and hence were not dealt with in connection with the PEEP. A few areas, such as completely non-interfering monitoring and automated system selection involving tradeoffs of hardware, software, OS support and run-time libraries, are problems that are currently beyond the state of the art and cannot really be dealt with under the PEEP philosophy of using existing tools and capabilities. As research progresses in monitoring and abstract modeling techniques (e.g., PAWS-style modeling at multiple levels of interpretation), support for these capabilities should be incorporated into the PEEP. Similarly, once more languages are generally available on parallel machines, language selection will merit further automated support.

For the purpose of prioritization, we analyze capabilities into three categories: *essential*, *recommended*, and *nice-to-have*. An *essential* capability must be present—one can't develop a program without it. Language translators (compilers and linkers) fall into this category. *Recommended* capabilities provide a marked improvement to the parallel software development process, but are not essential to development. A *nice-to-have* capability may be useful for a specific problem, but is neither essential nor generally recommended for parallel software development.

As the experiments on the Transputer attest, none of the tools we investigated are really essential. The support for parallel programming provided by the compiler/linker/debugger toolset of the Meiko CSTools, and similar vendor support software for parallel computers meets basic requirements. There are many limitations, such as only a few available languages, insufficient buffering capabilities for effective communications, or uneven support for various synchronization mechanisms. However, the lack of other capabilities can be compensated for by more programming effort. Without additional programming tools, parallel programs will be developed through trial and error as was done in the initial experiments.

- *Recommendation: Other commercial tool sets, such as EXPRESS, should be systematically evaluated.*

Due to budget considerations, other commercial tool sets and languages were not evaluated as part of this study. Providing alternatives to CSTools should be a goal of the PEEP, and any alternatives will need to be investigated prior to inclusion on the PEEP.

Thus, most parallel programming tools as investigated by the PEEP are rated as *recommended*, though in varying degrees. Parallel languages and compilers capable of optimizing parallel loops are *recommended* because they support software quality factors such as portability, reliability and maintainability without sacrificing efficiency (assuming

that the same language is available on multiple targets—not currently the state-of-the-art. In the future, Ada 9X may help in this regard, by providing a language standard that incorporates support for concurrency and distribution.)

Run-time library capabilities, such as micro-tasking and broadcast facilities, are also at this level of importance. Once a few languages supporting concurrency and basic communication mechanisms become available, more languages become just *nice to have*. Similarly, once an adequate development method is supported, more tools provide a kind of convenient redundancy that does not significantly improve the power of the platform. Parallel debugging and the current level of visualization support as provided by some vendors are tools of this kind—any one tool will suffice. Almost all the other tools considered for inclusion in the PEEP are also of this kind. This includes additional languages and all the various tools for modeling and identifying parallelism that were not fully evaluated.

We conclude that the capabilities of PProto and PAWS are *recommended* since they are unique in addressing modeling issues related to rapid prototyping and architectural comparison. In addition, some sort of high-level data flow specification language would also seem to be needed. Of the languages we tried, Id was quite successful in this regard, although another dataflow language with similar support would probably serve as well.

Thus, Id, CODE, PAWS and PProto are *recommended* techniques. A tool supporting transformation techniques similar to Meta-Crystal is also *recommended*, as is sufficient monitoring capability to do basic performance evaluation. More languages and tools to assist in such activities as visualization, design data capture and transformational configuration control including derivation histories are *nice-to-have* (though there are some researchers who would claim such facilities have always been essential). Ordinary *nice-to-have* tools are the myriad small tools that can help with documentation, verification, description, and measurement of software. One example is the CSN_Illustrate tool developed for use with the Meiko tools to depict code usage on the Transputer. The following table summarizes the conclusion for each tool:

Tool/Language	**Conclusion**
Ada 9X	Recommended
CODE/ROPE	Recommended
Crystal	Nice to Have
CSN_Illustrate	Nice to Have
CSTools *(or equivalent)*	Essential
Faust	Not Recommended
Id	Recommended
*Lisp	Nice to Have
PAWS	Recommended
PCN	Recommended
POKER	Not Recommended
PProto	Recommended
TANGO	Nice to Have

- *Recommendation: Work on PProto and PAWS should continue, and production quality versions of these tools should be integrated into the PEEP*

These two tools fit well with the methodology we have proposed, and there are very few efforts in the research community that offer comparable capabilities. We find these two tools to be of relatively high quality in reliability, usability and documentation, as compared to the spectrum of University tools we evaluated. They are not robust, production-quality products, however. We believe that bringing these tools to a state where they can be reliably used for large-scale programs with a minimum of support from the developers should have as much funding priority as further research on the underlying technology.

- *Recommendation: New developments in CODE, Id, Ada 9X and PCN should be followed for new developments and integrated into the PEEP as necessary.*

The field of parallel computing is expanding rapidly. It will be important to integrate new tools into the PEEP for evaluation as they are developed. Of the tools evaluated, Id and CODE are two projects where the work continues, and is developing interesting technology. Of the languages Ada 9X and PCN should be followed. With Ada 9X, the Ada language is improving its support of shared memory multiprocessors, and adding capabilities to provide support for distributed, or message passing architectures. PCN has some of the more advanced support for program monitoring, and good support for integrating existing code.

5.1.2 Integration Requirements

For the prototype, we have limited the amount of "wrapping" required to incorporate a new tool into the PEEP. Tool integration has been aided by the open systems framework provided by UNIX's process model and X's client-server model. Future versions of the PEEP should maintain this open systems' strategy, recognizing that the frameworks will evolve beyond what is available today.

In the area of common services, the object-oriented programming model has identified promising processing techniques in such traditional activities as databases and language processing. Concurrent development consisting of related processes cooperating through standardized interfaces is inherent to object oriented techniques as well as to many aspects of CASE technology. Recent trends in software development environments toward what are generally termed software process models indicate a possible direction for future PEEP growth. Taken together, all these kinds of research can be thought of as suggesting ways to improve compiling technology, to facilitate common tool interaction, or to make use of object oriented process control notations. Which technique or point of view would be best for the PEEP is the long term issue; achieving effective tool interaction in some way for the short term is also a serious problem.

Within this evolving framework, it should be possible to increase the degree of tool integration by defining uniform data and event formats.

Another aspect of integration is Configuration Management. This is crucial both for maintaining product consistency within large development efforts, and for maintaining a program and its variants within exploratory programming . In using PProto, we observed that prototypes would be easier to build if there was a mature cut-and-paste facility. As we discovered, one often wants to replicate a given node or graph fragment to use elsewhere. However, this is hampered by the need to specify input and output ports on a node (at all times). It would be helpful if there was a way to specify absolute and relative ports (as in the cells of a spreadsheet). Absolute ports would always refer to the same place, even when a node is duplicated. Relative ports could refer, for example, to an adjacent node.

A Configuration Management system should allow users to control multiple versions of a number of diverse kinds of entities and the relationships among them. Entities may be at any granularity within the system (as the above example suggests).

Throughout the SEPA program we experienced a need for better automated configuration management support. Much of our work on the sample problems was approached by making slight variants on an initial program, re-building, re-testing, and re-measuring performance. This cycle was repeated many times for each problem. There were no suitable facilities for isolating the differences between successive versions of the programs, and correlating the changes with the modified test cases or new performance results. Meaningful relationships between the changes and the performance impact of those changes had to be established and tracked by hand.

- *Recommendation: A comprehensive configuration management facility be added to the PEEP and integrated with all the tools that support our recommended methodology.*

The "Artifacts" system developed for DARPA's ProtoTech program would make an excellent starting point. This system maintains a network of fine-grained arbitrary relationships between source programs, test cases, results, and any other data in the system, and allows the automatic triggering of arbitrary tools upon changes to any of the data.

5.2 SEPA Long-Term Plan

This section contains specific recommendations for PEEP evolution and recommendations for further work by Rome Laboratory in this area.

5.2.1 Tool Enhancements

To use existing tools, the following improvements are needed:

- The ability to map Id programs semi-automatically to decomposed subproblems. A tool to do this would consist of lists of data and control structure transformations together with pre and post conditions needed to apply them. A method of supplying additional information about program data and operations, as well as intended architecture, would be one of the main interfaces for this tool. The effect would be

to allow the user to program a formalism similar to Crystal to allow Meta-Crystal transformations appropriate to the possible architecture to be applied. The output would be a Crystal-like program with enough architectural specificity to be translated to a concrete language such as Ada, C, *LISP or FORTRAN without any actual target dependencies, but with enough architectural dependence to be compiled without losing parallelism.

- The ability to keep derivation information in ROPE. Information on the user-supplied characteristics and transformations used to produce the concrete code would be kept with the program in a ROPE-like library system.
- More flexible synchronization conditions in CODE to allow better synchronization using target compiler capabilities. CODE 2.0 is supposed to support guard expressions which will do this.
- More TOAD's for more targets, particularly to support C for popular MIMD machines. There are supposed to be ways to get these with minimal (two month) effort. These could be developed as needed.
- The ability of target linkers and run-time monitor facilities to support intelligent monitoring and tracing. These already appear to have been prototyped in the TOPSYS project successfully; Meiko and other hardware vendors appear to intend to support full monitoring and tracing eventually. It is also possible this could be added to PCN.

Tools to supply more life-cycle coverage would include:

- More languages, such as C* and Linda. Ultimately the goal could be to allow any parallel language to be used on the PEEP. These could be procured individually as needed.
- More static and dynamic analysis tools, particularly in the support of Testing and V&V. These could be taken from existing DoD supported efforts and tailored to the PEEP, or procured individually as needed.
- More advanced design tools such as Meta-Crystal and those supported by the DARPA Common Prototyping Language effort, including Proteus in the area of languages and E-L in the area of transformational design support.

The issue of appropriate and useful documentation for user friendly access to the results of applied research projects needs to be kept very prominent. If possible, there should be follow up on the Faust experience to determine if there is any way to avoid the loss of the results of such an interesting project. In particular, the Sigma editor and the IMPACT monitoring design using automatic probe insertion would be important capabilities in the long term PEEP if they could be recovered.

5.2.2 PEEP Evolution

- *Recommendation: The parallel software development methodology evolved during the course of the SEPA contract should be supported by a well-integrated set of robust tools, and used experimentally by organizations outside Intermetrics to validate our assessment of its value*

To do this, the tools must first be made more robust, and reliable, and should offer a consistent user interface. They should then be integrated into the proposed methodology. This could be achieved through the development of a composite Users Guide that presents the methodology and the tools in an integrated manner.

5.2.3 Sustain PEEP Technology Base

> *Recommendation: A mechanism should be created for capturing and preserving the software that results from University research in parallel processing. When a tool is procured under sponsored research, there should be requirements on the delivery of adequate installation instructions to get all components running, and on the existence of an introductory user manual with examples. These systems should be archived with all such materials.*

The goal would be to preserve sufficient information to allow both meaningful evaluation and estimation of the value of continued support or extension. One clear and very disappointing conclusion of our work is that much promising University software is being lost. Parts of the very promising Faust system were not included on the source distribution tape, and could not be located by the researchers who remain at the university. The Molecule project had ended and the software was "put to rest."

Most researchers see published papers as the primary output of their work, and these papers are properly archived and readily retrievable. The software they develop, however, is not stored or catalogued in any systematic way. Often, all trace of a potentially valuable software system is expunged from storage upon graduation or transfer of the individual researcher.

Although little of this software is of production quality, it is nonetheless a very valuable resource. It can be used for more detailed understanding of the ideas, for productizing by third parties, or as a source of reusable components.

We believe that it would be a major benefit to the parallel processing community if a mechanism were established for capturing and preserving these software assets. Simply indexing the software against the published papers describing the research would be a sufficient retrieval mechanism. What is needed is a centralized collection activity. Provisions should also be made in government research grants to support this archiving.

We do not propose to change the intellectual property rights to any such software. The existing legal system for determining rights and ownership is workable. It may require negotiation with universities, but this can be done. However, without assurance that the software still exists in a usable form, this negotiation process is not worthwhile.

We believe that Rome Laboratories could fill a very valuable role by providing such a central archiving activity, at least for the products of government-funded University research in parallel processing.

- *Recommendation: Further research should be encouraged in specific areas where the SEPA program has been unable to find adequate existing tools*

We believe that TOADs should be created for more popular parallel architectures using the CODE system. Better tools for the visual display (animation) of the dynamics of parallel programs in execution should be developed. Tools for testing parallel programs should be created that will handle the non-determinacy inherent in these programs. Tools for designing parallel programs (before coding and independent of prototyping), and analyzing the requirements (independent of the design) are generally missing from the field and deserve more attention.

- *Recommendation: Systematic independent evaluation of emerging technology for parallel software development should be continued*

Reports on the development of new tools and techniques are being published continuously. We could have easily spent all our time surveying the literature and keeping up with the new developments. In terms of this project it was necessary to stop collecting data at some point, and start hands-on evaluations. Based on the hands-on evaluation we have found that the published literature on new parallel software tools and methodologies is not a very suitable basis for evaluating their usefulness. All papers report success. The usability of the languages, tools, or methodologies by third parties cannot readily be established by reading the papers. Further programs like SEPA, in which an independent third party installs the tool and uses it for simple problems, will help to calibrate the applicability of the new ideas that have emerged since our survey.

- *Recommendation: For the evaluation of the applicability of parallel software development techniques to command and control, a command and control benchmark should be developed.*

Most benchmark and example programs used for evaluating parallel tools and architectures are scientific and mathematical applications. As such they do not provide a good measure of such tools and architectures for command and control. A sample command and control problem would need to include aspects common to many C^2 applications such as real-time processing, large database processing, image processing, and simulation.

In order to provide a realistic sample, even a "small" C^2 application would be quite complex. As a result, a successful C^2 sample would need to be well documented with both requirements and design as well as a user's guide. A full set of input data would be required along with documented expected results. A retargetting manual would also be needed for rewriting the application in languages other than the initial implementation.

5.2.4 Next Generation PEEP

Using the base technology of a fully populated, highly functional PEEP, it is possible to take the next step in supporting high performance parallel computing:

- *Recommendation: that the PEEP be extended to support "architectural supercompiling"*

The supercompiler concept has been developed over the past decade and was named by Wolfe in 1982. It involves the application of uniform deductive methods from verification and inference research. Compiling with a supercompiler is the selective application of transformations to a program representation that is successively refined until it can execute on the desired hardware. Supercompiling provides a way of looking at the cooperation of limited transformation tools on a platform such as the PEEP as a kind of non-deterministic program mapping that resembles compiling.

The experiments we have conducted have led us to believe that the best long-term opportunity for automated support of parallel software development is implementation of the supercompiler notion. This follows from our recommended methodology: starting development with very high-level languages, such as Id, and refining programs in stages through more concrete languages, adding information about details of the target architecture in the later stages.

The effectiveness of the proposed methodology and the selected tools would be greatly enhanced if it were possible to mix languages at any level of this refinement process. Unfortunately, most of the tools that are available at present are tied to single languages, or small sets of languages.

One of the most useful concepts in the field is the use of human-guided, machine-facilitated transformations to reduce the higher-level, architecture-independent representations of programs to concrete form. This is the supercompiler notion.

At present, the Meta-Crystal system being developed at Yale comes closest to providing the desired mechanism. However, Meta-Crystal is tightly connected to the Crystal language, which does not meet the needs of most parallel software developers. We believe that the idea of human-guided successive transformations is too important to be linked to a single language. The PEEP should provide a framework for an evolving library of transformations that can be applied to programs written in any language.

The supercompiler concept for parallel applications requires a suite of language translators and other tools that operate on a common internal representation of programs. Developing such a suite of tools will be a major undertaking. It will require the cooperation of government, academia, and industry to create a sufficiently broad set of languages, analysis and simulation tools, and transformations. The first step must be to define a common representation that can be accepted by all parties.

Industry is not ready to accept a universal low-level representation for programs. SIMD vendors will not accept a standard that is, or seems to be, strongly oriented to MIMD architectures, and vice-versa. MIMD vendors who provide a shared memory model will not

accept a standard that is strongly oriented to message passing. Even if a consensus on low-level representations could be forced, it might be harmful to the further advancement of parallel processing technology — whether MIMD or SIMD, Shared Memory or Message Passing, Vector, Dataflow, or other architecture classes are best is still controversial. This should be resolved by open competition in the marketplace.

However, we believe that a *high-level*, architecture-neutral representation could be defined that would be acceptable to all parties. In our experiments under the SEPA program, we have found that a high-level dataflow representation of programs (using Id) is readily mappable to all classes of architectures. Furthermore, the tools that would be needed to map programs represented at this level to low-level representations for specific architectures are the very facilities that comprise a supercompiler system.

DARPA's ProtoTech program has developed technology that addresses this need— the Kernel CPL multi-language environment framework. This framework will need adaptation to meet the needs of the full range of parallel processing machines, but its effectiveness as a supercompiler framework for sequential languages has already been demonstrated, particularly by the work of Yale University, Intermetrics, and Software Options Inc.

ProtoTech has also developed a formal, mathematically oriented language for prototyping parallel software, Proteus. Although Proteus is tied to shared memory architectures, it merits further investigation as a member of a family of languages to be supported by the PEEP. There are many parallel programming languages and prototyping tools, so implementing Proteus on the PEEP is not of high priority, but the framework provided by the Kernel CPL technology is unique and very important.

The PEEP should be the framework for this integration. The experiment could be performed before adding the supercompiler technology discussed above, or could be deferred until that technology is present in the PEEP.

The necessary "productization" of the required tools could be done by enlisting the cooperation of the original developers or their funding agencies, by separately contracting with those organizations, or by having a PEEP follow-on contractor work with their source code. We expect that the latter method would yield more immediate results, but that establishing more mechanisms for helping university researchers to produce robust tools would be of more long-term value.

References

R. Acosta, "PProto: An Environment for Prototyping Parallel Programs", *Journal of Systems Integration*, Volume 1, pp. 339–365.

J. Allen and K. Kennedy, "A Parallel Programming Environment", *IEEE Software*, July 1985, Volume 2, Number 4, pp. 21–29.

W. Appelbe and K. Smith, "Start/Pat: A Parallel Program Toolkit", *IEEE Software*, Volume 6, Number 4, July 1989, pp. 29–38.

J. Arthur and D. Reed, "Prometheus: An Integrated Environment for the Development and Execution of Parallel Programs", *Proceedings of IEEE Conference on Parallel Processing*, 1984, pp. 44–50.

Arvind and R. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture", *IEEE Transactions on Computers*, Volume C–39, Number 3, March 1990, pp. 300–318.

S. Arya and B. Gaither, "Parallel Algorithm Development Workbench", *Proceedings of IEEE Conference on Parallel Processing*, 1988, pp. 11–17.

R. Bagrodia, K Chandy, E. Kwan, "UC: A Language for the Connection Machine", *Proceedings of IEEE Converence on Supercomputing*, 1990.

R. Bahkle and G. Snelling, "The PSG System: From Formal Language Definitions to Interactive Programming Environment", *ACM TOPLAS*, Volume 8, Number 4, October 1986, pp. 547–576.

T. Bemmerl, "An Integrated Portable Tool Environment for Parallel Computers", *Proceedings of the IEEE Conference on Parallel Processing*,1988, pp. 50–53.

M. Brown, "Exploring Algorithms Using Balsa–II", *IEEE Computer*,

Volume 21, Number 5, May 1988, pp. 14–36.

M. Brown and R. Sedgwick, "Techniques for Algorithm Animation",

IEEE Software, Volume 2, Number 1, January, 1985, pp. 28–39.

J.C. Browne, M. Azam, and S. Sobek, "CODE: A Unified Approach to Parallel Programming", *IEEE Software*, Volume 6, Number 4, July 1989, pp. 10–19.

J.C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a

Reusability-Oriented Parallel Programming Environment", *IEEE Transactions on Software Engineering*, Volume SE–16, Number 2, February 1990, pp. 111–120.

N. Carriero and D. Gelernter, "Coordination Languages and Their Significance", *CACM*, Volume 35, Number 2, February 1992, pp. 96–107.

N. Carriero and D. Gelernter, "Linda in Context", *CACM*, Volume 32, Number 4, April 1988, pp. 444–459.

M. Chen, "Compiling Parallel Programs by Optimizing Performance", *The Journal of Supercomputing*, Volume 2, July 1988, pp. 171–207.

M. Chen. "Transformations of Parallel Programs in Crystal", in *Information Processing*, Elsevier North-Holland, New York, 1986, pp. 455–462.

K. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley, New York, NY 1988.

L. Chang and B. Smith, *Classification and Evaluation of Parallel Programming Tools*, University of New Mexico, Department of Computer Science, November, 1990

J. Dongarra and D. Sorenson, *SCHEDULE: Tools for Developing and Analyzing Parallel FORTRAN Programs*, Technical Memo 86, Argonne National Laboratory, November 1986.

I. Foster and S. Taylor, *Strand: New Concepts in Parallel Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

E. Gabber, "VMMP: A Virtual Machine Mechanism for the Development of Portable and Efficient Programs for Multiprocessors". *Proceedings IEEE Conference on Parallel Processing*, 1989.

V. Guarna, D. Gannon, D. Jablonski, A. Malony, and Y. Gaur, "Faust: An Integrated Environment for Parallel Programming", *IEEE Software*, Volume 6, Number 4, July 1989, pp. 20–28.

V. Guarna, D. Gannon, Y. Gaur, and D. Jablonski, "Faust: An Environment for Programming Parallel Scientific Applications", *Proceedings IEEE Supercomputing Conference*, 1988, pp. 3–10.

R. Halstead, "MultiLisp: A Language for Concurrent Symbolic Computation", *ACM TOPLAS*, Volume 7, October 1985, pp. 501–538.

D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs", *IEEE Software*, Volume 2, Number 2, March, 1985, pp. 47–57.

C.A.R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

D.E. Knuth, *Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.

E. Jamiesson, "Characterizing Parallel Algorithms", in E. Jamiesson and D. Gannon, eds. *The Characteristics of Parallel Algorithms*. MIT Press, Cambridge, Massachusetts, 1987.

H. Jordan, "The FORCE", in E. Jamiesson and D. Gannon, eds. *The Characteristics of Parallel Algorithms*. MIT Press, Cambridge, Massachusetts, 1987.

M. Karr, *Equality, State and Logic*. Software Options Technical Report, Cambridge, Massachusetts, 1988.

T. Lehman amd M. Carey, "Query Processing in Main Memory Database Systems", *Proceedings of SIGMOD*, 1986, pp. 239–250.

M. Linton, "Implementing Relational Views of Programs", *Proceedings of ACM Symposium on Practical Software Development Environments*, April 1984, pp. 132–140.

D. Luckham, J. Vera, D. Bryan, L. Augustin, and F. Belz, *Partial Orderings of Event Sets and Their Application to Prototyping Concurrent Timed Sets*. DARPA Manuscript, Stanford University, December, 1991.

M. Marconi and D. Hare, "The PegaSys System : Pictures as Formal Documentation of Large Programs", *ACM TOPLAS*, Volume 8, Number 4, October 1986, pp. 524–546.

B. Miller, C. Morgan, et. al., "IPS–2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems*, Volume 1, Number 2, April 1990.

P. Mills, L. Nyland, J. Prins, and J. Reif, "Prototyping High-Performance Computing Applications in Proteus", *Proceedings of IEEE Conference on Supercomputing*, 1991, pp. 433–442.

T. Moher, "PROVIDE: A Process Visualization and Debugging Environment", *IEEE Transactions on Software Engineering*, Volume SE–14, Number 6, June 1988, Pages 849–857.

P. Newton and J. Browne, "The CODE 2.0 Graphical Parallel Programming Language", *Procedings of the ACM Conference on Supercomputing*, July 1992.

K. Nichols and J. Edmark, "Modeling Multicomputer Systems with PARET", *IEEE Computer*, Volume 21, Number 5, May 1988, pp. 39–48.

R. Nikhil, *The Parallel Programming Language Id and its Compilation for Parallel Machines*, MIT Computation Structures Group Memo 313, July 1990.

R. Olson, R. Crawford, and W. Ho, "A Dataflow Approach to Event-Based Debugging", *Software Practice and Experience*, Volume 21, Number 2, February 1991, Pages 209–229.

H. Ossher and W. Harrison, "Support for Rapid Change in RPDE", *Proceedings of the ACM Conference on Software Development Environments*, December 1990, pp. 218–228.

D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers", *CACM*, Volume 29, Number 12, December 1986, pp. 1184–1201.

M. Panamgi, W. Heusch, and G. Kaiser, "Debugging Multithreaded Programs with MPD", *IEEE Software*, Volume 8, Number 3, May 1991, Pages 37–44.

D. Pease, A. Ghafoor, et. al. "PAWS: A Performance Evaluation Tool for Parallel Systems", *IEEE Computer*, Volume 24, Number 1,January 1991, pp. 18–29.

T. Perry and G. Zorpette, "Supercomputer Experts Predict Expansive Growth", *IEEE Spectrum*, Volume 26, Number 2, February, 1989, pp. 26–33.

T. Pratt, "The Pisces 2 Parallel Programming Environment", *Proceedings of the IEEE Conference on Parallel Processing*, 1987, pp. 439–445.

T. Pratt, "Pisces: An Environment for Parallel Scientific Computation", *IEEE Software*, July 1985, Volume 2, Number 4, pp. 7–20.

S. Reiss, "Connecting Tools Using Message Passing in the FIELD Environment", *IEEE Software*, Volume 7, Number 5, July 1990, pp. 57–67.

S. Reiss, *Integration Mechanisms in the FIELD Environment*, Brown University, Technical Report No. CS–88–18, October 1988.

S. Reiss, "Working in the Garden Environment for Conceptual Programming", *IEEE Software*, Volume 4, Number 6, November 1987, pp. 16–27.

S. Reiss, "Graphical Program Development with Pecan Program Development System", Proceedings of the ACM Symposium on Practical Software Development Environments, *ACM SIGPLAN Notices*, May 1984, pp. 132–140.

D. Rosenblum, "Specifying Concurrent Systems with TSL". *IEEE Software*, Volume 8, Number 3, May 1991, pp. 52–61.

M. Rosing, R. Schnabel, and R. Weaver, *The DINO Parallel Programming Language*, University of Colorado at Boulder, Department of Computer Science, CU–CS–457–90, April 1990.

G. Sabot, *The Paralation Model*. The MIT Press, Cambridge, Massachusetts, 1988.

K. Schwan, R.Ramnath, S. Vasudevan, and D. Ogle, "A Language for the Construction and Tuning of Parallel Programs". *IEEE Transactions on Software Engineering*, Volume SE–14, April 1988, pp. 455–471.

J. Stasko, "TANGO — Algorithm Animation Designer's Package", from *BWE Programmer's Manual*, Brown University, July 1990.

Z. Segall, L. Rudolph, "PIE: A Programming and Instrumentation Environment for Parallel Processing", *IEEE Software*, Volume 2, Number 9, November 1985, pp. 22–37.

L. Snyder, D. Socha, "POKER on the Cosmic Cube: The First Retargetable Parellel Programming Language", *Proceedings of the IEEE Conference on Parallel Processing*,1986, pp. 628–635.

H. Stone, "Parallel Processing with the Perfect Shuffle", *IEEE Transactions on Computers*, Volume C–20, Number 2, February 1971, pp. 153–161.

H. Stone, "Dynamic Memories with Enhanced Data Access", *IEEE Transactions on Computers*, Volume C–21, Number 4, April 1972, pp. 359–366.

R. Taylor, *Analysis of Concurrent Software by Cooperative Application of Static and Dynamic Techniques*, Technical Report 196, University of California, Irvine, March 1983.

V. Turchin, "The Concept of a Supercompiler", *ACM TOPLAS*, Volume 8, Number 3, July 1986, pp. 292–325.

M. Wolfe, *Optimizing Supercompilers for Supercomputers*. Ph.D Thesis, University of Illinois at Champagne-Urbana, 1982.

J. Yand and Y. Choo, "Parallel-Program Transformation Using a Metalanguage", *Proceedings of ACM Conference on Principles and Practices of Parallel Programming*, 1991, pp. 11–20.

A. Xu and L. Hwang, "Molecule: A Language Construct for Layered Development of Parallel Programs", *IEEE Transactions on Software Engineering*, Volume SE–15, Number 5, May 1989, Pages 587–599.

Special References

C. Lightfoot, D. Sakai, T. Busse, J.Shelton, *Software Techniques for non-Von Neumann Architectures*. Computer Sciences Corporation, Final Technical Report RADC–TR–89–337, Rome Laboratory, Griffiss Air Force Base, New York, January 1990.

B. Breugge and P Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism", Proceedings of Software Engineering Symposium on High Level Debugging, *ACM SIGPLAN Notices*, August, 1983, pp. 34–44.

F. Leighton, *Introduction to Parallel Algorithms and Architectures*.

Morgan Kauffman Publishers, San Mateo, California, 1992.

S. Miller, *Survey of Parallel Computing* (Second Edition). Amherst Systems, Inc., Final Technical Report RADC–TR–89–68, Rome Laboratory, Griffiss Air Force Base, New York, June 1989.

R. Nikil, *Id Language Reference Manual*, Computation Structures Group Memo 284–2, MIT Laboratory for Computer Science, Cambridge, Massachusetts, July 1991.

R. Nikil, et. al., *Id World Reference Manual*, Computation Structures Group Memo, MIT Laboratory for Computer Science, Cambridge, Massachusetts, Noverber 1989.

Software Options, Inc, *E-L Definition*. Technical Report, Cambridge, Massachusetts, 1988.

Thinking Machines, Inc, **Lisp Reference Manual*. Cambridge, Massachusetts, 1986.

J. Werth, et. al., *CODE 1.2 User Manual and Turorial*. University of Texas at Austin, 1990.

M. Wolfe, *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, Massachusetts, 1989.

A. Experiment Descriptions

A.1 Experiment 1: Exploiting Inherent Parallelism

Parallel sorting is a classic problem. Very good sequential algorithms exist, such as radix exchange, that have essentially no inherent parallelism. For applications with very large quantities of data, or severe timing requirements, parallel approaches are indicated. Hardware methods with the necessary stochastic properties have been extensively studied, and are one solution. In principle, these could be programmed in software, but probably not efficiently except on dedicated mesh architectures.

Assuming that dedicated hardware is infeasible, traditional methods of sorting may be used with significant parallel speedup. This involves choosing an algorithm with significant intrinsic parallelism, or using a divide-and-conquer approach to partitioning the data. Techniques that divide the input into as many chunks as there are processors, sort each chunk, and merge the sorted results are good candidates for theoretical algorithm speedup, but are effective in practice only if communication costs can be kept relatively small. In a *mixed approach*, individual processors can use fast sequential algorithms and less refined communication techniques that are less optimal than programmable meshes or systolic arrays. This is the area where code replication techniques are useful on MIMD architectures.

We chose Quicksort for our experiment. Parallel sorting has been studied extensively, and other more parallel sorting algorithms are documented in the literature. However, our intent in this experiment was to try to parallelize a standard sequential algorithm with the tools available. Certainly, if we wanted the best possible results in terms of performance a different algorithm would have been chosen.

Our parallel sort routine involved a straightforward attempt to use the intrinsic parallelism of Quicksort in a dataflow language. Its formulation in Id selects an arbitrary element (which can be the first) and divides the input into two parts, consisting of elements that are less and greater than the selected one, and then applies the same function to the parts. Quicksort's basic formulation in Id is as follows:

```
(qsort lesseq-a) ++ a : (qsort greater-a)
```

where "++" is a list concatenation operation, ":" is a LISP-like *cons* operator, "a" is the selected element and "lesseq-a" and "greater-a" are the parts of the input values that are recursively sorted by Quicksort.

The strategies used for sorting are described in section 4.

A.2 Cooperative Concurrency

Searching algorithms are also well studied. Even more than in sorting, fast searching methods rely on techniques for organizing data that are inherently sequential. One parallelization approach would be to do a fast parallel sort, and do all searching using a sequential binary search algorithm. This approach has serious problems with the insertion

of new entries, making it inappropriate in real-time applications. Using multi-tasking to synchronize insertion and deletion by dividing input into chunks and merging similar data is as valid a technique as for sorting. However, merging is somewhat more complicated and expensive when searching since the point where the insertion takes place has to be protected until the merge is completed. Merging insert (write) functions with lookup (read) functions introduces synchronization issues. The intent of the second experiment was to investigate the synchronization issues associated with this type of algorithm. One simple way to achieve this is to associate parallel processes with parts of data or the memory they access. This is a powerful way to use multiple processors in real-time searching. It can be implemented like a multi-tasking system and adjusted dynamically using appropriate scheduling, but is also very adaptable to mid- and fine-grain parallel architectures.

A basic idea for such a search is to treat *n* processors as searching through a multi-threaded list, where parts of the list can be global and parts more local (shared by some or even non-shared). Each segment of the list can be treated as entered by a hash function that selects which buffer and processors to use. For local memory, the processor is determined by selecting its address space. For shared memory, synchronization needs to be associated with the insert actions. Processors could be differentiated on the basis of whether they have access to shared memory, and would have slightly different lookup and add actions. This would be an inherent difference between MIMD and more uniform SIMD architectures.

In its simplest form, a SIMD algorithm would probably be roughly a recursive Id program with two functions: one to add a value to the list based on some key that is a function of its value, and the other to lookup a value. There would be some main or environmental process adding values, and eventually multiple clients trying to access the values (in order to verify and/or operate on them). The Id functions we tried were:

```
type struct-elem = {record
      key = int;
      value = *0};

def lookup key struct =
 {case struct of
  nil = nil;
  | (struct-elem:rest) = {
      if (matching-key struct-elem.key key) then
       value = struct-elem.value;
      else if (possible-key struct-elem.key key) then
       value = (lookup rest key)
  else value = nil;
      in
       value}};

def add new-elem struct =
  {case struct of
   nil = new-elem;
  | (struct-elem:rest) = {
   if (possible-key struct-elem.key new-elem.key) then
       (struct-elem : (add rest new-elem))
      else (struct-elem : ( new-elem : rest ));
```

```
}};
```

Here, "struct" is an Id I-structure, built by "add" — "add" is written as if it returned a structure when what it really needs to do is insert into or append to the global I-structure from a controlling process. At the simplest level, "add" will insert elements whenever it encounters a position that is vacant. This can be thought of as encountering an end of tape marker and writing the next element. The keys can be thought of as relative addresses which match when they are equal and are possible matches when they do not exceed the last used address.

A.3 Target-Independent SIMD

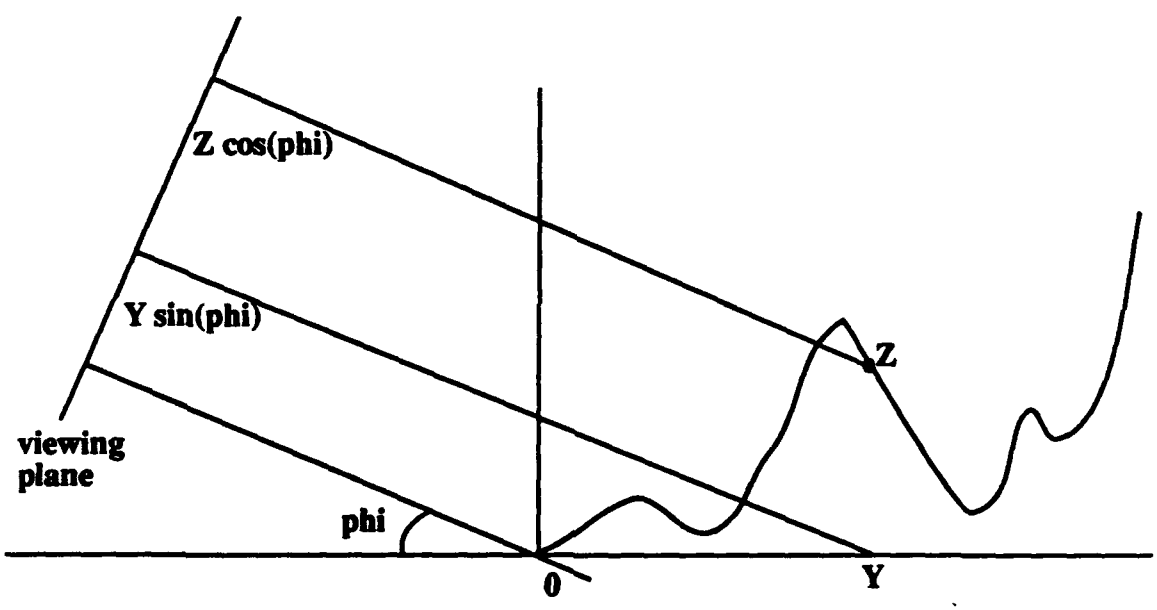
The image warping algorithm uses map data containing a height and color for a rectangular area and determines the data used for creating an image viewed from a particular angle that looks 3-dimensional to the viewer. The basic "Floating Horizon Algorithm" is as follows: if we have elevation and color data on, for simplicity, a N by N grid, we can use the floating horizon algorithm to display that data given a viewing angle of phi. We may think of the data as height function, h, mapped into 3-D space by:

$Z(i,j) = h(X(i),Y(j))$

where $X(i) = i$, $Y(j) = j$, $i,j = 0, 1, \ldots, N-1$

If the image is to be viewed at angle phi (the top view is given by phi=90), the projection is given by:

$(X,Y,Z) \rightarrow (X, Y * \sin(phi) + Z * \cos(phi))$



Given a cross-section as shown in the diagram, the height on the viewing plane of a particular point (Y, Z) is calculated using the equation above. If the calculated height is greater than the current horizon, then select the point, and make it the new horizon.

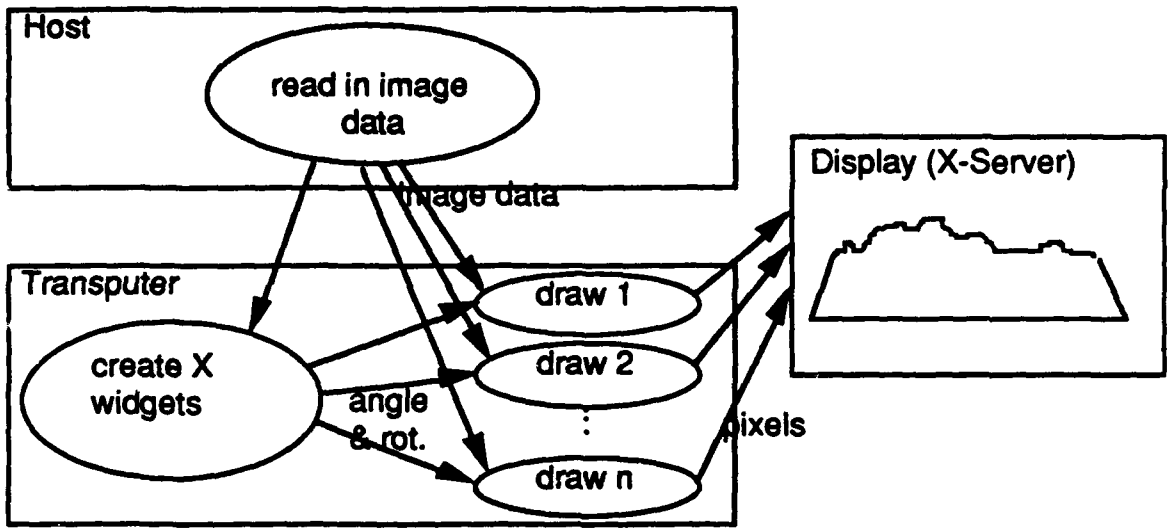
Otherwise, the point is below the horizon as viewed from angle phi, and it is ignored. This is done for all Y's to produce one column of the projected image. The same calculation is done for all columns (i.e. X's)

The pseudo-code for this algorithm may be expressed as:

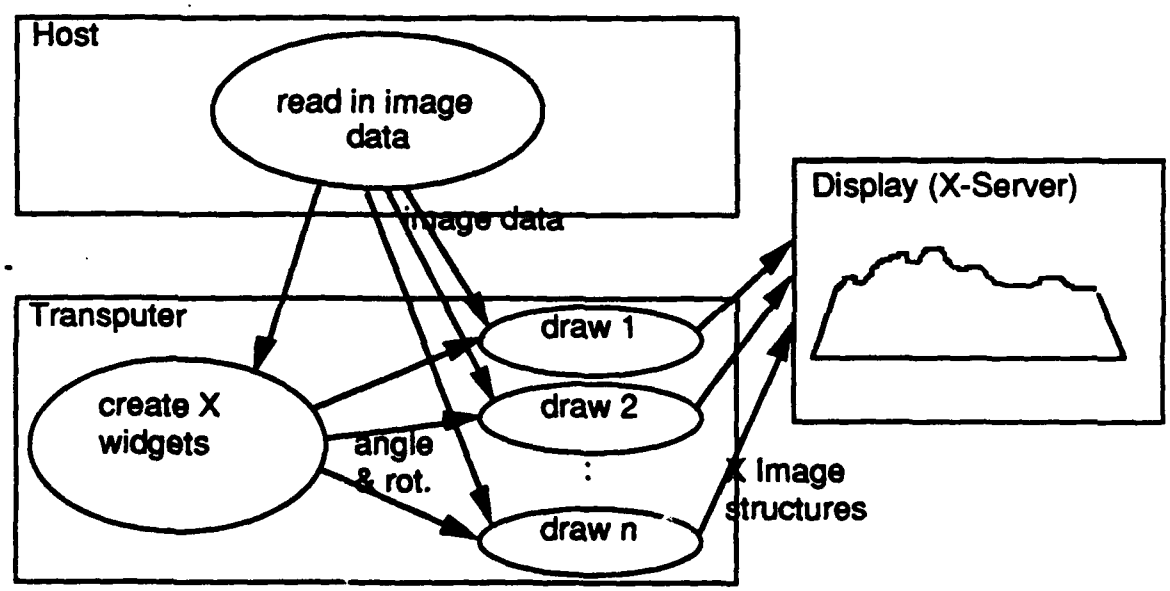
```
procedure DisplayHeightField (Z, C, N, phi) is

arguments:
    Z[][]    indexed square array of heights
    C[][]    indexed square array of color
    N   number of elements in a row/column of Z
    phi  view angle

variables:
    Im[][] indexed square array that represents the rendering

    i,j,k  integers
    p,p0     projected heights, screen y-coordinates
    horiz    integer

    function P(x,y) is
    begin
        return int[(y*sin(phi)) + (z*cos(phi))];
    end P;

begin
    for i in 0..N-1 loop
        — reset horizon for new column
        horiz := 0;
        for j in 0..N-1 loop
            — calculate projection for point
            p0:= P(j, Z[i][j]);
            p := p0;
            — fill in image if above horizon
            — and fill in intermediate pixels
            while p > horiz loop
                Im[i][p] = C[i][j]
                p = p-1
            end loop;
            — save new horizon, if necessary
            if p0 > horiz then
                horiz := p0
            end if;
        end loop;
    end loop;
end;
```

Note that we simplified the data partitioning by restricting the directions from which the image could be viewed, and by displaying an orthogonal view, rather than a perspective view. These simplifications do not change the basic algorithm, just the amount of data required by each processor.

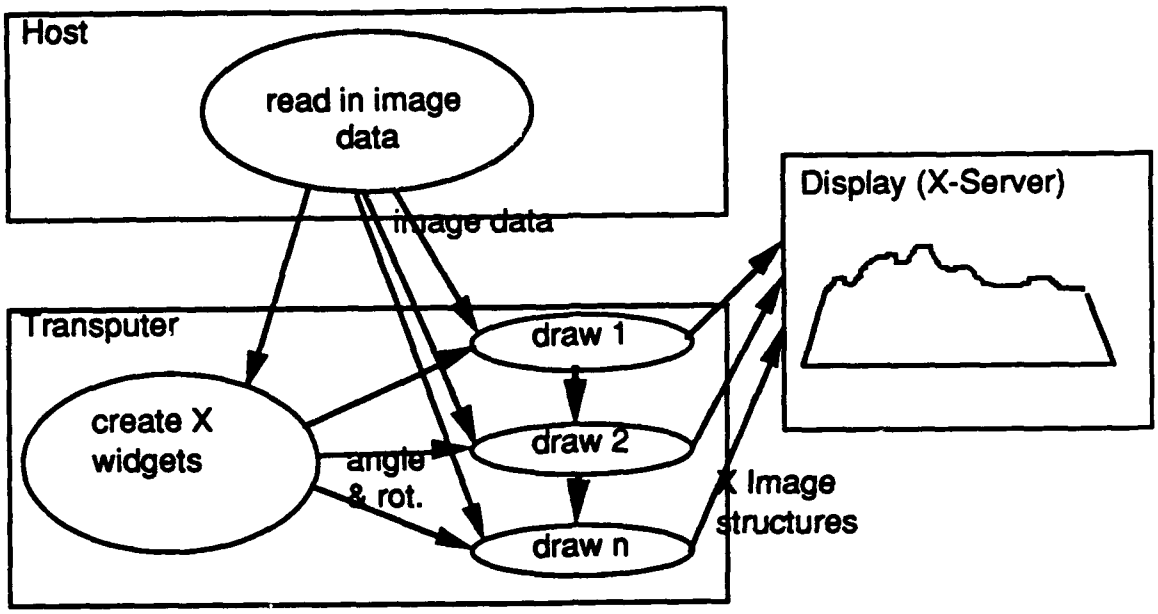
In implementing the image warping algorithm, we went through four iterations. The first implementation, shown in the figure below, resulted in poor performance. As individual pixels were sent from the trasnputers to the display, there was a severe communications bottleneck between the two machines.



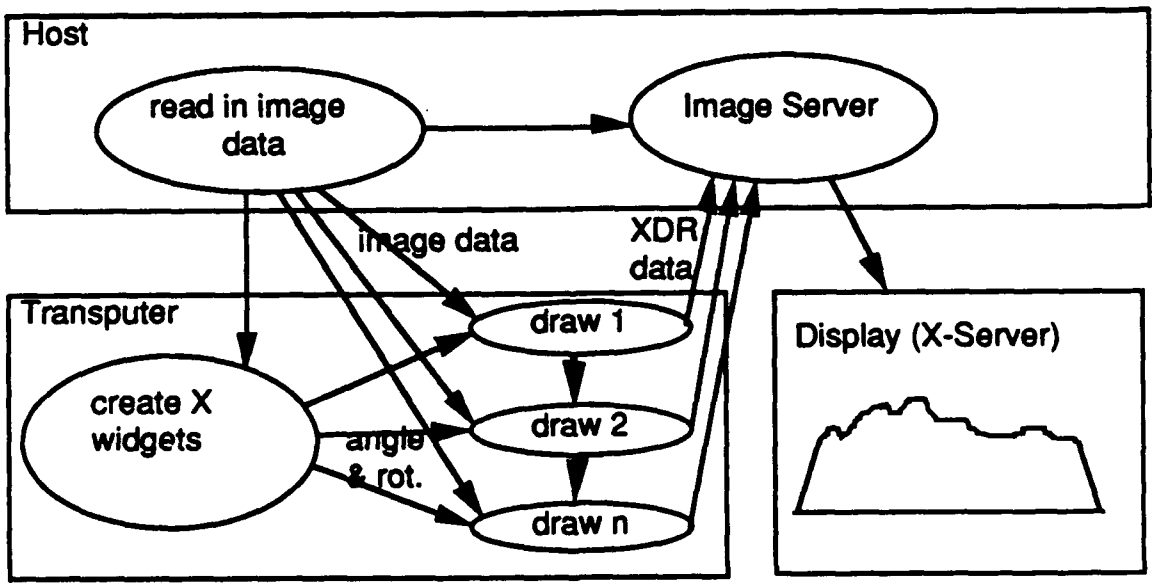
In the next implementation, most of the application remained the same, but instead of sending individual pixels, it sends X image structures to reduce the number of calls to the X-server. This had better performance, but there was still contention for the communication line to the X-server. As a result the transputer network would intermittently hang, and the data on the screen was sometimes corrupted. Since not much changed between the two implementations it took very little time to implement the second version based on the first.



The third implementation, shown below, added daisy chaining between the transputer nodes to eliminate the corrupted data. This was generally successful, however, the transputer network would still hang on occasion as the single-threaded X-server on the display could not handle the calls coming from multiple transputer nodes. Again, since the code was changed very little, it took very little time to implement the third version.



The final strategy was to eliminate the X calls from the transputer nodes entirely. This involved learning the XDR utilities for exchanging data between the transputers and the host. As a result, it took somewhat longer to implement this version of the algorithm than the previous two variations. However, it was worth the effort as this produced the most reliable implementation, and it was almost as fast as implementation three.



The following table shows the relative performance of the different implementations. The measurements given are seconds; the first number is the elapsed amount of processing time on the transputer; the second is the total time which includes processing on the host.

	Impl. 1	Impl. 2	Impl. 3	Impl. 4
Transputer time	93	13	3	2
Total time	93	15	7	9

B. Tool Summaries "Data Sheets"

This section contains the descriptions of all the tools and environments surveyed. It includes both tools that were included in the original survey report and other tools that were discovered later. It is organized into parallel environments, FORTRAN parallelism support tools, special purpose tools, and parallel languages. All tools are described in the present tense in terms of their capabilities even though, as discussed earlier in the report, a lot of these projects have ended, and the tools are no longer available.

Parallel Environments

CODE

Developed at the University of Texas, the Computation-Oriented Display Environment (CODE) provides high level parallelism support for a parameterized range of target architectures. It includes a graphical editor and monitoring facility for program graphs of schedulable units in (currently) three program languages, FORTRAN, Ada and C. The CODE effort is closely integrated with another environment project, the Reusability-Oriented Programming Environment (ROPE) at Austin. This shares the program database with CODE and allows intermixing of program components developed under either system. The program graphs are hierarchical compositions of other units in either the graph language or the base language. These four kinds of units are: controlling units, filters (used for constraints), dependency lists (involving structures of related units), and subgraphs.

Target architecture models are described in a TOAD (Translator Of A Declaration) which includes synchronization information and target configurations. TOADs exist for Ada, FORTRAN, and various extensions of C, for target environments running on a variety of parallel machines, including a Sequent Balance, a VAX cluster, a Cedar (clusters), an Intel Hypercube, and a Cray X-MP. Some executable FORTRAN modules that have been published use the Schedule execution environment for synchronization operations, indicating that this approach is amenable to supporting common target environments in a natural way.

The development path using CODE is to edit a graph of program components called Schedulable Units of Computation (SUC). This is done under control of an X-based menu system with fifteen icons for drawing graphs of SUC relationships. One icon supports general attribute editing actions and thus provides a fully non-linguistic model of data flow. Code in the SUC's themselves is in one of the supported languages, Ada, FORTRAN or C. Input data dependencies are modeled as input parameters. Output data dependencies can be output parameters in Ada, or read-write parameters in FORTRAN and C. Mutual exclusion uses Ada tasks or global variables synchronized by appropriate language extensions for the supported target. For FORTRAN, one possible informal standard is the Schedule runtime interface as initially done for the Encore Multimax at Argonne Labs. The Cray and IBM extensions use the multitasking supported by the vendors' FORTRAN-77 compilers. Any micro-parallelism features can be used by programming them into the code for the SUC's.

The main shortcoming of this approach is that peculiarities of the target language and architecture are exaggerated. The lack of portability in FORTRAN is made explicit in the need for a different TOAD for every combination of target and operating system. The optimization problems of Ada tasking makes synchronization very expensive. Each node depending on exclusive access to any data is modeled as a separate task. This situation should be considerably improved by using Ada 9X protected operations. Exclusive access to shared data can then be implemented with protected operations. CODE should be changed to support this improvement if Ada 9X is supported.

CODE's separation of target environments represents an important research effort, but its real application potential is limited. The main reservations CODE developers express are that it maybe difficult to model architectures significantly different from their initial set. In particular, fine grain parallelism is supported by the users knowledge of the problem and the compiler for the chosen target language. The decomposition of the algorithm as done by transformations or by other preprocessor approaches is done entirely when specifying the schedulable units. In some ways, this is ideal for developing programmer intuition and skills, since the approach provides rapid target specific performance information. The user can decompose his problem according to different architectural considerations, and execute the models on the actual hardware using the supported compilers. What cannot be easily done is to record the process of generating the different variants. What would be needed is support for experimentation in expressing the algorithm using different mappings, to complement CODE's capability of providing feedback on the effects of architectural decomposition for a particular algorithmic formulation.

The developer's of CODE are about to announce a new version that addresses some of these issues. They have unified the concepts related to specifying synchronization expressions, into classes of operations that permit more effective translation. This has the effect of decoupling transformations that potentially interfere for very general data dependencies. In effect, they are supporting the derivation of specific categories of synchronization operations from the more general relations of SUC's (now called just UC's) allowed in the first version of CODE. This should tend to support exactly the kind of successive refinement (under deterministic transformations) that is needed to address the variant issue raised above. Presumably, this involves improvements in ROPE to record related configuration items. This seems a likely direction that hopefully will be pursued in the future.

Express

Express is a commercial parallel programming environment designed to address basic development issues. It includes tools for programming, debugging and performance monitoring of distributed hardware platforms. We considered EXPRESS as a substitute for the Meiko CS Tools that come with the Transputer, but rejected it since it forces the transputers into a hyper cube configuration. However, EXPRESS is available on several platforms in addition to the transputer, including a network of workstations which may make useful in some applications for the PEEP.

Faust

Faust is a modern environment for supercomputers developed at the Center for Supercomputing Research and Development, at the University of Illinois, in Champagne-Urbana. Supporting Butterfly, Cedar, and Ardent targets, Faust is basically a supercomputer resident environment, but is "conventional UNIX compatible", and can be run on Sun workstations. It has functions for user interface, graph manipulation, and text processing. It provides a high-level graphical view of function and task relationships. Its multi-language editor supports FORTRAN and C with various extensions, as well as a Pascal-based functional language. It includes a graph browser and a dynamic parallel program monitoring tool. All its graphics use X for portability.

Faust seems to be the most immediately usable of the current environments for parallel programming in conventional languages. In particular, its program database and intelligent editor capabilities are well-integrated with the architecture-specific optimization capabilities needed to produce executable parallel code. Its editor, called Sigma, performs vectorizing optimizations based on knowledge of the target architecture, applying program transformations under user control. Optimizations have the results of dynamic analysis available, using information fed back from monitoring during execution.

Faust inserts program monitoring in a systematic way as part of its preparation of a parallel program for a given architecture. The information collected becomes part of the application program database as either performance statistics or data dependence information. The dynamic flow and data usage information can be used by optimizing tools for global flow analysis and for determining when particular program transformations can be applied during architecture mapping. Also, Faust provides a dynamic call graph animated view of a program execution that displays the dynamic execution of a parallel program in terms of its original subroutine call graph. Additions support integrated interactive display. Faust's Integrated Multiprocessor Performance Analysis and Characterization tool set, IMPACT, integrates performance data collection with display and analysis tools. IMPACT's event-display tool traces multitasking events and displays them in real time.

The Faust project appears to have minimal interest in exploration of the programming language problem. Emphasis is on extensions of FORTRAN and C (including concurrent C and C++), for its target architectures. Among the parallel environments, Faust definitely provides the most advanced support for vectorizing optimization. However, Faust has little capability to deal with algorithms that are parallel at a very high level, such as data flow and parallel graph algorithms. It concentrates on solidly supporting programmability in conventional languages for a few related architectures, but provides no basis for experimentation with different parallel programming languages or methodologies.

ISSOS

ISSOS is a programming environment developed at Ohio State University that supports prototyping of parallel applications. It also supports experimentation that is

directed towards improving the performance of parallel algorithms executing on the a network of UNIX machines.

The system's Program Construction System (PCS) uses an object oriented dialect of C called COOL to prototype the application. The PCS is equipped with a syntax directed editor and a COOL to C source code generator for eventual execution in a UNIX environment that supports mechanisms for process creation, scheduling and inter-process communication.

ISSOS tools help the programmer create new "program adaptations" that differ from the original with respect to performance. Static adaptations are implemented by including "Program Adaptation" statements in the program. These statements control load balancing and can effect the execution of a process. In addition to adaptation statements, the Adaptation Controller (AC) is aware of the physical characteristics of the target and can modify the allocation of resources to the running program accordingly.

The AC also does dynamic tuning by interpreting data about program behavior collected by Program Monitors which exist on nodes.

ISSOS is still very much a research effort, with even its language (COOL) in some process of evolution. That, combined with its emphasis on large grained parallelism, limits its applicability to the PEEP.

Omega/PegaSys

Omega was developed at Stanford University in the early 80's as a non-linguistic programming environment. It supported input of programs in mixed form, including traditional algebraic form for expressions, graphs for control structure, and icon representations for program structures. It combined this with a relational program database that supports development through successive refinement by providing relations as sequential interpretations.

Omega pioneered, and PegaSys continues the development of, the use of "software through pictures" and very high level non-traditional programming techniques. In PegaSys, the user describes how a program is built using a hierarchically structured collection of pictures called formal dependency diagrams (FDDs). PegaSys is implemented in Interlisp-D and runs on Xerox 1100-series personal computers, and the only language currently supported is Ada.

There are three major components included in PegaSys:

- Interface Tools
- Hierarchy Manager
- Program Verifier

The interface tools include a display management package and also a package that maps graphical operation into logical operations. Also included with these tools are graphical and textual structure-oriented editors, a pretty printer, and a database which

maintains FDDs and programs. The hierarchy manager ensures that each level of an FDD implements the levels above it correctly and also aids in the construction of the levels of an FDD. The program verifier determines whether an FDD is logically consistent with the Ada program it describes.

PARET

Developed at AT&T Bell Laboratories, PARET is a system description and simulation language oriented to architecture design and investigation. It supports graphical layout of processors and division of parallel functions among processing elements (PEs), switching elements (SEs), and communication elements (CEs) (in networks of CPU's). It primarily simulates machines with a MIMD architecture without shared memory. It supports graphical layout of these elements with arcs representing data and control flow, and allows association of capacities, buffers and timing delays with their interconnections. During simulation, information can be passed to analyzers of time dependent sequences to generate control traces and calibrated dynamic utilization displays called meters. Computation within a processor can be simulated by a delay, or the computation can actually be performed if written in C or C++.

The PARET user interface provides a fixed screen configuration that includes windows for the display of:

- the active parts of the system or subsystem graph
- the set up of simulation parameters and flow graph database
- the graphs or meters of significant activities

Nodes and meters provide significant measures of system performance and, typically, are used to determine if a given design is capable of satisfying a set of design goals. All nodes and meters assigned to one processor are shown in the same color.

This is a good example of a system simulation approach that would be needed for realistically modeling a wide variety of parallel computers. However, it is different from the above work on parallel programming because it concentrates on hardware description and provides no support for advanced software development methods.

PIE

The Programming and Instrumentation Environment (PIE) is a programming environment, developed at Carnegie Mellon University, that pioneered the application of environment concepts to parallel program analysis and development. PIE is composed of a Modular Programming (MP) metalanguage, a program constructor, and an implementation assistant. The MP metalanguage provides support for the efficient manipulation of parallel modules and fast parallel access to shared data constructs. The program constructor (PCT) provides the capability to generate efficient parallel programs. The PCT is itself composed of three elements, a MP-oriented editor (MPOE), a status and performance monitor, and a

relational representation system. The implementation assistant provides semantic support in the parallel program development cycle.

The language and concepts of PIE are very good, but are somewhat specialized in their support of a particular model of parallelism (shared memory multi-processing targets with medium to large grain parallelism). PIE is a good example of an early parallel programming system, however, the PEEP requires a more general ability to model other parallel methodologies.

POKER

POKER is a parallel programming environment developed at the University of Washington that uses a visual representation to describe parallel programs. Poker assumes an underlying message passing non-shared memory architecture.

Programs are constructed by creating and manipulating visual or textual representations of information about the program called "views". The heart of a program is its communication structure which is represented by a communicating graph. Nodes represent processes. Edges describe communication to ports on other nodes. Ports are also represented graphically as a view. The process views are defined textually in "Poker C" and labeled to correspond to vertices in the graph. "Poker C" is similar to regular C except it has an additional construct to describe inter-process communication. There are also I/O views that associate dangling edges with input or output streams.

A Poker program's "views" plus the source code for the processes are stored in a program database which is compiled and can be run on a simulator. The simulator is equipped with a "Trace view" that includes a timing model. Trace view lets the programmer stop the program, single step through it and monitor traced variables. Poker programs also run on several parallel computers including the Pringle parallel computer and the CalTech Cosmic Cube.

Poker has been used by over 50 institutions for 5 years and a large number of parallel programs have been written with it. This experience confirms the notion that visual support for parallel programmers improves productivity. This approach, as implemented, is limited because large programs are difficult to input graphically, and are confusing to view. A user is forced to construct large programs from smaller building blocks. Consequently Poker supports a paradigm for algorithm construction based on decomposition. Poker users typically decompose physical problems into a group of independent parallel programs that are each defined by a communication graph and its associated views. These programs are called "phases", and a group of them are executed to solve a complex problem. This approach encourages modularity, reusability, and graphical representation of program components. However, the synchronization between phase executions is inefficient and there is no mechanism to build single executable applications. The programmer is forced to run phases one after another.

A new parallel environment called Orca is being planned by the Poker researchers to correct these deficiencies.

Prometheus

Prometheus is a UNIX-based environment originally developed at Virginia Polytechnic Institute and State University, in Blacksburg. It has a Backus FP "combining forms" functional language, provides three different ways of constructing programs (top-down by graphical editor, bottom-up by language-sensitive editor, and iteratively by graph transformation), and executes by an extended Bourne-shell with a graphical debugger. It includes a Tool Description Language for functional extension.

For dynamic analysis, Prometheus provides four views of dynamic behavior: road maps, activity execution graphs, invocation trees, and frame usage. These can use explicit program points, or can make use of probes inserted into the interpreted code in a controlled way to monitor block entry and exit. These probes are maintained in a sensor-enable table that is checked each time a sensor is encountered. There is a standard format for transferring this information to the program database for post-mortem analysis. Probe insertion and enabling are both under user control, to keep intrusion to a minimum.

This is a powerful environment which could be a framework for program analysis, including investigation of parallelism. However, it includes no direct support for "very high level languages" and would require extensive use of its extension mechanism to model a variety of parallelism methodologies.

PSG

Developed at the Technical University of Darmstad, the Program System Generator (PSG) is probably the most thorough integration of programming language definition and verification techniques with environment technology. PSG generates language sensitive editors, an interpreter, and a fragment library system using formal language specifications for any user-defined language.

The main component of PSG is a full-screen editor which permits both text and structure editing. Prevention of both syntactic and semantic errors are guaranteed when using the editor in structure mode. In text mode, the editor guarantees the immediate recognition of these errors. PSG has been used to generate environments for ALGOL-60, Pascal, MODULA-2, and the formal language definition language itself. The main interest in PSG for the PEEP is its use of common type definitions in the tools generation process to guarantee interoperability of tools.

RPDE

This is a software process project at IBM Yorktown Heights that supports an object oriented tool framework for integration of program analysis and processing techniques. RPDE has been used successfully in:

- extending tool functionality,
- extending data domain of tools,
- supporting new languages,
- integrating separate environments,
- supporting new/different operating systems.

RPDE emphasizes common services along tools, including mappings from internal tool data representation to common interoperable formats that are a kind of temporary common file system. The initial efforts have shown that the common service framework covers almost all of the needed services for new processors.

In their article "Support for Change in RPDE", H. Ossher and W. Harrison wrote:

> The research issues being pursued in connection with Garden are similar to those of interest to us in building and using RDPE. Both systems employ a substantial functional framework of services, with an object-oriented definition of conceptual language structure on top. The developers of Garden have employed it to explore graphical programming, the creation of user concept structures, and visual output. We have employed RPDE to explore ways of exploiting the structure of professionally developed software to solve in-the-large programming problems. We have also devised and demonstrated the usefulness of some extensions to the object-oriented paradigm.

These are similar to the goals of the PEEP and indicate possible directions to pursue.

TOPSYS

TOPSYS is an integrated parallel programming environment that addresses monitoring on distributed platforms using a number of cooperating techniques and tools. It is currently only available for the iPSC/2; however, if the Sun workstation version currently being developed becomes available, it should be evaluated as a possible stand alone special purpose development vehicle for the PEEP.

FORTRAN Parallelism Support

FORCE

FORCE is an extension of FORTRAN for parallel programming of scientific applications on shared-memory multicomputers. It is currently not a supported tool, however, it is available "as is" to interested people. It is based on the UNIX tools sed and m4, and there are versions for Encore, Sequent, Alliant and Cray machines. It involves

parallel control structures which are expanded by target dependent macro's to architecture specific microtasking calls.

PAT

Developed at Georgia Tech, PAT (Parallelizing Assistant Tool) is a menu driven editor for analyzing and incrementally transforming FORTRAN programs to achieve greater usage of available parallel optimizations. It analyzes the program for specific features, including common data usage and looping behavior and allows the user to change the program under specific conditions. Some of the transformations can even be done automatically, and then subjected to user scrutiny for possible improvement or partial reversal. This effort shows the kind of user engineering that can be expected to play an increasing role in parallel programming support efforts. It can draw on compiler-related efforts that are basically optimization projects, such as the R^n project at Rice and several parallel compiling projects of hardware vendors, and on the program transformation work of such projects as Meta-Crystal at Yale and E-L at Harvard. PAT's use of an intelligent editor to control a semi-automatic program refinement process draws on work such as Interlisp and the relationship of EMACS and LISP that has a long history at MIT.

The model used by PAT has some important central properties:

1. subroutines are analyzed one at a time, with information from nested calls used as available;
2. loops with no dependencies are identified for optimization first;
3. loops with dependencies are analyzed incrementally under intelligent editor control using semi-automatic refinement;
4. line numbers from the original source are preserved throughout the improvement process to allow maximal use of programmer knowledge.

PAT supports three levels of interaction:

Manual Mode – where the user has complete control over transformations,

Power Steering Mode – where the tool makes suggestions using available information and follows user directions,

Auto Pilot Mode – where the tool will perform sequences of transformations independently.

The last mode is still a research area, but activity appears to be progressing. The program regions analyzed by the PAT under user control are:

- subprograms,
- loops,
- loops with governing if conditions,
- parallel regions (TBD).

An important aspect of subprogram analysis is control over the amount of called program information. Basically complete analysis is needed to handle all possible dependencies. However, this is very expensive and may normally not provide much information. The PAT solution will be to allow the user to decide how much of the call

graph to make use of. Full subprogram inlining is possible in the limit. The user may control how much of the call graph is included for analysis and make use of the multiple window browsing to coordinate which regions to be used. In this static analysis browsing mode, none of the program transformation actions may be invoked.

Program transformations are under control of a "sub-menu". Actions include adding to and deleting from source and moving sections of code. Dependencies can be affected by adding protection, adding alignment for loop indices, and replicating variables and assignments. Some of this can be automated, since if a dependence graph is a tree, all the dependencies may be removed by subscript alignment. This should be the kind of optimization that will increasingly be done automatically as PAT is improved. More globally optimal decisions such as process partitioning should continue to be done under programmer control.

More ambitious optimization is needed for real architectural tailoring. A more flexible approach would use PAT techniques to examine existing FORTRAN programs for opportunities for annotating parallel computations. A second phase similar to the FORCE tool for programming portable FORTRAN with parallelizable loops would then be needed.

Pisces

Developed at the University of Virginia, Parallel Implementation of Scientific Computer Environments (PISCES) is a FORTRAN-based environment with concrete language extensions for tasking and parallel blocks. It runs on a single processor VAX using UNIX processes for parallel execution, or a network of Apollo workstations. It depends on compiler-based analysis for higher-level program views and requires interpreter additions for dealing with different architectures. The main interest in Pisces is in the form of its task declarations, which are "clusters" having intrinsic properties (including "controllers", "subtasks", and message handlers). Its parallel blocks are concurrent C-style constructs. This project provides little basis for experimenting with either parallel languages or hardware. Its FORTRAN extension would provide a possible "intermediate language" to which very high level languages could be translated. Then, programs in this language would have to be translated further to lower level FORTRAN as supported by standard compilers. However, the sequential FORTRAN produced by the PISCES preprocessor may not be easily parallelized by another machine's parallelizing compiler.

R^n

This is Rice University's FORTRAN environment for parallel architectures. It has the most advanced compiler-based analysis of the environments surveyed. It utilizes most of the known static analysis and optimization techniques and performs vectorizing optimizations for most available parallel architectures in the context of a complete optimizer and historical program database. R^n is still a standard for supercomputer FORTRAN language environments. The developers have deliberately avoided ever becoming multi-language or

including any higher level formalisms in the source. All program development is done using a program database and associated analysis tools. The latest extensions support using execution information to improve optimization. By design, this effort provides no support for high level parallelism, and so, is primarily useful in the PEEP effort for its ideas.

Schedule

Developed at Argonne National Laboratories, to provide portability of scientific parallel applications, Schedule is a FORTRAN library of parallel support routines which allow application programs to perform synchronization and scheduling. It runs on VAX, Cray and various multicomputer hardware. It provides a way of dealing with limited operating system and architecture variations as a procedural extension to FORTRAN.

Special Purpose Tools

BALSA-II

BALSA-II is a program annotation and monitoring system under development at DEC System Research. It is an outgrowth of the BALSA program originally developed at Brown University. It involves programmer specification of significant program points, and use of programmed tracing routines to cause system monitoring and data collection at run time. The data obtained is used to monitor execution in whatever is determined to be the most critical or significant way. Typically, this involves data usage or execution frequency information that can be used to significantly improve algorithm performance.

BALSA-II provides a graphical interface which allows the user to vary the point of view under control of a mouse. It communicates with the program viewer to display the section of the program graph affected. The ability to view results is important for providing real data to guide performance improvements. It is not specifically related to parallelism, but could be used to explore algorithms executing in parallel by assigning monitoring elements to individual processing units. The importance of its features cannot be overemphasized in trying to characterize the behavior of a parallel system.

Trace data is often very difficult to interpret meaningfully, and often there is a need to relate behavior at different program levels. BALSA-II provides this kind of control, and includes a user interface that enables dynamic selection of monitored events and traces. The user can have the effect of "zooming" and "unzooming" to get a valid feel for how much activity is occurring in particular subsystems and subprocesses.

BALSA-II is a methodology for annotating and interpreting programs of any kind and does not depend on the language or the kind of problem. Initially, it makes use of an extended Pascal, but it can support any language. It makes good "meaningful use" of color in its graphics, shows the possibilities of graphics in program monitoring, and suggests the

importance of user-supplied high-level structuring information for programs in any implementation language.

Since the Garden system has taken many of the same concepts and integrated them into it's multi-lingual graphics front end, these concepts are available in either system.

Dalek

Dalek is a debugger developed at the University of California at Davis to run on top of traditional debuggers such as sdb. It is primarily an event oriented debugging language useful in any dynamic program analysis, including traditional testing. It allows logical operations, queries, conditional execution-time actions on events, and monitoring and profiling under user control. It can be used like compiler and linker-inserted probes for run-time monitoring and can perform dataflow analysis of event relationships like intelligent debuggers. It relies on user-supplied event definitions and can be used with any language. Dalek uses the techniques of traditional debuggers, but is more powerful in that its event language is a kind of monitoring meta-language. Its role as a sequential debugging/testing tool could be adapted to support parallel program tracing and performance analysis.

E-L

E-L (for Environment and Language) is a transformational framework developed at Harvard and supported by Software Options, Inc. for refining programs using a variety of software methodologies and for compiling to many targets. E-L supports:

- a flexible linguistic medium to allow the user to specify the domain of discourse
- an open-ended tool set to allow existing and new tools to interact easily
- a base of operations and principles of extension for manipulating programs

While based on principles of language extension, the E-L approach is different from standard language compiling in that programs are manipulated by tools whose base transformations are modified to produce interpretable E-L at successive levels of semantic refinement. The final level is a program using primitives close enough to the desired hardware operations that standard code generation techniques can be used to compile to executable programs.

E-L has been successfully used to model a UNITY compiling process that provides UNITY syntax and semantics, as well as a facility for structuring UNITY programs to allow checking of meaningful program composition.

FIELD

The Friendly Integrated Environment for Learning and Development (FIELD) is a interactive, UNIX workstation based programming environment that runs on top of X-windows. Developed at Brown University, it provides a variety of tools for program and data visualization and offers a consistent and integrated graphical interface. FIELD's main

feature is a central, selective broadcasting mechanism that lets new tools be integrated into the system with little effort in the future.

The tools currently integrated into the FIELD environment include a syntax directed editor, debugger, cross referencer, data structure displayer, Make interface, profiler, and general system viewer.

GARDEN

Developed at Brown University, GARDEN is a "visual programming environment" that has been used to model a wide variety of parallel programming methodologies. It supports:

- rapid implementation of design languages to facilitate "conceptual programming"
- object-oriented program graphs
- textual interface with Lisp-like Functional Programming
- primitives for object properties including concurrency
- translation to C to achieve compiled-code performance of the simulation

GARDEN has been used to model data flow systems, finite state path languages, Petri nets, a CSP-like ports language, Linda and a MultiLisp system. Extensive engineering of user interfaces has been done to support monitoring. There is a capability for supporting many windows, monitoring specifically programmed events, and the interpreter allows any desired arrangement of processes and processors to be monitored at varied levels of description.

Like many of the parallel environments, GARDEN supports instrumentation either automatically, by inserting non-intrusive probes at significant points, or by programming. This data can be used with a program viewer for value tracing, or to monitor significant activities, or can be analyzed to generate various kinds of control traces or dynamic utilization displays. For object monitors, GARDEN communicates with the program viewer to display the section of the program graph affected.

As a programming environment GARDEN is incomplete in that it supports neither the reading of source code for the modeled languages into the GARDEN environment nor the output of objects from the system into their textual source format. This problem can be solved by defining methods for both reading and writing source along with the other objects for the modeled language.

Hypertasking

Hypertasking is a software parallelization tool for the Intel hyper cube. It is currently available as an unsupported tool from Intel for the iPSC/2. Currently it does not run on any other parallel hardware.

IPS-2

Developed at University of Wisconsin-Madison, IPS-2 is a performance measurement tool for parallel and distributed programs. Measurements can be displayed at different levels of abstraction including the program's behavior as a whole, that of processes, procedures within processes, or primitive machine level activity.

Programmers do not need to modify their programs to use IPS-2. The raw data is gotten through the use of instrumentation probes which are contained in the run-time system and operating system kernel. The user chooses to use them via compiler and linker options. Their output is kept in a data pool and analyzed when requests are made through the user interface.

IPS-2's graphical user interface lets the user assign executable images to processes, processes to machines, select statistics for display and begin the execution of the program interactively. IPS-2 then displays the requested performance statistics. Additionally the tool locates bottlenecks by performing Critical Path Analysis. IPS-2 can also perform Phase Behavior Analysis which automatically identifies phases in the execution history of the program so that they can be studied more carefully.

IPS-2 has been used to provide data for analytical performance models of parallel systems, and to measure the characteristics of parallel database join operations, parallel searches, and network flow programs. User feedback suggests the tool is easy to use and useful.

MPD

Multi-Threaded Debugging is work done at Columbia University using Mach C Threads. The methodology uses global control flow analysis to identify partial orders of events, and to construct predecessor automata. These automata are used to instrument programs and provide meaningful characterization of debugging and trace information. The debugging language allows the user to assist in analysis and to use identified events in path expressions to control monitoring activity and filter trace data.

This is a general, language-independent, debugging technique, including sufficient source analysis to support tracing of parallel execution. It can be used with other language preprocessor and transformation techniques to provide support for visualization used for either algorithm understanding or performance analysis.

Olympus

This is an interactive modeling system, however, its graphical interface is Sunview not X. It is unusual for people to still be using Sunview, so it might be worth investigating whether the developers have moved to X since we last talked to them.

PADWB

Parallel Algorithm Development Workbench is a tool that facilitates the evaluation of algorithms by simulation. The user models the communication overhead, inter-connect strategy, CPU, etc. to obtain a graphical description of the system's behavior. The output shows the relationship between communication and CPU usage including idle times and periods of overlapped processing and communication for all processors involved.

The modeling is decomposed into three layers: Network, Cluster, and Processor layer. There is a software interpreter that accepts the code under test as input and a Graphics post-processor that gather's combined input from the other layers and generates output.

Basically the algorithm is interpreted by the "software interpreter" which generates an instruction stream to the "processor flow model". This layer does instruction counts and accounts for the relative speed of different instructions. This stream is, in turn, fed into the "cluster model" which models a group of a processor's contention for shared resources. It is here that time required for bus access, memory access and I/O are accommodated.

The inter-connection between processors, and the cost of communication, is modeled by a "network model". This section is the heart of the simulation as it relates to parallel processing. It can represent various inter-connection methods and demonstrate their suitability to a proposed algorithm.

All the models are process oriented next event simulations written in Simula with DEMOS extensions and in C. Some component behaviors are modeled by stochastic assumptions. Performance is a concern with this type of simulation especially when very detailed models are used. Another consideration when assessing the value of such experiments is the assumptions that are made with respect to exogenous events. These assumptions will to a large degree, determine the accuracy of the results.

Work on this tool at Gould is on going and as of 1988, the network layer was incomplete, although parallel experiments were being conducted by using the processor layer. The experiments were small, involving known problems (matrix multiply) with only a few processors (2).

PAWS

Developed at Syracuse University, PAWS (Parallel Assessment Window System) is an integrated X-Windows based toolset for evaluating existing or hypothetical machine architectures running a common application. It translates source programs from a high-level source language into the form of a dataflow graph. It maps this intermediate form into an executable characterization using target machine description models that currently reflect a SIMD (Thinking Machine's CM-2), a shared memory MIMD (Encore Multimax) , and a message passing MIMD (hyper cube) architecture. It supports user-controlled monitoring that relates dynamic information back to the graph of the source program.

The source languages supported are SISAL, a single assignment language based on stream operations, and Ada. Programs are translated to a graphical dataflow language IF1, an intermediate language for SISAL. The partitioning of processors and evaluation of performance information all make use of user controlled heuristics and actions performed during walking this graph. The machine descriptions include processors, memory characteristics and interconnection topologies. These are created using an interactive tool that allows a complete hierarchical description of the target hardware. This is also in the form of a graph, with a structured arrangement of information about processor, memory and network/connection characteristics. Query parameters allow reference to data from these structures. The query types for performance data include times for arithmetic and logical computation, communication costs and network overhead.

An interactive display tool provides the user interface for accessing all the PAWS tools. This is a hierarchical menu-driven system, supporting multiple windows for programming, interpreting and evaluating parallel algorithms. Called the IGDT, this displays graphs hierarchically, allowing the user to select a node and expand or contract the field of view to display as many nodes as desired. The same interface is used to describe all the application code, architecture description and assessment options. Similar menus support specification of compiler options when the Ada compiler is invoked and selection of performance metrics during evaluation.

The normal mode of using PAWS for assessing an algorithm is to run a compiled program on a theoretical architecture assuming an ideal number or arrangement of processors for the program first. Then the program is run on a variety of architectures more closely approximating intended targets. The effective speedup or other cost metric can be determined from the monitoring data collected for each run. The algorithm can be debugged using the same kind of facilities on the ideal machine, or on adaptations to reveal different possibilities of data and processor usage.

The architecture description framework is just a graph structure, built with the same tools as the basic PAWS facilities. This structure includes relationship to a predefined architecture characterization structure. This describes:

- the number and flexibility of different functional units,
- the number of processors,
- memory bandwidths and memory hierarchies,
- the types of interprocessor communication mechanisms.

There are five areas hierarchically partitioned into a fine enough detail to provide desired granularity of measurement. These areas are computation, data movement, communication, control and I/O.

Both parallelism and execution profiles are generated/computed by walking the program dataflow graph. The walk traverses the graph recording and evaluating each node's performance and statistical data. To handle compound nodes, the walk recurses, thus allowing statistics for individual procedures and blocks to be recorded and maintained individually and evaluated hierarchically. Recursive calls in the application are implemented as bounded loops, with a bound based on estimation or actual data from previous runs.

PAWS provides a way of modeling almost any architecture. By using IF1 and performing all analysis on an arbitrary graph, any level of computation can be modeled and then mapped to an architecture using graph pattern matching. Data for comparing performance on categories of architecture could be obtained by adjusting the architecture graph, thus allowing the equivalent of general simulation. Information on processor usage and communication overhead can be obtained in a structured way with a flexible machine description. This facility is an important technique for evaluating processor decomposition and interconnection strategies.

The system has an Architecture Characterization Tool that lets a user specify the architecture. A machine-independent version of the application (a data dependency graph) is then generated from a standard high-level language with the Application Characterization Tool, and is "executed" on the machine. The performance of the proposed architecture is profiled with a Performance Assessment Tool and a graphic display is implemented via an Interactive Graphical Display Tool. These tools generate parallelism profiles and speedup information in an intuitive graphical format.

PAWS is unique in that it provides for an assessment of the synergy between an application domain and its architectural platform. With these tools a user can determine the efficacy of a particular machine organization for his particular problem set.

PProto

PProto (for Parallel Prototyping system) is a CASE tool with graphical input of parallel systems specified as functional elements mapped to parallel architecture models for simulation and evaluation. Functional elements are refined into interpretable components and used to debug and measure models of algorithms on various architectural models. PProto provides:

- prototyping capabilities addressing functional, structural, timing and behavioral abstraction issues, allowing multiple versions of a system prototype to be modeled and evaluated
- concurrent execution model involving systems of multiple communicating processes simulated on very general multi-processors
- support for top-down incremental development via component reuse
- support for studying software/hardware mapping tradeoffs through identifying architectural resource impacts
- sophisticated simulation capabilities, including utilities for interpretation, scheduling, resource modeling, instrumentation, animation and debugging

A PProto specification is a hierarchical data flow graph consisting of processor elements, data storage units, communication connections and ports. The behavior of each leaf of a model is described with a simple structured programming language (SDDL —Structured Data Definition Language) which is supported by the PProto interpreter. Tools include:

- graph editor, for producing data flow graphs,
- behavior editor, for editing SDDL node behaviors,

- schema editor, for menu-based editing of data storage units,
- reuse facility, for saving components of models,
- architecture editor, for graphical editing of hardware topologies,
- mapping editor, for specifying software/hardware component mappings,
- prototype simulator, for execution, animation and debugging.

PProto is a process description language and persistent database that supports modeling and performance analysis for parallel hardware design. It uses behavioral modeling, with process models and a concurrency notation that includes guards and actions similar in spirit to Dijkstra's guarded commands. PProto allows direct mapping of these high-level specifications to target hardware descriptions that can include processors, memories and buses as basic configuration items. It can perform simulation analysis with mechanisms similar to debugging commands, and supports user programmed monitoring at the model level. It reports statistics on communications, including size of buffers, which enables detection of bottlenecks.

TANGO

Tango (Transition based ANimation GeneratiOn) is an object oriented algorithm animation system developed at Brown that lets the programmer illustrate the "interesting events" within a program without having to write any low level graphics code. This package can be used with the FIELD programming environment described earlier.

VMMP

The Virtual Machine for Multiprocessors (VMMP) is a software package that provides a set of services to write parallel applications for MIMD multiprocessors in a machine independent fashion. This is done by abstracting common requirements of parallel algorithms into a set of procedure calls whose implementation are hidden from the user.

The author identifies 2 paradigms for parallel algorithms: "tree computations" and "crowd computations". Tree computations are a strategy which solves problems by breaking them into subproblems and assigning the subproblems to child processes. "Crowd computations" are groups of processes executing the same code on distributed data and synchronizing with messages. In other words a message passing based data-parallel approach. VMMP also offers a set of functions that creates shared objects available to all processes. Shared objects are provided so that the programmer can distribute data, collect results etc.

An example of a tree computation entry points is Vcall. Vcall creates a child process. Vwait allows the parent process to block until its children terminates.

An example of a crowd services is Vcrowd. Vcrowd creates some number (which is passed in as an argument) of processes all executing the same code. Vsend allows the programmer to send a message to any or all processes within a crowd. There are functions

to specify a topology to a crowd to help optimize communication. VMMP supports a ring, torus, mesh, n-cube and tree.

VMMP supports associate functions that operate on distributed data via the Vcombine entry point. Associate operations are used with shared data objects which can be created with Vdef.

The VMMP hides low level communication and synchronization requirements of a problem from the programmer. Instead he is offered a set of high level operations that are typically used in parallel algorithms. These parallel constructs, however, have known communication and synchronization requirements for a particular architecture. This allows run-time optimizations to be made transparent to the user. Never the less, only a good marriage between hardware and parallel coding paradigm can insure an efficient program. For instance, shared memory operations entail a high communication overhead on a MIMD machine which uses discrete memory for each processor. A general disadvantage of this kind of approach is that the relationship between a chosen construct and its efficiency is not obvious.

The VMMP has been implemented on two shared memory multiprocessors (an ACE and MMX) and three uni-processors, and is being implemented on a distributed memory multi-processor. It is used as the intermediate language for a portable parallel Pascal Compiler.

The main advantage of this system is it portability. It is appropriate for medium and large grain parallelism.

Parallel Languages

*C**

C* is a data-parallel language that is based on C and C++. C* was developed at Thinking Machines Corporation for the Connection Machine. The aggregate data structure is the "domain". A new statement type, the "select" statement, defines parallel execution within domains.

The sequential code that each processor executes in parallel is standard C code. The index within a domain specifies the virtual processor that a domain element resides on. Incrementing or decrementing a domain index identifies the adjacent processor. This gives domains a grid-like shape.

C* is completely synchronous. Virtual processors execute their instructions as if the host processor were broadcasting the instruction at all the processors in typical SIMD fashion. MIMD operations can be executed by dereferencing pointers to functions because dereferencing operations occur in parallel.

It is interesting that C* programs have been used successfully to develop data parallel programs on a shared memory MIMD machines (Sequent), and on a hyper cube (NCUBE 64). These efforts support the contention that data-parallel programming has applicability to a broad class of architectures.

Crystal

Crystal is a functional language developed at Yale University which supports "macro" data parallel programming. Crystal assumes a macro-parallel machine model with the following characteristics :

1. The machine consists of processors, each with local memory, named by a coordinate system (an index set) that will be mapped to the processor space of the machine by the compiler. Each processor can have local variables and read only constants.
2. Processors can execute two kinds of operations: they perform a computation (by executing processing functions) or they communicate with other processors (via communications functions). Processing functions operate on local variables. They take as arguments, other local variables, constants and remote arguments passed in from other processors. These remote arguments are gotten from other processors by executing communication functions. Remote argument values are sometimes the product of merging values (a many to one mapping) from many processors.
3. The crystal computational model includes the notion of a time step. During a time step a processor can execute a processing function without communication with any other processor. In other words a time step is a period of parallel computation that does not require synchronization.
4. A processor, via communication, can allocate and deallocate groups of processors to execute subroutines.

Crystal borrows constructs like sets, tuples and the notion of aggregate operators from older languages like SETL and APL. It is a functional language that uses conventional mathematical notation to describes a problem as a system of possibly recursive equations over an index set. The indices correspond to virtual processors.

Crystal's approach to compiler optimization takes advantage of the language's equational structure to automate program transformation. The notion is to generate equivalent programs from the original that have more efficient patterns of inter-process communication and data distribution.

Optimization efforts are helped by the language's functional nature which simplifies data dependency analysis. Most importantly the language has two constructs to help the programmer indicate to the compiler the data locality and recommended processor organization for a problem's solution. These are the "index domain" and the "data field".

An index domain is a data structure that describes the communication costs within the index set. It gives the index domain a "shape", and each index a position in a logical space. The data field is used to describe distributed data structures. The compiler can optimize the

assignment of data fields to index domains. When this changes the index domain the compiler uses a "domain morphism" to map the old index domain to the new, more efficient one. Once this is done, the equational nature of the language allows the compiler to transform the program mechanically into an equivalent program that matches the more optimal index domain.

Programmer insight into the nature of the algorithm is needed to choose the most efficient index domain, and index morphisms. This means that except for a narrow set of problems, a Crystal programmer must understand the parallel nature of the solution, and be explicit with respect to data locality and inter-processor communication as embodied by the index domain and data field constructs.

DINO

DINO (for Distributed Numerically Oriented Language) is an extended parallel programming language similar to C*. Users declare a virtual parallel machine that is best suited to the algorithm, as well as a global data structure and communication patterns, using procedures running concurrently as parallel processors. The language is C augmented with high-level parallel constructs that allow the programmer to specify the data decomposition scheme and communication pattern that is best for the particular algorithm. The DINO compiler transforms the program expressed in terms of (virtual) data parallel processes into an efficient program whose number of processes is the same as the number of available physical processors. This reduces the overhead of many virtual processes and is very useful on MIMD machines that have a small number of physical processors and relatively slow inter-process communication. DINO is intended to be used primarily to write data parallel programs on MIMD distributed-memory machines. It enables massively parallel SIMD-like numerical computation to be programmed in a pseudo-SIMD fashion. Its effectiveness depends on the process contraction optimization performed by its compiler.

Id

This is a high level language which supports fine grain parallelism and deterministic behavior. Id is a data flow language and its approach to parallelism is quite different from other languages in allowing implicit parallelism. Id assumes that the programmer should not be forced to identify and express the parallelism in a problem. Rather, the language's operational semantics should allow the compiler to extract implicit parallelism.

Id also supports the notion that algorithms written in it should be determinate. Typically, parallel execution of multiple processes are asynchronous because of variations in scheduling and processor speeds. Determinate results require the program to manage its own synchronization. Functional languages automatically guarantees deterministic behavior. Id is a functional language augmented with a determinate, parallel, data structure called an "I-structure". I-structures resemble arrays and possess a property called "non-strictness". Non-strictness increases the opportunity for parallelism because elements in a data structure

can be read before the all the elements have been defined. This fits the operand driven characteristic of data flow architectures because operands that are data structure elements can be used as soon as they are ready.

Id programs are compiled into dataflow graphs, which are the machine language for the MIT Tagged-Token Dataflow Architecture. Id is very well suited for programming this kind of non-Von Neumann target machine. Its non-strict semantics (which are at the heart of its ability to capture fine grain parallelism implicitly) make it difficult to partition code into a number of sequential threads that can be run on conventional sequential and parallel architectures. The tagged-token dataflow architecture uses instruction-level synchronization to do all scheduling of computation. This dynamic scheduling insures optimal use of processor elements. Communication is subsumed in parameter transmission. Synchronization is a requirement of execution of any function or operation.

Id is based on the premise that parallel programming cannot be made comparable to sequential programming by extending existing languages. One problem is that the extensions are often architecture-specific and reflect fundamental properties of memory hierarchies, interconnection topologies, and non-uniform address spaces. Another problem is indeterminacy--the problem of data races on access to shared variables requires complicated coding and debugging methodologies. The program cannot be guaranteed to avoid time and configuration dependent behavior. Another problem is that in many applications, it is not possible to maximize use of a highly parallel machine. The problem of optimizing partitioning, scheduling and synchronization activity is difficult even when the application can be described deterministically. The challenge of using faster and more massively parallel machines itself leads to attempts to solve more complex problems better. In this environment, no single problem solving technique is adequate.

This is the environment where functional languages with implicit parallelism match the problem domain best. A user's coding style is not affected by considerations of parallel processing resources. In Id, opportunities for parallelism arise from evaluating the arguments of a recursive function in parallel, and from concurrently executing the producer and consumer of a data structure. This concurrency never gives rise to nondeterminacy because there is no "updateable" data storage and hence no possibility of data races. Id allows exploitation of parallelism in expressions, iteration, recursion and allows a higher degree and finer level of parallelism than is practically obtainable from other approaches. In addition, Id contains basic array structuring facilities that are new in purely functional languages, and allow vector-based parallelism (the basis of most numerical parallelism) to be exploited fully. Id addresses this by a combination of language features, compiling technology and innovative tagged hardware that yields a powerful unified solution.

The unit of synchronization is the I-structure. These are indivisibly written once only, and can be accessed for reading only after they have been written. They are the fundamental synchronization mechanism for all computation. Additional persistent arrays called M-structures provide support for non-deterministic parallel graph processing or multiprocessing and for resource management. To avoid idling, it is necessary for the processor to interface with data flow using minimum overhead and to switch rapidly to other threads when data is not available. Dataflow hardware features instruction level forks and

joins to allow large numbers of threads to execute on a single processor. Thread-scheduling is data-driven--an instruction is never attempted until its data are known to be available. Threads may be as short as a single instruction so that parallelism may be exploited at the finest grain. All long-latency requests are "split-phase" transactions: each request carries with it a thread descriptor which is returned to the processor along with its response. Threads may thus be resumed regardless of the order of arrival of responses. I-structures cease to exist when they are no longer referenced; M-structures have global properties associated with their managers.

Programming in Id involves a mixture of mathematical and logical styles, with most known programming language features available is some form. Arithmetic operations and expressions, as well as basic ALGOL-tradition control structures are available. LISP-style recursive function definitions based on efficient list processing are the fundamental units of computation. Id includes a full library of basic arithmetic and string manipulation functions. Its functional style depends fundamentally on list structures and basic list operations. It includes array and record structures with a full set of basic operations. A pattern matching notation is used to do a variety of list, array and record selection operations.

Linda

Linda is a high level set of language extensions for parallel programming developed at Yale University. As described by its developers in a *Communications of the ACM* article:

> "In the research community, discussion of parallel programming models focuses mainly on message-passing, concurrent object oriented programming, concurrent logic languages, and functional programming systems. Linda bears little resemblance to any of these."

Linda involves the "explicit creation of parallel execution threads" by modeling activities as a collection of active and passive "tuples". Active tuples are a bit like processes. Passive tuples can be thought of as data available to anyone that wants it.

There are four operations defined over "tuple space": out creates a passive tuple (some data). In selects an existing tuple that matches a criteria and removes it from "tuple space". Should the sought for tuple not exist, the "in" statement blocks until the data becomes available. Read acts like "in" except it reads the tuple, but does not remove it. Eval creates an active tuple (data and some code that acts on it). When the code finishes executing, the result is a new, passive, tuple left in tuple space.

Linda operations can be added to any language, and have been used with FORTRAN, C and C++, Scheme, MODULA-2 and others. Linda promotes an "uncoupled programming style" that does not require any specific relationships (client-server, master-slave). All the main models of parallel programming have been successfully programmed in Linda-C, with noticeable improvements in understandability and program usability. The Linda object manager is a "super operating system" that performs dynamic object management under concurrency constraints. This is a very complex interpreter, that runs on Sun and VAX workstations.

Linda's authors feel that the language is not suited to SIMD architectures, but would be: "an excellent language for massively parallel, fine grained, asynchronous machines". Linda systems are commercially available for use on homogeneous networks of Sun workstations, shared memory MIMD machine manufactured by Encore and Sequent, as well as the distributed memory Intel iPSC/2 hyper cube.

Linda is especially suitable for modeling in a very general environment that does not require programmer interaction with resource control. Modern distributed MIMD environments are probably the area of its greatest potential.

**Lisp*

*Lisp is a parallel extension of the Common Lisp language, and has the same syntax and style as Common Lisp. *Lisp adds one major data type to Common Lisp: the parallel variable, or pvar. This is an abstract data object that represents the concept of a value stored in the memory of each processor on the CM. *Lisp also adds a large number of functions and macros that operate exclusively on pvars. Among these operators are parallel equivalents for many of the operators available in Common Lisp, as well as special-purpose operators that perform such CM-specific tasks as processor selection, interprocessor communication, and scanning.

*Lisp is available in two versions: as an interpreter/compiler combination for the CM hardware, and as a stand-alone simulator. The *Lisp interpreter and compiler are extensions of the existing interpreter and compiler in Common Lisp, and *Lisp programs are written and compiled no differently from Common Lisp programs. The *Lisp simulator runs entirely on the front-end computer and simulates the operations of an attached CM. Code developed using the *Lisp simulator can always be ported directly to the *Lisp interpreter/compiler on the CM hardware with few modifications. However, code compiled using the simulator must be recompiled to run on the hardware.

Molecule

Molecule is a language effort that involves defining new classes of program constructs called "molecules". A molecule belongs to one of three levels of program abstraction. The highest level uses data-flow semantics. These molecules are used to write applications in a machine independent way. The second level is the "mode" layer. Molecules of this class correspond to the architectural characteristic of the target machine and the kind of synchronization and communication facilities that are appropriate with that style of hardware. For instance, molecules in the mode layer might be concurrency primitives (send/receive) for use on a discrete memory MIMD message passing machine. Each class of "mode" molecule represents a kind of intermediate language that is tailored to a particular kind of parallel machine. A program transform tool called a "molecularizer" converts the application layer molecules into a program expressed in the molecules for the a mode.

The lowest level of molecule is the machine layer, which is the language for which a compiler on the target exists. A "source code to source code pre-compiler" is envisioned that would take the output of the molecularizer, and translate it into the appropriate language for the compiler automatically.

The emphasis of this project is to support a programming methodology that allows a programmer to write in a data flow language. The researcher feels that such a data flow language is the most user friendly, and totally divorced him from the architectural requirements of the execution environment. Then tools exist to perform transforms on that language into an intermediate, and then a machine language suited to the class of machine of interest.

The layered software development approach outlined above has been tried successfully on a parallel matrix inversion problem using a data-flow and parallel dialect of PAL (a Pascal like language) for the application and mode layer. The machine layer language is iPSC hyper cube C with message passing extensions.

MultiLisp

MultiLisp is a version of a Lisp dialect called Scheme developed at MIT. Variants of the language are available on a several parallel computers including the Encore Multimax, the BBN's Butterfly, and the Alliant FX/8.

Parallel versions of Lisp typically support concurrency by adding an extension called a "future" to the base language. The form, (future X), immediately returns a future for X plus creates a task to evaluate X. The future of X is can be thought of as a place holder for the evaluated value of X. It can be used until the value becomes available. The future should eventually resolve to a value when the child task spawned to evaluate X completes its work.

If a task "T" tries to use a future of X, when it, in fact, needs the resolved value, "T" will suspend until the child process is through computing it. For some operations, like the movement of a value from one location to another, as in an assignment statement, or the passing of a parameter, the future can be used while the value is being resolved. This allows a style of computation similar to dataflow.

The future can be used to perform data parallel operations. For instance, the MultiLisp library function pmapcar takes the future of a function for every element in a list. This results in the same function being evaluated in parallel for all elements in the list.

Critics of MultiLisp contend that the language is not transparent: it is difficult to know when to use the future construct or how much it will cost.

Occam

Occam is based on the Communicating Sequential Process (CSP) construct developed by Hoare. An Occam program consists of active agents called "processes" that communicate with each other via one way, point-to-point links between processes called "channels". There are three kinds of processes: assignment, input and output. Assignment processes change the value of variables. Input processes read from channels, while an output process writes to them. Such communication is synchronized, i.e. the inputting process will wait for another process to output to the channel.

These elemental processes are like statements in a conventional language. They can be combined into a series of actions that occur sequentially with a "seq" statement. Or they can be forced to occur in parallel with a "par" statement. Such combinations are themselves processes, and, as such can be used in a par or seq statement. A par process blocks until all the processes within it complete.

Occam is strongly typed and the scope of objects is limited to the process immediately following the declaration. Processes can only communicate with channels. Occam does not support shared variables.

Occam also supports an iterative construct, "while" and two control flow constructs: "if" and "alt". The "if" is similar to "if" statements in conventional languages. The "alt" statement selects one of a group of alternatives depending on the state of a channel.

Programming style in Occam connects large processes via a channel. The data is read from a channel, manipulated by inserting processes after the receipt of the data, and output to a different channel. It was developed to program the INMOS transputer and as such, is a good match for MIMD architecture with discrete memory where the communication costs to the adjacent processors is cheap.

Paralation Model

The Paralation model is specification for language extensions consisting of a new data structure — (a "paralation") and new operations ("elementwise evaluation", "mapping" and "match"). They comprise a parallel programming model that "can be combined with any base language to produce a concrete parallel language." The resulting code is easy to compile efficiently because the constructs contain explicit non-machine dependent information about data locality, appropriate processor topology, and synchronization requirements. It is machine independent because compilers can choose to use or ignore this information depending on the machine organization.

There is a version of Paralation lisp based on common lisp that runs on the Connection Machine. A version of Paralation C is under development.

A "paralation" is composed of "fields". A field is composed of elements, which are pieces of data that are all the same size. Every paralation has one field that contains its indices. An index numbers can be thought of as specifying a "site" or processor.

A rough analogy to a paralation is a table where each column is a field. Each row has an index number that specifies the "site" or processor that the row specified by that index number is processed on.

The model guarantees locality for data used at a processing site, and for all the processing sites within a paralation. This means all the elements in a row will be stored close together in memory. Additionally, all the processors associated with a particular paralation are logically "close" to each other. The distance between sites within a paralation is determined by the "shape" of the paralation. The shape is under programmer control. For example, if a paralation were shaped like a ring and had 3 sites, the shape captures the fact that site 1 is logically 1 site away from both site 2 and site 3.

The Paralation model supports three operations: "Elementwise evaluation", "mapping" and "move". Elementwise evaluation is an operation that acts on a field of the paralation. It performs the same operation on each element and creates a new field containing the result. Since each element in a field resides at a different site, operations occur in parallel. The elementwise evaluation is complete when all the sites finish executing this code. Elementwise evaluation is the only construct for synchronization supported between sites. Any code that requires synchronization within the elementwise evaluation is considered an error.

Data is moved between paralations by the "move" operator. Elements to be moved are determined by a "mapping". A mapping is created by a "match" operation. Match operations allow the programmer to identify elements within one paralation that are to be associated with elements in a different paralation. The "move" can be thought of as a kind of parallel assignment statement which over-writes a field in a paralation with data from another paralation.

The Paralation model is compatible with any parallel architecture, but the author contends that "a crossbar-based shared memory machine is not the ideal target". This is because the shared memory implementation "ignores the locality properties of paralations and fields." The author goes on to state that "this is inherent in the idea of shared memory hardware, which hides the cost of communication." In other words, information about the locality of a paralation is of no use if the cost of communication from one processor to another is always the same. This model is best suited to medium or fine grained MIMD or SIMD machines.

PCN

PCN (Program Composition Notation) is an imperative block-structured expression language with guarded commands for parallelism. It attempts to combine the functional style of Lisp for structuring blocks and statement segments, with a concise style similar to C for composing statement and expression sequences. The declaration and block structure is intended to map to data flow graphs, and the composed segments to the actions of nodes in this graph. Relating segments to nodes allows monitoring and tracing to be done according to the structure specified by the program, and gives a programmable way to help make execution sequences meaningful for dynamic and post mortem analysis. Two tools Gauge

and Upshot are part of the interpretation system intended to this monitoring facility to provide execution profiles (e.g., to improve load balancing) and collect trace information to assist in performance monitoring and tuning. Upshot can be used to do complicated event tracing for post mortem analysis, debugging, and even testing in the presence of potential race conditions or deadlock.

The PCN system is a combination of a language, interpreter and library. It supports high-level programming and debugging as well as the special kinds of performance assessment already highlighted. It allows interface to FORTRAN and C programs via interface directives, and is intended to facilitate the connection of commonly used sequential segments using some simple notions to develop a parallel program:

- definitional variables, for controlling the exchange of information between concurrently executing potentially interfering program segments
- concurrent composition, for building parallel programs by composing segments into systems having a number of independently executing threads
- controlled nondeterministic choice, for expressing intrinsic parallelism of program segments, following Djistra's notion of guarded commands where the guards need not be mutually exclusive
- encapsulation of state change, by restricting use of data structures subject to state change to program segments where they are not shared

This composition strategy is very similar to the notions of CODE, except that hierarchies of sequential segments can be built and synchronization of shared data is based on a distributed rather than structured global data basis. This is generally more efficient, and certainly more programmable. PCN is oriented towards making parallel system programming a practical reality, and supporting monitoring and tuning in a style similar to current sequential development methods.

Proteus

This is a wide-spectrum language being developed at Duke with a range of parallel control constructs, intended to provide a complete set of possible notations for programming parallel algorithms. It supports a set of synchronization primitives intended to allow mutual exclusion on concurrent objects, with mechanisms provided to map the control primitives to MIMD, SIMD or specialized architectures. It supports specification of process sequencing and time constraints to allow flexible target independent real-time programming. It competes with languages such as Crystal which encourage refinement, but supports more specific and constrained concurrency implementation.

RAPIDE

Developed at Stanford University, RAPIDE is a process-structured expression language with guarded commands for parallelism. It attempts to combine the functional style of Lisp for structuring blocks and statement segments, with a process-oriented structure similar to VHDL for composing statement and expression sequences. The process

structure is intended to map to data flow graphs, and the composed segments to the events associated with nodes in this graph. Relating events to nodes allows monitoring and tracing to be done according to the structure specified by the program, and gives a programmable way to help make execution sequences meaningful for dynamic analysis and verification.

Strand-88

Strand is a single assignment language, with a functional programming style, supporting six programmable communication methods that model various kinds of parallel architecture. The fundamental notion is communicating processes, which synchronize using data flow constraints according to data availability. Strand is a parallel programming tool that includes a development environment and parallel programming libraries. Strand computes via reduction of a pool of extremely light weight processes defined by evaluating guarded-Horn clauses. Processes communicate by reading and writing shared variables (communication channels) which can only be defined once. A computation step involves removing a process from a pool. If the process is predefined, then it is executed immediately; otherwise a reduction attempt is made. This involves data flow synchronization. The process must wait until its constraints are satisfied; it then changes state and forks new goal processes, which are added into the process pool. This process continues until the pool is empty. Computation is then complete.

The programming system supports a variety of virtual machines and load balancing tools to ease the programming tasks and allow applications to scale when additional hardware becomes available. It is transportable across different MIMD shared or distributed memory machines. To enable the reuse of existing code, Strand supports a multi-lingual mechanism which consists of: (1) a logic programming approach that allows users to specify parallel program constructs and synchronization in a simple way, and (2) a foreign interface which allows a program construct to be implemented in existing sequential C and FORTRAN or by specialized hardware.

UC

Developed at UCLA, UC is based on the UNITY model of parallel programs. Development of UC was driven by a desire to separate efficiency concerns from programming issues when writing a parallel program. It is easier to develop and maintain a program if the language presents a simple virtual machine model to the programmer. On the other hand, execution costs are optimized by taking advantage of specific architectural features of a parallel processor such as the inter connection topology. However, with most parallel programming languages the modification of the code to exploit a particular architectural feature changes the meaning of the program perhaps causing a bug, or possibly obscuring the algorithm making the program hard to maintain.

UC is an extension of C with a few high level parallel constructs allowing the algorithm to be expressed in an abstract, high-level language. It adds one data type -

index_set, one operator - reduction, and four constructs to express dependencies among statements - par, oneof, seq and solve. It also disallows goto statements and only allows C pointers to be used to pass an array (or array slice) as an argument to a function.

An index_set represents an ordered set of integers, each of which must be a constant expression. The reduction operator is used to perform a binary operation on a set of operands. The par construct indicates that the following statements may be executed in parallel, while the seq construct indicates the statements must execute sequentially. The solve construct may be used to write programs that solve a set of proper equations. The oneof construct represents a non-deterministic choice. Initially the compiler generates a default mapping of the program data-space to the target architecture that allows a programmer to develop a correct prototype.

A separate map section of the program can be specified to override the default data mappings of the compiler. The data mapping can be improved to better match the underlying architecture without changing the correctness of the program. As a result the program is easier to develop and maintain while execution remains competitive with other languages.

UNITY

UNITY is the specification language used by Chandy and Misra in their 1988 book on parallel program design. It was intended as a way to unify parallel programming and verification in a single algorithm definition and proof system. It provides a means of expressing algorithms precisely in an architecture independent manner and provides a methodology for using architecture dependent refinements to produce proofs of program completion. This is the background for a formal system for verifying parallel programs.

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.