

TM-1028

Second Joint ICOT/DTI-SERC Workshop
on Decomposition of Parallel Applications
and Benchmarking and Evaluation
of Parallel Systems

by
S. Uchida

February, 1991

© 1991, ICOT

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Second Joint ICOT/DTI-SERC Workshop
on
Decomposition of Parallel Applications
and
Benchmarking and Evaluation of Parallel Systems

Monday, October 15 ~ Wednesday, October 17, 1990
Shiba Park Hotel & ICOT
Tokyo, Japan

Workshop Chairpersons: Dr. Paul Refenes (Dpt. of Trade & Industry)
Dr. Shunichi Uchida (ICOT)

Secretariat: Takashi Kurozumi (ICOT)
Hiroshi Hara (ICOT)

Content

1. Preface	S. Uchida, P. N. Refenes
2. Details of the Workshop (Schedule and Participants)	
3. Presentations	
3-1 Overview of the Final Stage R&D of FGCS Project	S. Uchida
3-2 Overview of Parallel Computing Research in the UK and Europe	P. N. Refenes
3-3 PIM Architectures and R&D Status	K. Hirata
3-4 Evaluation of the KL1 Implementation on the Multi-PSI	Y. Inamura K. Nakajima
3-5 Benchmarking and Evaluation of Software Systems the EDS and Flagship Project	P. Townsend B. Proctor P. Watson
3-6 Performance Aspects of the EDS Parallel Processing Machine	S. J. Cockcroft M. Ward
3-7 Research on parallel Inference Systems in the Fifth Generation Computer Systems Project	T. Chikayama
3-8 KL1 Programming environment-PIMOS-	H. Yashiro
3-9 Progress in the Development of the Data Diffusion Machine	D. H. D. Warren
3-10 A Programming Environment for Parallel MIMD Machines	P. Evan
3-11 Decomposition of Parallel Applications for SIMD Machines	G. Manning
3-12 Parallel Application Program Research at ICOT	N. Ichiyoshi
3-13 MGTP: A Hyper-matching Model-Generation Theorem Prover with Ramified Stacks	R. Hasegawa
3-14 Parallel Programming in LSI CAD Systems	K. Taki
3-15 Parallel Programming in Genome Analysis System	K. Nitta

3-16 Constraint Logic Programming and Its Parallel Implementation:	
Guarded Definite Clauses with Constraints	D. Hawley
.....	A. Aiba
3-17 Mapping Applications onto Various Parallel Architectures	
using Functional Language and Program Transformation	J. Darlington
3-18 Panel discussion	K. Furukawa
.....	D. H. D. Warren
.....	J. Darlington
.....	T. Chikayama
.....	N. Ichiyoshi

4. Guide to the Japan and UK Demonstrations

- 4-1 Japanese Demonstrations
- 4-2 UK Demonstration

Preface

It has been eight years since the Fifth Generation Computer Systems Project started. The main aim of the project was to develop new computer technologies for knowledge information processing unifying "knowledge processing" technology and "parallel processing" technology using logic programming.

It was several years before the start of the project that researchers of U.K. and Japan realized the importance of logic programming as a link between these two technologies. Since then, researchers in both countries have been exploring the new research field of logic programming in co-operation with each other.

In these eight years, a great deal of research effort has been made toward the goal of FGCS throughout the world and produced many excellent results. The Second Joint ICOT/DTI+SERC Workshop is the evidence of this progress and of international co-operation. Through the presentations and discussions made in this workshop, we are sure that the researchers could convince themselves of the appropriateness of their research direction and sharpen the focus for the future.

These proceedings consist of papers and lecture notes that can be divided into four groups. First, there are overviews of the final stages of the FGCS project, and of Parallel Computing Research in Europe and the U.K. These overviews summarize progress in the areas of Architecture, Operating Systems and application.

Second, there are papers on the topic of Benchmarking and evaluation of parallel systems. These papers describe both benchmarking strategy and specific benchmarking examples from ICOT.

Third, there are papers dealing with the problem of Decomposition and mapping of parallel applications. These papers describe methods and techniques for decomposing and mapping applications to architectures. They also describe software tools.

Fourth, there are papers describing the demonstrations done by both Japan and U.K. sides. These papers introduce what kind of problems can actually be handled by the current technology.

It is our sincere hope that the workshop proceedings and lecture notes will serve as a valuable reference service for the researchers being engaged in the parallel processing and knowledge processing in both countries for years to come.

Finally, we would like to thank you the many people from U.K. and ICOT who provided high quality input to the proceedings and the discussion sessions.

Shunichi Uchida
Research Department
ICOT

Paul Refenes
Information Technology Directorate
DTI

Second Joint ICOT/DTI-SERC Workshop

1 Date and Place

Oct. 15 (Mon)	Shiba Park Hotel
Oct. 16 (Tue)	ICOT Annex
Oct. 17 (Wed)	ICOT Annex

2 Topics of the workshop

- 1) Decomposition of parallel applications
- 2) Benchmarking and evaluation of parallel systems

3 Participants and their presentations

1) UK side

- D.H.D. Warren (U. Bristol): Progress in the Development
of the Data Diffusion Machine
- J. Darlington (Imperial C.): Mapping Application onto Various Parallel
Architectures using Functional Language and
Program Transformation
- P. Refenes(DTI): Overview of Parallel Computing Research
in the UK and Europe
- P. Evans (MEIKO): A Programming Environment for Parallel MIMD Machines
- G. Manning (AMT): Decomposition of Parallel Applications for SIMD Machines
- P. Townsend (ICL): Benchmarking and Evaluation of Software Systems
- the EDS and Flagship Projects
- C. Hughes(LOGICA)
- D. Watson(PARSYS)
- C. Sharpington(Thorn-EMI)

2) Japan side

- H. Tanaka (U.Tokyo): Welcome
K. Hirata (ICOT-1L): PIM Architectures and R&D Status
Y. Inamura (ICOT-1L) and K. Nakajima (Mitsubishi): Evaluation
of the KL1 Implementation on the Multi-PSI
T. Chikayama (ICOT-2L): Research on Parallel Inference Systems
in the FGCS Project
H. Yashiro (ICOT-2L): KL1 Programming Environment - PIMOS -
N. Ichiyoshi (ICOT-7L): Parallel Application Program Research at ICOT
R. Hasegawa (ICOT-5L): MGTP: A Hyper-matching Model-Generation
Theorem Prover with Ramified Stacks
D. Hawley and A. Aiba (ICOT-4L):
Constraint Logic Programming
and Its Parallel Implementation:
Guarded Definite Clauses with Constraints
K. Nitta (ICOT-7L): Parallel Programming in Genome Analysis System
K. Taki (ICOT-7L): Parallel Programming in VLSI CAD Systems
A. Goto (NTT)
S. Uchida (ICOT): Overview of the Final Stage R&D of FGCS Project
K. Furukawa (ICOT)
K. Fuchi (ICOT)
and a few more researchers from ICOT and manufacturers

4 ICOT demonstration (3rd day afternoon)

4.1 Parallel systems

(Written in KL1 and running on the Multi-PSI/PIMOS)

- Small benchmarking programs (Pentomino, Bestpath)
- Go-playing program: GOG
- LSI CAD programs (Logic simulation and Routing)
- Legal reasoning program
- Genome analysis programs

4.2 Sequential systems

(Written in ESP and running on the PSI/SIMPOS)

- Constraint logic programming language: CAL
- Molecular biological database in Kappa

5 Time table

Second Joint ICOT/DTI-SERC Workshop

(Tokyo, Oct.15-17, 1990)

	10/15 (Monday)	10/16 (Tuesday)	10/17 (Wednesday)
9:30	@ Shiba Park Hotel	@ ICOT Annex	
10:00		Warren (50)	Darlington (50)
10:10	Tanaka>Welcome (10)		
10:20			Break (10)
10:30	Uchida (40)	Evans (40)	Mini Panel (90): Darlington, Warren, Furukawa, Chikayama
10:50		Break (20)	
11:00	Refenes (40)	Manning (40)	
11:20	Break (20)		
11:30			
11:50	Hirata (40)		
12:00			
12:30		Lunch	
13:30			
14:00		Ichiyoshi (40)	ICOT Demonstrations (Yoshioka)
14:10	Inamura/ Nakajima (40)	Hasegawa (40)	
14:40		Coffee Break (30)	
14:50	Townsend (50)		
15:20		Taki (50)	
15:30	Coffee Break (30)		
16:00	Chikayama (50)	Nitta (40)	
16:10			
16:50	Yashiro (40)	Hawley/ Aiba (40)	
17:30			
17:50			
18:30			
	Reception @Shiba Park Hotel		Farewell Party @"Chugoku Hanten"

6 Participants

[British Participants]

Name	Organization	Department	Position
<i>Academia</i>			
Dr John Darlington	Imperial College	Computing	Professor
Dr David H.D. Warren	Bristol University	Computing	Professor
<i>Government</i>			
Dr Paul Refenes	Department of Trade & Industry	Information Technology Directorate	Consultant
<i>Industry</i>			
Dr Geoff Manning Dr Patrick Evans	Active Memory Technology MEIKO	Board	Managing Director Software Engineering Manager
Dr David Watson Dr Paul Townsend	PARSYS ICL-Manchester	Board	Managing Director Parallel Systems Manager
Mr C. Sharpington Dr Clifton Hughes	Thorn-EMI LOGICA		Principal Consultant

[Japanese Participants]

Name	Organization	Department	Position
Dr Hidehiko Tanaka	U of Tokyo	Electrical Eng.	Professor
Dr Hanpei Koike	U of Tokyo	Electrical Eng.	
Dr Atsuhiko Goto	NTT	Software Lab	Senior Researcher
Mr Rikio Onai	NTT	Software Lab	Supervisor
Mr Kenichi Yamazaki	NTT	Software Lab	Research Engineer
Mr Kazuhiro Kazama	NTT	Software Lab	Research Engineer
Mr Katsuto Nakajima	Mitsubishi	Info.Sys.Lab	Senior Researcher
Mr Hiroshi Nakashima	Mitsubishi	Info.Sys.Lab	
Mr Toshiaki Tarui	Hitachi	Central Lab	
Mr Takayuki Nakagawa	Hitachi	Central Lab	
Mr Koichi Kumon	Fujitsu Lab		
Dr Tsutomu Maruyama	NEC	C&C Sys. Lab	
Mr Takashi Usuki	Sony		
Dr Syuichi Sakai	ETL		
Mr Kenji Nishida	ETL		
Mr Hiroichi Hiroshige	ICOT		Executive Director
Dr Kazuhiro Fuchi	ICOT	Res.Center	Director
Dr Kōichi Furukawa	ICOT	Res.Center	Deputy Director
Mr Takashi Kurozumi	ICOT	Res.Center	Deputy Director
Mr Yoshihisa Ogawa	ICOT	Res.Plan.Dpt.	Manager
Dr Shunichi Uchida	ICOT	Research Dpt.	Manager
Dr Ryuzo Hasegawa	ICOT	4&5th Lab	Deputy Manager
Mr Kenji Ikoma	ICOT	Research Dpt.	Deputy Manager
Mr Tsutomu Yoshioka	ICOT	Research Dpt.	Managing Researcher
Dr Kazunori Ueda	ICOT	2nd Lab	Senior Researcher
Mr Kunjaki Mukai	ICOT	3rd Lab	Senior Researcher
Dr Takashi Chikayama	ICOT	2nd Lab	Chief
Dr Katsumi Nitta	ICOT	7th Lab	Chief
Dr Kazuo Taki	ICOT	1st (7th) Lab	Chief (Deputy Chief)
Mr Kazumasa Yokota	ICOT	3rd Lab	Chief
Mr Yuichi Tanaka	ICOT	6th Lab	Chief
Dr Akira Aiba	ICOT	4th Lab	Deputy Chief
Mr Nobuyuki Ichiyoshi	ICOT	7th Lab	Deputy Chief
Mr Masayuki Fujita	ICOT	5th Lab	Deputy Chief
Dr Keiji Hirata	ICOT	1st Lab	Researcher
Mr Yū Inamura	ICOT	1st Lab	Researcher
Mr David J. Hawley	ICOT	4th Lab	Researcher
Mr Hiroshi Yashiro	ICOT	2nd Lab	Researcher

Second Joint ICOT/DTI-SERC Workshop PROGRAMME

(Tokyo Oct.15~17, 1990)

10/15 (MONDAY)

At Shiba Park Hotel (Botan-no-ma)

10:00~11:30	Morning Session (1) <div style="text-align: right;">Chairperson: K. Furukawa</div>
10:00~10:10	H. Tanaka: Welcome
10:10~10:50	S. Uchida: Overview of the Final Stage R&D of FGCS Project
10:50~11:30	P. Refenes: Overview of Parallel Computing Research in the UK and Europe
11:30~11:50	{ Break (20 min.) }
11:50~12:30	Morning Session (2) <div style="text-align: right;">Chairperson: N. Ichiyoshi</div>
11:50~12:30	K. Hirata: PIM Architectures and R&D Status
12:30~14:00	{ Lunch (90 min.) } <i>at Ivy room (Shiba Park Hotel Annex 2F)</i>
14:00~15:30	Afternoon Session (1) <div style="text-align: right;">Chairperson: D.H.D. Warren</div>
14:00~14:40	Y. Inamura and K. Nakajima: Evaluation of the KL1 Implementation on the Multi-PSI
14:40~15:30	P. Townsend: Benchmarking and Evaluation of Software Systems - the EDS and Flagship Projects
15:30~16:00	{ Coffee Break (30 min.) }
16:00~17:30	Afternoon Session (2) <div style="text-align: right;">Chairperson: C. Hughes</div>
16:00~16:50	T. Chikayama: Research on Parallel Inference Systems in the FGCS Project
16:50~17:30	H. Yashiro: KL1 Programming Environment -PIMOS-
18:30~	{ Reception } <i>at Rose room (Shiba Park Hotel Annex 2F)</i>

10/16 (TUESDAY)

At ICOT Annex

9:30~11:00	Morning Session (1) Chairperson: P. Refenes
9:30~10:20	D.H.D. Warren: Progress in the Development of the Data Diffusion Machine
10:20~11:00	P. Evans: A Programming Environment for Parallel MIMD Machines
11:00~11:20	{ Break (20 min.) }
11:20~12:00	Morning Session (2) Chairperson: T. Chikayama
11:20~12:00	G. Manning: Decomposition of Parallel Applications for SIMD Machines
12:00~13:30	{ Lunch (90 min.) }
13:30~14:50	Afternoon Session (1) Chairperson: A. Goto
13:30~14:10	N. Ichiyoshi: Parallel Application Program Research at ICOT
14:10~14:50	R. Hasegawa: MGTP: A Hyper-Matching Model-Generation Theorem Prover with Ramified Stacks
14:50~15:20	{ Coffee Break (30 min.) }
15:20~16:50	Afternoon Session (2) Chairperson: Y. Inamura
15:20~16:10	K. Taki: Parallel Programming in VLSI CAD Systems
16:10~16:50	K. Nitta: Parallel Programming in Genome Analysis System
16:50~17:30	Afternoon Session (3) Chairperson: K. Hirata
16:50~17:30	D. Hawley and A. Aiba: Constraint Logic Programming and Its Parallel Implementation: Guarded Definite Clausus with Constraints

10/17 (WEDNESDAY)

At ICOT Annex

9:30~10:20	<p>Morning Session (1)</p> <p style="text-align: right;">Chairperson: K. Ueda</p>
9:30~10:20	<p>J.Darlington: Mapping Applications onto Various Parallel Architectures using Functional Language and Program Transformation</p>
10:20~10:30	{ Break (10 min.) }
10:30~12:00	<p>Morning Session (2)</p>
10:30~12:00	<p>Mini Panel: (Title to be announced)</p> <p>Coordinator: K.Furukawa</p> <p>Panelists: D.Warren, J.Darlington, T.Chikayama, N.Ichiyoshi</p>
12:00~13:30	{ Lunch (90 min.) }
13:30~17:50	<p>Demonstrations</p>
13:30~13:40	<p>Overview of the demonstrations</p> <p>1) Parallel systems</p> <p style="padding-left: 40px;">(Written in KL1 and running on the Multi-PSI/PIMOS)</p>
13:40~14:15	<p>Pentomino-Packing Piece Puzzle Solver</p> <p>Bestpath-Shortest Path Problem Solver</p>
14:15~14:30	<p>Go-playing program: GOG</p>
14:30~15:00	{ Coffee Break (30 min.) }
15:00~15:30	<p>LSI CAD program (Routing)</p> <p>LSI CAD program (Logic Simulation)</p>
15:30~15:55	<p>Functional Programming Environment</p>
15:55~16:00	{ Break (5 min.) }
16:00~16:25	<p>Legal Reasoning program</p>
16:25~16:50	<p>Genome Analysis programs</p>
16:50~17:00	{ Coffee Break (10 min.) }
17:00~17:25	<p>2) Sequential systems</p> <p style="padding-left: 40px;">(Written in ESP and running on the PSI/SIMPOS)</p>
17:00~17:25	<p>Constraint logic programming language: CAL</p>
17:25~17:50	<p>Molecular Biological Database in Kappa</p>
18:30~21:00	<p>{ Farewell party }</p> <p>at "Chugoku Hanten" (Chinese restaurant)</p>

Overview of the Final Stage R&D of FGCS Project

Shunichi UCHIDA
Institute for New Generation Computer Technology (ICOT)

September 30, 1990

Extended Abstract

FGCS project was substantially started from June 1982. Roughly speaking, it aimed at the R&D of following three major technological goals;

1. Knowledge processing
2. Parallel processing
3. Combination of above two using Logic programming

Since then, the project has developed following hardware and software systems for the research on parallel processing;

1. Sequential inference systems
 - 1984 PSI-I attaining 37 KLIPS (KL0) and ESP language and SIMPOS
 - 1986 PSI-II attaining 330 KLIPS (KL0)
2. Parallel inference systems
 - 1985 GHC
 - 1986 Multi-PSI/V1 attaining 1 KLIPS x 6PE (FGHC), and Parallel interpreter of FGHC
 - 1988 Multi-PSI/V2 attaining 150 KLIPS x 64PE (KL1), and PIMOS/V1 and small benchmark programs
 - 1990 VLSI chips and CPU boards for final PIM modules, and PIMOS/V2 and many small application programs
 - 1992 Final PIM modules, 256-512PE x 3 modules, attaining 300-400 KLIPS x N (KL1) = Target/module: 200-500 MLIPS and PIMOS + KBMS and medium scale experimental application systems

For the research on knowledge processing, a variety of experimental software systems have been built on the sequential inference system.

- A DBMS based on the nested relational model; Kappa-II, and an experimental system of deductive and object-oriented DB

Extended abstract

- An experimental system for natural language understanding; DUALS, and a tool-kit for NL research; LTB
- Constraint logic programming language; CAL
- Expert systems for VLSI design and diagnosis of electronic systems
- Theorem prover and a computer aided proof system; CAP
- A Go-game playing system; GOG

In the final stage, we plan to reconstruct some parts or some functions of these software systems on the parallel inference systems. Furthermore, we try to include such new application areas as genetic information analysis and legal information retrieval.

An image of our target system is illustrated in Fig-1. Our research activities in the final stage is illustrated in Fig-2. The trend of hardware technology which we used for our inference systems is illustrated in Fig.-3.

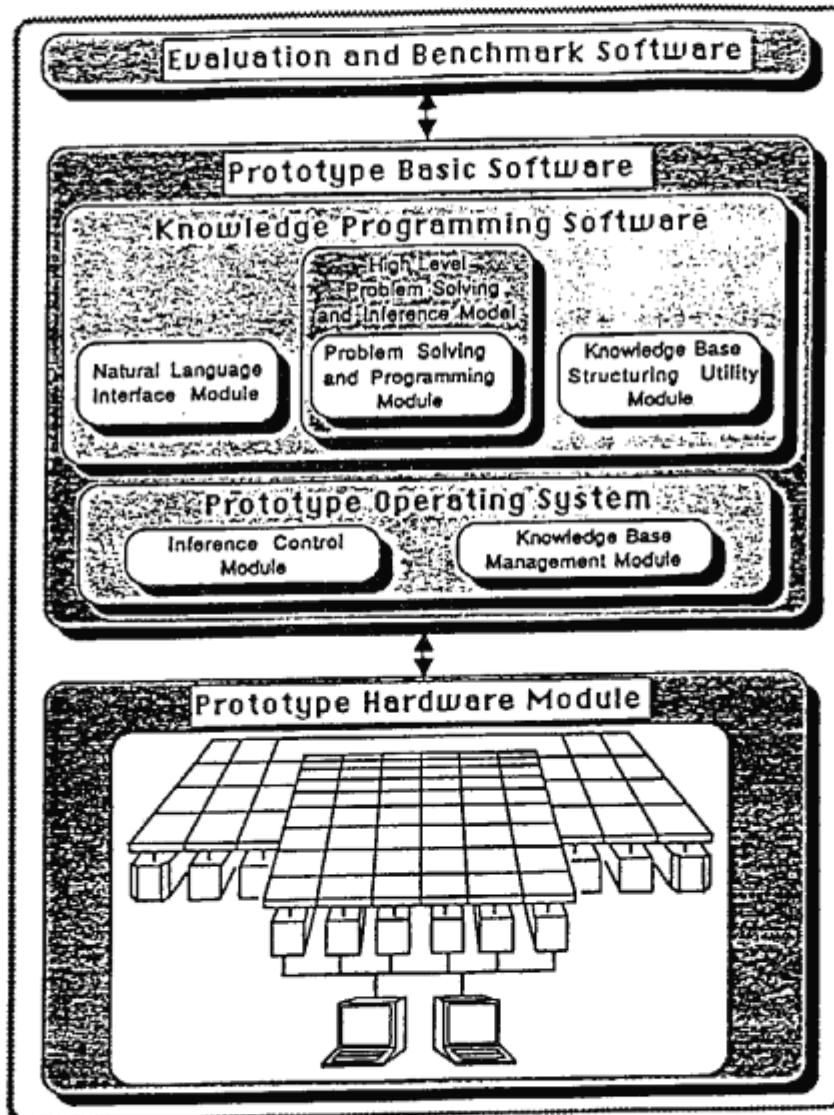


Fig.-1 The structure of the FGCS prototype system

Extended abstract

The experimental parallel application systems are important research topics newly added in the final stage.

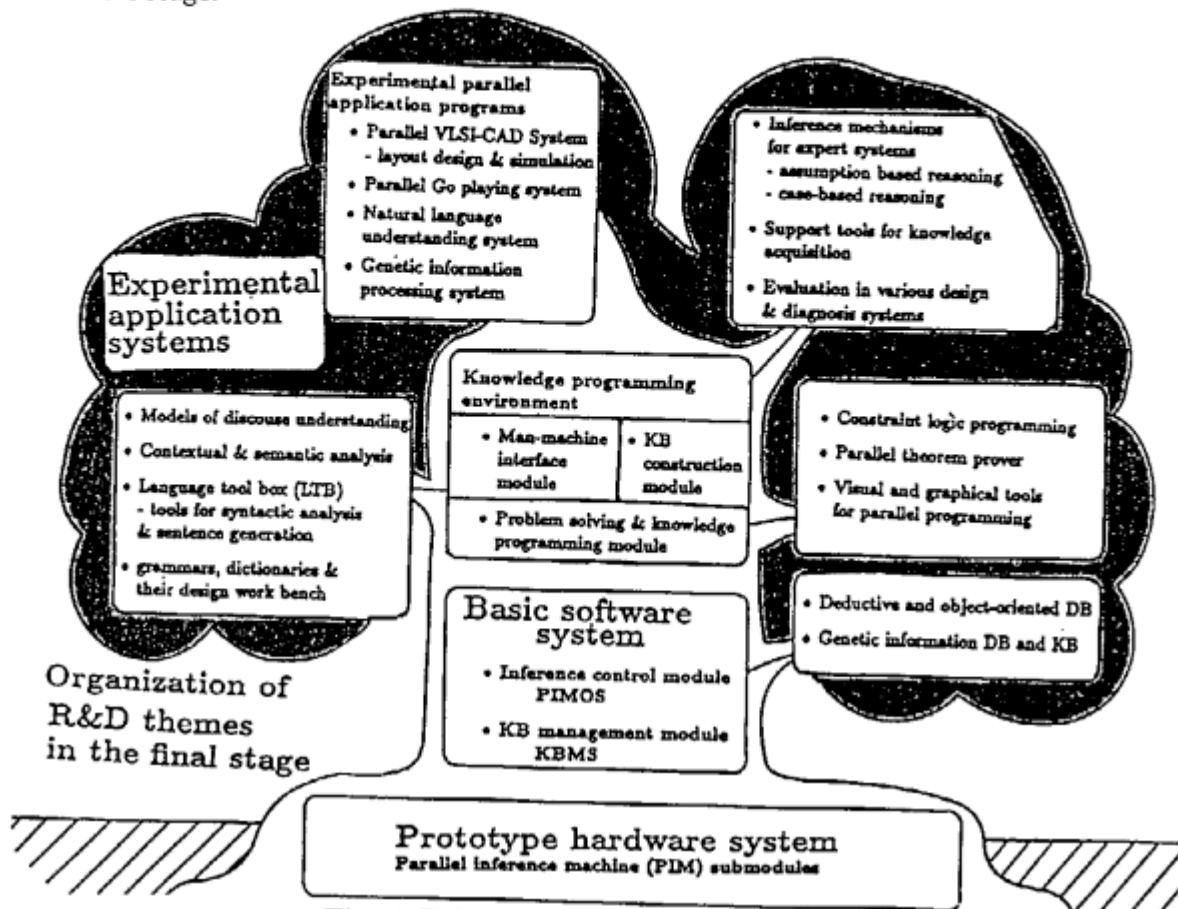


Fig.-2 R&D themes in the final stage

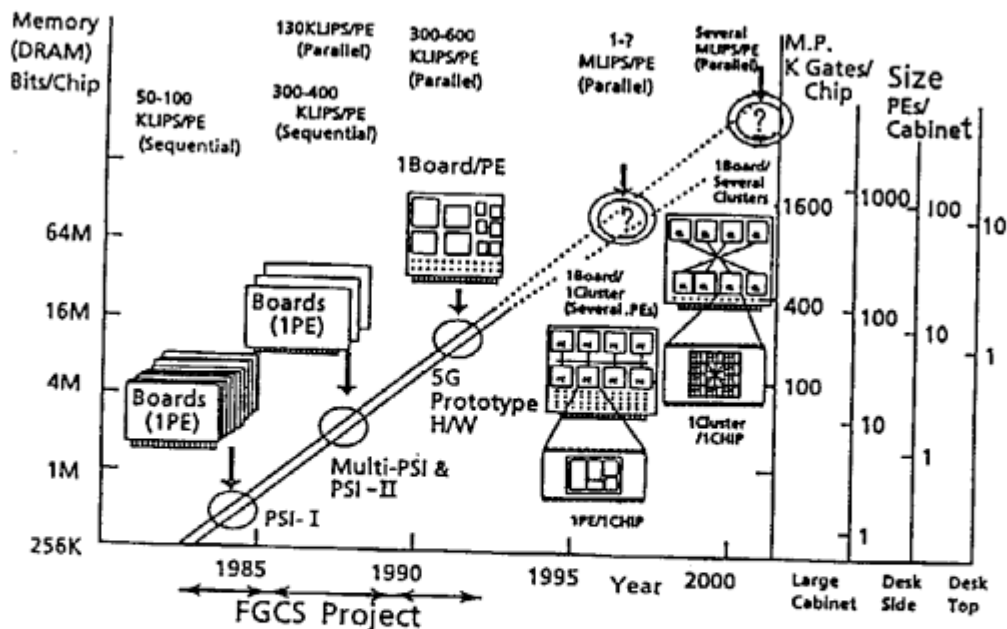


Fig.-3 Trend of the hardware technology used for inference machines (by T. Kurozumi)

Extended abstract

The organization of ICOT has been changed and extended. Recently, we have started international collaborative works with ANL, NIH and SICS using the sequential and parallel inference systems. These are summarized in Fig.-4 and Fig.-5.

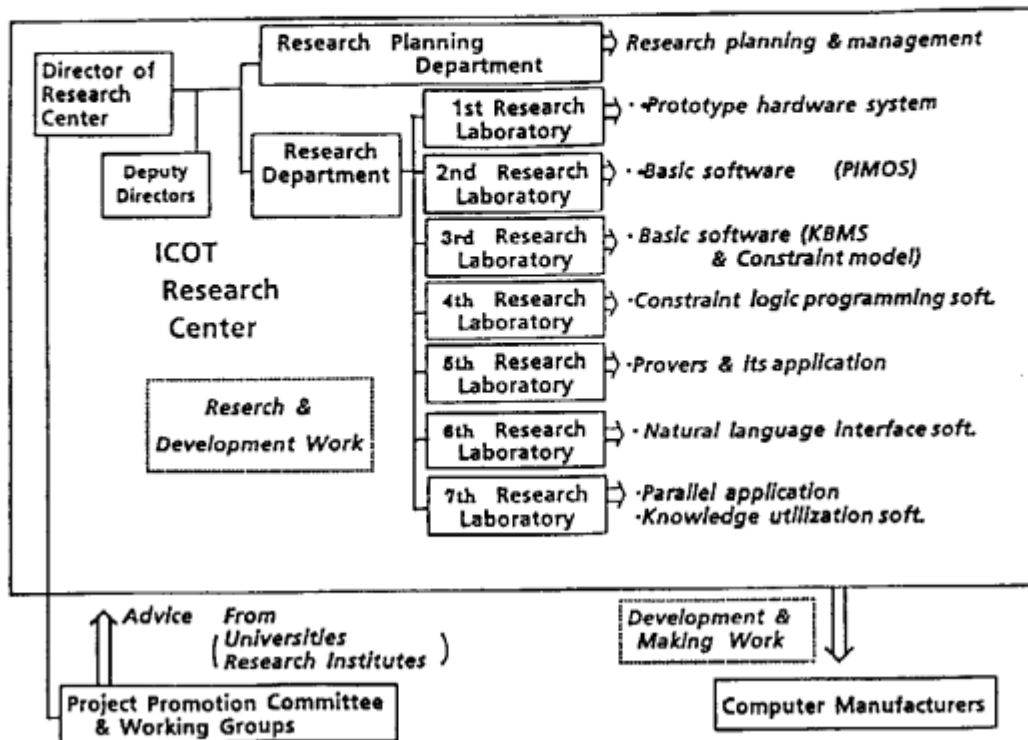


Fig.-4 Organization of ICOT research center (by T. Kurozumi)

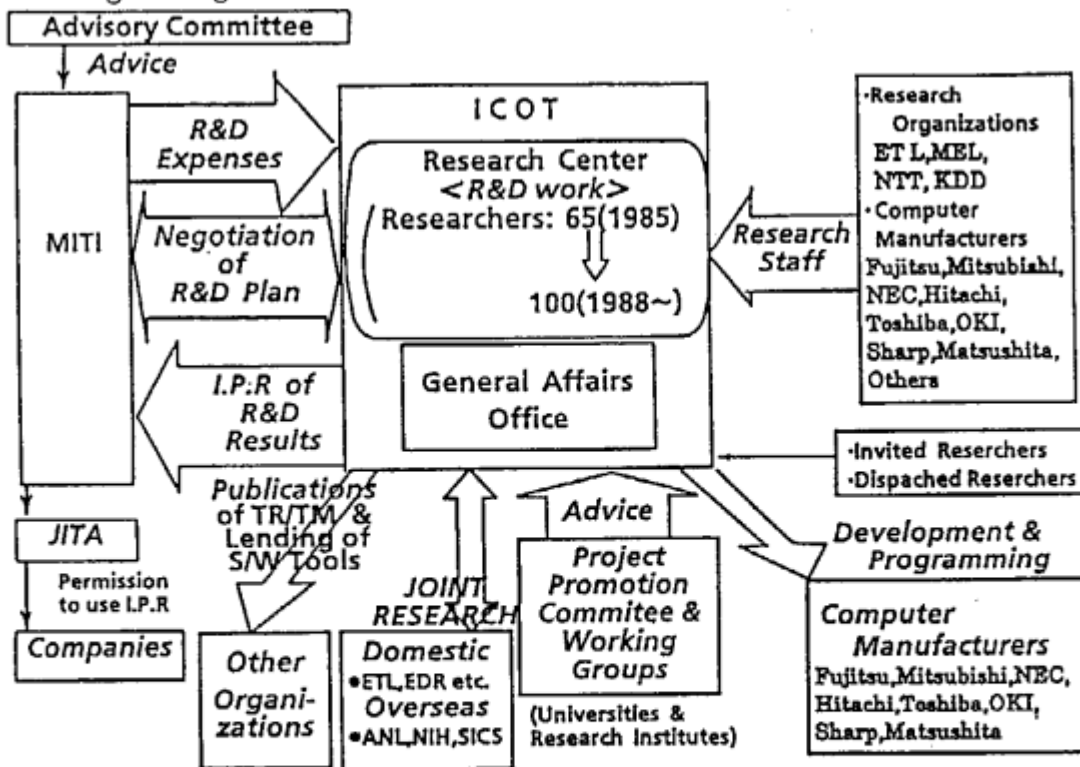
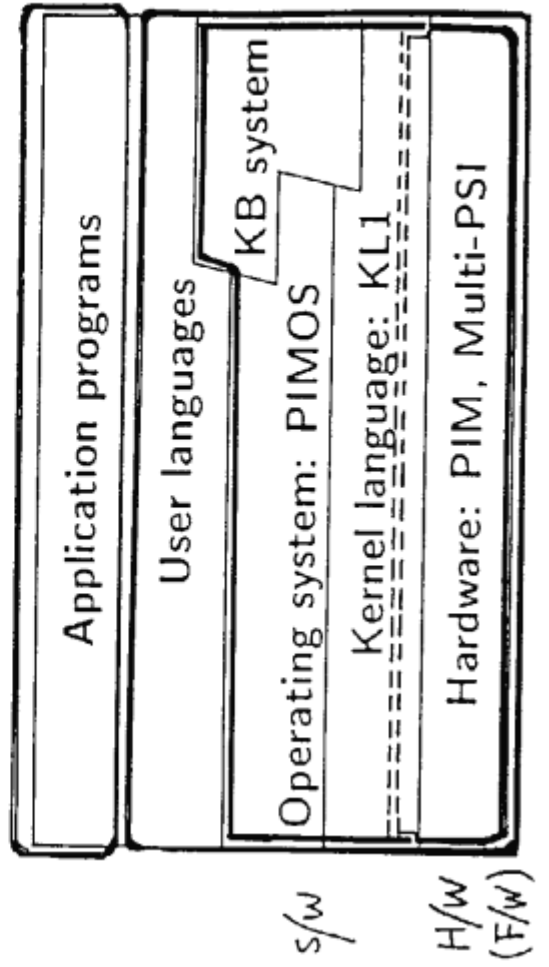
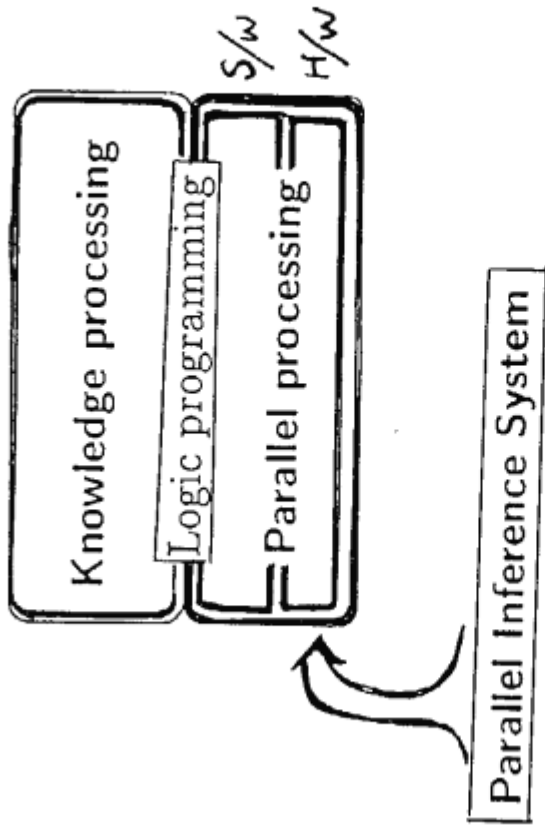


Fig.-5 Organization of FGCS project (by T. Kurozumi)

Overview of
the Final Stage R&D of
FGCS Project

Shunichi Uchida
ICOT

General Framework of R & D



Sequential Inference Systems

- 1984 PSI-I 37 KLIPS (KL0/V1)
 - ESP language and SIMPOS/V1
- 1986 PSI-II 330 KLIPS (KL0/V2)
 - SIMPOS/V2 and Kappa-I
- 1988 -SIMPOS/V5 and Kappa-II
 - Pseudo Multi-PSI and PIMOS-S
 - **About 300 PSI-II Machines and Network
- 1991 PSI-III \Rightarrow 1.2 MLIPS (KL0/V2)
 - SIMPOS/V7 + UNIX
 - Domestic and international network link
 - to access PIM systems at ICOT

Knowledge Programming Software on PSI/SIMPOS

- 1987 - 1990
 - Knowledge Representation Languages:
 - CIL, CRL and Quixote
 - KBMS based on Deductive and O-O DB
 - Constraint Programming Languages: CAL
 - Mathematical and Meta-programming Systems:
 - CAP, EUODHLOS, ARGUS, etc
 - NL Understanding Systems and Tools:
 - DUALS and LTB
 - Many Expert Systems:
 - VLSI CAD Systems
 - Go playing system
 - CASE Systems, etc.

Parallel Inference Systems

1985 **GHC**

1986 **Multi-PSI/V1** 1 KLIPS x 6PE (FGHC)

Parallel interpreter of FGHC

1988 **Multi-PSI/V2** 150 KLIPS x 64PE (KL1

(2 - 5 MLIPS)

PIMOS/V1 and small benchmark programs

1990 **Final PIM Chips and CPU Boards**

PIMOS/V2 and many application programs

1992 **Final PIM System** 300-600 KLIPS/PE (KL1

(PIM model/p \Rightarrow 200 MLIPS/512PE)

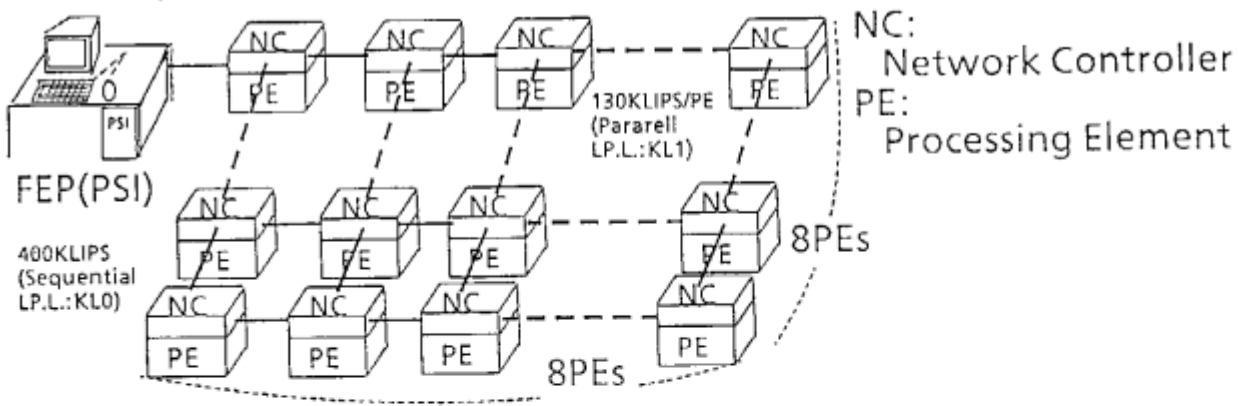
5 modules: model-p, m, c, k, i

Total 1072PE's = 512 + 256 + 256 + 32 + 16

PIMOS/V3 + KBMS(Kappa-P)

Parallel Application Systems

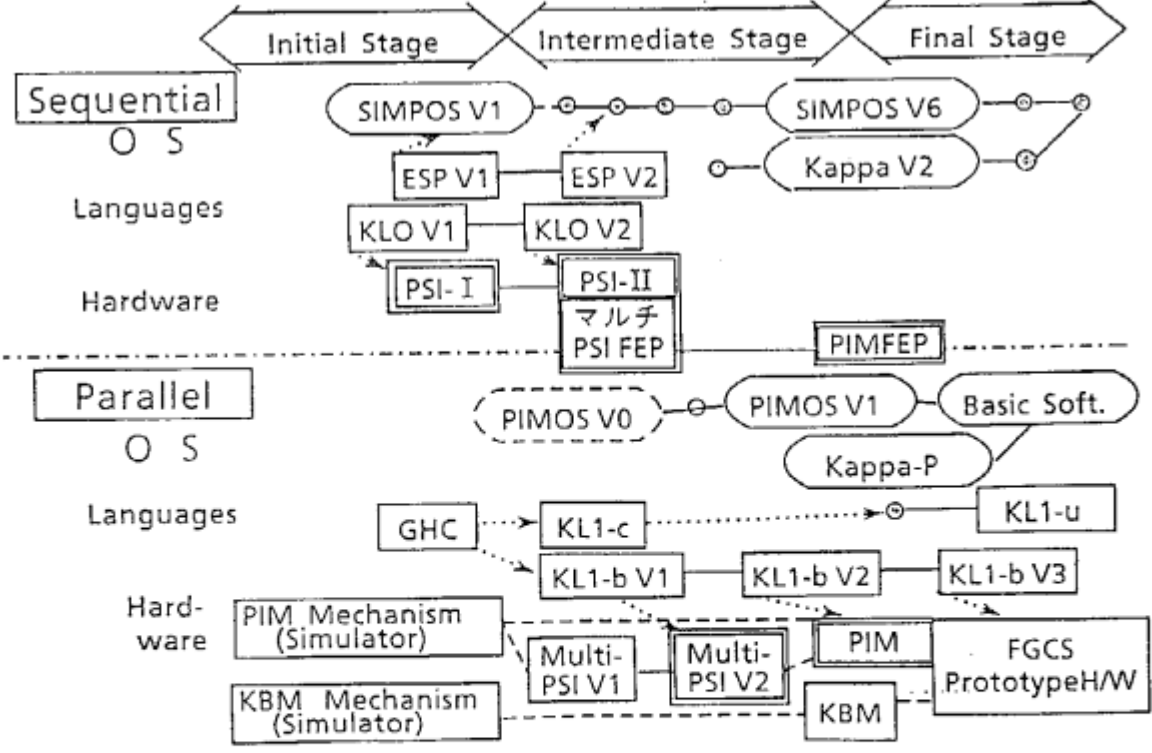
Experimental Parallel Inference Machine: Multi-PSI/v2



- 64 PEs max (PSI-II CPU each)
2 ~ 5MLIPS/system (ave)
- Machine language : KL1-b
- Memory : 16 Mw / PE
(80MB)

- Network :
-2-dimensional mesh,
-message exchange
-routing functions
-5MB/s x 2directions/ch

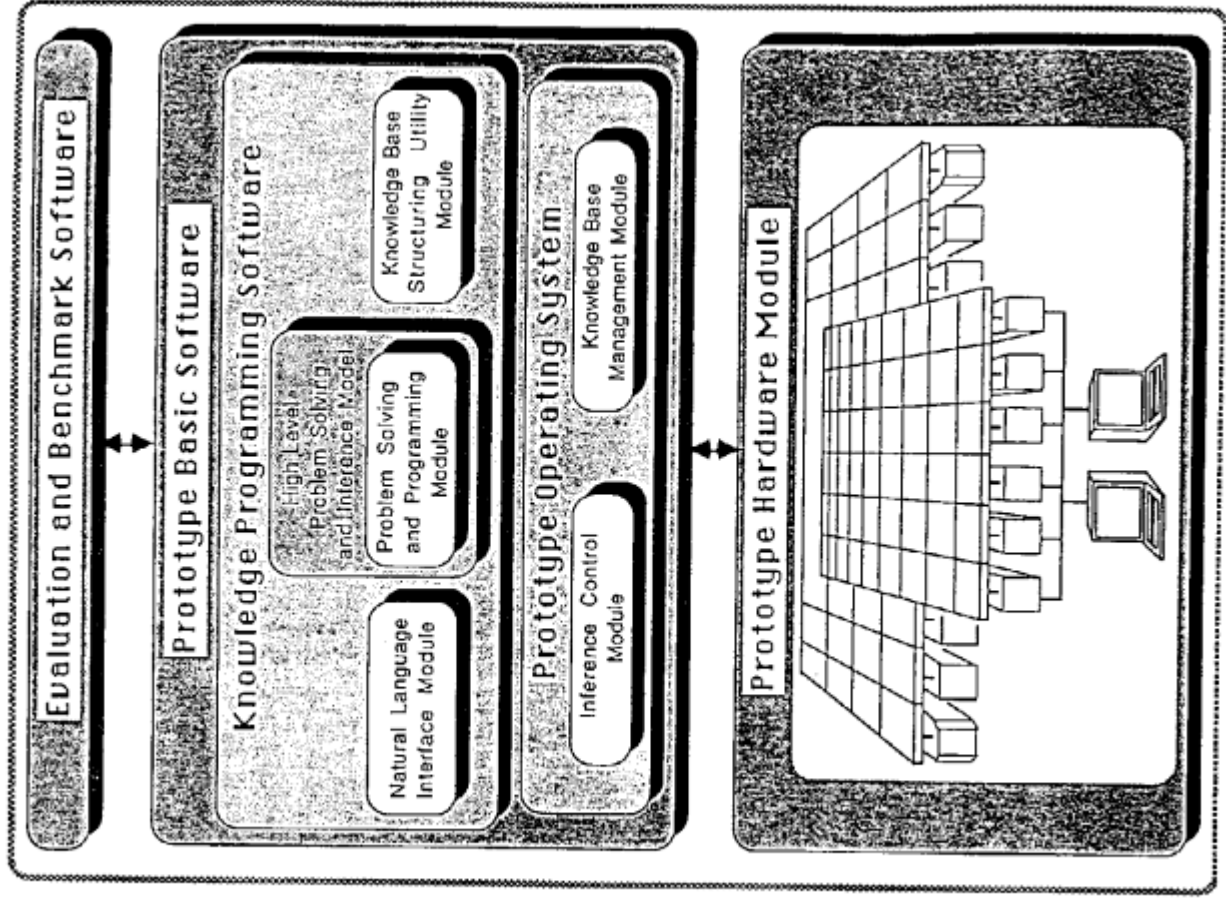
R&D Steps of System Software, Kernel languages & Hardware



Main research targets in the Final Stage

"To establish a parallel logic programming environment to be able to open for large scale application problems"

1. Large-scale high-speed PIM modules
2. Reliable multi-user operating system
3. Parallel DBMS
4. Collection of parallel programming tools and skills through the development of application programs

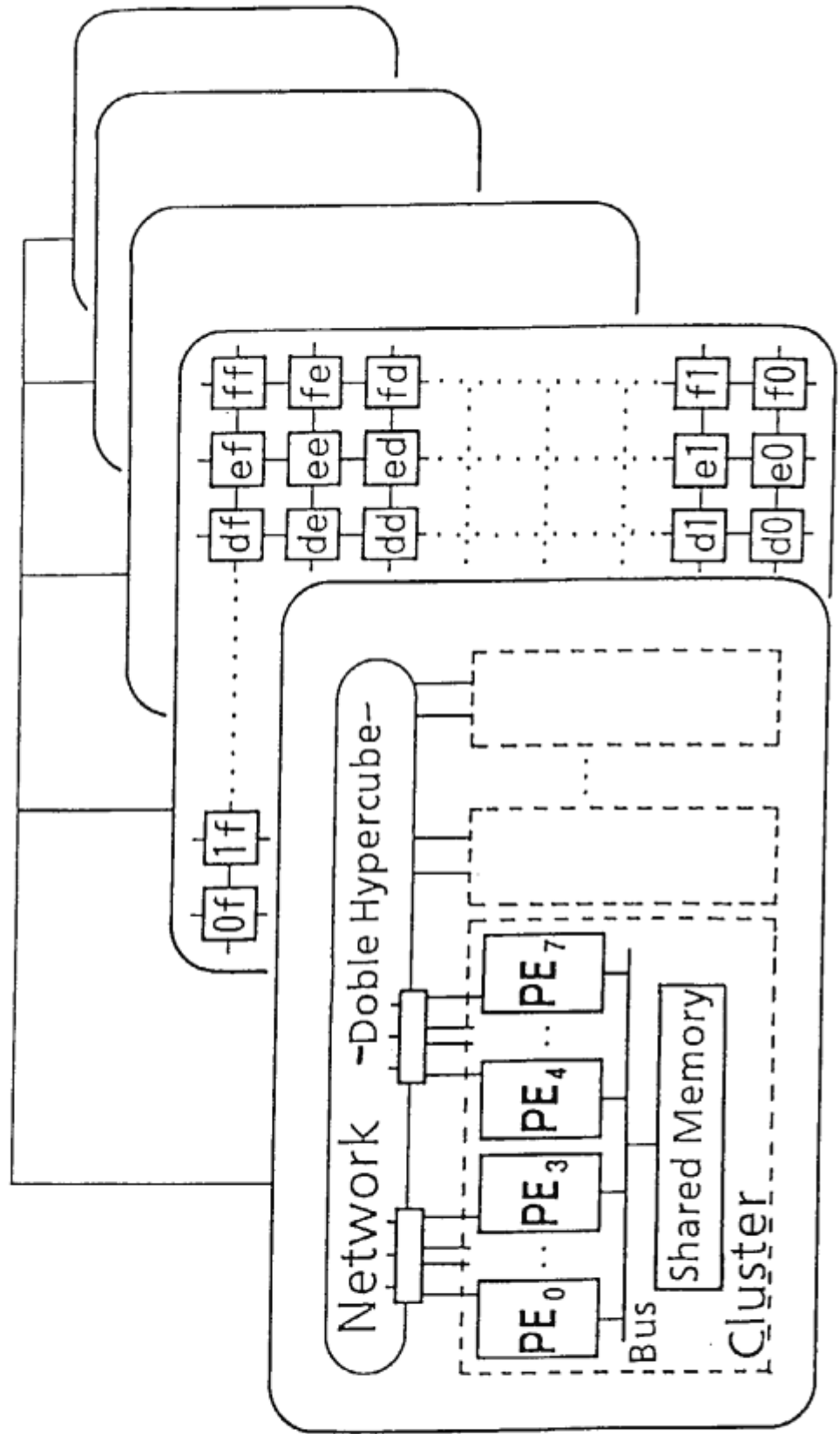


1. PIM hardware modules

and KL1 language processors

- Completion of PIM H/W development and evaluation of H/W:
Chips, CPU Boards, Clusters, Cabinets, and Total systems
- Completion of distributed implementation of KL1 language processors in the cluster and total system:
Memory management, Message passing control, and etc.

Prototype System Configuration
 — Parallel Software Development Environment —



2. Operating system, PIMOS

and KL1 programming environment

- Improvement and extension of the core of PIMOS:
Resource management, Remote Access control, and User management
- KL1 programming environment:
Debugging tools including performance debugging, Optimization techniques in Compilers, and etc.
- Parallel programming skills:
Algorithms, Paradigms, and Strategy for job division and priority control

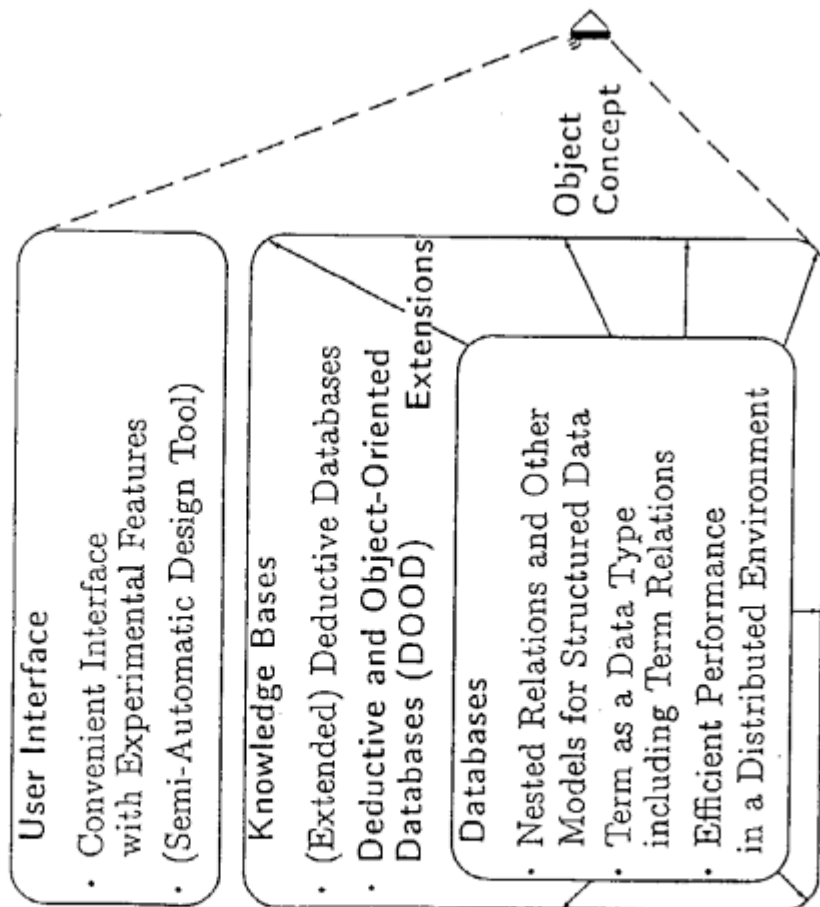
3. Parallel DBMS and its extension to KBMS

- Development of parallel DBMS, Kappa-P based on Nested Relational Model
- Implementation of relational algebraic operations on parallel and distributed environments, Multi-PSI and PIM
- Design and implementation of KBMS based on Deductive DB and Object-Oriented DB
- Design and implementation of a knowledge representation language, Quixote for the KBMS

Current Status of the Kappa System

- Kappa-I was implemented in August, 1987
 - 60,000 Lines in ESP
 - Several β -Test Users
- Kappa-II was implemented in March, 1989
 - 125,000 Lines in ESP
 - More Efficient Performance Than Kappa-I
 - Main Memory Databases
 - User Definable Interface
 - Practical Constructors (List, Bag)
 - Widely Released
- Kappa-P will be implemented in March, 1991
 - Written in KL1
 - Parallel Version of Kappa-II
 - (More Efficient Performance Than Kappa-II?)

Basic Policies



4. Application systems for parallel inference
system including DBMS/KBMS

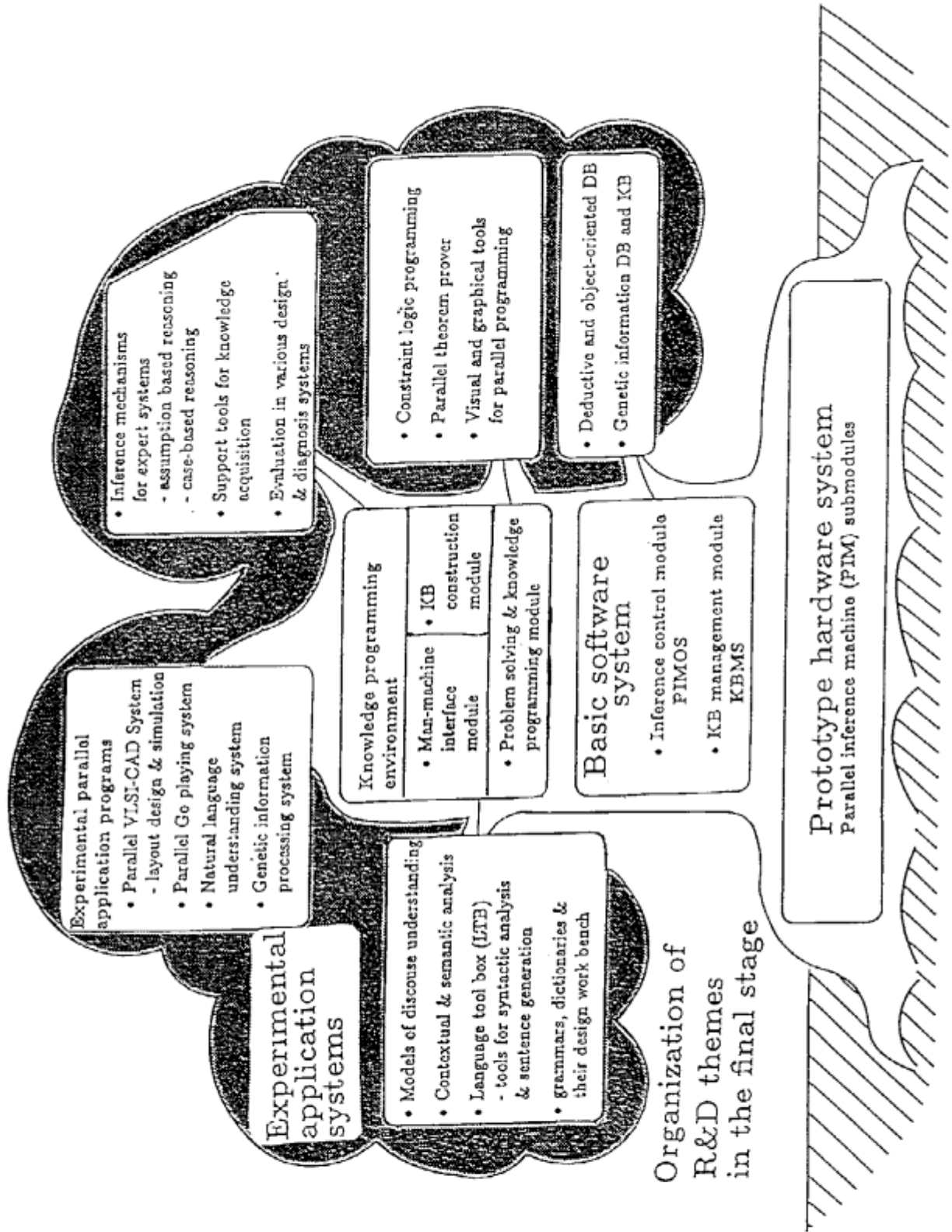
- (a) Small benchmarking programs:
Best-path, Pentomino and Tume-Go programs
- (b) Routing and logic simulation programs for VLSI
CAD systems
- (c) Parsers for lexical and syntactic analysis in
NL understanding systems
- (d) Parallel Theorem Provers
- (e) Parallel implementations of Constraint Logic
Programming Language, GDCC

(f) Parallel Go playing system, GOG

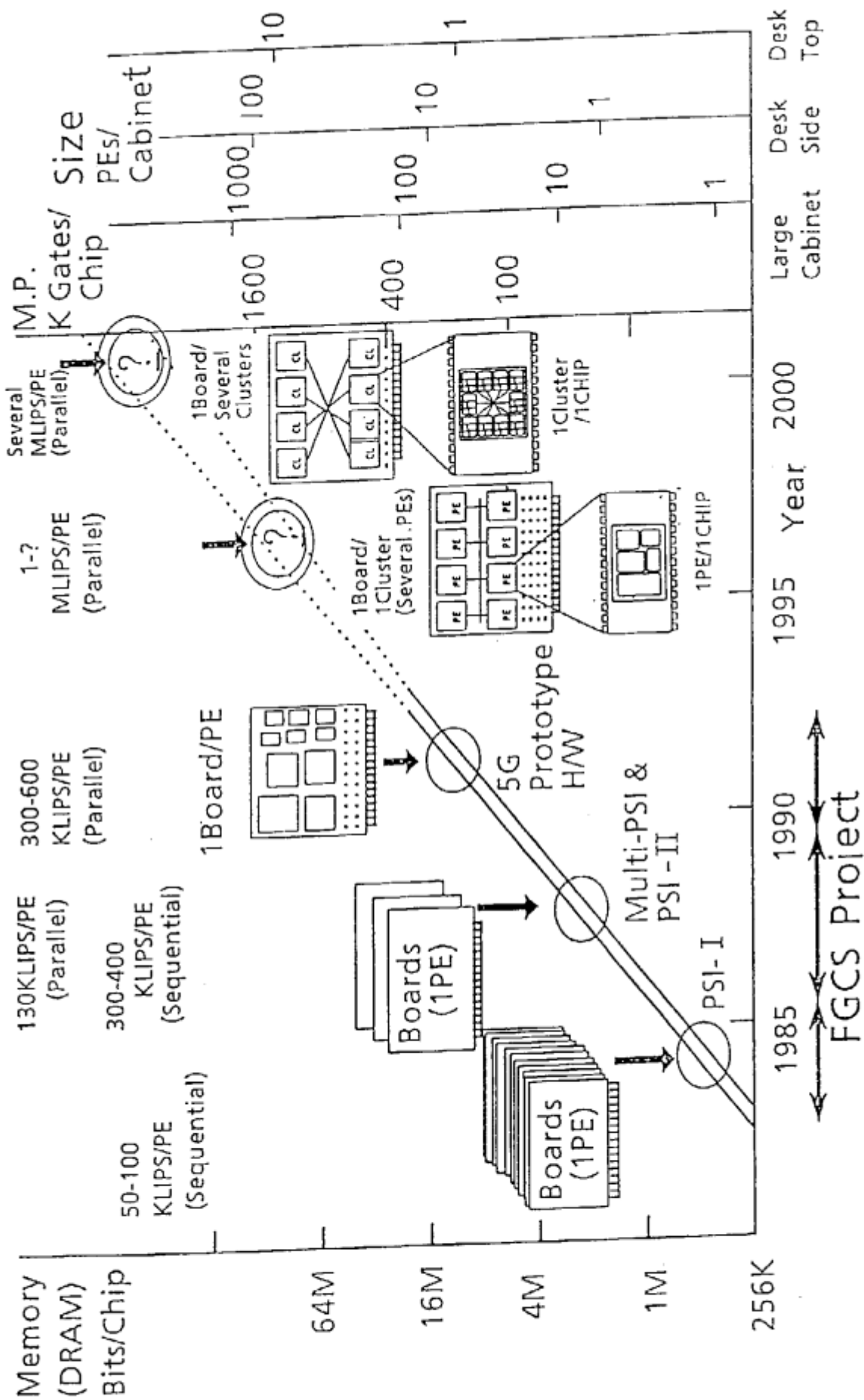
(g) Legal reasoning programs

- (h) Programs for multiple sequence alignment of
DNA or protein sequences for genetic infor-
mation analysis

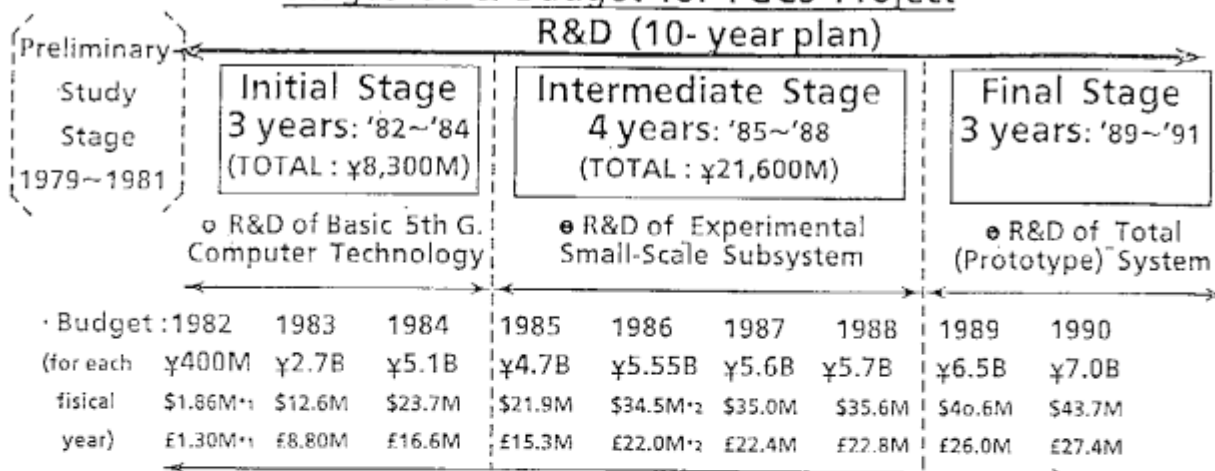
- (i) Evaluation of Kappa-II using GenBank data
and description of metabolic reactions us-
ing the knowledge representation language,
Quixote



TRENDS of LSI TECHNOLOGIES & 5G MACHINES



Stages of & Budget for FGCS Project



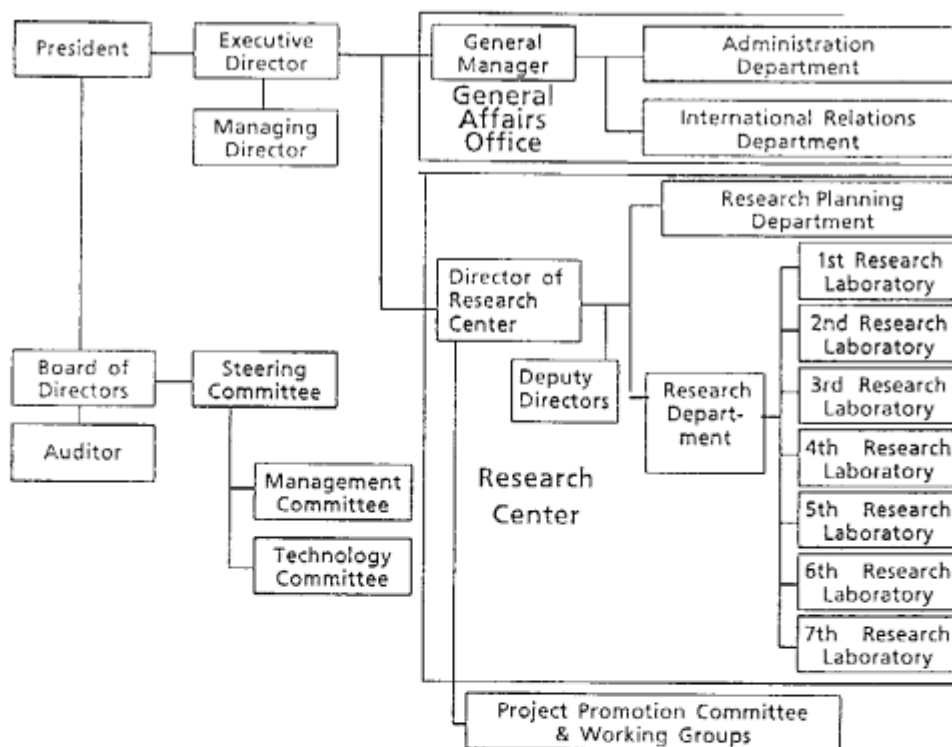
· R&D are carried out under the auspices of MITI.

(All budget are covered by MITY.)

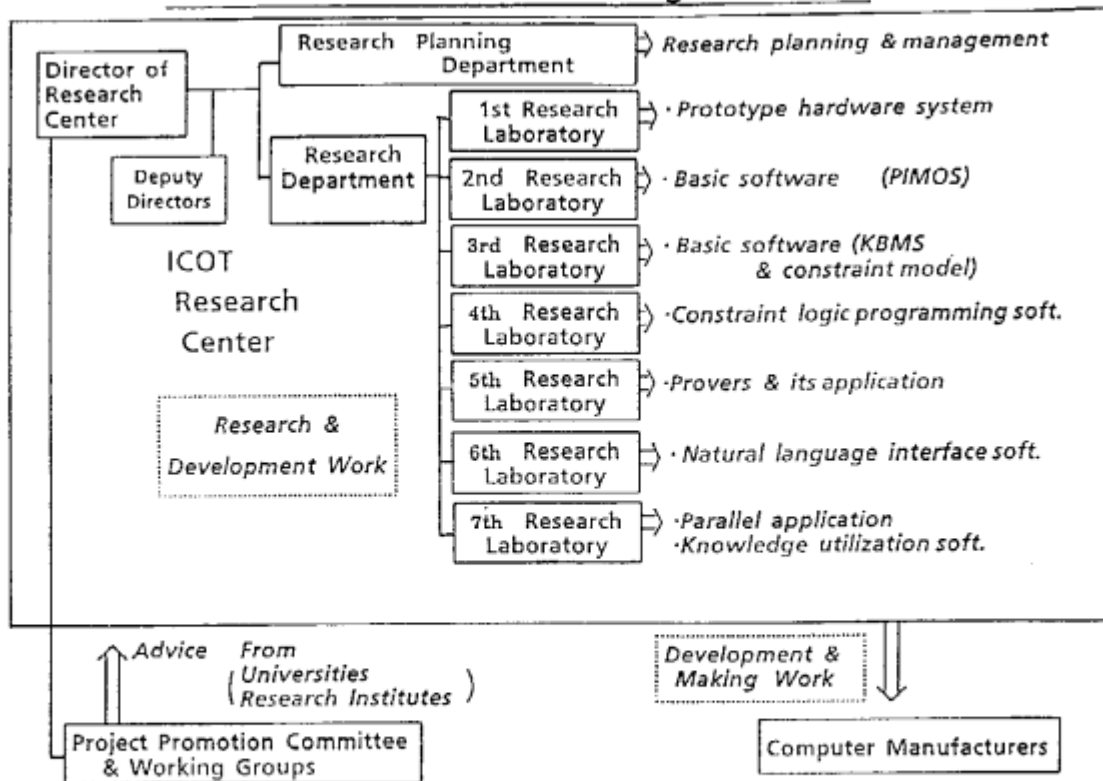
*1 \$1 = ¥ 215, £1 = ¥ 307 (1982~1985)

*2 \$1 = ¥ 160, £1 = ¥ 250 (1986~1989)

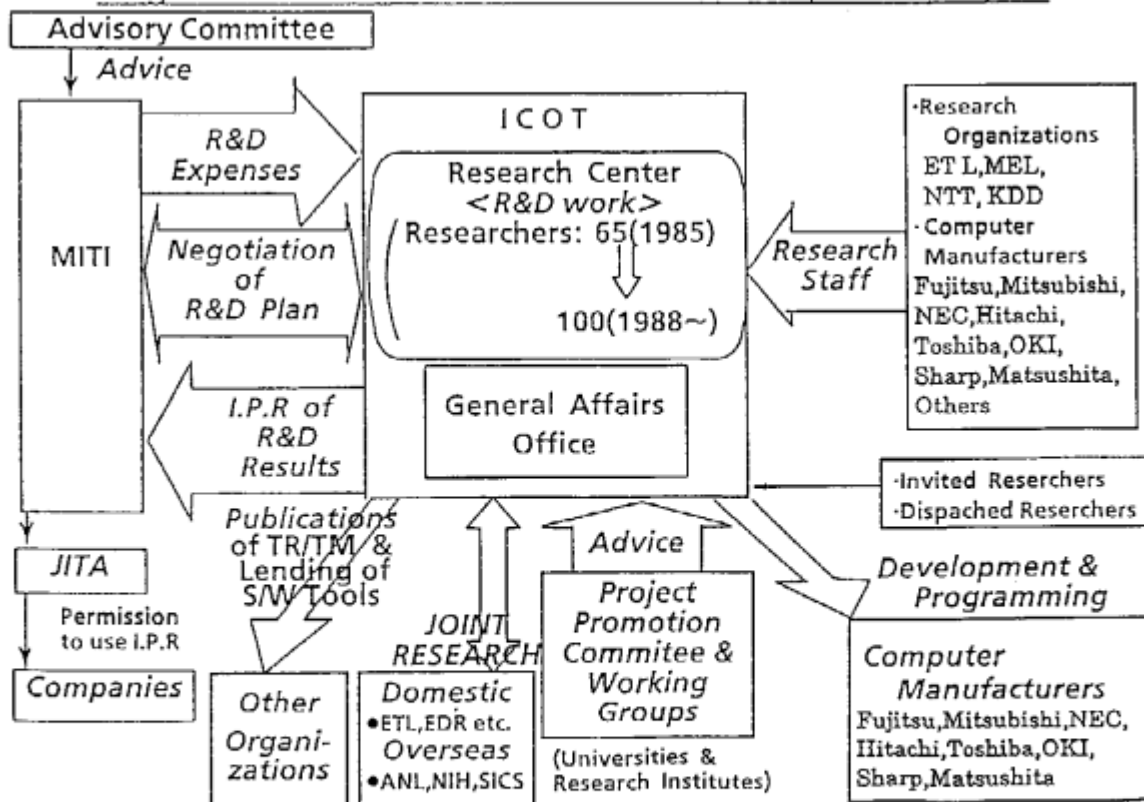
ICOT Organization



ICOT Research Center Organization



Organization of Fifth Generation Computer Project



International Co-operation & Diffusion Activities of R&D Results

• Preparation and Distribution of Technical Papers

(e.g., ICOT Journal, Technical Reports and Memoranda(1,400 TR/TMs))

• Conferences and Symposia

- Sponsorship of the International Conference on FGCS
(1981, 1984, November 1988, 1st-5th June 1992(planning))
- Joint Symposia and Workshops(co-sponsorship)

.....U.S.-J, France-J, Sweden-Italy-J, U.K.-J

- Invitation of Experts (Researchers)
to ICOT for short periods for research exchange (56 researchers(1982-1989))
- Acceptance of Researchers from U.S.(NSF-backed), France(INRIA
-backed)& U.K(DTI-backed) to ICOT for half to 1 year (based on agreement)
- Dispatch of researchers (universities, ICOT)
to international Conferences and Meetings to present technical papers
- Acceptance of visitors to ICOT (Researchers, Journalists, etc.)
- Joint Research with other OrganizationsANL, NIH, SICS,

OVERVIEW OF PARALLEL COMPUTING RESEARCH IN THE UK AND EUROPE

PAUL REFENES

INFORMATION TECHNOLOGY DIVISION

DEPARTMENT OF TRADE AND INDUSTRY

ABSTRACT

Parallel and novel architecture research is a research area which is becoming increasingly important for the IT Industry as a whole. The majority of the novel programming styles and computational models which are expected to enhance programmer productivity (e.g. Functional, Logic, Object-Oriented, etc) have a strong requirement for powerful compute engines which is beyond the capabilities of traditional sequential processors. In addition an increasing number of new applications are emerging which are only possible to tackle due to the availability of powerful parallel computers.

The research area of parallel and novel architecture encompasses four key technologies:

- . hardware architecture: this technology includes both the design of microprocessors/microcomputers and the architecture of networks of such processors.
- . basic system software: including operating system kernels, memory management, message through-routing, etc.
- . parallel application development tools: such as decomposition tools, mapping tools, load balancing, etc.
- . neural computing: soft information processing systems and massively parallel and distributed architectures.

This paper reviews the state of the art in UK and European research in these four areas.

OVERVIEW OF PARALLEL PROCESSING IN EUROPE & THE UK

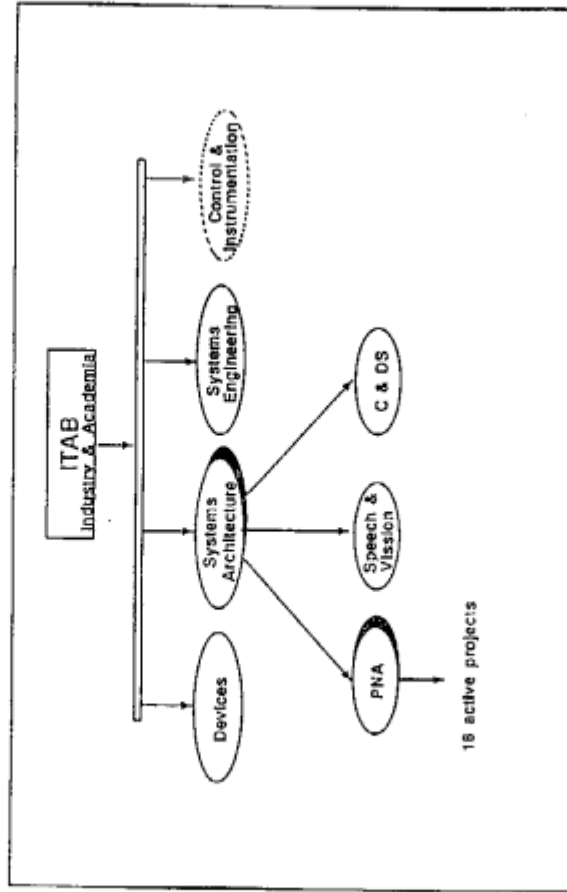
P. N. REFENES

INFORMATION TECHNOLOGY DIVISION
DEPARTMENT OF TRADE & INDUSTRY

CONTENTS

- IT research in the UK
- IT research in Europe
- Parallel Architecture research in Europe & the UK
 - Symbolic processing
 - Numeric processing
 - Basic system software
 - Parallel application development tools
 - Neural computing
- Parallel Application Centres Programme

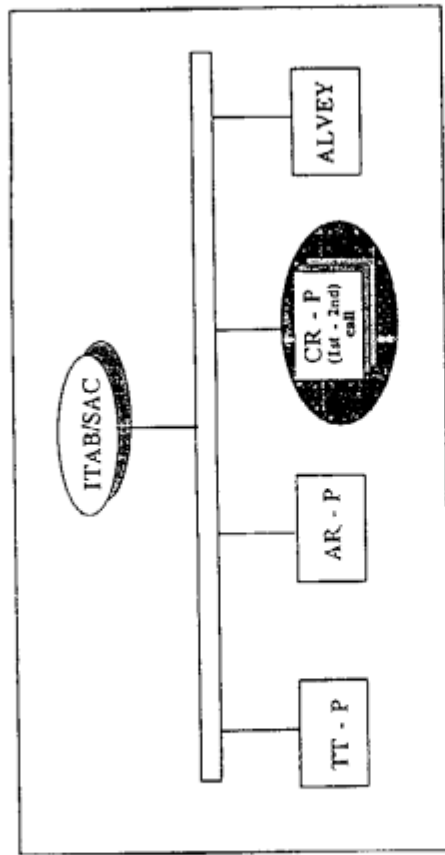
IT research in the UK - ITAB



- Two active calls + old ALVEY projects
- Special Initiatives
 - Parallel applications Programme
 - Neural Computing

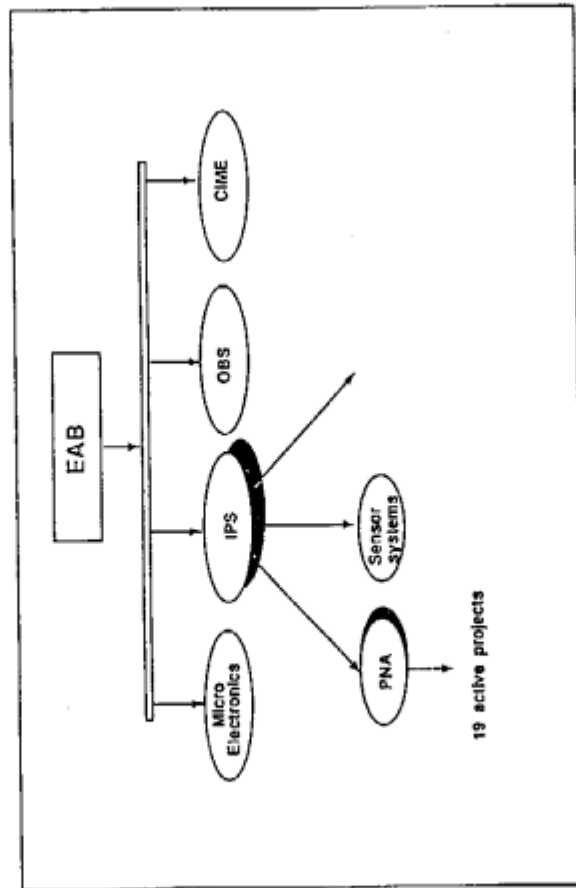
ITAB/SAC ACTIVE PROJECTS

IT research in Europe - ESPRIT



TT - P : Technology Transfer Programme (within SAC)
 AR - P : Academic Research
 CR - P : Collaborative Research
 ALVEY: Projects which are still active

SAC funding - this report deals only with the collaborative/uncollected projects from the 1st and 2nd call.



- Two active calls
- Much Larger Projects

ESPRIT		EPAB-SAC				
ADVANCED ARCHITECTURE	#Phys	US\$m	UK %	PNA	#Phys	Ln
• Symbolic Processing	3	2.8	10.9	• Symbolic processing	3	1.0
• Numeric Processing	3	7.2	27.8	• Numeric processing	1	0.5
• Basic System Software	5	4.3	13.9	• Basic System Software	3	3.0
• Application Development Tools	3	1.7	5.3	• Application Development Tools	5	4.0
• Neural Computing	5	2.5	8.2	• Neural Computing	6	2.2
TOTAL	19	12.4			18	10.7
SECTOR-BASED SYSTEMS		SPEECH & VISION				
• Vision Systems (IPS)	8	5.0	22.3	• Vision Systems	5	4.8
• Speech Systems (IPSS)	5	3.9	16.6	• Speech Systems	2	0.3
• Vision Systems (ORIS)	7	2.1	34.4	• Vision Systems	-	-
• Speech Systems (ORIS)	1	0.1	19.6	• Speech Systems	-	-
TOTAL	21	10.86		TOTAL	7	5.1
CDS		CDS				
• Communications	5	2.5	14.3	• Communications	-	-
• Distributed Systems	7	5.0	21.6	• Distributed Systems	3	2.8
• Methods & Tools	10	1.6	6.9	• Methods & Tools	5	1.4
• Application Demonstrators	9	2.9	9.7	• Application Demonstrators	-	-
TOTAL	31	12.0		TOTAL	8	4.2

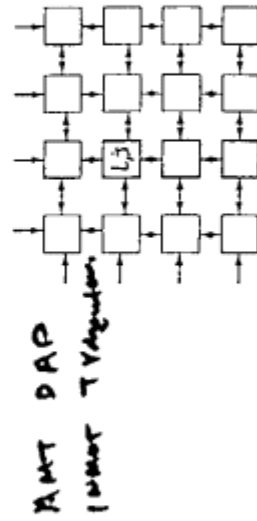
PARALLEL DECOMPOSITION

GEOMETRIC DE-COMPOSITION (of MIMD, SIMD)

- Parallels is obtained by distributing the DATA such that the original geometric-structure is preserved
- Each PE processes MODERATE amounts of data and SYNCHRONISES between computational steps with others
- OFFERS good speed-up if: data dependencies are characterised by locality of reference
- LIMITING factor is the communication overheads

EXAMPLE (Chaotic relaxation)

$$A[i, j] = f(A[i-1, j], A[i, j+1], A[i+1, j])$$



PARALLEL DECOMPOSITION

ALGORITHMIC DE-COMPOSITION (cf. MISD)

- Parallelism is obtained by de-composing the ALGORITHM into smaller steps.
- STEPS are simple and performed repetitively in a loop
- Offers speed-ups proportional to the # of processors used
- LIMITING factor is the slowest element in the pipeline

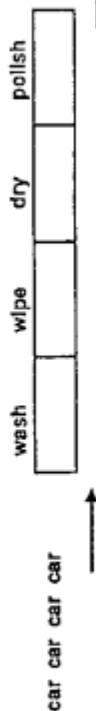
EXAMPLE →

```
do (9 to 5) {
```

```
  wash(car)
  wipe(car)
  dry(car)
  polish(car)
  next(car)
```



can - ?



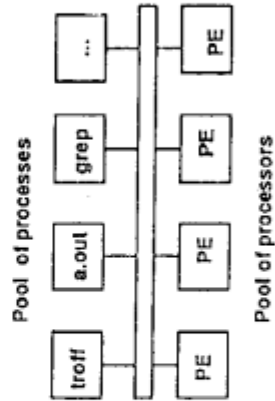
PARALLEL DECOMPOSITION

FARMING DE-COMPOSITION (cf. MIMD)

- Parallelism is obtained through a pool of processors executing from a pool of processes.
- Little synchronisation is required between processors except for the initial delivery of data
- OFFERS speed-ups proportional to the # of processors used
- LIMITING factor is the contention for access to the pool

EXAMPLE (Unix processes) →

```
% troff file & a.out program & grep x file > junk &
```



NUMERIC PROCESSING SYSTEMS

CHARACTERISTICS

- * Highly regular data structures -> geometric parallelism
- * Highly regular operations -> algorithmic parallelism
- * Static de-composition
 - >
 - static processes
 - static control structures
 - static languages (Fortran, Occam, Parle, etc)

ESPRIT		ADVANCED ARCHITECTURE				PARALLEL NOVEL ARCHITECTURE			
NUMERIC PROCESSING: Architectures		MECU	UK	\$UK	NUMERIC (SIMD) PROCESSING		UK	EM	
GENESIS (2447)	Distributed memory parallel Super Computer for numerical applications BULL	9.06	6.28	69.3	SIMD/MIMD (1428)	High speed SIMD/MIMD interconnect	AMT MEIKO	0.5	
NAOPIA (2288)	Optical processing architecture in industrial applications THOMSON-CSF	1.0	0.0	0.0					
GPMIMD (5404)	General purpose MIMD machines INMOS	10.71	5.72	53.4					
TOTAL		20.77	12.0	57.8	TOTAL			0.5	

TABLE III: Numeric Processing Architectures

SYMBOLIC PROCESSING SYSTEMS

CHARACTERISTICS

LANGUAGES

- * are VHL mostly with IMPLICIT Parallelism
- * notable tends towards explicit Parallelism to restrict ||ism to manageable levels.

ARCHITECTURES

- * highly (micro) programmed control flow devices
- * need to support dynamic control structures (stacks, etc)

PARALLELISM

- * mainly farming de-composition (reduction systems)
- * also some geometric parallelism in logic (unification)

ESPRIT ADVANCED ARCHITECTURE				ITAB PARALLEL & NOVEL ARCHITECTURE				
PARALLEL APPLICATION DEVELOPMENT TOOLS		MECU	UK	£UK	APPLICATION DEVELOPMENT TOOLS		UK	£M
IMSE (2143)	Integrated Modelling Support Environment STC	3.0	1.07	35.67	COOTS (1059)	Object Orientated Languages on Parallel Transputer - Arrays	RSRE INMOS THORN EMI	1.1
REX (2080)	Reconfigurable and Extensible parallel and distributed systems. STOLLMAN	9.99	1.7	17.02	DAPW (1438)	Programming Work Bench for a Massively Parallel Computer (DAP)	AMT QMW Intercept MEIKO	0.5
FAAST (5212)	Fault-Tolerant Architecture Nixdorff	6.58	0.22	3.3	FSPP (1452)	Fortran for Scalably parallel processors		0.9
					FLARE (2117)	Functional Languages Applied to Realistic Examples	BT Logica	0.6
					GRAS-PARC (2172)	Graphical Environmental for Supporting Parallel Computing	NAG Quintel	0.9
TOTAL		19.57	2.99	15.3	TOTAL			4.0

TABLE VI: Parallel Application Development Tools

5

ESPRIT ADVANCED ARCHITECTURES				ITAB PARALLEL NOVEL ARCHITECTURE				
NEURAL COMPUTING		MECU	UK	£UK	NEURAL COMPUTING		UK	£M
PYGMALION (2059)	Neurocomputing: Basic tools, Application tools and Applications THOMSON CSF	2.5	0.45	18.0	GANNET (1886)	Neural Network generation adaptation using evolutionary techniques	LOGICA MEIKO	0.2
ANNIE (2092)	Industrial application of Artificial Neural Networks UK AEA	2.5	1.16	46.4	LINNET ()	Neural Network Awareness Club	LOGICA TSB	
MLT (2154)	Machine Learning Toolkit. and Industrial Applications NIXDORF	6.99	1.68	24.03	SRT (1057)	Speech recognition techniques using Market Chains Neural Networks	SINTEX CAL PARSYS	1.1
GALATHEA (5293)	Continuation of PYGMALION OM. Architectures, tools and applications THOMSON CSF	8.5	0.84	9.9	LNNCI (1005U)	Logica Neural Nets Characteristics and implementation	BRUNEL	0.2
NNF (5433)	Neural network systems for forecasting and diagnosis (B)	1.9	0.0	0.0	CONNET (2163)	Neural Networks for control of real-time systems	DOWTY LOGICA	0.5
					EX-STATIC (2167U)	Simulation of dynamic architecture for neural networks	CAM BRIDGE	0.2
TOTAL		22.39	4.13	18.4	TOTAL			2.2

TABLE VII: NEUROCOMPUTING

PARALLEL APPLICATION CENTRES PROGRAMME

AIM:

To establish 4-5 centres to support the tools and methods for the development of parallel applications.

ACTORS:

Universities :- hosts for the centres

Parallel systems (software & hardware) vendors:- joint projects for further systems development

End Users :- collaborative projects for technology transfer and demonstrators

TOTAL COST:

up to 40m PS.

PIM Architectures and R & D Status

Keiji Hirata

Institute for New Generation Computer Technology

hirata@icot.or.jp

1 Introduction

We have been developing parallel inference machines (PIMs) and its firmware in the Japanese FGCS project [Goto 89a], [Goto 89a]. Our research goal is to prove that a logic programming framework is most effective for knowledge processing. In the first step, we would show that application programs and the operating system (PIMOS), that are all written in KL1¹, can work efficiently on a KL1 engine. The PIM hardware, the firmware, and KL1 language processor make the KL1 engine. My presentation reports the current status of the development of the KL1 engine.

When starting the PIM development, we did not have enough experience to select one of several alternative PIM architectures which can give the best performance for executing KL1 programs. Thus, the purpose of the development of more than one PIM was to examine and compare the technical issues for different architectures. First, the machine architectures and the features of each PIM are presented. Then, the implementation of the KL1 engine on the PIMs is reviewed. Next, the current status of the development of the hardware is reported. Last, how each module of the firmware works to execute KL1 programs is described. Then, the progress report on the firmware is made.

2 Comparison of Five PIMs

Five PIMs are now being manufactured; *PIM/p*, *PIM/c*, *PIM/m*, *PIM/i*, and *PIM/k*. We will compare the specifications of these PIMs to each other with respect to the factors listed below. The following four tables show the comparison (Tables 1, 2, 3 and 4). The technical factors which are concerned with the PIM architectures are:

- **intra-cluster configuration**

We classify the PIMs with respect to the configuration within a cluster (Tab. 1). Here, a *cluster* is part of the machine structure, and consists of 10 or so processing elements and a shared memory connected by a bus. In *PIM/p*, every four NIs are connected to a router, which works as a node in a global network.

- **inter-cluster configuration**

There are many possible methods and topologies of the networks which connect the clusters to each other: bus, hypercube, crossbar, mesh, omega, tree, and so on (Tab. 2).

¹KL1 is a concurrent logic programming language and was developed in ICOT.

Table 1: Intra-cluster Configuration

	Number of PEs	Number of NIs	Comment
PIM/p	8	8	each PE has NI
PIM/c	8	1	NI is connected to a bus
PIM/m	1	1	
PIM/i	8	1	NI is connected to a bus
PIM/k	16	1	one of 16 PEs has NI

(PE = processing element, NI = network interface)

Table 2: Inter-cluster Configuration

	Topology	Number of Clusters	Total number of PEs
PIM/p	hypercube	64	512
PIM/c	crossbar	32	256
PIM/m	mesh	-	256
PIM/i	-	2	16
PIM/k	-	2	32

Notice that the PIM/m architecture does not include a part named "cluster" in fact. In the table 1, one PE with a NI is regarded as a cluster for comparison. Actually, each PE of PIM/m has its own private memory and there is no shared memory over the entire machine.

- **KLI-oriented instruction set and processor configuration (Tab. 3)**
Whether a processor is a RISC or a CISC is one of major concerns. If a processor is a RISC, its compiler must generate efficient machine codes in particular. In case of a CISC, we have to support micro programming. With respect to its architecture, a tag architecture may make a symbolic manipulation efficient.
- **coherent cache**
An architecture which can exploit the data locality is possibly effective to the KLI

Table 3: Specification on Processing Element

	Instruction set	Cycle time	LSI fabrication	Line interval
PIM/p	RISC + macro instruction	60 ns	standard-cell	0.96 μm
PIM/c	CISC (micro programmable)	50 ns	gatearrays	0.8 μm
PIM/m	CISC (micro programmable)	60 ns	standard-cell	0.8 μm
PIM/i	RISC	100 ns	standard-cell	1.2 μm
PIM/k	RISC	100 ns	custom	1.2 μm

Table 4: Specification on Coherent Cache

	Protocol	Number of states
PIM/p	invalidation (Illinois)	4
PIM/c	invalidation (modified Illinois)	5
PIM/m	-	-
PIM/i	broadcasting	6
PIM/k	invalidation (modified Berkeley) hierarchical	4

execution. So, we suppose an architecture which can keep the locality high; there are ten processing elements, which are connected to a shared memory through coherent caches and a bus. The coherent caches is necessary to decrease the bus traffic. All of the PIMs adopt the write-back coherent cache method (Tab. 4).

Notice that PIM/k has the hierarchical cache; a PE of PIM/k has its first cache, the four PEs share a second cache, and the four second caches share a global memory.

- I/O channel

Input and output to storage devices make the PIMs more practical. We have the two extremes; each processor has its own I/O channel, and an entire system has one I/O channel. The former configuration can obtain more disk throughput but the amount of the hardware increases and the control is more difficult.

A plan of how much disk capacity the PIM experimental versions provide is as follows. A cluster of PIM/p (8 PEs) has two SCSI channels and 2.32GB disks totally. In PIM/m, every 8 PEs have a SCSI channel and a 600MB disk.

3 Current Status on Hardware Development

We report the current status concerned with the PIM/p hardware. Currently we are assembling the experimental version and are checking the operations on the hardware level. The next refined version of this experimental one which consists of 1 clusters (8 PEs) will first start working at ICOT in March of 1991. We hope that PIMOS on PIM/p (128 PEs) will start running in the latter half of that year.

As for the production model of PIM/m, two sets of 16PE system will arrive at ICOT in April of 1991. We hope that PIMOS on PIM/m (256 PEs) will be able to run before the end of that year.

The experimental systems of PIM/c, PIM/i and PIM/k are now being fabricated by each manufacturer.

4 How to Implement KL1 Engine on PIMs

In this section, we describe how to compile KL1 programs into the PIM machine codes. First, a KL1 program is compiled into an intermediate code, *KL1-B*, which corresponds to the WAM instruction of Prolog. Here, we have three execution methods (Fig. 1). The first

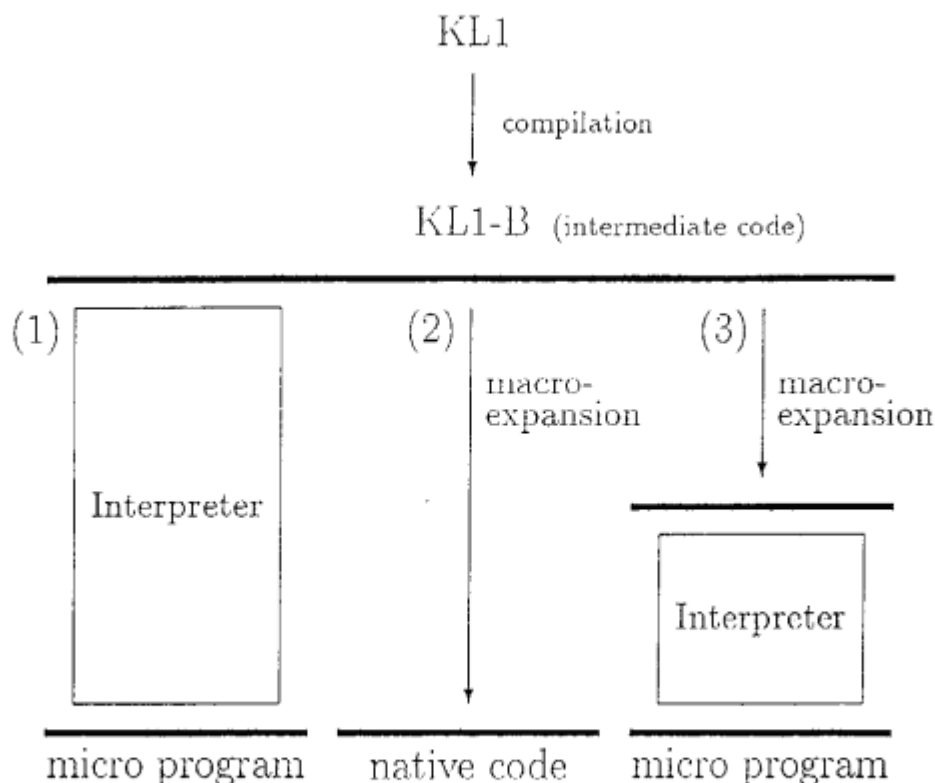


Figure 1: Three Methods for Executing KL1-B Instructions

method (Fig. 1 (1)) is to interpret the KL1-B codes directly as a high-level machine language by using a micro programming technique. Method (2) is to macro-expand (or compile) the KL1-B instructions into native codes². After that, the native codes are linked with run-time libraries and, then, can be executed. Method (1) is suitable for PIM/m and PIM/c, and (2) for PIM/i and PIM/k. Method (3) is located in the middle position; a KL1-B instruction is further macro-expanded (or compiled) to yet another intermediate codes lower than KL1-B. PIM/p adopts a mixed way of these three methods (1), (2) and (3).

Next we show the organization of software modules of the KL1 engine (Fig. 2). To write the firmware for the five PIMs efficiently and commonly as possible, we introduce a virtual PIM hardware which is an abstraction of the five PIMs, and design a language for describing the firmware on the virtual hardware. We call the firmware *VPIM* (Virtual PIM) and the language *PSL* (PIM System Descriptive Language). VPIM is an abstract machine for KL1-B. PSL is an extension of the C language. Statements of PSL can be source codes to be compiled and, in the same form, macro-definitions as well. Thus, the KL1-B expander does macro-expansion of KL1-B instructions with regarding VPIM (firmware in PSL) as the macro-definitions (Fig. 2 (a)). Actually, some KL1-B instructions are fully macro-expanded or compiled, and others are macro-expanded down to another intermediate level lower than KL1-B (Fig. 2 (c)). Depending on each PIM architecture, we have to find out the appropriate intermediate level and to use the KL1-B expander and the PSL compiler properly. On the other hand, the PSL compiler compiles the VPIM source codes (not macro-expanded) to the real-machine codes for the KL1-B interpreter (Fig. 2 (b)). In detail, a

²Indeed, the language KL1-B is designed so that the macro-expansion of KL1-B codes generates a sequence of machine codes.

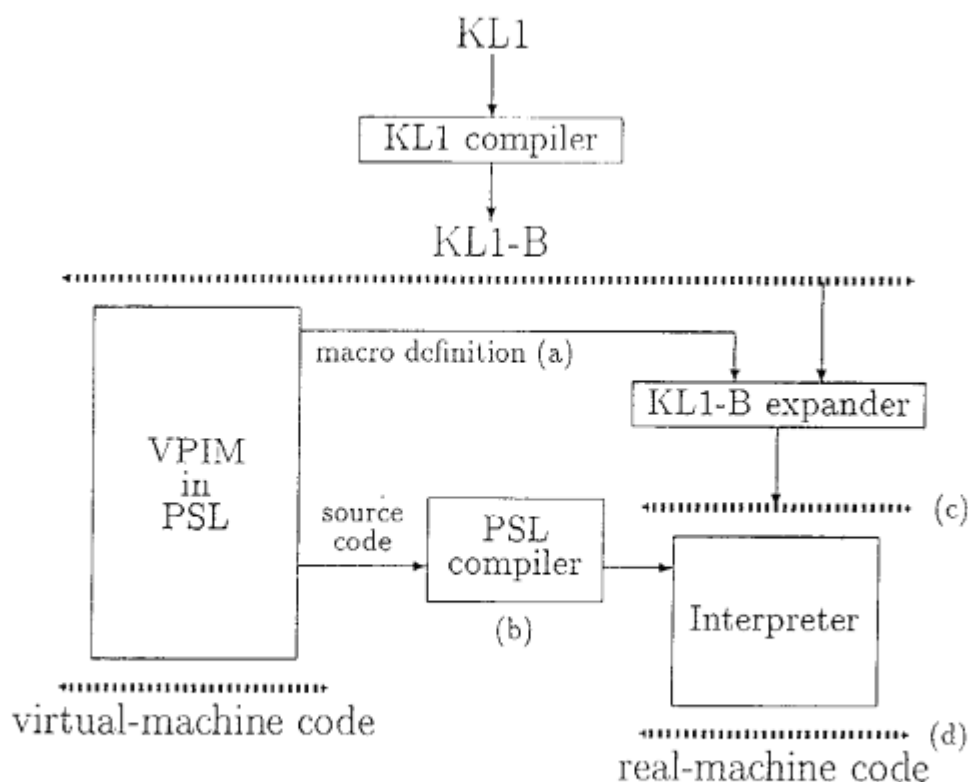


Figure 2: KL1 Language Processors

segment of VPIM codes corresponding to the intermediate-level instructions (c) is compiled to the native codes for the virtual PIM hardware once. The native codes of the virtual hardware can be translated to real-machine codes with little effort (Fig. 2 (d)).

Furthermore, VPIM has a useful feature; VPIM itself can be compiled as a C program with slight modification. Thus, we can easily prove the correctness of VPIM on conventional computers (the SUN workstation, Sequent Symmetry, and so on).

Actually, in order to implement the KL1-B expander easily, the part of VPIM which corresponds to macro-definition for the KL1-B expander is written in KL1, not in PSL. Thus, the KL1-B expander can be used as a self-expander also.

5 Current Status on Software Development

We show the current status on the PIM software development as follows.

- **VPIM:** Now we have just released version 0.5, which does not only implements almost all the basic intra-cluster functions (goal reduction, goal management, garbage collection, and so on) but also includes the following new features: packetized inter-cluster communication, and distributed resource management. Version 1.0 will be released in Dec. of this year which will include the SCSI device driver.
- **KL1 compiler, PSL compiler and KL1-B expander:** Prototyping of these compilers and expander is almost finished. These compilers can be used as self-compilers on the real PIM hardware as well. Their performance will be improved.

- **Miscellaneous:** Bootstrap routines and software for service processors are being developed. The bootstrap routines include the system initializer, IPL, and so on. The software for service processors includes debuggers, tracers, and statistic measurers.

The compiled code of the append program can run on a PIM/p hardware simulator, actually. Thus, our linker and loader work well, too.

6 Concluding Remarks

Our future plan is as follows: assembling and testing the machine hardware, and improving performance of the KL1 engines. We also would like to explore the possibility of other methods for efficient KL1 parallel implementation.

Acknowledgement:

We would like to thank the researchers who have been working on the PIMs and the Multi-PSI.

References

- [Goto 89a] A. Goto: Research and Development of the Parallel Inference Machine in the FGCS Project, In *Parallel Processing and Artificial Intelligence*, Eds. M. Reeve and S. E. Zenith, pp.65-96, John Wiley & Sons (1989).
- [Goto 89b] A. Goto: Developing the Parallel Inference Machine, In *Proc. of Joint Japanese-American Workshop on Future Trends in Logic Programming*, ANL-89/43, Argonne National Laboratory, pp.51-55 (Oct. 1989).

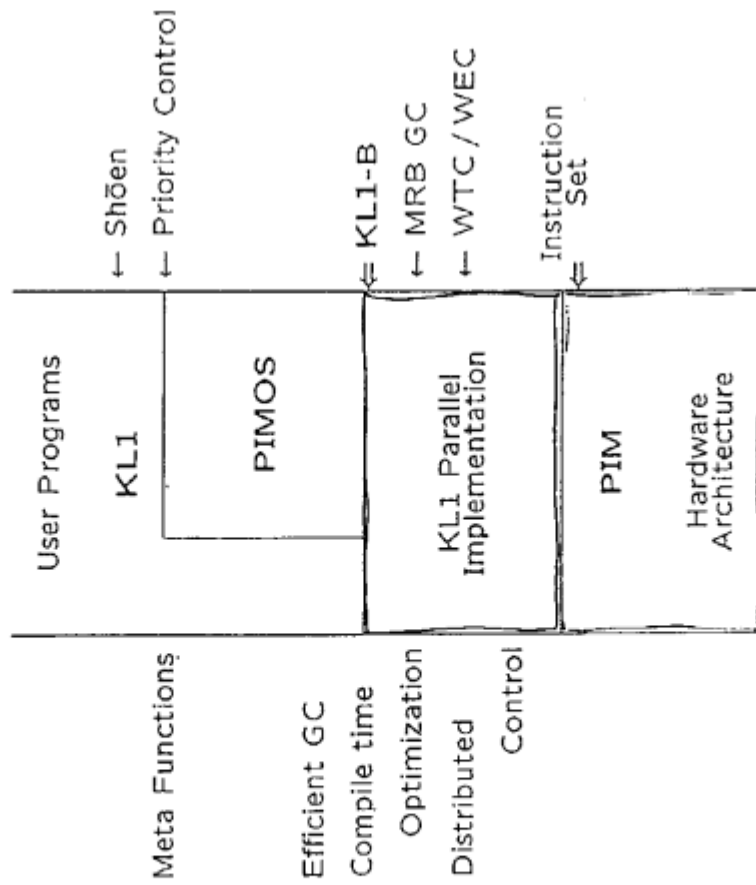
PIM Architectures and R&D Status

Keiji Hirata
(ICOT)

Outline

1. Machine Architectures and their Features
 Five PIMs
2. Current Status on Hardware Development
3. Software Configuration
 Firmware and Language Tools
4. Current Status on Software Development

Global Configuration of PIM System



Research Issues

- KL1 goal reduction (unifications)
 - with efficient memory management
 - ⇒ Save memory consumption
 - ⇒ Reuse memory area
- Parallel goal scheduling, and load distribution
- Distributed resource management

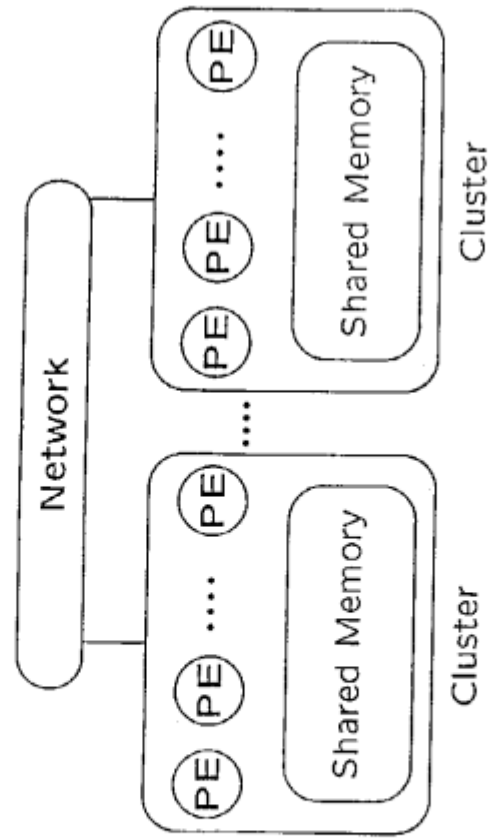


PIM Architecture

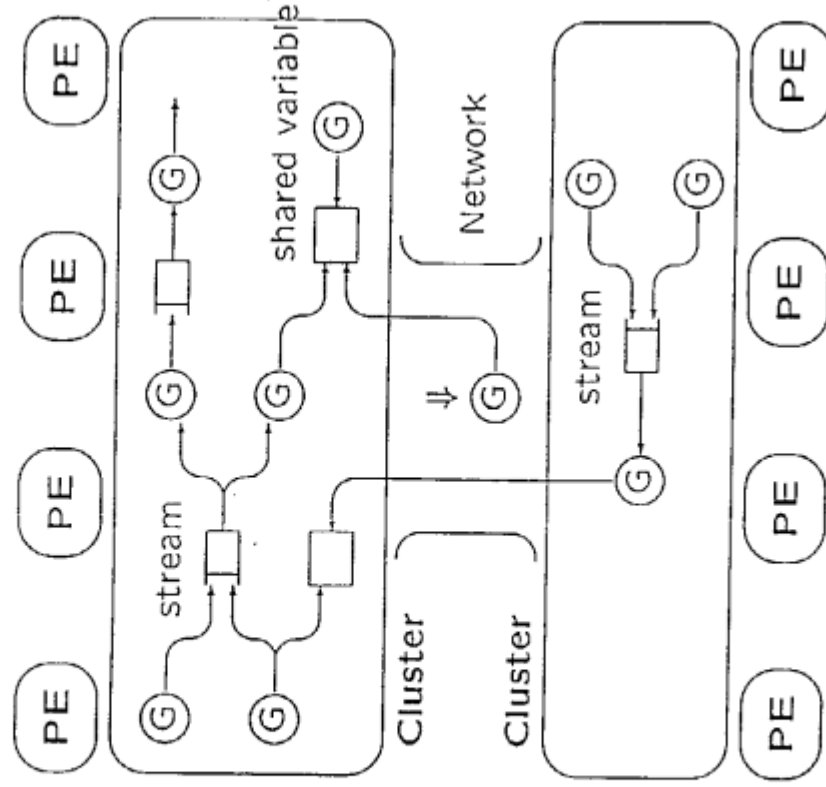
Localization in parallel goal reduction

- Reduce communication traffic
- Enhance memory access locality

Abstract Configuration of PIM



Parallel KL1 Goals in PIM



Alternatives in KL1-B Implementation

- (1)
- KL1-B interpretation by microprogram (HLIC/CISC).
 - Dispatching overheads to micro-instructions.

- (2)
- Expanded compiled code in RISC-like instruction set.
 - Code fetch overheads

- (3)
- Small subroutines in RISC-like instructions for each KL1-B instruction.
 - Overheads in subroutine call and return.
 - Overheads in parameter passing.

RISC + HLIC/CISC →

RISC-like instruction set

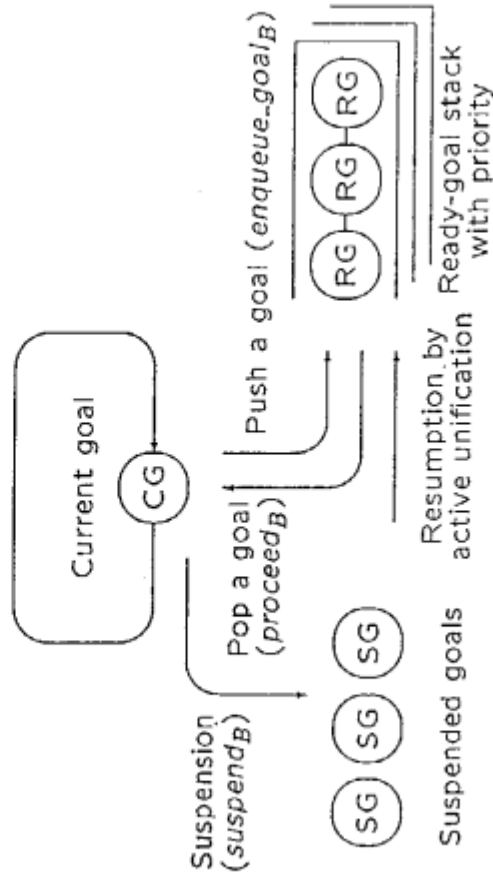
with Conditional-Macro-call instructions

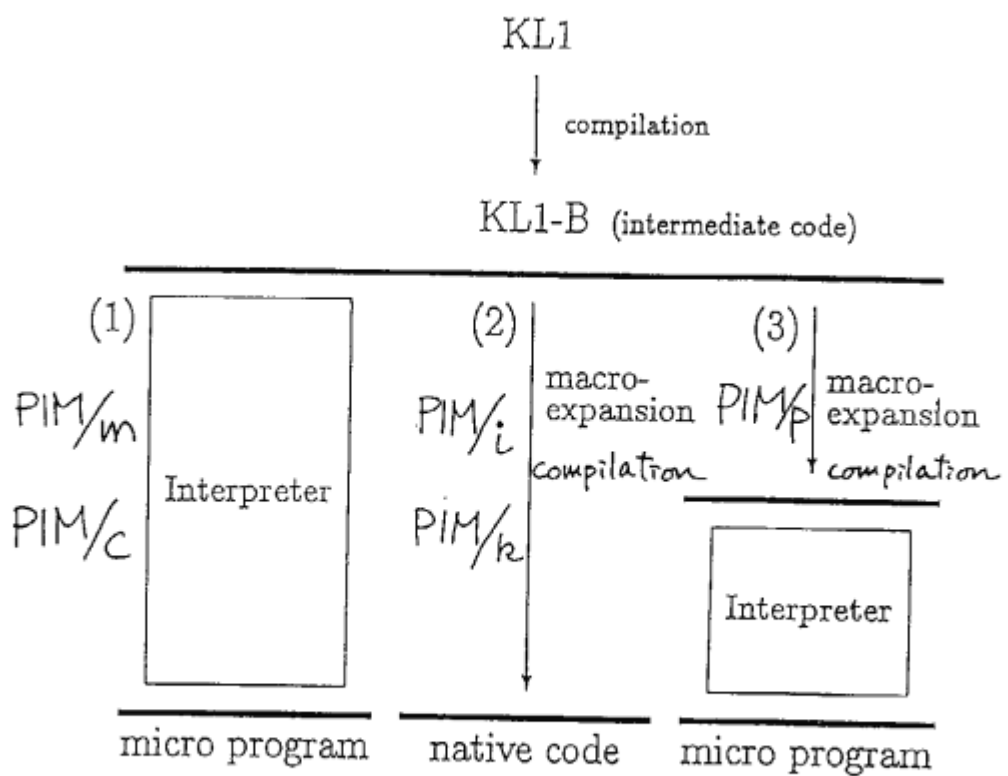
- + macro-body in local memory
- + indirect registers

KL1-B: Abstract machine instruction for KL1

- Goal reduction by register.
- Non-busy waiting goal scheduling.
- Incremental garbage collection by MRB.

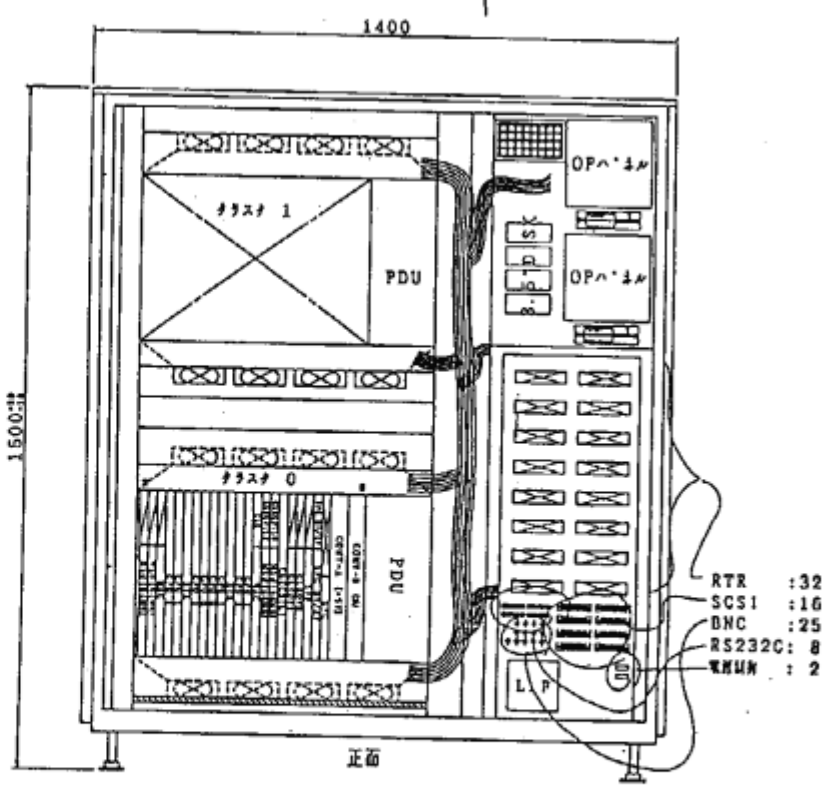
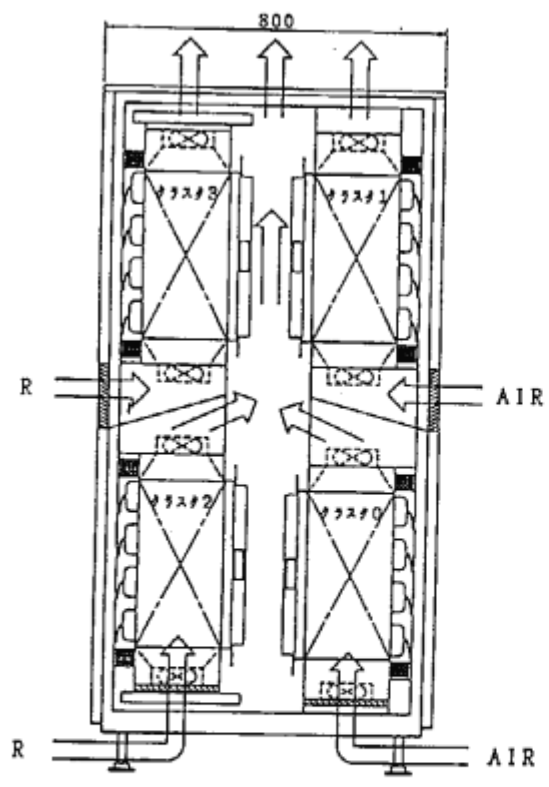
Continuous reduction (*execute_G*)



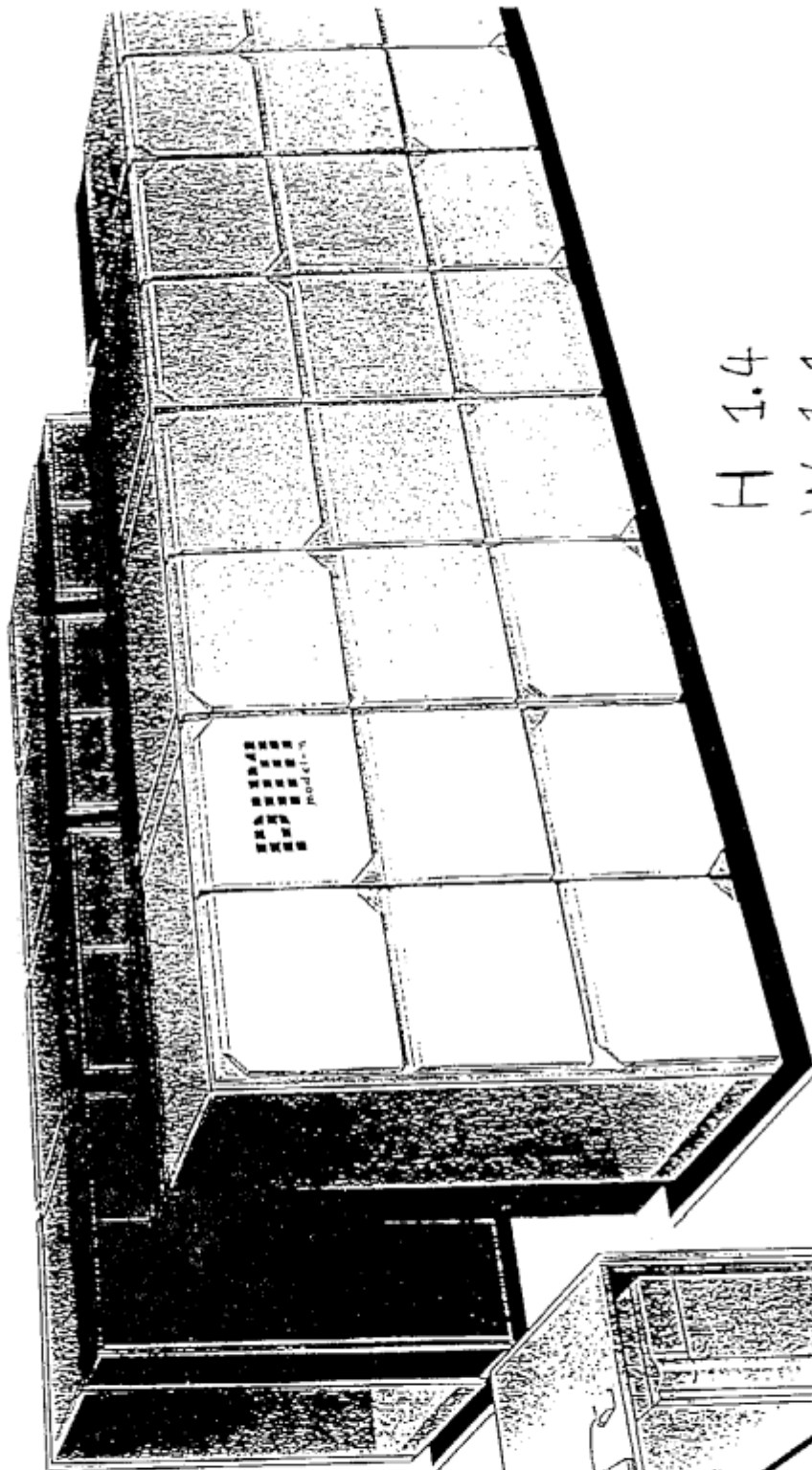


Three Methods For Executing KL1-B Instructions

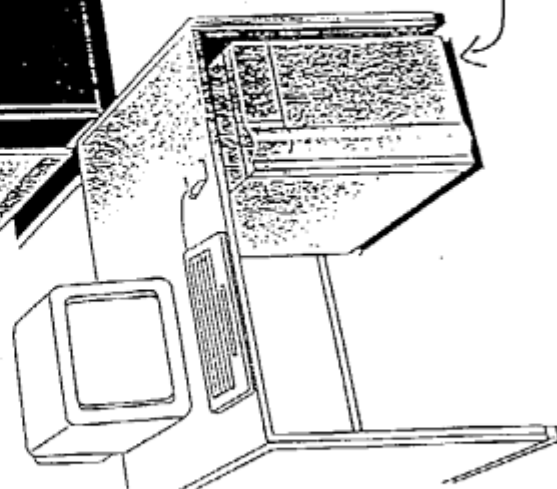
4 Clusters 32 PEs PIM/p



PIM/m



H 1.4
W 1.1
D 0.87



PSI-II

PIM/p Fujitsu

PIM/m Mitsubishi

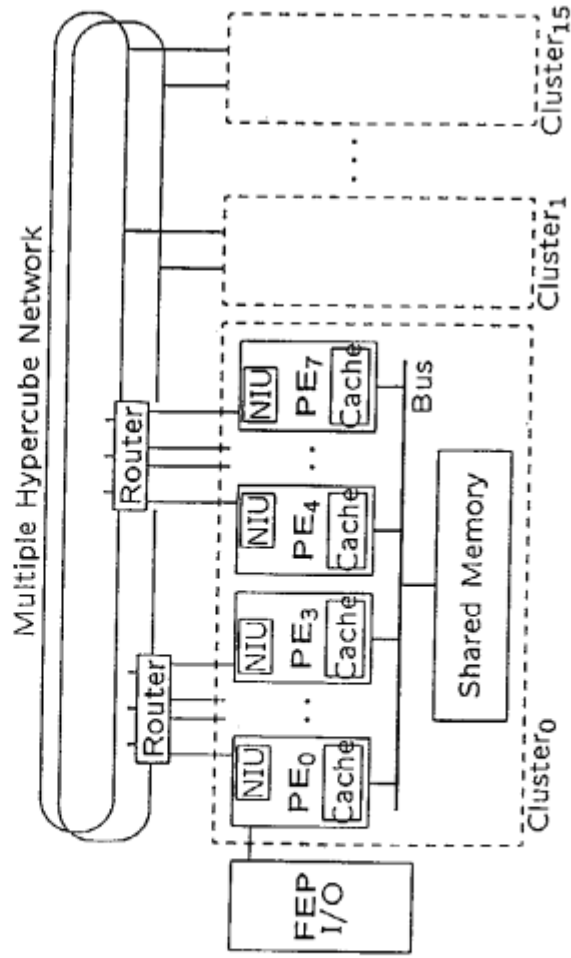
PIM/c Hitachi

PIM/i Oki

PIM/k Toshiba

Global Configuration of the PIM/p

- Cluster: 8 PEs + shared memory
- 16 clusters connected by multiple hyper-cube network
work
(512 PEs in total)



PIM/p Hardware Architecture

- Machine instruction set for KL1
 - ⇒ Short machine cycle by 4 stage pipeline (Hopefully 1 instruction / 50nsec cycle)
 - ⇒ Direct support for MRB GC and free list
 - ⇒ Conditional macro-call instructions
- Coherent local cache for KL1
 - ⇒ Efficient communication within a cluster
 - ⇒ Low cost hardware lock
- High performance inter-cluster network
 - ⇒ Efficient both for short and long messages
 - ⇒ A network port on each PE to send and receive messages where required

PIM/p Global Specification

Processing Elements

Execution	One cycle pipeline by 4 stages.
General Registers	40 bit × 32 W
Internal Instruction Memory	50 bit × 8 K W
Cache Capacity (each for code and data)	64 KB 256 column, 4 set, 32 B/block, 2 block/sector
Cache Protocol	Write back, Invalidation, Special commands for KL1

Cluster

Number of PE	8
Shared Memory	256 MB

Network

Topology	Doubled Hyper-cube (Max 6 dimension)
Throughput	Max 20 MB/sec in each link (40 MB/sec for cluster)

Inter-cluster Configuration

	Topology	Number of Clusters	Total number of PEs
PIM/p	hypercube	64	512
PIM/c	crossbar	32	256
PIM/m	mesh	-	256
PIM/i	-	2	16
PIM/k	-	2	32

Intra-cluster Configuration

	Number of PEs	Number of NIs	Comment
PIM/p	8	8	each PE has NI
PIM/c	8	1	NI is connected to the bus
PIM/m	1	1	
PIM/i	8	1	NI is connected to the bus
PIM/k	16	1	one of 16 PEs has NI

(PE = processing element, NI = network interface)

Specification on Processing Element

	Instruction set	Cycle time	LSI fabrication device	Line interval
PIM/p	RISC + macro instruction	60 ns	standard cells	0.96 μm
PIM/c	CISC (micro programmable)	50 ns	gate arrays	0.8 μm
PIM/m	CISC (micro programmable)	60 ns	cell base	0.8 μm
PIM/i	RISC	100 ns	standard cells	1.2 μm
PIM/k	RISC	100 ns	custom cells	1.2 μm

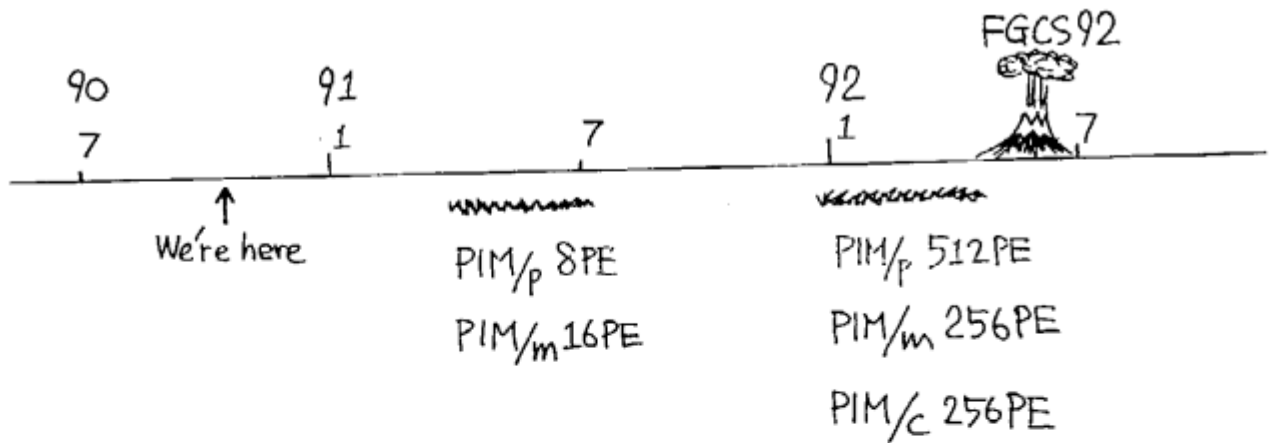
Specification on Coherent Cache

	Protocol	Number of states
PIM/p	invalidation (Illinois)	4
PIM/c	invalidation (modified Illinois)	5
PIM/m	-	-
PIM/i	broadcasting	6
PIM/k	invalidation (modified Berkeley), hierarchical	4

Specification on Disk Channel

	Number of PEs / Channel	GB / Channel
PIM/p	4	1.160
PIM/c	1	-
PIM/m	8	0.6
PIM/i	8	-
PIM/k	1	-

PIM Arrival Schedule



(as of Oct. '90)

VPIM (Virtual PIM)

implementing KL1-B engine
firmware on virtual PIM hardware

implementation techniques of MultiPSI V2
written in PSL (PIM Descriptive Language)

VPIM

Version 0.5 (Sept. '90)

- goal reduction
- goal scheduling
- priority control
- MRB GC
- global GC

• inter-cluster communication (prototype)

• basic Shoen functions
(distributed execution control)

Version 1.0 (Dec. '90)

- exception handling for PIMOS
- multipacket
- full Shoen functions
- self-system support
- SCSI device driver

Miscellaneous

- KL1 compiler, PSL compiler, KL1-B expander
- self-linker, self-loader
- bootstrap routines
- Service processor software
- debuggers, tracers (firmware level)

keywords:

VPIM

PSL.

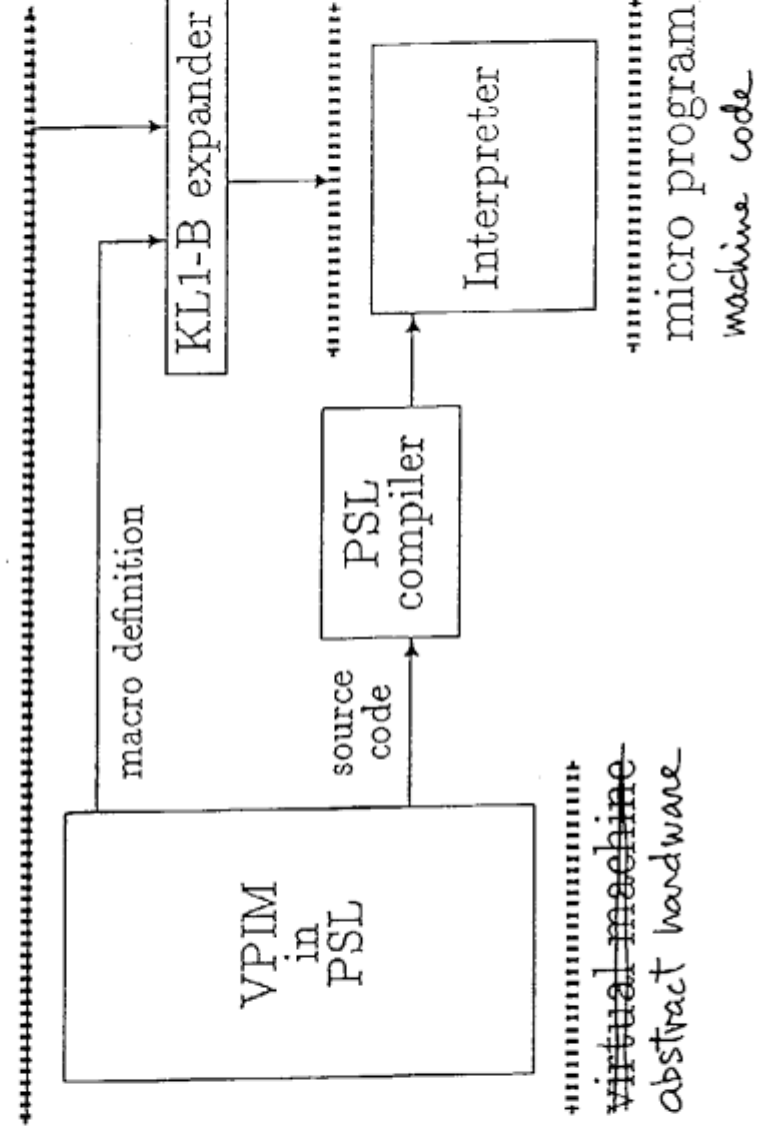
KL1-B

KL1

KL1 compiler

KL1-B

KL1-B Abstract Machine



KL1 Language Processors

Evaluation of the KL1 Implementation on the Multi-PSI

Yū Inamura

Institute for New Generation
Computer Technology
1-4-28 Mita, Tokyo, 108, Japan
inamura@icot.or.jp

Katsuto Nakajima

Mitsubishi Electric Corporation
5-1-1 Ofuna, Kamakura, 247, Japan
nak@isl.melco.co.jp

Abstract

The Multi-PSI, a loosely-coupled multiprocessor running the concurrent logic programming language KL1 (kernel language version 1), has been developed for conducting parallel non-numeric software research and for testing various new implementation techniques for concurrent logic languages.

This paper reports some measurement results in terms of intra- and inter-processor operations of the Multi-PSI system, and they show the basic performance of the distributed implementation of KL1.

We also measured performance and communication overheads in benchmark programs, and it was ascertained that the overall communication overheads were acceptably small.

1 Introduction

The Multi-PSI has been developed in the Japanese FGCS project as a testbed for the implementation of the concurrent logic language KL1 [2]. Up to 64 processing elements (PEs), each of which is identical to the CPU of the personal sequential inference (PSI) machine [8], are connected by an 8×8 mesh network having automatic routing capability.

A distributed KL1 system [6] on the Multi-PSI is not only the language system but also an operating system kernel. The design goal was to obtain overall high performance including garbage collection overhead. The task and resource managements are both decentralized.

This paper gives the measurement results of the costs of intra- and inter-processor primitive operations in the system. It is rare to implement concurrent languages on the actual parallel machines, so we think these measurement results are useful as the basic data for designing parallel machine hardware and the implementation of concurrent languages.

In terms of intra-processor operations, we measured some primitive operation costs such as *goal fork*, *unification*, and *suspension*. We also measured the effectiveness of some optimization techniques, which were devised to get rid of some disadvantages of logic programming languages as compared with conventional procedural languages.

In terms of inter-processor operations, actual communication overheads in two benchmark programs are also measured, in addition to the analysis of the inter-processor message handling costs[7].

2 Intra-processor Operation Costs

2.1 Append Speed

An *append* (list concatenation) program is often used as a benchmark program for logic programming languages. Append program written in KL1 follows:

```
append([X|X1],Y,Z) :- true | Z=[X|Z1], append(X1,Y,Z1).  
append([],Y,Z) :- true | Z=Y.
```

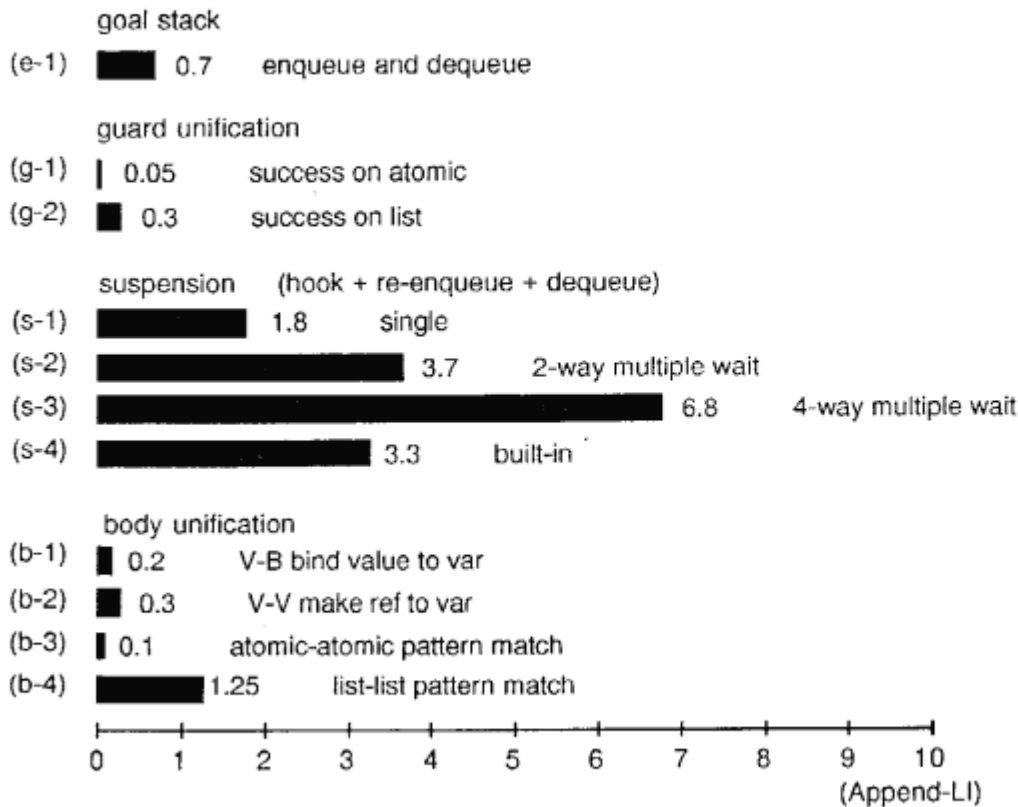


Figure 1: Cost of Typical Intra-processor Operations

The cost of one reduction (iteration) of the first clause is 39 steps of the micro instructions in the best case (no suspension, etc), and the speed turns out 128 KRPS (Kilo Reduction Per Second) assuming no cache miss.

In this paper, we define the cost above (about 8μsec) as one *append-LI*(Logical Inference) or one *LI* to normalize our measurement results in the following sections for comparing each items.

2.2 Basic Operation Costs

Figure 1 shows the costs of typical primitive operations in KLI programs.

The cost of enqueueing a goal to the goal stack and dequeuing it is about 0.7 *append-LI*. As the *append* loop is performed in tail recursion optimization (TRO), the gain of TRO in *append* is $(0.7/1.7) \times 100 = 40\%$.

There are four typical cases in a guard unification: success to test an atomic data (g-1) or an structure data such as a list (g-2), and non-success because the caller variable is not instantiated (g-3) or the value is mismatched (g-4). (g-3) causes suspension of the goal if there is no alternative clause for the call. We can avoid most of (g-4) by clause indexing compilation.

Non-busy wait mechanism is used for goal suspension. The goal is connected to the causal variable and waits for its instantiation. We call this operation *goal hooking*. Suspension in Figure 1 includes the sum of the costs for hooking, resuming (re-enqueueing) and dequeuing the goal (s-1). If there are two unbound variables each of which may allow to commit a clause, the goal is hooked to both variables to construct an

Table 1: Performance improvement with structure reuse

	No reuse (KRPS)	Frame reuse (KRPS)	Element reuse (KRPS)
Append	110	128	146
Qsort	95.8	108	113
Primes	55.9	59.8	61.2

RPS: reductions per second

OR-wait suspension (s-2). (s-3) is the case of an OR-wait suspension of four variables. KL1 body built-in predicates also suspend if one of their input arguments is uninstantiated (s-4).

Most body unifications in KL1 are; (b-1) binding a value to an unbound variable or (b-2) making a reference pointer from an unbound variable to another. Note that (b-1) does not include the cost for re-enqueuing goals hooked to the instantiated variable.

It is rare to perform a pattern matching between atoms (b-3) or structures (b-4), and the latter is not optimized in the current implementation.

2.3 Effectiveness of Optimization Techniques

In this section, we give measurement results in terms of some optimization techniques, intending to get rid of the disadvantages of logic programming languages, as compared with conventional procedural languages.

The techniques are (1) destructive update of structure data, and (2) built-in stream merger, each of which is based on the multiple reference bit (MRB) management scheme[1]. The reader is referred to [5] for details of these techniques.

2.3.1 Destructive Update of Structure

The effect of the structure reuse was measured with several small benchmark programs, such as *Append*, *Quick-sort*, and *Prime number generator*. Each program was compiled in three ways and the execution speed of each is measured. The three ways are:

1. Without any structure reuse;
2. With structure frame reuse;
3. With structure element reuse.

Table 1 shows the measurement results.

The performance improves by 10% to 30% with structure element reuse in these small benchmarks. This improvement is mainly brought about by the decrease of the number of instruction steps when running these small benchmarks. However, structure reuse can also reduce the frequency of memory access, and this can increase the performance particularly on shared memory machines such as the parallel inference machines (PIMs)[4].

2.3.2 Built-in Stream Merger

The cost of the built-in merge procedure is compared with the merger defined with KL1, to find out how the built-in merger improves the performance.

Figure 2 shows the cost of merging one element, with the input list's MRB both on and off.

The difference between the KL1 merger and the built-in merger with MRB on can be regarded as the effect of the built-in merger representation, which can be realized even in the implementation without the MRB mechanism. The four times performance was attained by the introduction of the built-in merger.

The difference between the built-in merger with the MRB off and on is the effect of the MRB. Execution with the MRB off is twice as fast as that with it on.

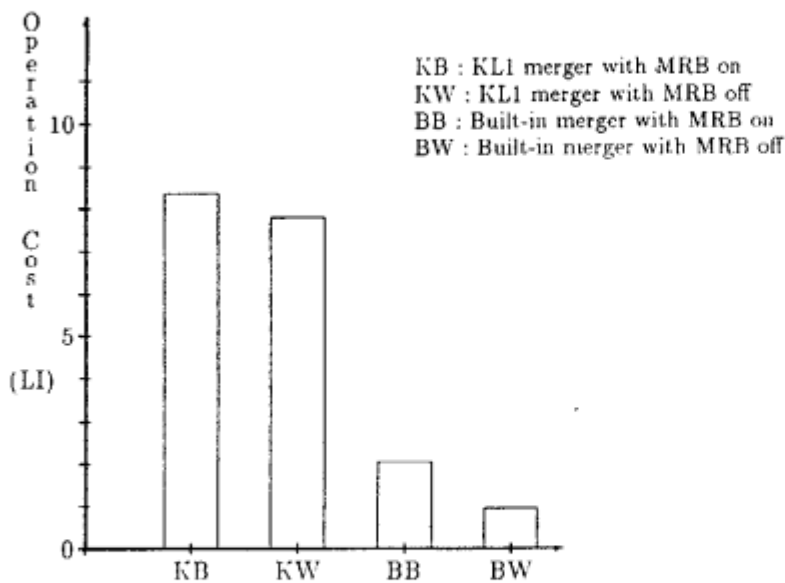


Figure 2: Performance of the merge operation

Although the performance improvement with the built-in merger without MRB scheme seems to be sufficient, we are convinced that MRB mechanism is necessary, since merge operation is used quite frequently in KL1 programs, and influences the overall system performance.

3 Inter-processor Operation Costs

3.1 Message Handling Costs

Figure 3 shows the costs for handling typical messages.

A `%throw` message is used for load distribution, and a goal (process) is transferred to another processor by this message.

A `%read` message is sent to refer the data object existing in another processor. The value of the data object is transferred by an `%answer_value` message as the response to the `%read` message. More details

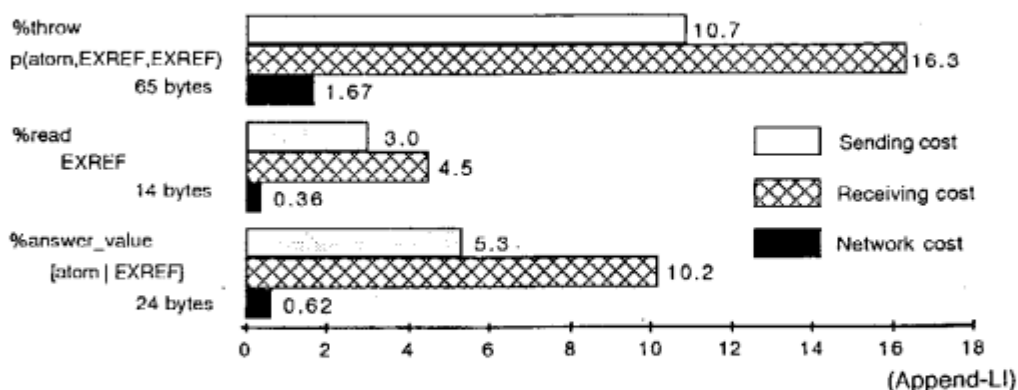


Figure 3: Cost of Typical Inter-processor Operations

Table 2: Message Frequency and Reductions

Pentomino (39.3 KRPS on 1 PE)

Num of PEs	4 PEs	16 PEs	64 PEs
execution time (msec)	54,63	14,62	4,35
total reductions ($\times 1000$)	8,317	8,332	8,340
reductions/sec (KRPS)	152.2	570.1	1,919.4
reductions/msg	221	108	88
msg bytes/sec ($\times 1000$)	14.5	108.1	440.5

Bestpath (23.4 KRPS on 1 PE)

Num of PEs	4 PEs	16 PEs	64 PEs
execution time (msec)	10,655	4,062	1,691
total reductions ($\times 1000$)	987.7	1213.6	1,505.2
reductions/sec (KRPS)	92.7	298.8	890.1
reductions/msg	21.9	11.7	6.2
msg bytes/sec ($\times 1000$)	114.0	692.5	3,854.3

about inter-processor operations are found in [6].

The measurement condition follows:

- The costs of sending and receiving a 65-byte `%throw` message whose three arguments are an atom and two external pointers, which point to the data objects existing in other processors, as in a typical situation.
- The costs of sending and receiving a 14-byte `%read` message requesting the contents of an external pointer and a 24-byte `%answer_value` message answering the request. The returned data is a list whose CAR is an atomic data and the CDR is an external pointer.

The routing hardware have 5M bytes/sec of the bandwidth for transmitting messages. Compared with the network costs (hardware capability: 1.67/0.36/0.62 append-LI for 65/14/24-byte), the sending and receiving costs of the microprogram execution are quite large. They include the costs of address translation, encoding and decoding messages, and distributed goal management and other resource management.

3.2 Measurements with Benchmark Programs

We took measurements for two different types of benchmark programs.

- **Pentomino:** A program to find out all solutions of a packing piece puzzle (Pentomino) by exploring the whole OR-tree. Two-level dynamic load balancing is employed [3].
- **Bestpath:** A 160×160 grid graph is given together with non-negative edge costs. The program determines the lowest cost paths from a given vertex to all the other vertices of the graph by performing a distributed shortest path algorithm. The vertices are represented as KLI processes, and they communicate with each other to determine the shortest paths.

3.2.1 Message and Reduction Profile

Table 2 shows the execution time, the total reductions and the message frequency, etc. The message sending rates on 64 PEs are: one message per 88 reductions in Pentomino, and one per 6 reductions in Bestpath.

The average network traffic can be calculated from these figures. Relative to the 5 Mbytes/sec processor-processor channel bandwidth of the Multi-PSI network hardware, the average traffic on a processor-processor channel is very small: 0.08% (Pentomino) and 0.3% (Bestpath) of the bandwidth. We expect that the

network hardware will not be a bottleneck in inter-processor communication even if the system scales up to 1K processors.

3.2.2 Runtime Analysis and Speed-up

Figure 4 shows the average breakdown of execution steps of all processors and speed-up ratio with both benchmarks.

In the case of Bestpath problem, the idling ratio is not worse than that of Pentomino, however, the computing ratio is much worse because large inter-processor communication overheads and large cache miss penalty exist. Thus the speed-up ratio of Bestpath problem is worse than that of Pentomino, as shown in figure 4.

4 Conclusions

We reported some measurement results of intra- and inter-processor operation costs of the Multi-PSI machine.

With regard to intra-processor operations, it was known that some primitive operations, such as typical guard unification, are performed very fast, using tagged architecture of the hardware. It was also known that some measurement results prove the effectiveness of the optimization techniques we devised.

Concerning inter-processor communications, it was ascertained that the network hardware is not fully used, and most communication overheads are brought about by the micro steps for message handling. We now have plan to optimize these operations by re-writing the micro codes relating to message handling, and expect that the execution steps will be reduced much.

However, even now, the dynamic characteristics of two benchmarks show that inter-processor communication can be limited without affecting the workrate of processors by contemplating the load balancing strategy. We think both language implementations and application softwares cooperate together in order to accumulate much more experiences concerning concurrent logic programming languages.

References

- [1] T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, Vol. 2, pp.276-293, 1987.
- [2] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.230-251, ICOT, Tokyo, 1988.
- [3] M. Furuichi, K. Taki and N. Ichiyoshi. A Multi-Level Load Balancing Scheme for OR-Parallel Exhaustive Search Programs on the Multi-PSI. In *Proc. of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 1990.
- [4] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.208-229, ICOT, Tokyo, 1988.
- [5] Y. Inamura, N. Ichiyoshi, K. Rokusawa, K. Nakajima. Optimization Techniques Using the MRB and Their Evaluation on the Multi-PSI/V2. In *Proceedings of the North American Conference on Logic Programming 1989*, pp.907-921, 1989.
- [6] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, T. Chikayama. Distributed Implementation of KI.1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, pp.436-451, 1989.

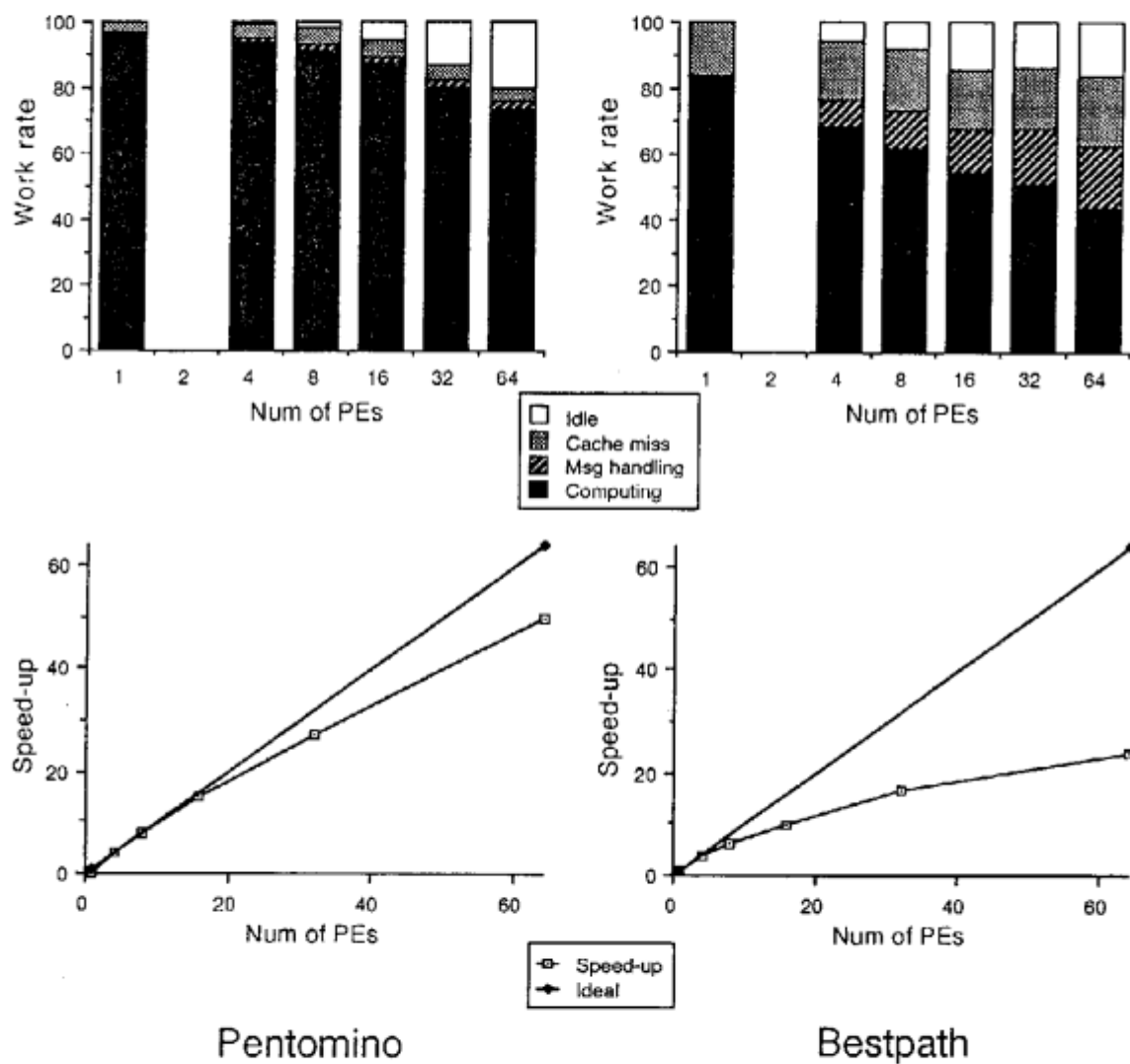


Figure 4: Runtime Analysis and Speed-up

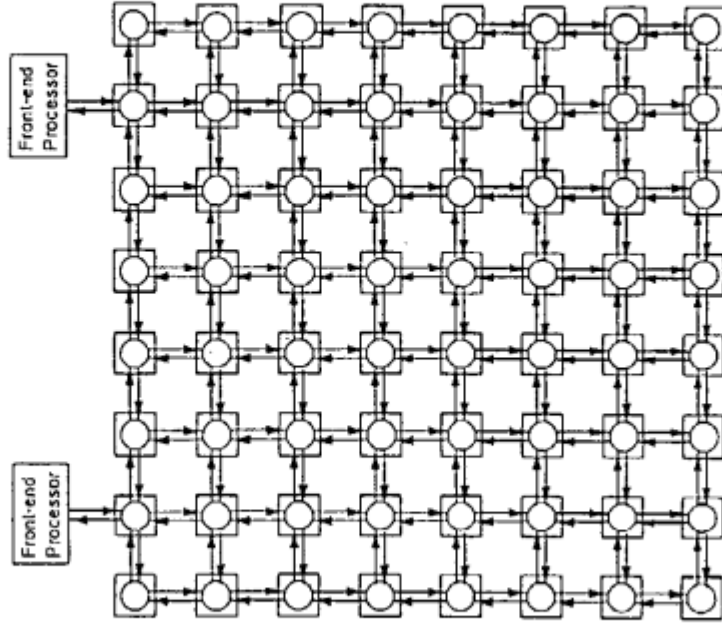
- [7] K. Nakajima, N. Ichiyoshi. Evaluation of Inter-processor Communication in the KL1 Implementation on the Multi-PSI. In *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. I, 1990.
- [8] H. Nakashima and K. Nakajima. Hardware architecture of the sequential inference machine : PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, pp 104-113, 1987.

Evaluation of the KLI Implementation on the Multi-PSI

Yū Inamura
ICOT

Katsuto Nakajima
Mitsubishi Electric Corp.

1. Overview of the Multi-PSI



- Loosely-coupled Multi-Processor
- Max. $8 \times 8 = 64$ Processors

1. Overview of the Multi-PSI

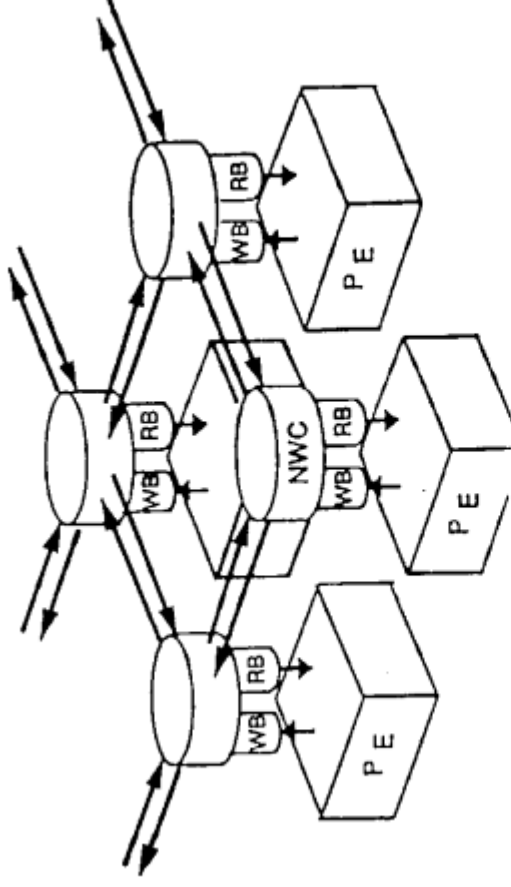
2. Overview of the KL1 and Its Implementation

3. Performance Evaluation

(a) Effectiveness of Optimization Techniques

(b) Evaluation of Inter-PE Operations

4. Summaries



NWC (Net-Work Controller) :

- Automatic Routing
- 5 M Byte/sec/channel

PE (Processing Element) :

- Same as PSI-II CPU (Cycle Time = 200 nsec)
- Tag Architecture
- Main Memory = Max. 80 M Byte/PE
- 130 KRPS (Reduction Per Second)/append/PE

Overview of the KLI Language

- KLI \Rightarrow Concurrent Logic Language
H :- $G_1, \dots, G_m \mid B_1, \dots, B_n$.
 - Dataflow Synchronization
 - Single Assignment

Features of the KLI Implementation(1)

- Optimization Techniques
 - \Rightarrow To Get Rid of the Disadvantages (As Compared with Conventional Procedural Languages)
 - * Destructive Update of Structure Data
 - * Stream Merger

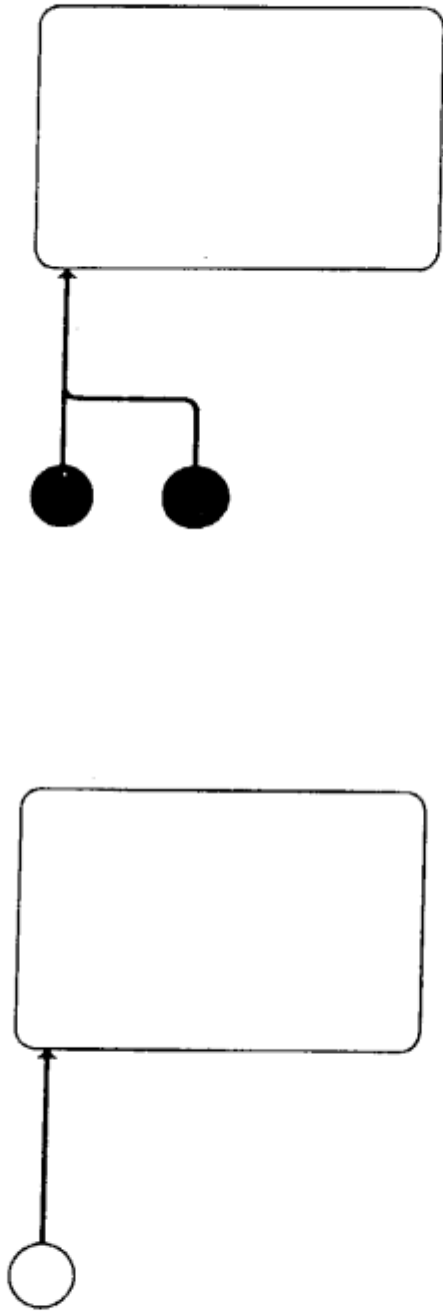
(1) Destructive Update of Structure

- Update of Structure
 - Done by Copying in Naive Implementation
 - Done with Constant Overheads by Mutable Array
Though the Overheads are Still Large
 - The MRB Enables Destructive Update

The Overview of the MRB Scheme

The MRB (Multiple Reference Bit):

- One Bit Information Attached to Pointers
- Indicating Whether Other Pointers Exist or not.
 - OFF \Rightarrow Single-referenced
 - ON \Rightarrow Maybe Multiple-referenced

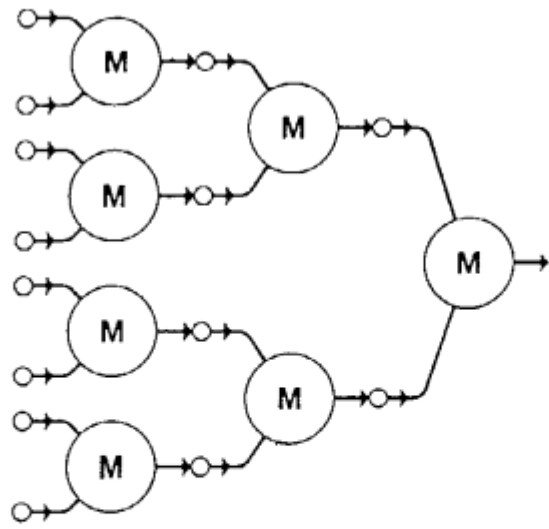


Possible Conditions of the MRB Status

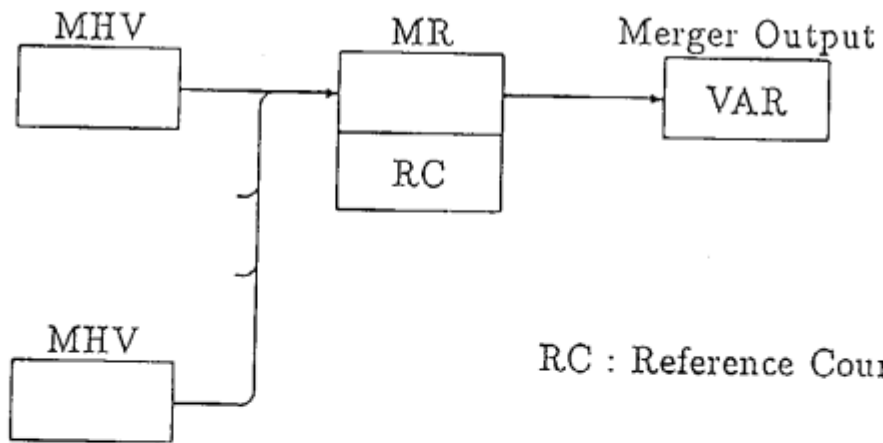
(2) Constant-time Stream Merger

```
append([], Y,Z) :- true | Z = Y.  
append([X1|X], Y,Z) :- true |  
      Z = [X1|Z1], append(X., Y, Z1).  
  
set_vector_element(V, N, OldElm, NewElm, NewV)
```

```
merge([], In2, Out) :- true | Out = In2.  
  
merge(In1, [], Out) :- true | Out = In1.  
  
merge([X|In1], In2, Out) :- true |  
      Out = [X|Out2], merge(In1, In2, Out2).  
  
merge(In1, [X|In2], Out) :- true |  
      Out = [X|Out2], merge(In1, In2, Out2).
```

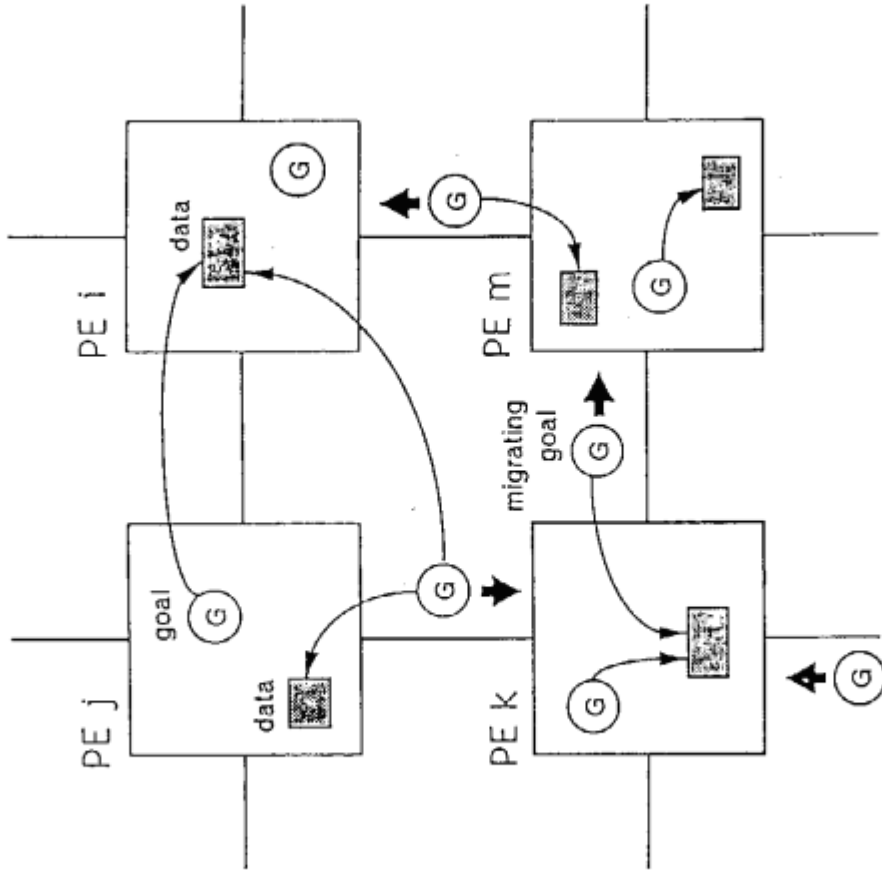


Tree of Binary Merger Processes

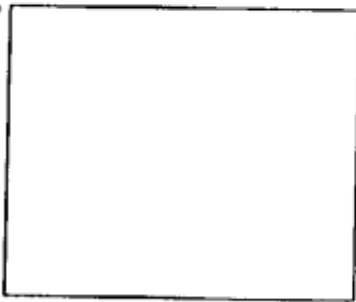


Features of the KL1 Implementation(2)

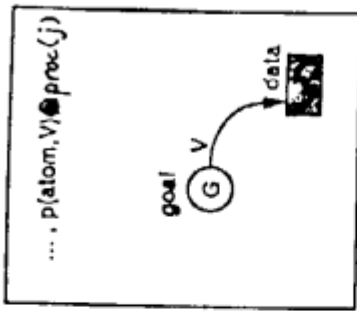
- Inter-processor Operation
 - Load Distribution by *throw_goal* pragma
 - ⇒ `goal@processor(N)`
 - External Reference Management
 - ⇒ To Keep Consistency of Logical Variables



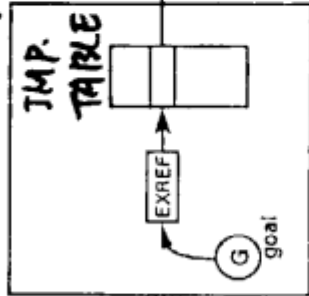
PE J



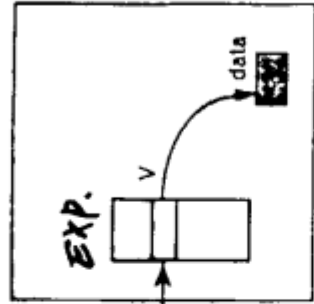
PE I



PE J

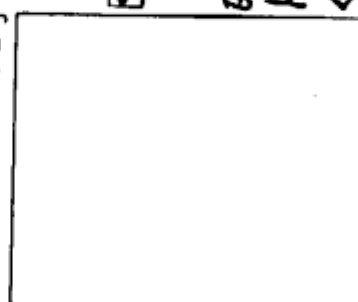


PE I

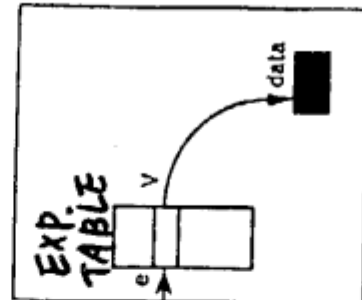


EXT. REF.

PE J

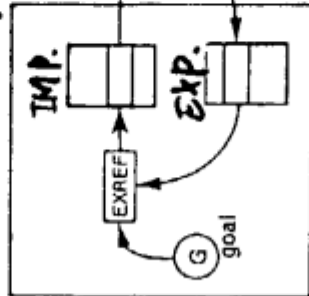


PE I

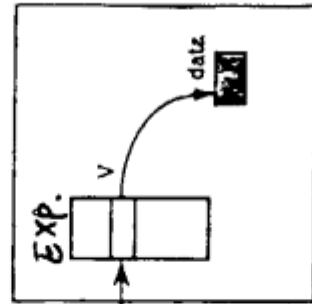


EXTERNAL REFERENCE <i, e>

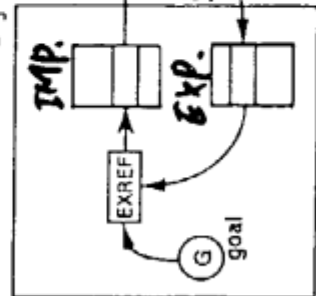
PE J



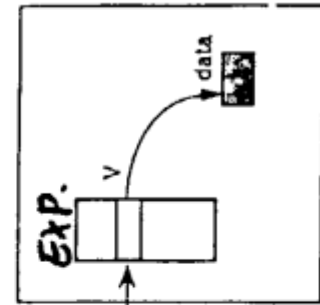
PE I



PE J



PE I



Measurement Results

(1) Destructive Update of Structure

	No		Frame		Element	
	Reuse	110 K	Reuse	128 K	Reuse	146 K
Qsort		95.8 K		108 K		113 K
Primes		55.9 K		59.8 K		61.2 K

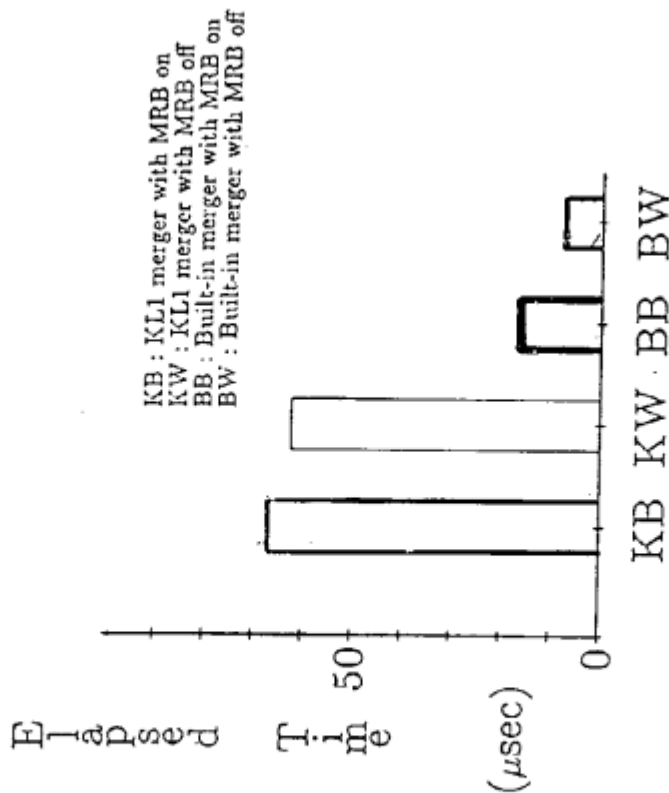
Unit : Reduction Per Second

```

append([], Y,Z) :- true | Z = Y.
append([X1|X],Y,Z) :- true |
    Z = [X1|Z1], append(X, Y,Z1).

```

(2) Stream Merger



Communication Cost Analysis

3. Performance Evaluation

Loosely-Coupled \Rightarrow Large Inter-PE Communication Cost

3.1 Evaluation of Hardware & Implementation

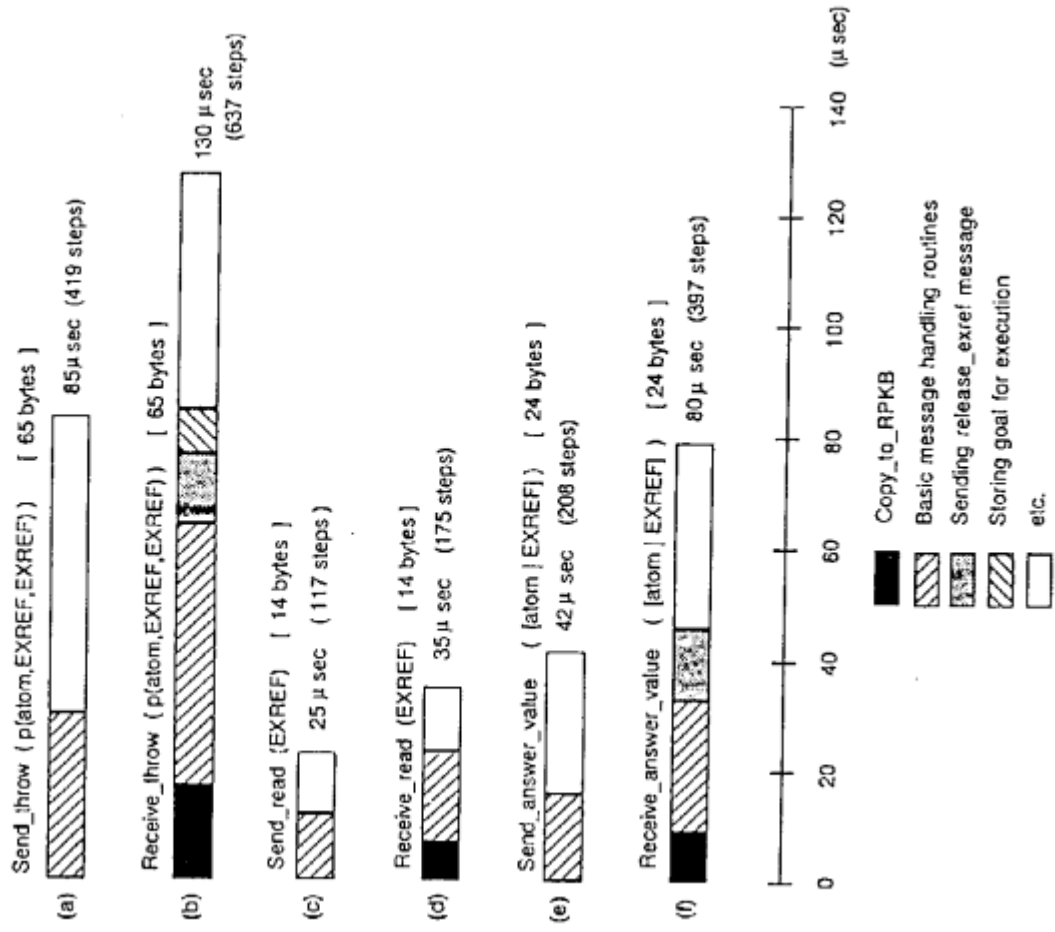
\Rightarrow by Communication Cost Analysis

\Rightarrow (Static Analysis)

3.2 To Find out the Limitation Factor of Performance Improvement

\Rightarrow by Communication Overhead Analysis

\Rightarrow (Dynamic Analysis)



Basic message handler (Hardware Operation)

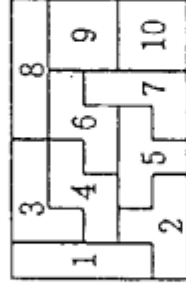
- Message Header Addition
- 32 Bit \rightarrow 8 Bit Translation (Hardware Spec)
- Message Tail Addition
- Copy_RPKB(Saving Message from Network Hardware to Memory)

KL1 Encoder / decoder (KL1 Operation)

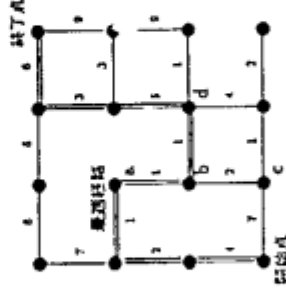
- Arguments Translation (Including Translation to External Reference)
- Translation of Shōen Attributer and Execution Environment
- Translation of Code

3.2 Communication Overhead Analysis (Measurements Using Benchmarking Programs)

3.2.1 Benchmarking Programs



- Pentomino (Packing Piece Puzzle)

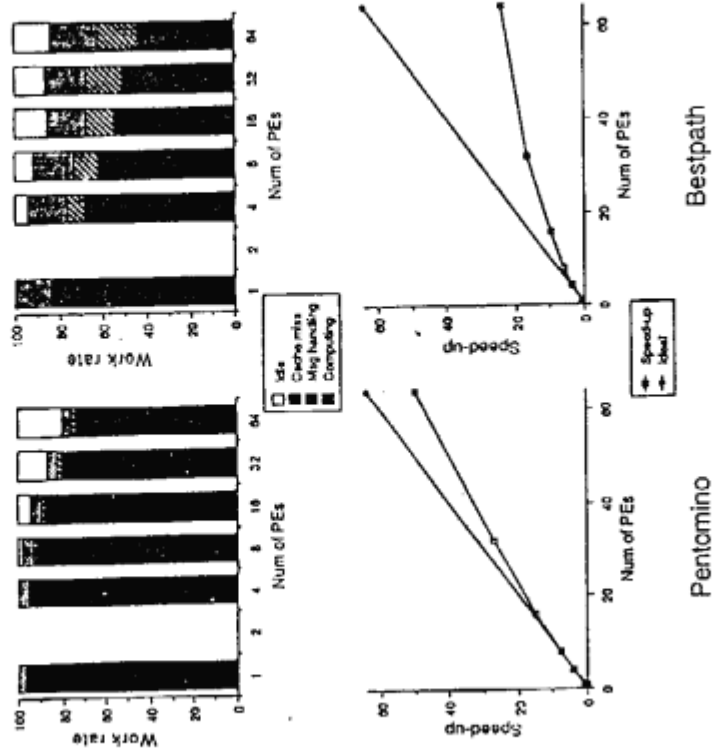


- Shortest Path Search

3.2.2 Dynamic Frequency of Messagees

Pentomino (39.3 KRPS / 1 PE)			
Message	4 PE	16 PE	64 PE
Total Message Number	37,625	77,506	94,750
%read	27.7 %	31.9 %	32.4 %
%answer-value	27.7 %	31.9 %	32.3 %
%release	18.2 %	15.9 %	14.4 %
%unify	6.3 %	9.7 %	11.7 %
%throw-goal	3.2 %	2.2 %	1.9 %
etc.	16.9 %	8.4 %	7.3 %
Mean Length (Byte/msg.)	21.1	20.4	20.2
Execution Time	54,634 ms	14,615 ms	4,345 ms
Total Reduction	8,317 K	8,332 K	8,340 K
RPS	152.2K	570.1K	1,919.4K
Reduction/msg.	221	108	88
Msg. Freq. (Byte/s)	14.5 K	108.1 K	440.5 K

Communication Overhead



Shortest Path (23.4 KRPS / 1 PE)				
Message	4 PE	16 PE	64 PE	
Total Message Number	45,005	103,424	241,404	
%read	27.5 %	27.8 %	27.7 %	
%answer.value	22.6 %	23.2 %	23.6 %	
%unify	18.8 %	18.6 %	18.0 %	
%release	13.8 %	13.9 %	13.9 %	
%throw-goal	13.8 %	13.9 %	13.9 %	
etc.	3.5 %	2.6 %	2.9 %	
Mean Length (Byte/msg.)	27.0	27.2	27.0	
Execution Time	10,655 ms	4,062 ms	1,691 ms	
Total Reduction	987.7 K	1,213.6 K	1,505.2 K	
RPS	92.7K	298.8K	890.1K	
Reduction/msg.	21.9	11.7	6.2	
Msg. Freq. (Byte/s)	114.0 K	692.5 K	3,854.3 K	

3.3 Evaluation of Network Hardware

3.3.1 Network Traffic on Benchmarks

Pentomino on 64 PE:

$$\frac{(\text{Total Message Number: (Byte/s)}) \times (\text{Mean Communication Length})}{(\text{Total Channel Number})} = \frac{441 \text{ KB/s} \times 2}{224} = 3.9 \text{ KB/s} = 0.08\% \times 5 \text{ MB/s}$$

Shortest Path on 64 PE:

$$\frac{3,854 \text{ KB/s} \times 1}{224} = 17.2 \text{ KB/s} = 0.3\% \times 5 \text{ MB/s} = (63 \text{ KB/s on 1 PE})$$

Summaries

We have revealed the evaluation of the KL1 implementation on the Multi-PSI.

3.3.2 Anticipation for Larger Scale System

Condition:

- 1,000 PE (32 × 32)
- Fine Grain(Message Frequency = Bestpath) = 6.2 Reduction/msg.
- Random Communication (Mean Communication Length 21.3)

$$\frac{(\text{Total Message Number (Byte/s)}) \times (\text{Mean Communication Length})}{(\text{Total Channel Number})} = \frac{63,000 \text{ KB/s} \times 21.3}{3,968} = 346 \text{ KB/s} = 6.9\% \times 5 \text{ MB/s}$$

- Basic operations are improved up to 8 times by the optimization techniques
- Communication overhead is very small in OR-Tree search problem
- Even in higher communication frequency (6 reduction/message), communication overhead is less than 30 %
- Network traffic is low
⇒ Scalable to 1000 processors.

Benchmarking and Evaluation of Software Systems

the EDS and Flagship Projects

Paul Townsend, Brian Proctor and Paul Watson

ICL,
Wenlock Way, West Gorton,
MANCHESTER, M12 5DR.

Abstract.

In this position paper we briefly review the role of the Benchmarking and Evaluation of Software Systems, with particular reference to two of the parallel computer system projects in which ICL has recently been involved: Flagship (1985-89) and EDS (1988-).

Benchmarking and Evaluation has an important role to play in the design of all computer systems, but this is particularly true of parallel architectures, whose performance characteristics are currently far less well understood than are those of conventional serial architectures. There are two reasons for this:

- Computer designers have accumulated 40 years of experience in the behaviour of serial computers, whose basic design has changed little during that time. Parallel computer designers have only a few years experience to draw on, and the range of architectures has been far more diverse, making it difficult to apply the experience gained from one parallel machine to the design of another.
- The exploitation of parallelism adds an extra dimension of complexity to the behaviour of a computer system.

For these reasons great stress has been placed on benchmarking and performance evaluation in the Flagship and EDS projects.

The Place of Benchmarking and Evaluation In the Development Process

A benchmark provides a method of exercising an interface in a computer system for one of 4 reasons [1]:

- **Performance Evaluation.** Benchmarking is one part of the wider activity of performance evaluation, which also includes measurement and tuning of existing systems, and the modelling, by simulation or analytical techniques, of systems yet to be implemented. Six different types of Performance Evaluation activity have been identified [2]:
 - **Simple Primitive Performance Tests** to measure the fastest achievable execution rate for a particular primitive operation in order to verify that the implementation satisfies the design requirement.
 - **Synthetic Primitive Performance Tests** to measure the variability of primitive performance with variation in the environment, to ensure that when system overheads are taken into account, the implementation satisfies the design requirement.

- **Application Development Benchmarks** to compare the performance of different systems when performing a particular type of work in order to validate that the system under development is likely to satisfy user requirements.
 - **Exemplar Demonstrations** to demonstrate the system in the best possible light, so that performance measures made are those on which the project wishes the system to be judged externally. An example of this is the use of nfib to measure function calls per second in a functional programming system.
 - **Synthetic Workload Benchmarks** to compare the performance of different systems when performing work which simulates a real customer workload.
 - **Real Workload Benchmarks** to compare the performance of different systems when performing a real customer workload.
- **Verification and Validation** of the system components and the integrated system at each stage of the development.
 - **The Demonstration of System Capabilities** at appropriate points in the project development.
 - **System Tuning** to achieve optimum performance of the integrated system. In order to assist this, both the Flagship and EDS systems include performance monitors which provide a graphical display of user selected measures, for example the processor utilisation of each PE in the system.

The Flagship Project

The Flagship project [3] [4], supported by the UK Alvey programme, designed and implemented a parallel computer system aimed mainly at the execution of declarative languages. The lack of widespread use of these languages by application writers created benchmarking and performance analysis problems as there were no application development or real workload benchmarks. This made it difficult to judge the performance of the system relative to both other declarative language implementations, and to conventional serial language implementations.

The project had to create its own application benchmarks mainly by recoding a selection of the Gabriel LISP benchmarks in HOPE+. However, these are still relatively small and do not represent commercial applications.

The EDS Project

The ESPRIT II European Declarative System (EDS) project has the aim of developing a parallel system to provide an evolutionary upgrade to existing (sequential) business/ commercial IT areas. The application to Relational Database Management Systems (RDBMS) has been chosen as the prime focus of the project because it is a key component of the IT business of the project participants.

In order to appear to customers as an evolutionary upgrade, it is a requirement that the parallel RDBMS must run existing database applications. This has been achieved by supporting the standard database languages, for example SQL. Therefore the most important benchmarks are those in the RDBMS area, and these will be used to compare the performance of the EDS RDBMS with other systems.

Because of the commercial importance of on-line transaction processing (OLTP) and database management systems (DBMS), 34 IT organisations set up the Transaction Processing Council

with the aim of defining standard, scalable OLTP/DBMS benchmarks. Their first output TPC "A" was a standardized low complexity transaction processing "Debit-Credit" benchmark. They are now in the final stages of standardizing TPC "B", a batch version of TPC "A". Work is currently proceeding on two new benchmarks, one to measure commercial On-Line Transaction Processing performance, the other for high complexity management information systems.

The selection of application and workload benchmarks for the EDS RDBMS is therefore made straightforward by the presence of these TPC benchmarks, which will be used to measure the system. One early experiment which gave confidence in the EDS design was to measure the performance of a version of *Debit-Credit* on a prototype RDBMS implementation on the Flagship system. Its performance could then be compared with that of several conventional serial systems, and this showed that a parallel system like Flagship or EDS could perform RDBMS computations with a better performance to cost ratio. This experiment is reported in reference [5].

Synthetic workload benchmarks, however carefully designed, cannot completely characterize all applications, and so within EDS there is a large activity in porting commercially important database applications onto the EDS system.

One example of an EDS application is the *Planes* Geographic Information System. This was originally written in Cobol with embedded calls to ICL IDMSX and TPMS to provide the user interface and database handling. The system is currently being modified to utilise the EDS RDBMS. It is considered to be a good example application because:

- The user databases are large, typically several Gigabytes.
- It is an existing, widely used commercial application.
- It needs more processing power, particularly to cope with a range of interactions from simple data retrieval to complex queries.

References

- [1] A.D. Kitto, *The Role of Benchmarking in the Development Environment*, ICL Flagship Project Internal Document FLAG/WP/8PE.900, (1987).
- [2] A.D. Kitto, *Performance Evaluation Strategy Definition*, ICL Flagship Project Internal Document FLAG/SD/8PE.001, (1987).
- [3] I. Watson, J. Sargeant, P. Watson and V. Woods. *The FLAGSHIP Parallel Machine*, CONPAR 88, BCS Workshop Series, CUP, pp. 125-133, (1988).
- [4] P. Townsend. *Flagship Hardware and Implementation*, ICL Technical Journal, 5(3), (1987).
- [5] S.J. Cockroft, M. Ward. *Performance Aspects of the EDS Parallel Processing Machine*, Proc. of UKCMG Conf., Glasgow (1990).

Performance Aspects of the EDS Parallel Processing Machine

S.J. Cockroft, M. Ward
5G Systems, ICL
Wenlock Way, Manchester. M12 5RA

3rd January 1990

Abstract

Conventional computer systems' architectures and programming techniques place inherent limitations on performance. Such systems cannot process ever-increasing amounts of data and information without reaching a 'saturation point', when demand exceeds system capability. This paper outlines the EDS (European Declarative System) parallel machine architecture that has been designed to support the needs of the database-related business application areas of the 1990s. Results are given for a database benchmark executed on a 15 node parallel machine (Flagship), and a method of obtaining and presenting performance data from the machine is shown.

CONTENTS

- 1 INTRODUCTION
 - 1.1 Background
 - 1.2 Objectives
 - 1.3 Machine Architecture: An Overview
- 2 APPLICATIONS AND PERFORMANCE
- 3 EDS ARCHITECTURE
 - 3.1 Target System
 - 3.2 Parallelism
 - 3.3 Diagnostics
- 4 DEBIT-CREDIT BENCHMARK
 - 4.1 Requirements
 - 4.2 Flagship Implementation
- 5 PERFORMANCE MEASUREMENT
 - 5.1 On-line Measurement
 - 5.2 Post-run Measurement
- 6 PERFORMANCE METRICS
- 7 CONCLUSIONS
- 8 BIBLIOGRAPHY
- 9 APPENDIX
 - A.1 EDS System Diagram
 - A.2 Diagnostics Interface Display
 - A.3 DebitCredit Runtime Display
 - A.4 Meters Runtime Display
 - A.5 'DebitCredit' Performance Metrics

1 INTRODUCTION

1.1 BACKGROUND

The Flagship¹ parallel processing machine has been built by ICL in conjunction with UK universities under an Alvey contract and has been used extensively to obtain performance information. This information has been used to design its successor, the EDS machine, which is being developed to respond to the needs of leading commercial users of Information Technology. ICL is the lead contractor in the EDS project (ESPRIT II EP2025), and is supported by partners in France (Bull), Germany (Siemens and ECRC), Italy, Spain, Portugal, Greece and the U.K.

1.2 OBJECTIVES

The current growth in advanced database-related business applications is approximately 30% per annum. If this trend continues then it is anticipated that in the mid 1990s, conventional (sequential) computer systems will have insufficient capability to process the vast amounts of data and information that this will involve.

The project is concerned with developing a form of parallel computing which will be appropriate to the information system needs of the partners' business customers through the 1990s. In order to meet this requirement the technology must be competitive with alternative approaches, must conform to prevailing standards and must be capable of being an integral part of contemporary systems. Parallel techniques are therefore being developed and applied within the fields of advanced database management systems and application systems for business professionals.

By the end of 1990, the first prototype EDS machine comprising 4 Processing Elements (PEs) will be available for experimental work. During 1991, an EDS machine containing 64 PEs will become available. The EDS project comes ends in 1992, at which time system integration will have been completed with applications demonstrated on the machine.

The EDS project is influenced by experience gained from the Flagship project. A prototype Flagship machine has been used to execute a transaction processing benchmark (the results of which are presented in appendix

¹An Alvey project resulting in a 15 element parallel machine [6] to support Hope⁺ [7].

A.5) at a rate of 43 tps (at 30% PE utilisation). The performance target for a 256 PE EDS machine is 12,000 tps (at 30% PE utilisation).

1.3 MACHINE ARCHITECTURE: AN OVERVIEW

The very nature of conventional systems prevents the exploitation of parallelism; they are not efficiently extensible, and the complexities and costs involved usually restrict such configurations to duals or quads.

There are several factors to be considered in parallel systems development: portability, extensibility, security, performance, cost/performance, availability and integrity. Of these, performance is the most important parameter and is usually the focus of interest.

The trend in microprocessor development is towards very high performance rates (33 - 50 MIPS). The trend in semiconductor store is towards increased volume with little improvement in access time. Store bandwidth is therefore rapidly becoming a system bottleneck, even with wide data buses, especially when multiple users share access to common data areas. We have therefore chosen to have distributed store architecture. This architecture is defined by the Process Control Language (PCL) [5], which is the common interface through which all subsystems utilise the parallel features of the machine. It also provides a multi-level context model with light-weight threads, efficient and reliable message passing, and mechanisms for exception handling, scheduling and load-balancing in a highly parallel system.

PCL is the main interface to the EDS Machine Executive (EMEX). By having a common PCL and EMEX which controls parallelism for the execution models of the various subsystems, some major benefits are achieved. Firstly, it simplifies communication and synchronisation between multiple language subsystems (i.e. subsystems implemented in different programming languages) within a single application. Secondly, it dictates overall machine control (scheduling resources as necessary between the separate subsystems, and in a multi-user environment, between individual instances of a subsystem), and thirdly, a single set of mechanisms allows opportunities for optimisation between the EMEX and the supporting hardware.

The EDS database subsystem, EDS language subsystems and a UNIX subsystem run on top of the PCL interface. The language subsystems being developed are parallel versions of Common Lisp, Prolog and C/C++. Lisp and Prolog have some implicit parallelism and, by building on the features

of PCL, further parallelism may be explicitly defined (in a declarative style), within the language subsystems. These two languages are used to build the advanced applications with C/C++ being used as the system implementation languages.

Within the chosen market-place, the relational database management system is a major consumer of processing power. SQL is the interface to the relational database management system (rdbms). The parallelism in the implementation of the rdbms is transparent to the application using SQL. Also, the size of the work involved in satisfying the SQL query is sufficiently large to make an SQL server architecture viable. The EDS machine is therefore a server for SQL which receives requests from the host system. Initially, the focus of the applications will be towards medium concurrency, medium complexity transactions.

2 APPLICATIONS AND PERFORMANCE

A set of industrial and exemplar applications have been chosen to provide a realistic test-bed to assess the implications of porting applications, application development and operational performance. The work is centred on database application areas and includes a Geographic Information System, a public network management system, a language translator and a behavioural logic simulator.

The typical unit of performance used in conventional systems is MIPS (Millions of Instructions Per Second), which is related to the speed of execution of the system instruction set. However, it is only partly related to performance, since a raw MIPS figure does not take into account the capability of the instruction set nor the system hardware. Benchmarks are more realistic performance indicators, but these can lead to yet more problems. For example, the resources required to execute the benchmark in full may not be available due to the prototype nature of the system under test. Also, if the benchmark is scaled down to fit prototype resources, the potential to exploit parallelism may be severely restricted. The benchmark used on Flagship is an adaptation of TPC BENCHMARK[™] A [2] (derived from [1]) called TPC-A [3] which involves debiting and crediting bank accounts in a transaction processing environment. A more detailed description of TPC BENCHMARK[™] A and TPC-A is presented in section 4.

3 EDS ARCHITECTURE

3.1 TARGET SYSTEM

The EDS machine is a multiprocessor with an initial design target of up to 256 Processing Elements (PEs) as shown in figure 1 (appendix A.1). Each PE uses an advanced commercially available RISC chip-set (Fujitsu H40) driving a local store (from 64 Mbytes to 2 Gbytes) using Mbus at 142 Mbytes/sec. The PEs communicate over a custom designed delta-network.

In the prototype machine there will be up to 64 PEs each having 64 Mbytes of storage representing 4096 Mbytes in total. Some of these elements are used to support diagnostic, host system and I/O connections. An entire data-base may be store resident (volatile) and distributed across the available local stores. The route to disc store is via the I/O element (which is also used to establish the data-base, perform updates, etc.).

3.2 PARALLELISM

Data and processing activities are distributed across the PEs which can cooperate with each other by sending messages across the delta network [8]. Whilst this may appear to be a 'bottleneck', the message protocols are very simple, requiring only a destination PE number to establish a route. Full availability (i.e. the ability for PE(x) to connect to an available PE(y) regardless of any other established connection) across the network would present an horrific interconnect problem for 256 PEs. A delta-network configuration is therefore used, which affords a practical compromise, enabling concurrent message passing when routes are not blocked.

Flagship was based on the 'fine to medium grain' approach to processing units of work. This resulted in significant overheads when scheduling such units, thus restricting performance. Although Flagship has disc store access, there is no backing store implementation which is a further restriction in the execution of benchmarks. The EDS machine architecture is based on a 'coarse grain' approach to processing, which embodies the performance and efficiency of conventional processing but retains the ability to distribute work dependent on the prevailing conditions.

There are still issues to be resolved concerning work distribution which require further development. On Flagship, work can be exported from busy PEs (when a certain activity level is reached) to the least busy PE (each PE propagating its activity level back through the network). This algorithm can be modified using a 'strength of feeling' (sof) parameter. Exporting work toward specific data held on a remote PE (static routing) requires maximum sof. A minimum sof results in work being exported to the least busy PE (dynamic routing). Intermediate values specify an activity threshold, below which static routing occurs and when reached, dynamic routing is employed [9].

With data-base applications, simple searches/queries can probably be directed toward specific PEs (static routing), however, more complex queries present other difficulties. These queries may include searches, inferences, further searches, etc., resulting in highly dynamic forms of parallelism. This will inevitably result in computations being physically separated from the required data structures. Some work in this area is being carried out at Manchester University [4].

3.3 DIAGNOSTICS

The DE (Diagnostic Element, Figure 1, appendix A.1) provides facilities for a variety of activities, from commissioning hardware and kernel, to machine support. In the early versions of the EDS machine, all input/output will be via the DE. The facilities provided will include bootstrap, error reporting and handling, basic engineers' facilities and simple input/output. Each PE has local diagnostic facilities to support the element. Communication between the DE and PEs is via the Network and an RS232 Diagnostic and Control Interface (DCI). The Network is the main communications path, with the DCI being used for the low level functions such as reset and bootstrap.

The single engineer's console provides access to all the diagnostic facilities of the machine. Access for multiple users is provided via the host connection, which has only a subset of the diagnostic facilities for security reasons.

The diagnostics also support machine simulation. The user interface to the real machine or a machine simulator is very similar and is depicted for Flagship in Figure 2 (appendix A.2). Input is a combination of keystrokes (bottom window) and function-select 'push-button' icons. Responses are displayed in the middle window and system status information in the top

window.

The more difficult aspects of diagnostics are in the application domain at the execution model level. The longer an application executes before errors are manifested, the more complex the diagnostics becomes due to the dynamic nature of the processing. In this situation, the expertise of the application writer is required. Diagnostic facilities require 'break-point' setting to enable the user to specify conditions to stop the processing and halt the machine. The strategy used on Flagship is to stop at a point prior to an error condition, then single step through function execution with the machine displaying monitored information. At each step the user is able to access machine state information. This approach will, however, require some refinement for EDS, due to the coarser function granularity.

4 DEBIT-CREDIT BENCHMARK

TPC BENCHMARK™ A is a benchmark agreed by major concerns in the computing industry (i.e. ICL, Bull, IBM, DEC, RTI, Oracle etc.). At the time experiments were being carried out on Flagship the benchmark was essentially as follows, but has since been superceeded.

4.1 REQUIREMENTS

These are essentially as follows.

A single transaction is as follows :-

- Begin transaction
 - Read 100 bytes from the terminal
 - Update account X:
 - * set new balance = balance + delta
 - * test new balance is greater or equal to zero
 - * write new balance to account X
 - Write to history file
 - * account id, teller id, branch id, delta, time stamp
 - Update teller T
 - * set new balance = balance + delta
 - Update branch B
 - * set new balance = balance + delta
 - Write 200 bytes to the terminal
- Commit transaction

The Database requirements are as follows.

- Account records : 10,000,000
- Teller records : 10,000
- Branch records : 1,000
- History records : 2,590,000 of transactions

Response time requirements are that 95% of transactions completed must have a response time of less than one second.

At least 15% of transactions should be carried out at a different branch from where the account resides.

4.2 FLAGSHIP IMPLEMENTATION

There are essentially two problems concerning the implementation of TPC BENCHMARK[™] A. The first is the prototype nature of the Flagship machine and the storage size for the above database. If account records are typically 100 bytes, then this alone represents 1000 Mbytes of storage. We can scale down the store size and maintain the same Account/Teller/Branch ratios correspondingly. The second problem is that to achieve a high level of concurrency and parallelism, we need a large number of branch records to avoid contention and deadlock situations, i.e. access contention for the same branch record.

The EDS architecture does not place any restrictions on applications but in this instance, greater concurrency for experimentation purposes can be achieved by increasing the proportion of teller and branch records by at least an order.

The implementation for Flagship (called DebitCredit) is as follows :-

- Account records : 200,000
- Teller records : 3,000
- Branch records : 3,000

The history records are omitted because of store limitations and the absence of a backing store implementation. However, the process of creating such a record would be relatively trivial requiring the creation of a local pointer reference to the transaction data which is appended to a list of such references.

Most of the transactions are remote from the data, the requirement being at least 15% of such transactions. Flagship performance can thus be improved since directing more transactions toward the data would reduce the amount of network traffic per transaction. Also, Flagship store management and packet formats have restricted the maximum width of B-trees to 80 entries whereas extra width would reduce the depth of the tree data and the corresponding access time.

5 PERFORMANCE MEASUREMENT

Performance measurement is a function of the Diagnostic Element (DE) using information periodically received from each PE. The following tools exist to extract the performance dynamics of applications.

5.1 ON-LINE MEASUREMENT

Figure 3 (Appendix A.3) depicts the Flagship DebitCredit tool. This is invoked prior to execution of DebitCredit. It shows total number of transactions so far and the current average transactions per second. The various icons are displayed dynamically for demonstration purposes to give the observer an impression of the DebitCredit transaction activity.

Figure 4 (Appendix A.4) depicts the Flagship meters tool. This is invoked via the DE and is configurable to include monitoring of the various units within each Processing Element. These include store usage, work load (units of work still to do), work done so far (number of rewrites), network traffic, function call trace (user defined), PE idle time (no work to do) and locality (the proportion of store accesses local to the PE). Part of the activity within each PE is to maintain statistical information which is periodically transmitted to the DE (the period is user defined). Expertise in the application domain is required to interpret the graphs against application activity. The user will analyse work distribution, network utilisation, store utilisation etc, with a view to improving overall application performance when behaviour is not as anticipated.

Execution can be suspended at any point to enable monitoring as described in the following section.

5.2 POST-RUN MEASUREMENT

The DE can extract information from each PE whilst an application is executing. The information can then be displayed when convenient using the meters tool, as if the application were currently executing.

There are also statistical display facilities used for the analysis of performance attributes.

6 PERFORMANCE METRICS

DebitCredit metrics extracted from the 15 Processing Element Flagship machine are presented in Appendix A.5. They show response times and transactions per second obtained with varying degrees of concurrency (number of concurrent transactions) up to the maximum obtained with 15 Processing Elements.

Any work exported by a Processing Element is sent to the least busy Processing Element. In this example, this is not the best strategy, hence some optimisation is required.

7 CONCLUSIONS

The DebitCredit results obtained with the Flagship machine of 43 tps (at 30% PE utilisation), have encouraging implications for the EDS project. The inclusion of history records will obviously degrade this performance, however, the results were obtained under 'worst-case' conditions and the following optimisations are possible.

- Most transactions were carried out remotely from where the data was actually held resulting in large amounts of network traffic to both read and then update data.
- The distribution of global access data can be improved to restrict data searches, although this would involve some duplication of information.
- 'Intelligent' search algorithms can be used, since these are currently sequential.
- The granularity is too fine, resulting in the unnecessary exporting of small units of work away from the associated data, again increasing network traffic.

Customised hardware to assist with PE communication will give further performance benefits over the firmware based model of Flagship.

The performance target for a 256 PE EDS machine of 12,000 tps (at 30% PE utilisation) is a consequence of the high performance of each individual PE. The architecture does not place any restrictions on applications but some will derive more benefit than others when executed on an EDS machine. For example, an application such as TPC BENCHMARKtm A will derive less benefit than the class of applications used to process vast amounts of data and information which have far more potential to exploit the parallel resources of the machine.

8 BIBLIOGRAPHY

- [1] Tom Sawyer & Omri Serlin
DebitCredit Benchmark - Minimum Requirements and Compliance List
Codd & Date Consulting Group, 22/Jun/88
- [2] TPC BENCHMARK™ A
Draft 6-PR Proposed Standard, Transaction Performance Council
21/Aug/89 (latest version is 10/Nov/89)
- [3] S.M. Kellet, ICL
Definition Of The EDS TPC-A Benchmark
Flagship project internal document: EDS.DD.11I.0004, 08/Nov/89
- [4] I. Watson, Manchester University
Simulation of a Physical EDS Machine Architecture
EDS project internal document: EDS.WP.3I.iw01, 13/Nov/89
- [5] P. Istavrinos, Siemens
Specification of the Process Control Language (PCL)
EDS project internal document: EDS.DD.1S.0007, 11/Dec/89
- [6] P. Townsend, ICL
Flagship Hardware and Implementation
ICL Technical Journal, May/87
- [7] N. Perry, Imperial College, London
Hope⁺
Flagship project internal document: IC/FPR/LANG/2.5.1/7, Feb/88
- [8] T. Hall, ICL
Flagship Network Functional Overview
Flagship project internal document: FLAG/DD/4NE014, Jan/89
- [9] I. Watson, V. Woods, P. Watson, R. Banach
M. Greenberg, J. Sargeant, Manchester University
A parallel Architecture for Declarative Programming
Proc 15th Annual Int. Symp. on Computer Arch., Honolulu, Hawaii,
May/89

9 APPENDIX

A.1 EDS SYSTEM DIAGRAM

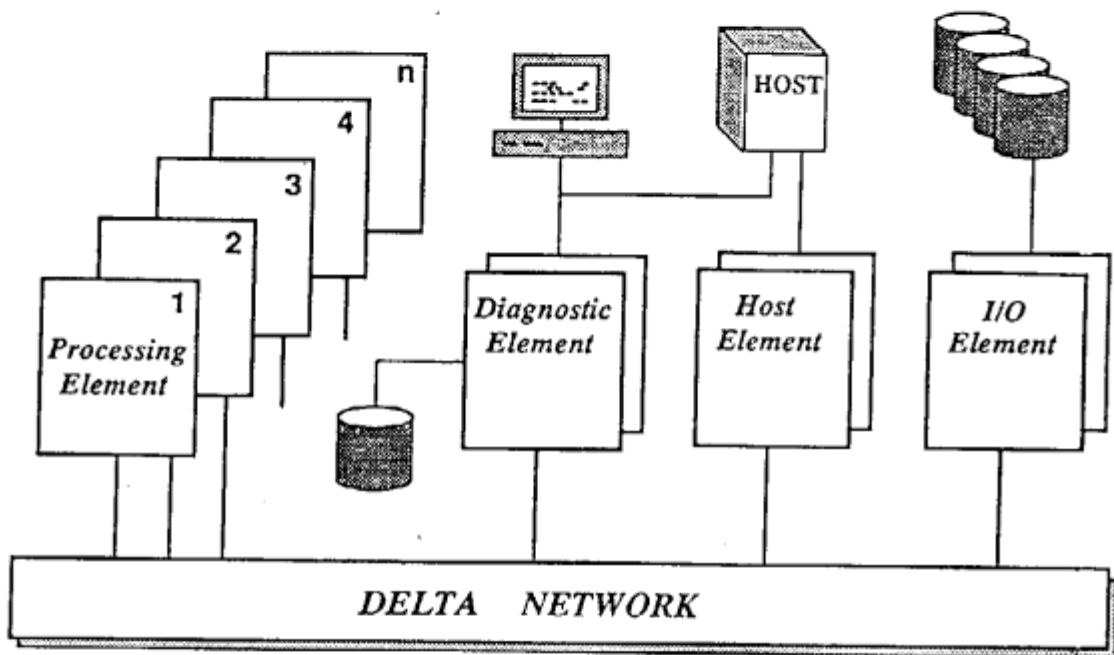


Figure 1: EDS System Diagram

A.2 DIAGNOSTICS INTERFACE DISPLAY

```

FLAGSHIP GDM 2.19 09/11/01 CPR. ICL 1987 model.r9b5mon
Node      0      1      2      3      4      5      6      7
232      msg      msg          conn
S/V       +      +      -      on      -      -      -      -
Stat      halt     halt
Max       cont     cont
Step      0      0
STO       6      6
EPQ       0      0
HS        0      0
NET       0      0
REVR      0      0
Tot ReVr:0 , Current PID:FFFFFFFF
Mon Dec 11 09:18:32 1989
Open complete steve on sc75 as user steve for Application using
rig em2.
Resetting the Emulator, please wait.
Sending resets
Resetting serial lines
Soft reset complete
gfeffc00
gfeffc00
Loading net loader
.Loading via Net, model.r9b5mon
Services Node 84980 bytes loaded.
IPL complete, Initialising PE's.
[ wait stop start again help abort log u inp quit ]
em> open rig1
em> ipl model.r9b5mon
em> init
em> [

```

Figure 2: Diagnostics Interface Display

A.3 DEBITCREDIT RUNTIME DISPLAY

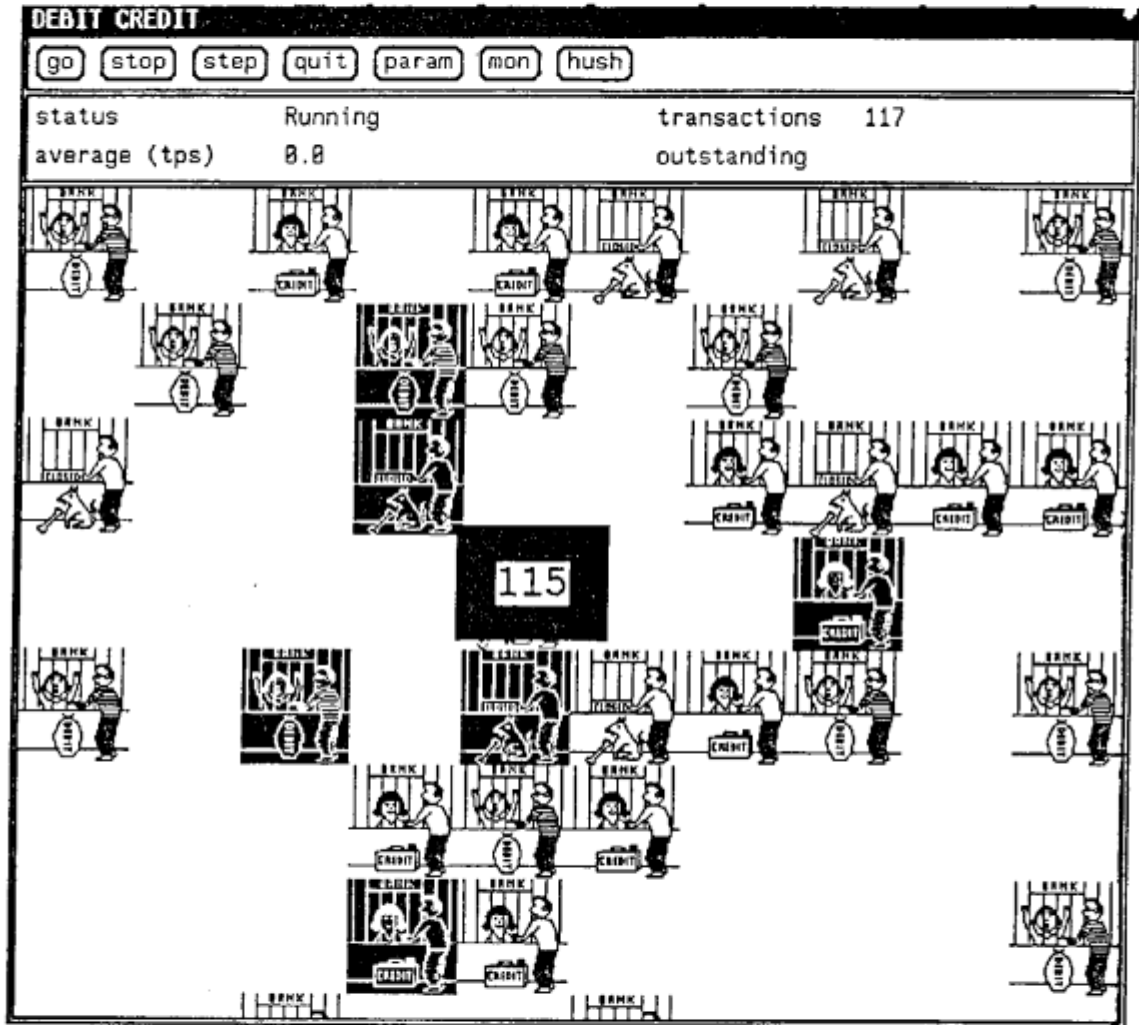


Figure 3: DebitCredit Runtime Display

A.4 METERS RUNTIME DISPLAY

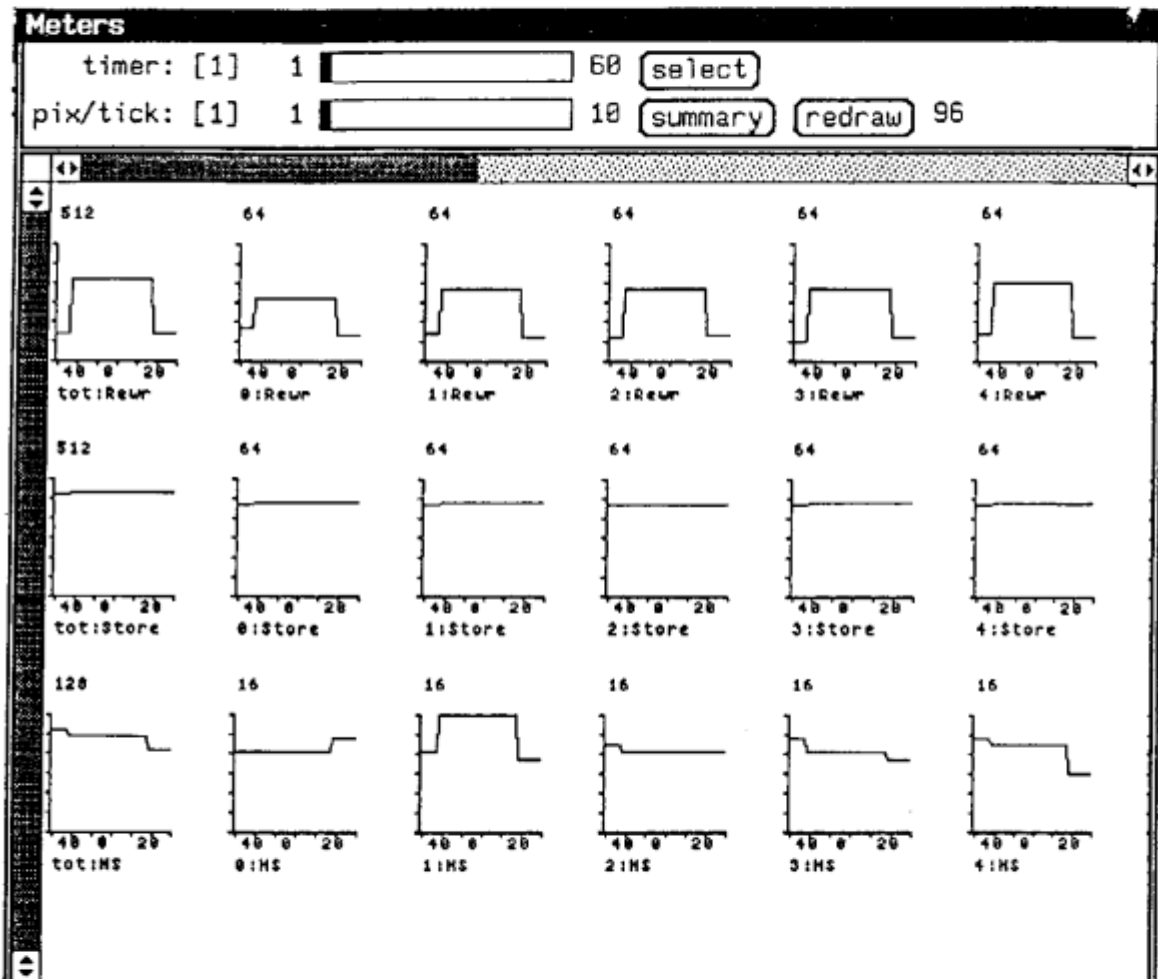


Figure 4: Meters Runtime Display

A.5 'DEBITCREDIT' PERFORMANCE METRICS

Steady state is achieved after 1 minute approx.

Average run time is 3 minutes approx.

The last table has accounts set to 3000 to compare with results obtained using Flagship processing elements having minimum store configuration.

Changing the number of tellers and branches does not affect performance as much as increasing the number of accounts. The latter increases the depth of the B-trees accessed during each transaction.

Accounts	Tellers	Branches	Transactions	Concurrency
200,000	3,000	3,000	19,030	31
Average t.p.s. = 85.5				
Transaction Response Times				
Time range		Transactions	Percentage	
0 - 199ms		4593	24.14	
200 - 399ms		9653	50.73	
400 - 599ms		2762	14.51	
600 - 799ms		864	4.54	
800 - 999ms		388	2.04	
1000 - 1199ms		152	0.80	
1200 - 1399ms		146	0.77	
1400 - 1599ms		106	0.56	
1600 - 1799ms		101	0.53	
1800 - 1999ms		84	0.44	
2000 and more		181	0.95	
Average = 85.5 ms		Maximum = 4400 ms		Percent < 1 sec = 95.95

Accounts	Tellers	Branches	Transactions	Concurrency
200,000	3,000	3,000	19,679	50
Average t.p.s. = 96				
Transaction Response Times				
Time range		Transactions	Percentage	
0 - 199ms		3857	19.60	
200 - 399ms		8677	44.09	
400 - 599ms		3116	15.83	
600 - 799ms		1108	5.63	
800 - 999ms		501	2.55	
1000 - 1199ms		292	1.48	
1200 - 1399ms		306	1.55	
1400 - 1599ms		238	1.21	
1600 - 1799ms		237	1.20	
1800 - 1999ms		89	0.45	
2000 and more		1258	6.39	
Average = 527 ms		Maximum = 4580 ms		Percent < 1 sec = 87.70

Accounts	Tellers	Branches	Transactions	Concurrency
200,000	3,000	3,000	19,478	101
Average t.p.s. = 113.0				
Transaction Response Times				
Time range		Transactions	Percentage	
0 - 199ms		2576	13.23	
200 - 399ms		6739	34.60	
400 - 599ms		2630	13.50	
600 - 799ms		1111	5.70	
800 - 999ms		454	2.33	
1000 - 1199ms		260	1.33	
1200 - 1399ms		221	1.13	
1400 - 1599ms		417	2.14	
1600 - 1799ms		623	3.20	
1800 - 1999ms		799	4.10	
2000 and more		3648	18.73	
Average = 904 ms		Maximum = 6300 ms		Percent < 1 sec = 69.36

Accounts	Tellers	Branches	Transactions	Concurrency
200,000	3,000	3,000	19,870	150
Average t.p.s. = 130.5				
Transaction Response Times				
Time range		Transactions	Percentage	
0 - 199ms		1335	6.72	
200 - 399ms		5062	25.48	
400 - 599ms		2458	12.37	
600 - 799ms		1201	6.04	
800 - 999ms		891	4.48	
1000 - 1199ms		706	3.55	
1200 - 1399ms		708	3.56	
1400 - 1599ms		745	3.75	
1600 - 1799ms		941	4.74	
1800 - 1999ms		971	4.89	
2000 and more		4852	24.42	
Average = 1161 ms		Maximum = 5900 ms		Percent < 1 sec = 55.09

Accounts	Tellers	Branches	Transactions	Concurrency
200,000	3,000	3,000	29,191	251
Average t.p.s. = 138.0				
Transaction Response Times				
Time range		Transactions	Percentage	
0 - 199ms		212	0.73	
200 - 399ms		3391	11.62	
400 - 599ms		3145	10.77	
600 - 799ms		2037	6.98	
800 - 999ms		1340	4.59	
1000 - 1199ms		1169	4.00	
1200 - 1399ms		1141	3.91	
1400 - 1599ms		1251	4.29	
1600 - 1799ms		1354	4.64	
1800 - 1999ms		1342	4.60	
2000 and more		12809	43.88	
Average = 1836 ms		Maximum = 9480 ms	Percent < 1 sec = 34.69	

Accounts	Tellers	Branches	Transactions	Concurrency
* 3,000	3,000	3,000	21,564	150
Average t.p.s. = 144.5				
Transaction Response Times				
Time range		Transactions	Percentage	
0 - 199ms		2801	12.99	
200 - 399ms		5206	24.14	
400 - 599ms		2141	9.93	
600 - 799ms		1098	5.09	
800 - 999ms		939	4.35	
1000 - 1199ms		805	3.73	
1200 - 1399ms		837	3.88	
1400 - 1599ms		920	4.27	
1600 - 1799ms		1163	5.39	
1800 - 1999ms		1193	5.53	
2000 and more		4461	20.69	
Average = 1069 ms		Maximum = 5560 ms	Percent < 1 sec = 56.51	

Flagship 1985 – 1988

UK Alvey Programme
Built in conjunction with Manchester University
Architecture by Manchester University
Developed and built by ICL
Motorola 68020 based
Operating system written in Hope+
Based upon Graph Reduction
Computational model written in C
Test programs and exemplars written in Hope + or C

Benchmarking and Evaluation

ICL

ICL 1988/89 Benchmarking and Evaluation Page 11

EDS 1989 –

Esprit Programme
Parallel architecture ICL, Siemens and Bull
Evolutionary route for current applications
Parallel machine. ICL, Siemens
Parallel Operating system based upon Chorus. ICL, Siemens
Parallel relational database. ICL, Bull
LISP. Siemens
Prolog (Elipsys) ECRC

Benchmarking and Evaluation

ICL

ICL 1988/89 Benchmarking and Evaluation Page 12

Benchmarking and Evaluation in the Development Process

Benchmarks are used in 4 ways:-

1. Performance Evaluation:
enables comparison between systems
2. Verification and Validation:
proves design is to specification
3. Demonstration of System capabilities:
good for visibility
4. System Tuning:
Used to optimise performance

Performance Evaluation – 1

1. Performance Evaluation
 - a. Simple primitive performance test.
e.g. measure operating system time to task switch
 - b. Synthetic Primitive performance test
e.g. Garbage collection. An artificial environment is provided in which to best measure garbage collection with minimal interference from system features not under investigation.
 - c. Application development benchmarks
e.g. matrix multiplication
 - d. Exemplar demonstrations
e.g. Text retrieval from the works of Shakespeare
 - e. Synthetic workload benchmarks
Here work simulating a real customer workload is performed.

- f. **Real workload benchmarks**
Here a real customer workload is run for comparison with other systems.

Performance Evaluation – 2

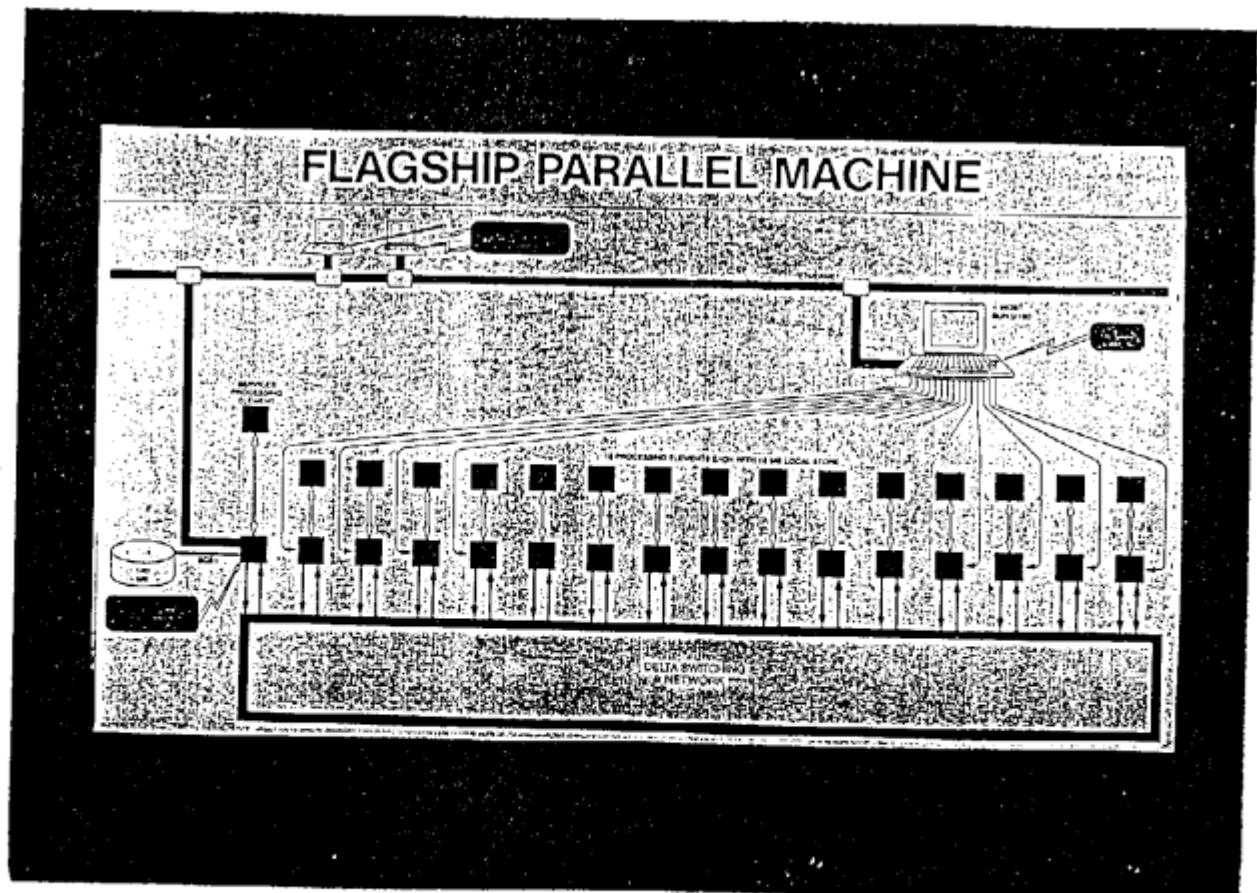
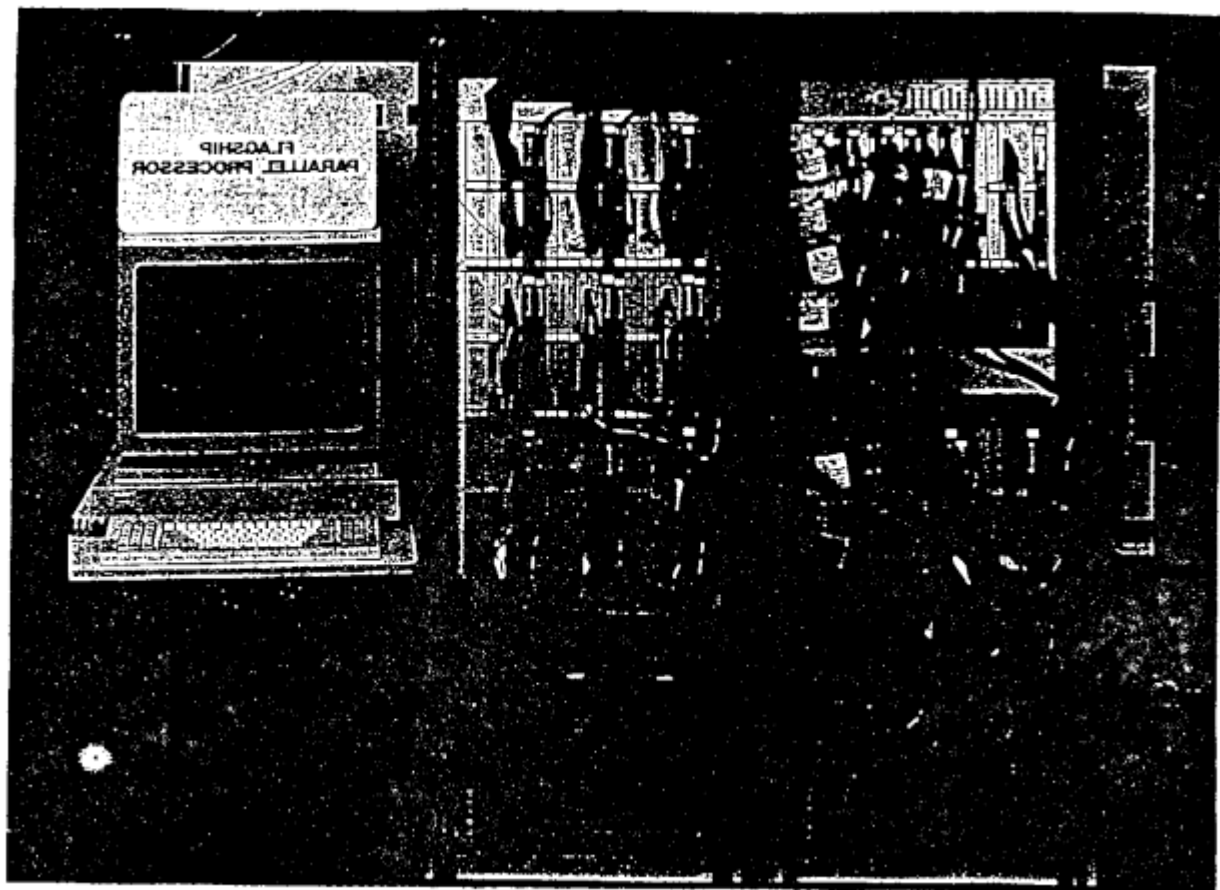
2. **Verification and Validation**
Of the system at each stage of development. Benchmark should be included in the validation suite.
3. **Demonstration of System capabilities**
at appropriate points in the project development. Presentation interface important
4. **System Tuning**
To achieve optimum performance of the integrated system. This requires the system to provide performance monitors.

Software Benchmarking for the Flagship and EDS projects

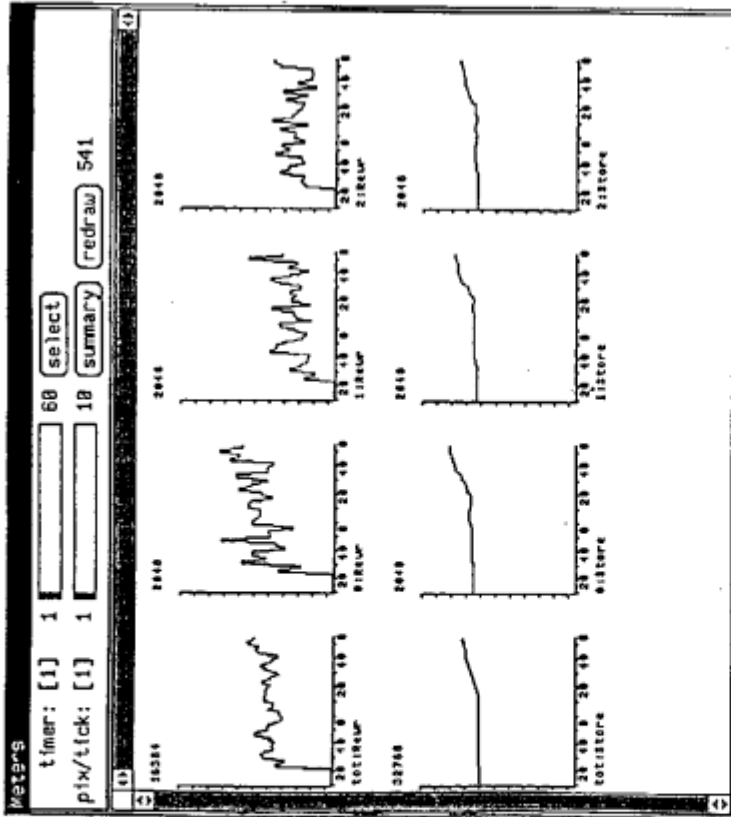
FLAGSHIP

Initially simple performance test, exemplar
demonstrations

e.g. Nfib, Nqueens, Triangle, Boyer



Meters Run-time Display



Customer impact

NFIB, NQueens, Triangle etc alright for comparison within research community

BUT!

What does this mean to our customers?

Benchmarks need to be relevant to applications

Database is the centre of most of our commercial systems

Mainframes cannot provide sufficient RDBMS performance

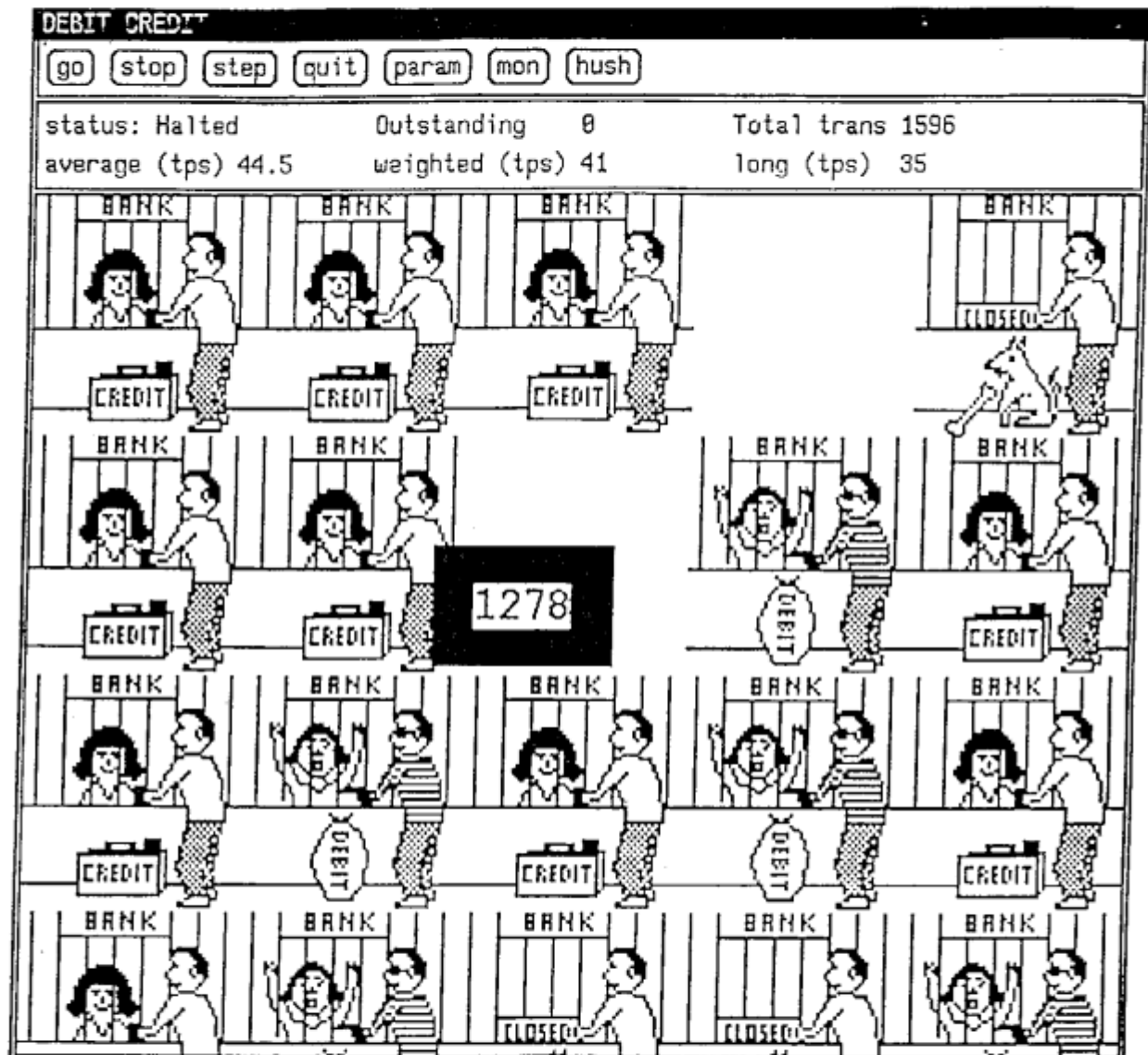
RDBMS technology extensions require more performance

ESQL hides parallelism from the application

Exploit our parallel expertise on Database technology

Debit/Credit an international benchmark

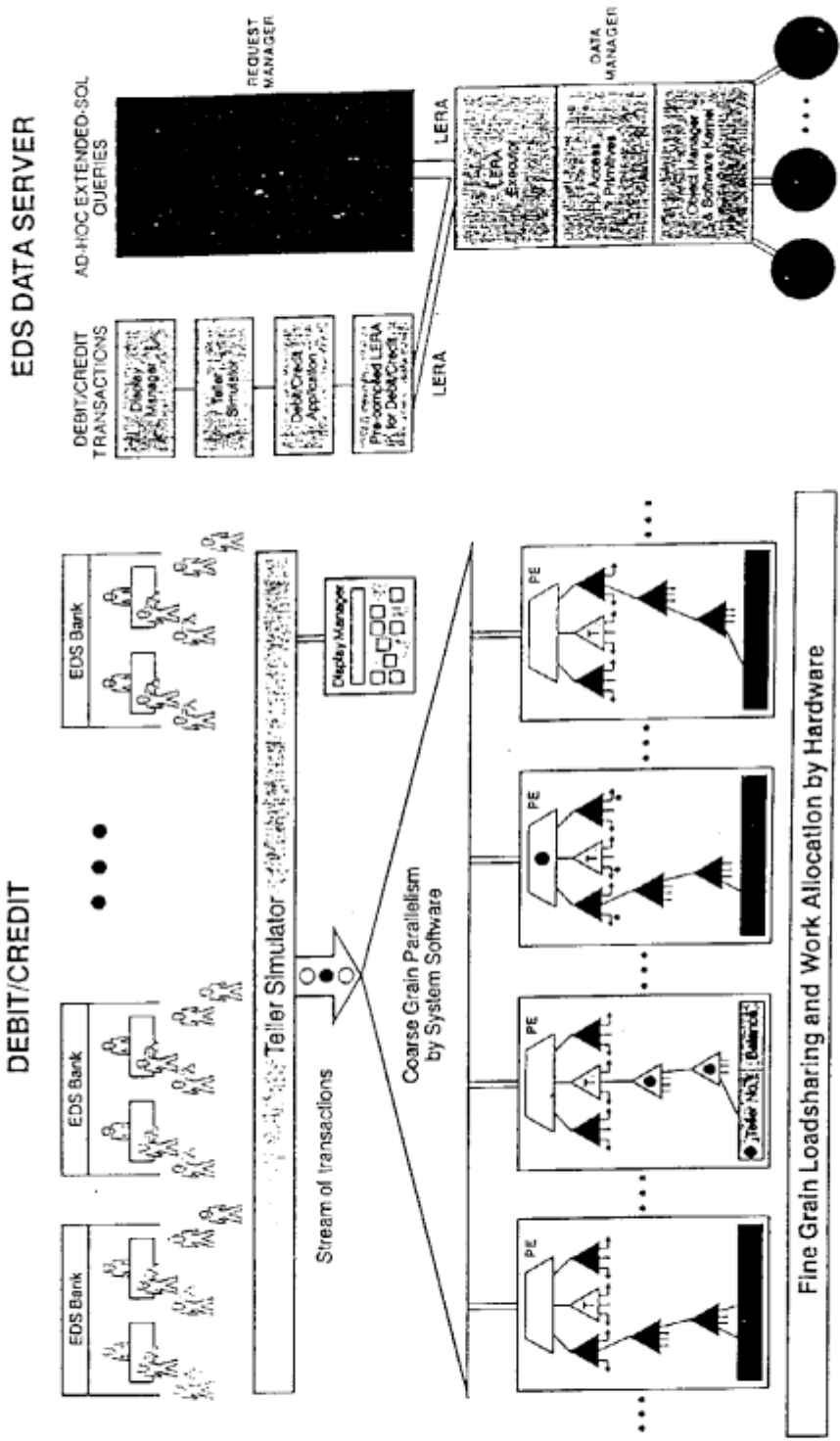
Debit-Credit Run-time Display



Benchmarking and Evaluation

ICL

101090pt : slides \ p(10.90) \ page 1

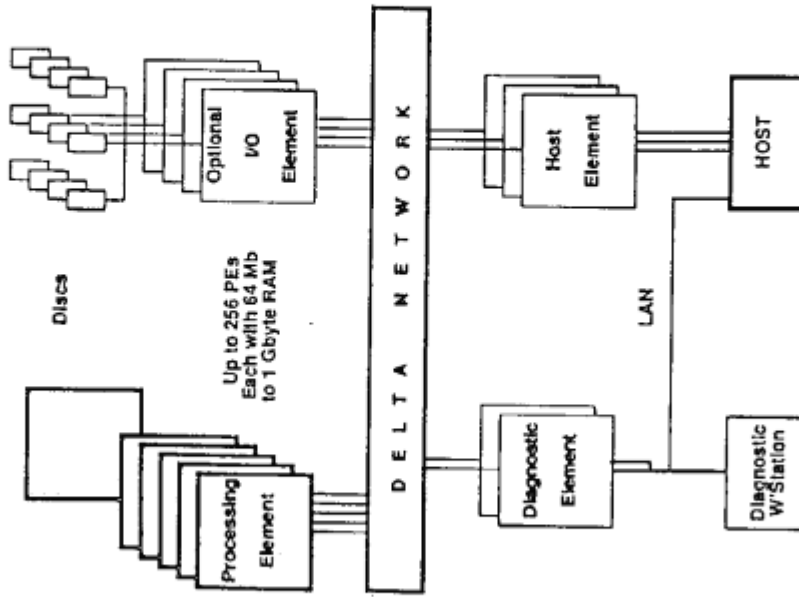


EDS PARALLEL DATABASE MANAGEMENT SYSTEM

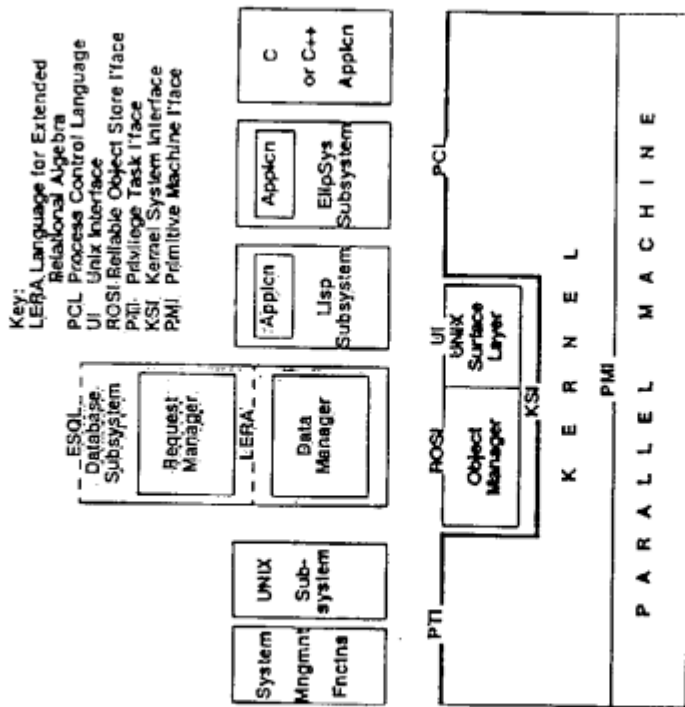
The EDS Approach

- Main Focus: Relational Database
- High level application programming interfaces
 - Database: Extended SQL
 - Decision support applications
 - Lisp
 - Prolog + constraint handling
- System programming
 - C++/C
- Application environments
 - UNIX
 - Proprietary host OS
- Distributed store multiprocessor
 - Up to 256 processing elements
 - Message passing
- Technology
 - High performance RISC
 - Large RAM store: up to 1 GBytes/PE
 - Optional Local discs

EDS Machine Architecture



EDS INTERNAL ARCHITECTURE



EDS Benchmarking - 1

1. Performance Evaluation Benchmarks:
 - a. Simple primitive performance test:

Database: Object Creation, Taking Locks, Tree Walk.

Kernel: Thread creation/Termination, TaskCreation/Termination, raw message passing performance, exception handling.
 - b. Synthetic Primitive Performance test:

Many simple threads sending and receiving at messages, to benchmark message passing at the system level.

EDS Benchmarking – 2

c. Application Development Benchmarks:

PEM (Portfolio Club Experimental model).

Simple database used to test each facility.

e.g. relational operators such as join, filter.

EDS Benchmarking – 3

d. Exemplar Demonstrations

PSIM

Parallel version of a sequential behavioural simulator.

Simulator is used to simulate mainframe computer designs.

The exemplar demonstrates both the feasibility and performance of the design.

EDS Benchmarking – 4

e. Synthetic Workload Benchmark:

TPC (Transaction Processing Performance

Council)

34 members, including ICL

Various standards and emerging benchmarks
Benchmarks clearly define the conditions for
running them.

TPC-A

TPC-B draft3, May90

TPC-Teradata-Dss, draft1, May90

TPC-Order-Entry

TPC-A and TPC-B are variants of debit/credit

EDS Benchmarking – 5

e. Synthetic Workload benchmark (continued):

TPC- Order-Entry Benchmark Specification.
(DEC)
(mid-weight read write transaction)

Wholesale supplier

Represents commercial applications in the
1990s.

Workload: managing orders

Can be naturally distributed without structural
changes to transactions.

ICL

Benchmarking and Evaluation

07/000001 / volume 1 / part 08.001 / Page 11

ICL

Benchmarking and Evaluation

07/000001 / volume 1 / part 08.001 / Page 11

EDS Benchmarking – 5

- e. Synthetic Workload (continued)
 - TPC- Teradata-DSS
(heavyweight complex queries)
 - Suite of commonplace decision-support queries
 - Taken from actual customer applications.
 - Downsized to the low-gigabyte range:-
 - Broadens their industry-wide usefulness
 - Eases of implementations.
 - Incorporate retail sales, insurance, banking,
pharmaceuticals, and communications
industries
 - Gathered together under one common physical
database.

Benchmarking and Evaluation

ICL

EDS Benchmarking – 6

- f. Real workload Benchmark
 - PLANES; Application
 - A current ICL application with many customers.
 - Geographic Information system.
 - A public utilities application
 - Provides information on Property , streets,
electrical supply etc.
 - Parallel machine compared with current ICL
mainframes and workstations.

Benchmarking and Evaluation

ICL

EDS Benchmarking

2. **Verification and Validation**
Here we will include many of the above benchmarks in the testsuite particularly TPC-B
3. **Demonstration of system capabilities.**
Here again we are using many of the above benchmarks as milestones in our development. They are included in our planned demonstrations to managers and reviewers.
4. **System Tuning**
Each Benchmark specified can be used to tune the aspect of the system for which it is designed.

Conclusions

- **Classify and know the purpose of each Benchmark**
- **Plan benchmarks early in the project**
- **Use international standards or help to evolve them**
- **Select benchmarks relevant to the planned application domain**
- **Build a Benchmark suite for revalidation after system changes**

Research on Parallel Inference Systems in the Fifth Generation Computer Systems Project

Takashi Chikayama
Institute for New Generation Computer Technology

Abstract

This paper describes the outline and the current status of the parallel inference system research and development in the fifth generation computer systems (FGCS) project of Japan, mainly focusing on its basic software part.

The fifth generation computer systems project is aiming at establishing the basic technology required for knowledge information processing systems. The parallel inference system is a part of the project to establish parallel processing hardware technology for providing massive processing power and basic software technology for effectively utilizing such hardware for knowledge information processing systems.

The nature of software cannot but change drastically when the underlying computation mechanism changes from sequential to parallel. From this viewpoint, the approach conventionally taken to establish parallel software technology has problems in trying to continuously enhance the technology for sequential processing. Although such continuity has its own merits, a completely different approach should be taken for massively parallel computer systems available in near future.

We analyze the problems of the conventional approach and present how such problems has been (and are planned to be) solved in the parallel inference systems of the FGCS project. The current status of the research and development is also reported.

1 Introduction

The fifth generation computer systems project is a national project of Japan, aiming at establishing the basic technology required for realization of knowledge information processing systems. The following two are most important to achieve the final objective of the project.

- Problem solving methods for intelligent processing
- Processing power for implementation of the above methods

The parallel inference system, as a part of the project, is aiming at establishing both hardware and software technologies for the latter.

Recent evolution of the hardware technology shows around 4 times increase in implementation density every 3 years. Extrapolating the recent density increase, moderately estimating it as 10 times in 6 years, 100 processors can co-reside in one chip in 2008. As the design cost is getting more and more crucial in total cost, the repeatability in such one-chip multi-processor will have great cost advantage to a complicated processor system occupying one whole chip or more, even if the both systems had the same performance. Thus, in early 21st century, multi-processor systems will be advantageous, not only in absolute processing power, but also in cost effectiveness even in small systems such as palm-top computers.

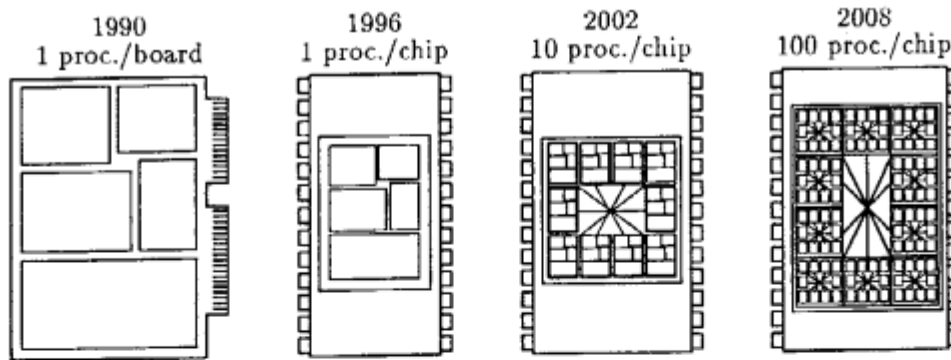


Figure 1: Expected Evolution of Parallel Processing Hardware Technology

The software technology for parallel processing, however, still remains premature. Especially, technology for building parallel software to solve complicated problems in the area of knowledge processing is far from satisfactory. This, we think, is at least partly due to the problems in the approach to parallel processing software technology conventionally taken, that is, trying to augment already available sequential processing technology. We thus propose an approach to establish software technology totally redesigned for parallel processing, including algorithms, programming languages and operating systems.

2 Problems of Conventional Approach

With the conventional approach, already available sequential technology is expected to be more smoothly converted to parallel technology. This approach might really be appropriate for small scale parallel processing systems, but it is not adequate for highly parallel systems.

Problems in trying to naively convert sequential technology for highly parallel processing are described below.

2.1 Algorithm

Most of the sequential algorithms depend too much on globally accessible memory with constant time access overhead. In the highly parallel environment, it is impossible to provide memory globally accessible in constant-time with reasonably small constant. Such algorithms cannot be easily tailored to a parallel version. The best algorithms for sequential systems are often not even decent algorithms for parallel systems, especially, for highly

Table 1: Two Approaches to Parallel Software Technology

Approach	Conventional	Required
Technology	extension of sequential technology	novel parallel technology
Execution concurrent sequential	when specified default	default when specified
Resultant system	artificial & awkward	natural
Technological continuity	better	worse

parallel systems.

The algorithms for parallel systems should be designed from scratch. The design must always take parallelism and, often more importantly, communication locality into account.

2.2 Language

In the conventional approach, programming languages originally designed for sequential processing are augmented with several additional features for use in parallel systems. Concurrent execution is explicitly specified by certain additional language constructs or, often, such features are not a part of the language but are merely provided as library routines. Such explicit specification often makes programs awkward and, more importantly, makes it more difficult for later reorganization for better load distribution and lower communication overhead.

Another problem in using such languages is in frequent bugs in synchronization as it is also explicitly specified. It is error-prone human beings who are responsible for proper use of such features.

Programming languages for parallel systems should be designed from scratch so that it suits best for concurrent execution. The language constructs should imply concurrent execution in principle and sequentiality should be specified only when needed. Synchronization should also be implicitly embedded in the language construct.

2.3 Operating System

In the conventional approach, the operating systems designed for sequential systems are augmented with certain primitives for parallel execution, leaving its basic design left unchanged. This was possible because, even for sequential systems, the operating system is designed to run processes concurrently.

There are two major problems, however, with this approach. One is that the interface of the operating system with the user programs is still based on sequencing, such as notifying the completion of requested operations by the completion of an operation, typically a supervisor call. In parallel systems where application software also has internal currency, this often causes synchronization problems. Another problem is that the management policy of the sequential operating systems is highly optimized for sequential processing, such as centralizing all the required management information, which is far from optimal for highly parallel systems. If the problem were confined to the internals, the user interface could have been preserved. Unfortunately, the interface specification is strongly influenced by the management algorithm. Expressing processes by an integer called *process number* is a typical example.

Operating systems for parallel systems should be designed for parallel systems from scratch. The user interface should also be designed anew to be consistent with the design of the concurrent programming language; Sequentiality should not be a part of the design of the interface. Distribution of management, as far as possible, is required to avoid bottlenecks. Such consideration will also affect the design of user interface.

3 Design Issues of Parallel Inference Systems

Based on the comparison of two approaches to establish parallel software technology, an effort to reconstruct all the required technology is taken in the design of parallel inference systems in the FGCS project. This section describes several characteristic design issues of the parallel inference system project.

3.1 Algorithm

When designing practical parallel algorithms, we should not forget that we have only finite number of processors. Assume that an algorithm has sequential computational complexity $c(n)$ and average parallelism $p(n)$, where n being the size of the problem. The total execution time $t(n)$ by this algorithm, excluding communication overhead and with ideal distribution, might be roughly given as $t(n) = c(n)/p(n)$. This $t(n)$ provides a criterion to compare various algorithms. With this criterion, an algorithm with higher complexity but with still higher parallelism is considered to be a better algorithm.

This, however, is only valid when $p(n)$ is smaller than the physically available parallelism p_p . When $p(n)$ becomes significantly larger than p_p , the criterion becomes $t(n) = c(n)/p_p$. With limited physical parallelism, algorithms with more rapidly increasing $c(n)$ can never be a better algorithm for large n regardless of $p(n)$.

This leads to a conclusion that a sequential algorithm can beat any parallel algorithms, if the latter has slight increase in total computational complexity. Thus, when designing a parallel algorithm, we must often consider a hybrid strategy, using a parallel algorithm in higher levels but switching to a sequential algorithm when the physically available parallelism is used up.

3.2 Language: KL1

We have designed a concurrent logic programming language named KL1 as the kernel language of the system [2]. The KL1 language is based on the flat version of GHC [7] with various extensions. As the GHC language is a concurrent language in its nature with implicit synchronization and without side-effects, it gives an ideal basis for designing a language for the parallel processing system.

The principal extensions to Flat GHC made in KL1 are as follows.

Meta-level Control: The meta-level control features are mandatory for describing an operating system in KL1. It is also beneficial for sophisticated program control for application software. The features provided by KL1 include, the “shoen” mechanism which controls a group of processes as a whole, the priority mechanism for controlling speculative computation.

Efficient Execution: The fundamental source of inefficiency in most “pure” languages is that they do not allow destructive assignment, excluding the possibility of fully utilizing the merit of random access memory. KL1 provides arrays with constant time update without disturbing its pure semantics[1]. This means that any imperative algorithms can be expressed in KL1 retaining the same computational complexity.

3.3 Operating System: PIMOS

A parallel inference machine operating system called PIMOS [2] is developed based on the policy described above. PIMOS is written in KL1 language as a collection of many dynamically created processes communicating one another via streams [5]. The interface between PIMOS and user programs is also streams.

The unit of resource management is called task, which is implemented using the shoen mechanism of KL1 described above. Tasks can nest arbitrarily many levels, forming a tree structure distributed to multiple processor, avoiding the management bottleneck problem.

Table 2: Available and Planned Parallel Inference Systems

	System	Hardware	# of proc.	Peak speed
1987	PDSS	conventional	1	~10 KRPS
1988	PIMOS/S	PSI-II	1	~150 KRPS
1988	PIMOS/M	Multi-PSI	64	~10 MRPS
1991	PIMOS/P	PIM	~512	>100 MRPS

RPS: reductions per second

4 Current R&D Status and Plans

Immediately after the experimental model of the parallel inference machine Multi-PSI[6], KL1 language implementation [4] and PIMOS running on it became available, several experimental application software projects started, some of which will be reported in other presentations in this workshop. Some of the programs resulted in almost linear speed-up, and some has yet to achieve good speed-up.

Several sets of the Multi-PSI systems with the newest version of the KL1 language implementation and PIMOS are currently in use at ICOT and several other related research institutes. The KL1 implementation on Multi-PSI with its maximum configuration of 64 processors has the peak performance of about 10 million reductions per second.

Newer versions of the parallel inference hardware PIM [3] is currently under development. A version with maximum configuration is expected to have up to 512 processors and more than ten times the performance of the Multi-PSI system.

5 Conclusion

In the research and development of the parallel inference system in the Japanese FGCS project, we took the approach to reconstruct the whole parallel processing software technology from scratch, rather than gradually modifying conventional sequential technology. Experimental versions of hardware, language implementation and operating system have been made available to application software research and several application projects are now going on.

From our research and development experience, we can say that describing concurrent software for parallel hardware is not difficult when an appropriate programming language is used. Almost no synchronization problems are found during even during the development of the operating system.

On the other hand, making programs run efficiently on parallel hardware is another thing. No universally applicable efficient load distribution method is found so far. We have to accumulate much more experience with various application software to establish the basis of practical parallel processing software technology.

References

- [1] T. Chikayama and Y. Kimura. Multiple reference management in flat GHC. In *Proceedings of 4th International Conference on Logic Programming*, 1987.
- [2] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, Tokyo, Japan, 1988.

- [3] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the parallel inference machine architecture (PIM). In *Proceedings of FGCS'88*. Tokyo, Japan, 1988.
- [4] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.
- [5] E. Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. ICOT Technical Report TR-004, ICOT, 1983. Also in *New Generation Computing*, Springer-Verlag Vol.1 No.1,1983.
- [6] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama and K. Taki. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *Proceedings of FGCS'88*, 1988.
- [7] K. Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. ICOT Technical Report TR-208, ICOT, 1986.

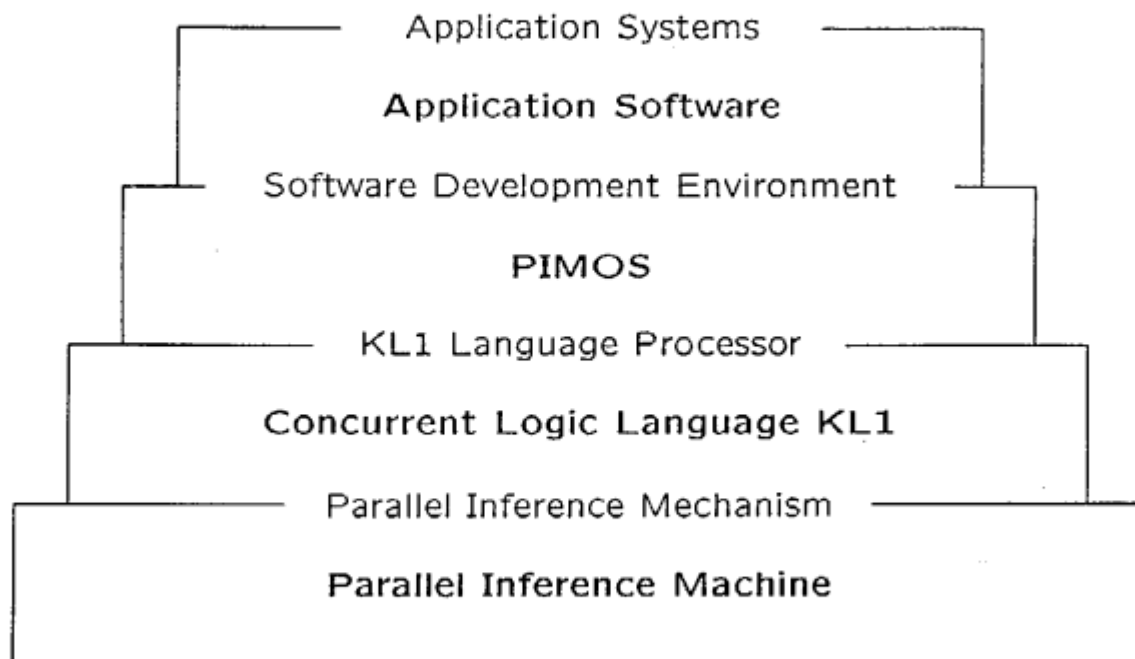
Research on Parallel Inference Systems in the FGCS Project

- What we are aiming at
- Why a new language or an operating system
- What we have achieved so far
- What R&D plan we have

Takashi Chikayama

Institute for New Generation Computer Technology

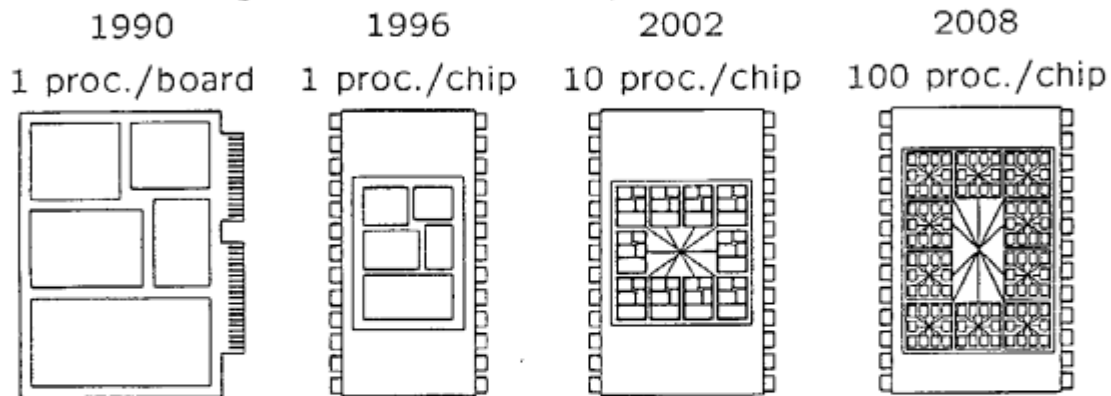
Structure of the Parallel Inference System



Evolution of Parallel Processing Technology

Hardware Technology: 4 times more density every 3 years

— estimating it as 10 times in 6 years...



→ Parallel processing will become advantageous, not only in absolute processing power, but also in cost effectiveness

Software Technology: Parallel processing technology for complicated software is quite premature

Two Approaches to Parallel Software Technology

Approach	Conventional	Required
Technology	Extension of sequential tech.	Novel parallel technology
Algorithm	Seq., modified	Newly designed
Language	Seq., modified	Newly designed
Concurrency	When specified	Default
Sequentiality	Default	When specified
Resultant system	Artificial & Awkward	Natural
Technological continuity	Better	Worse

Comparing Two Approaches (1) — Language

Approach	Conventional	Required
Language Synchronization	Sequential, modified Explicit	Born concurrent Implicit
Program structure		
Sync. bug	Frequent	Almost Never

Concurrent Logic Language KL1: its base language, GHC

Head :- *Guard* | *-Body*.
synchronization & condition execution

- **Born to be a concurrent language**
→ Basic features are designed for concurrent execution
- **Data flow language**
Conditioning and synchronization are indivisible
→ Can never make erroneous decision due to sync. failure
No side-effects
→ Can never lose data by overwriting
- **Processes as tail-recursive goals**
External devices as processes running unknown programs

Concurrent Logic Language KL1: Extensions to GHC

Priority: Parallel processing requires two forms of ordering

- Strict ordering forced by data dependency → Data-flow
- Preferred ordering for efficiency → Priority
 “In any order or in parallel, but preferably...”

Meta-level: Monitoring and controlling computation

- Mandatory for an operating system
- Beneficial for describing sophisticated problem solving strategy

Concurrent Logic Language KL1: Efficient Implementation

Efficiency drawbacks of “pure” languages

- Cannot “update” structure elements
 → Increased computational “complexity”
- Cannot use the same memory area by overwriting
 → Larger working set size

Optimized implementation

to be competitive with procedural languages

- New memory management scheme (MRB):
 → Efficient incremental garbage collection
 → Arrays with constant-time element updating

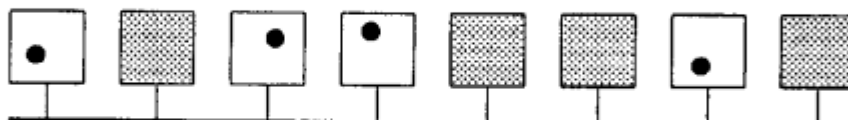
Software Development Environment for KL1

- **Synchronization problems:** → Inreproducible anomalies
 - Much less frequent ← Pure concurrent language
- **Deadlock detection:**
 - Automatic detection ← Data-flow language
 - Detection (in part) by static analysis (under development)
- **Tracing parallel processes:**
 - Selective tracing based on control flow
 - Selective tracing based on data flow (planned)
- **Performance debugging:**
 - Visualization of program behavior (under development)

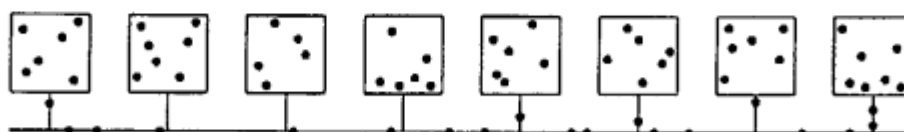
Comparing Two Approaches (2) — Algorithm

Good balance of load and reduction of communication are hard to realize at a time.

- Not enough distribution → Many idling processors



- Too much distribution → Too much communication



Comparing Two Approaches (2) — Algorithm (continued)

- For higher parallelism
→ Divide the problem into small parts and distribute them
- For lower communication rate
→ Do not distribute computation requiring the same data

Approach	Conventional	Required
Algorithm	Seq., modified	Newly designed
Trade-off between Parallelism & Locality	Often conflict each other	Can be made optimal

**A good sequential algorithm is often
not even a decent parallel algorithm**



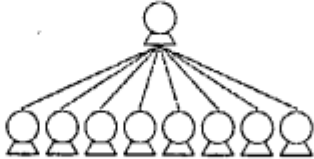
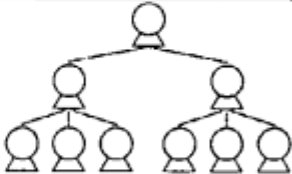
Algorithm: Automatic Load Distribution?

- Additional burden to algorithm designers
→ Automation of load distribution is desirable
- Appropriate distribution principle depends heavily on problems
→ General purpose automatic distribution is very difficult

⇒ **Load distribution libraries:**

Collection of load distribution algorithms found to be effective
through experiences with various application software

Comparing Two Approaches (3) — Operating System

	Conventional	Required
Management	Centralized	Distributed
Example	Task table	Tree of tasks
Small scale systems		
Large scale systems		
Parallelism in management	Small → bottleneck	Large
Communication	Frequent	Less frequent

Operating System: Problems of Conventional Schemes

Optimized for sequential processing

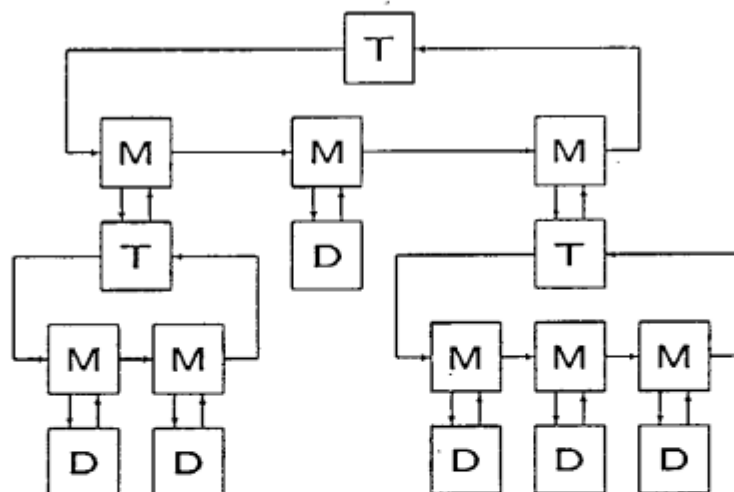
- Basic notions based on sequential processing
 - Large-grain processes as the unit of management
 - Processes with large inertia
 - Controlling dependency by execution order
 - Limitation on parallelism
- Operating system itself optimized for sequential processing
 - Centralized management
 - More communication
 - OS can be the bottleneck

Operating System: Redesigning for Parallel Processing

PIMOS employs schemes optimized for parallel execution

- Basic notions suited to parallel execution
 - Group of fine-grain processes as the management unit
 - Processes can be light-weighted
 - Data flow dependency
 - Better parallelism
- Operating system itself optimized for parallel processing
 - Hierarchical management
 - Distributed processing avoiding bottleneck
 - Reduced amount of communication

PIMOS: Hierarchical Resource Management



T: Task management process
M: Resource monitoring process
D: Device handler process

Parallel Inference Systems of FGCS

	System	Hardware	# of proc.	Peak speed
1987	PDSS	Conventional	1	~10 KRPS
1988	PIMOS/S	PSI-II	1	~130 KRPS
1988	PIMOS/M	Multi-PSI	64	~ 8 MRPS
1991	PIMOS/P	PIM	~512	>100 MRPS

RPS: reductions per second

From Our R&D Experience...

- Describing **concurrent** systems is **easy** in KL1
Almost no synchronization bugs
- Describing **parallel** systems is **difficult** even in KL1
No universally applicable load distribution method
→ Trial and errors with various problems
- **“Patching”** won't work with concurrent systems
If designed properly, programs can be clean
If designed poorly, programs will never work
- Social systems provide good models of parallel processing
⇔ Natural systems are optimized for higher parallelism

KL1 Programming Environment

— PIMOS —

Hiroshi Yashiro†, Koichi Nakao‡, Ryoza Kiyohara†,
Kumiko Wada† and Takashi Chikayama†

† Institute for New Generation Computer Technology
‡ Applied Technology Co., Ltd.

Abstract

In the Japanese fifth generation computer systems project, the parallel inference machines PIMs and the operating system PIMOS are being developed to provide the computational power required for high performance knowledge information systems. They are designed to run a concurrent logic programming language KL1 which is based on a flat version of GHC.

This paper gives an overview of the KL1 programming environment provided by PIMOS.

1 Introduction

In the Japanese fifth generation computer systems project, the parallel inference machines, PIMs[1], are being developed to provide the computational power required for high performance knowledge information systems.

Several models of PIMs are currently under development. The execution speed of the largest systems is expected to be between 100 MRPS(reductions per second) to 1 GRPS for practical applications.

As an experimental version of the parallel inference machine, Multi-PSI[2] has been developed to promote software research and development for parallel inference systems. One Multi-PSI system consists of up to 64 processing elements with separate local memory, connected by a two-dimensional mesh network.

In total, one Multi-PSI system has about 5 GB of memory and up to 10 MRPS of processing speed for simple programs. Currently, about 10 systems are being used in development of the operating system and experimental parallel application software.

The Multi-PSI and PIMs are designed to run a concurrent logic programming language KL1 efficiently. The KL1 language is based on a flat version of GHC[4] and has extensions that allow a meta-level execution control mechanism (the *Shōen* feature) and a priority specification mechanism[3]. The language is planned to be used in both system and application software of the machines.

The parallel inference machine operating system, PIMOS[3], is designed to control highly parallel programs efficiently on PIMs and to provide a comfortable software development environment for the KL1 language. A preliminary version of PIMOS has been developed and is currently in use on the experimental machine, Multi-PSI.

2 PIMOS and KL1

2.1 The KL1 Language

The KL1 language is a concurrent logic programming language based on a flat version of GHC which allows only unification and built-in predicates in the guard part of a clause. This eliminates the need for nested binding environments and management of tree structured control information, which makes efficient implementation considerably easier. However, the GHC language itself does not have enough power for efficient implementation of operating systems or application programs that require sophisticated control mechanism. Thus several extensions are made to the language, mainly for enabling meta-level execution control mechanism (the *Shōen* feature).

2.2 Communication Mechanism

In a parallel environment, it is not simple to manage various resources, such as CPU time or input/output devices. For example, the status of particular set of data can not be changed at a time when the operating system processes the data in parallel with user programs.

In such a case, conventional operating systems on sequential computers simply suspend the execution of the user program while the operating system processes the data. However, in a parallel computing environment where PIMOS runs, such a control of stopping all the user programs cannot be implemented without losing the advantages of parallel computing.

To solve this problem, we have introduced a programming style in which a process is considered as an object[6]. The process has an internal data and variables for interprocess communication. We call such variables *streams*. When a user wants to read or write the data internal to the operating system, that data can be accessed only through the streams. PIMOS employs this communication style with user programs.

In the KL1 language, interprocess communication where a message is sent and received through the streams can be implemented by means of instantiation of a shared variable. For example, in a case of reading a character string from keyboard, the KL1 program for PIMOS and the user can be written as follows.

```
?- pimos(Req), user(Req).

user(Req) :-
    true |
        Req = [getb(N,String)|ReqT],
        .....

pimos([getb(N,String)|ReqT):-
    true |
        readFromKbd(N,KBDString),
        KBDString=String,
        pimos(ReqT).
```

Figure 1: Interprocess communication between user and PIMOS

The user and PIMOS can communicate using a shared variable *Req* as a stream. First, the user sends a message *getb/2* that requests for reading *N* characters. When the PIMOS receives the *getb/2* message, it reads *N* characters from the keyboard to *KBDString* (*readFromKBD/2*). Then, the user receives the *String* instantiated to the *KBDString*. In the succeeding request, *ReqT* is used as shared variable for communication.

We call this programming style such as *pimos/1* “*process*”. PIMOS allows the user to access resources that PIMOS manages, such as keyboard, file and so on, in this way through streams.

2.3 The Shōen

Elements of KL1 programs in a system, which are called goals, form one logical conjunction. Therefore, the whole system may fail because of a failure in a user program. This problem comes from the fact that the PIMOS runs in the same level as the user programs. We provide the meta-control mechanism to simplify the management of the user programs in the KL1 language. We call this mechanism *Shōen*.

Shōen is a grouping construct for KL1 goals and a meta-logical unit to control and monitor the KL1 goals in it. It has a pair of streams, named *control stream* and *report stream*. The control stream is used to *start*, *stop* or *abort* the goals from outside the *shōen*. *Termination* of execution or events that occurred inside the *shōen*, such as a failure, deadlocks, or an exception, are reported on the report stream from inside the *shōen*. The *shōen* feature is similar to the meta-call feature of Parlog[5], and can be considered to be a language-embedded version of the meta-interpreters seen in systems based on Concurrent Prolog[7]. (Figure 2) *Shōen* can be nested by arbitrary levels.

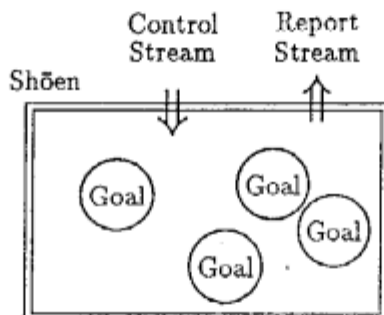


Figure 2: Shōen

PIMOS supervises user programs using the *shōen* facility. The exception reporting mechanism is also used for establishing the communications path from the user to PIMOS. Monitoring the report stream, PIMOS can recognize the users' request for a communication stream.

2.4 Hierarchy of Resource Management

KL1 provides only the control mechanism of the computational behavior. In the computer system, we also have other kinds of resources to manage, such as input/output devices. PIMOS provides hierarchical management for such resources using nested *shōens*.

PIMOS provides the notion of *task* as resource management unit. The task is a *shōen* with a corresponding supervisor process inside PIMOS, which controls the utilization of resources within the programs running in the task. PIMOS currently provides functions to access resources, such as files and display windows, provided by the operating system (SIMPOS) on the front-end-processor (PSI-II). A file system for disk drives directly connected to PIM is also under development.

3 Programming Environment

In a programming system, we generally need the following.

- Language processor (e.g. compiler, interpreter)
- Debugging system (e.g. debugger, tracer)
- Analyzer (e.g. profiler, system monitor)

PIMOS provides all of these functions on a physically parallel computer Multi-PSI. This section describes each of these functions.

3.1 Shell

The shell is the top level interface of the KL1 program execution. The shell handles a chunk of user programs so that they can be controlled. We call it a *job*. The shell provides a communication method called *pipe* between tasks belonging to a job. (Figure 3)

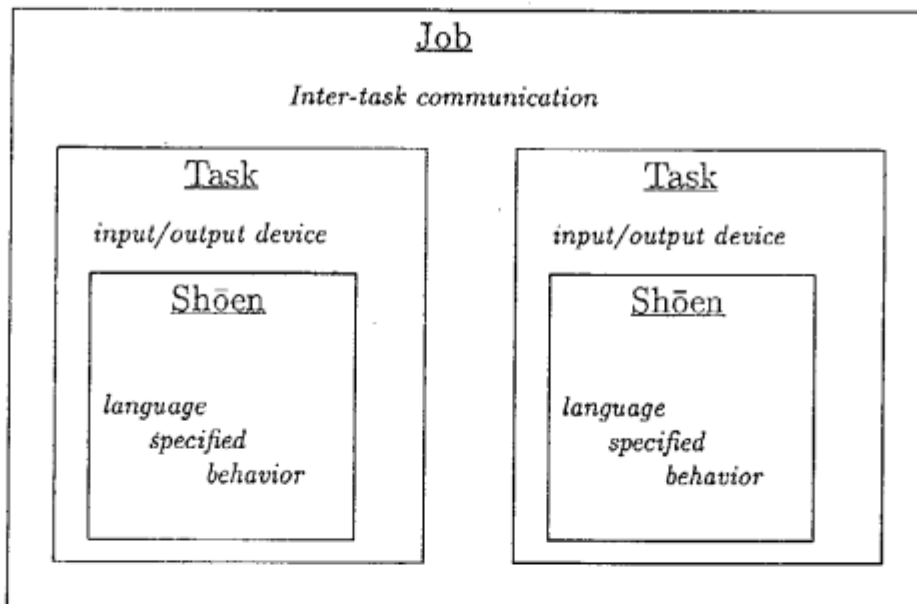


Figure 3: Hierarchy of Resource Management

3.2 KL1 Compiler

PIMOS provides the resident KL1 compiler on Multi-PSI. KL1 programs are compiled into abstract machine instructions, which are executed by the microcode. The object code modules are registered to the program database of PIMOS.

3.3 KL1 Listener

The KL1 listener is also a top level interface of the KL1 program execution, similar to the top level of interactive Prolog systems. The functions provided are as follows focusing on the debugging functions.

Goal Execution: The user can execute goals consisting of user-defined predicates and built-in predicates from the listener top level.

Tracing Function: The KL1 stepping tracer described below can be called from the listener.

Execution Profiler: The listener can collect profiling information as described below.

Detection of Perpetual Suspensions: The listener reports the perpetual suspensions (suspensions that will be never resolved, such as deadlock) that are detected during the garbage collection[8]. The detected goal is the goal that is the "cause" of the causality relationship of perpetual suspension.

Inspecting and Monitoring Functions: The inspector and the variable monitor (see below) can be called from the listener top level or from the tracer.

3.3.1 Tracing Mechanism

In ordinary execution of KL1 programs, when a goal is picked up from the goal pool and reduced to its children, the resultant goals are put back into the goal pool. On the other hand, when a traced goal is reduced, the resultant goals are not put back to the goal pool immediately; instead, information of the reduction is reported to the tracer. The reported information includes subgoals and the identifier of the traced parent goal. (Figure 4)

This identifier is unique corresponding to each traced goal. Keeping the correspondence of the identifier and the goal (predicate and arguments), the tracer can report which goal is reduced.

The tracer presents the parent goal, the subgoals and their arguments to the user. The user can specify for each subgoals whether to trace or not. The goals specified to be traced are given a new identifier and marked for tracing, and all subgoals are put back into the goal pool again. As the goals not specified to be traced are executed ordinarily, the overhead of tracing is minimized.

3.4 Inspector

The inspector is a tool to debug the KL1 program; it analyzes data structures and displays them interactively. Any KL1 data item can be inspected. With the help of the inspector, the user can analyze the contents of complicated data structures and analyze the state of the arguments of the goal during tracing. (Figure 5)

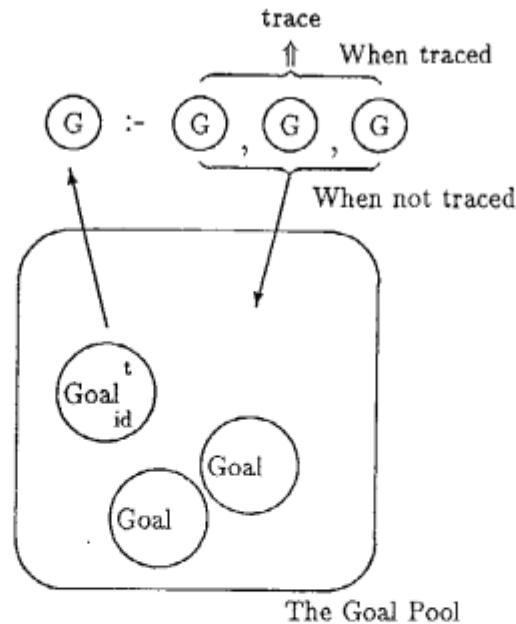


Figure 4: KL1 Goal Execution

3.5 Variable Monitor

For inspecting the future value of a currently uninstantiated variable, a data-driven monitoring process can be initiated from the inspector or the tracer. The value of the monitored variable is reported on its instantiation. It is also possible to incrementally monitor the instantiation of a list structure, which is useful for tracing communication streams.

3.6 Performance Analyzer

PIMOS has two kinds of performance analyzer. Each of these programs displays the program behavior by means of graphic interface.

Performance Meter: The performance meter displays work rate of each processors dynamically. It is useful for gross performance debugging.

Shōen Profiler: In the listener, more detailed information of dynamic analysis for performance can be obtained. It provides numbers of reductions and suspensions of goals in a certain shōen in a certain period of time on a certain processor. In addition, PIMOS provides the graphic interface of this information.

4 Conclusion and Future Plans

The programming environment provided by the PIMOS for KL1 language programs are described. These facilities are currently used on the Multi-PSI systems for development of experimental programs and PIMOS itself.

```

@004096 29   layered: filter([ & , & |E1],1,1,W)
41   * (1)   filter([ & , & |E1],1,1,W)? inspect 1 ← inspecting subgoal 1
filter([3*[' & '],4* V1 |T1],1,1,W)> me           ← list elements
0 : filter
1 : [3*[' & '],4* V1 |T1]
2 : 1
3 : 1
4 : W
filter([3*[' & '],4* V1 |T1],1,1,W)> 1           ← go down elements 1
0 : 3*[ & |Q1]
1 : 4* V1
tail : T1

```

Figure 5: Example of Inspector

Design and implementation of more sophisticated software development facilities are planned. Currently planned extensions include the following features.

Static analyze function: In KL1, the execution of a program can be suspended unexpectedly because of some mistakes in the programs. Currently, there is a simple analyzer such as variable checker, which examines the number of appearances of each variable in one clause, but it is not enough. A static analyzing method, which examines arguments relation of input/output in all clauses, is proposed[9].

Processor Profiler: We are developing the performance debugging facility which examines the behavior of each processor, for example the numbers of communication packets with other processors. It will provide another viewpoint for understanding the behavior of program execution.

References

- [1] A. Goto, M. Sato, K. Nakajima, K. Taki and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.208-229, ICOT, Tokyo, 1988.
- [2] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, T. Chikayama. Distributed Implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, pp.436-451, 1989.
- [3] T. Chikayama, H. Sato and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.230-251, ICOT, Tokyo, 1988.
- [4] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.
- [5] I. Foster. Parlog as a Systems Programming Language. *Ph. D. Thesis*, Imperial College, London, 1988.

- [6] E. Shapiro and A. Takeuchi. Object Oriented Programming in Concurrent Prolog. In *New Generation Computing*, Vol.1, No.1, pp.25-48, 1983.
- [7] E. Shapiro. Systems Programming in Concurrent Prolog. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, 1984.
- [8] Y. Inamura and S. Onishi. A Detection Algorithm of Perpetual Suspension in KLI. In *Proceedings of 7th International Conference on Logic Programming*, 1990.
- [9] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *Proceedings of 7th International Conference on Logic Programming*, 1990.

KL1 Programming Environment

— PIMOS —

Hiroshi Yashiro*, Koichi Nakao**, Ryoza Kiyohara*,
Kumiko Wada* and Takashi Chikayama*

*:Insitute for New Generation Computer Technology

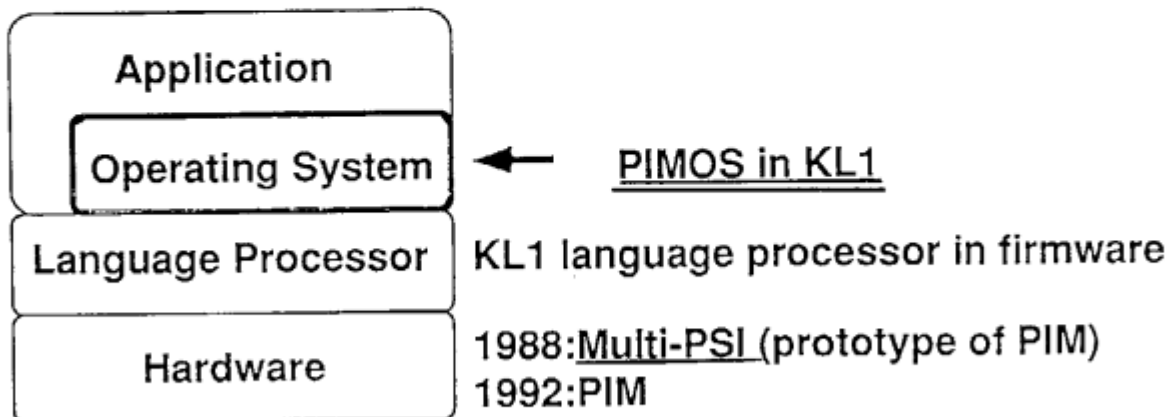
** : Applied Technology

Contents

1. Introduction
2. KL1 and PIMOS
 - 2.1 The KL1 Language
 - 2.2 Resource Management
 - 2.3 Hierarchical Resource Management
3. Programming Environment
 - 3.1 Compiler
 - 3.2 Debugger
 - 3.3 Analyzer
4. Conclusions

Fifth Generation Computer Systems Project (1982-1992)

→ R&D of Parallel Computer System

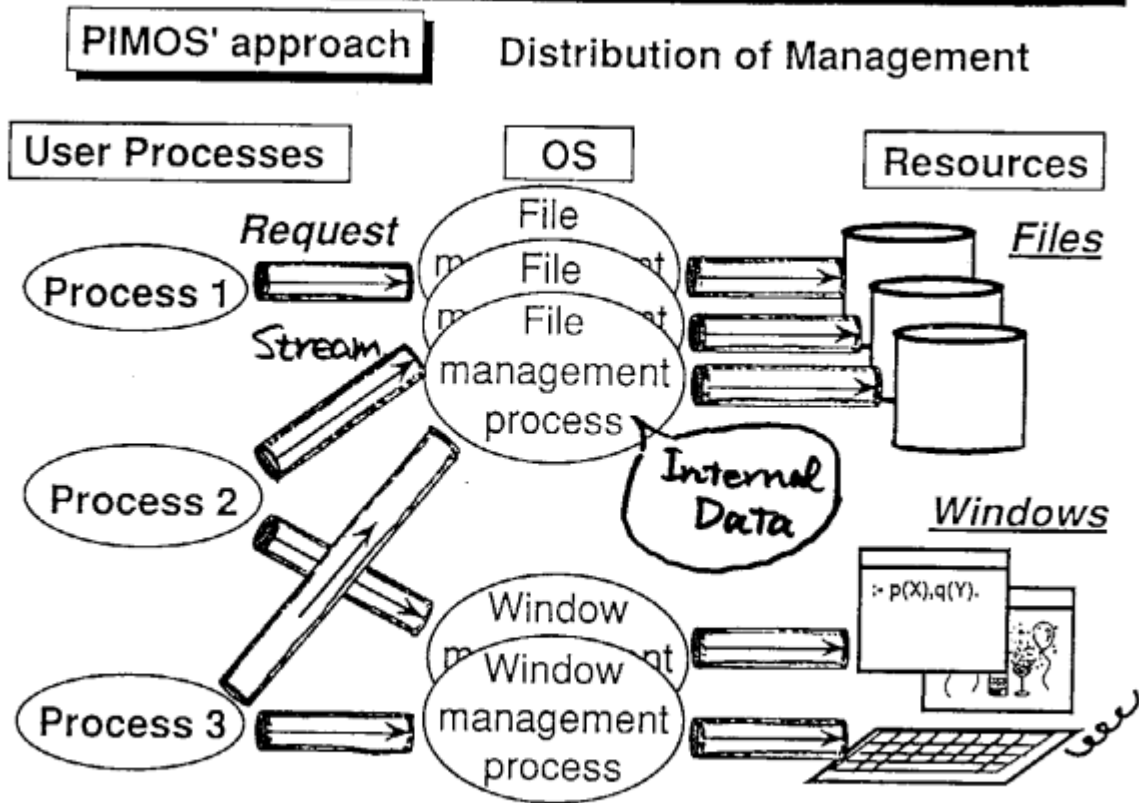


PIMOS
Parallel Inference Machine Operating System

- To control highly parallel programs efficiently.
- To provide a comfortable programming environment for the KL1 language.

Currently, PIMOS is running on the Multi-PSI.

Resource Management(2)



Communication Mechanism

Stream Interface between User and PIMOS.

→ KL1 programming style

user(Req) :-

Req =
[getb(N,String)|ReqT],

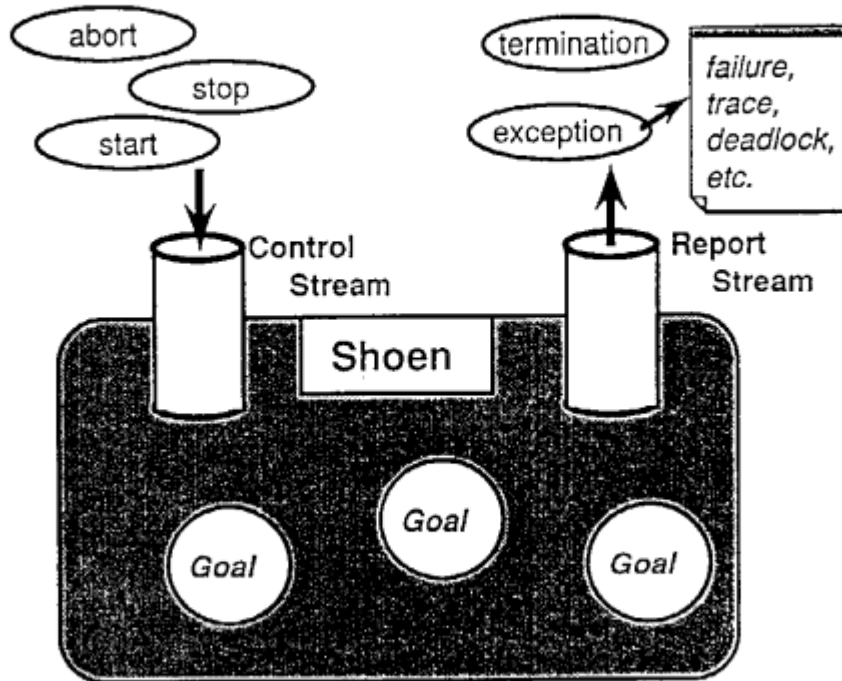
pimos([getb(N,String)|ReqT]):-
 readFromKbd(N,KBDString),
 KBDString=String,
 pimos(ReqT).



For succeeding request → ReqT

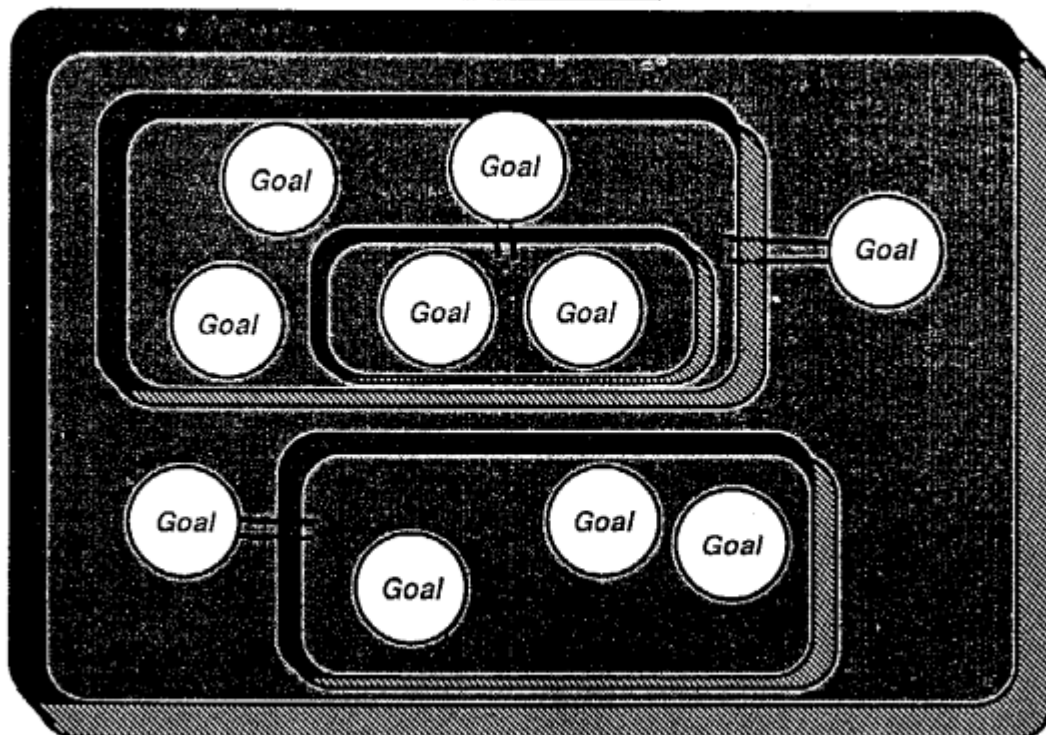
Shoen(1)

Meta-control mechanism



Shoen(2)

Shoen can be arbitrarily nested.



Task(1)

- Shoen : language defined behavior
→ provided by KL1 language processor.
- Task : Input/Output devices
→ provided by PIMOS kernel.

Shell for yashiro

```
yashiro>> ts |grep("job")
Line 5: 12 , "Job_1" , running, 1000000000, 50019
Line 7: 20 , "Job_2" , running, 1000000000, 46432
Line 9: 6 , "Job_0" , running, 1000000000, 37859
```

```
OK? yashiro>> ts
```

Task Status

```
+-----+-----+-----+
!TaskID|TaskName|Status|RscLimit|ConsumedRsc|
+-----+-----+-----+
9 , "shell:shell" , running, 2000000000, 401203
12 , "Job_1" , running, 1000000000, 50025
9 , "listener:go" , running, 2000000000, 46159
20 , "Job_2" , running, 1000000000, 46450
9 , "shell:go" , running, 2000000000, 42692
6 , "Job_0" , running, 1000000000, 37865
9 , "shell:shell" , running, 2000000000, 34019
40 , "Job_6" , running, 1000000000, 50599
9 , "ts:go" , running, 2000000000, 14501
17 , "grep:go" , running, 2000000000, 17999
45 , "Job_7" , running, 1000000000, 3702
9 , "ts:go" , running, 2000000000, 50
```

```
yashiro>> status
```

```
6 --> running ts|grep("Job")
```

```
2 --> running shell
```

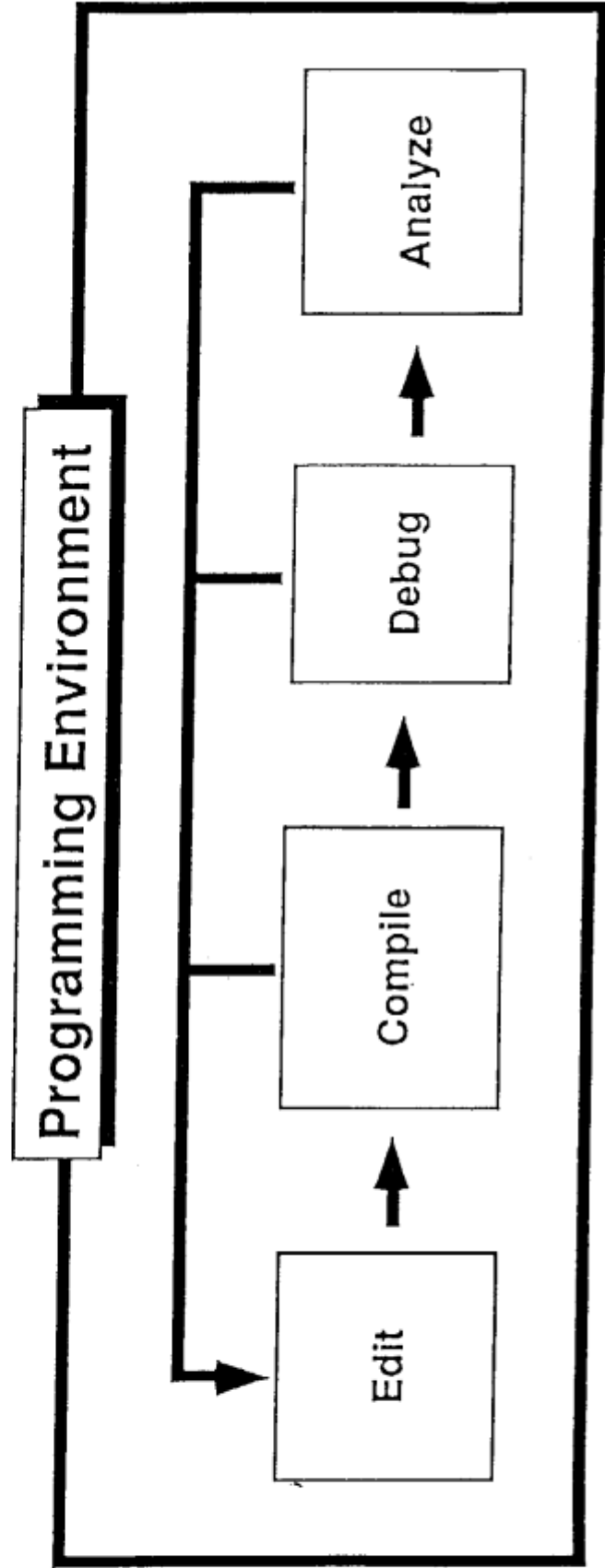
```
1 --> running listener (at(600,280), char(50,20), ">sys>font>test_11")
```

```
yashiro>> █
```

Job Status

Programming Environment

- Language Processor (such as Compiler, Interpreter)
- Debugging System (such as Debugger, Tracer)
- Analyzer (such as Profiler, System Monitor)

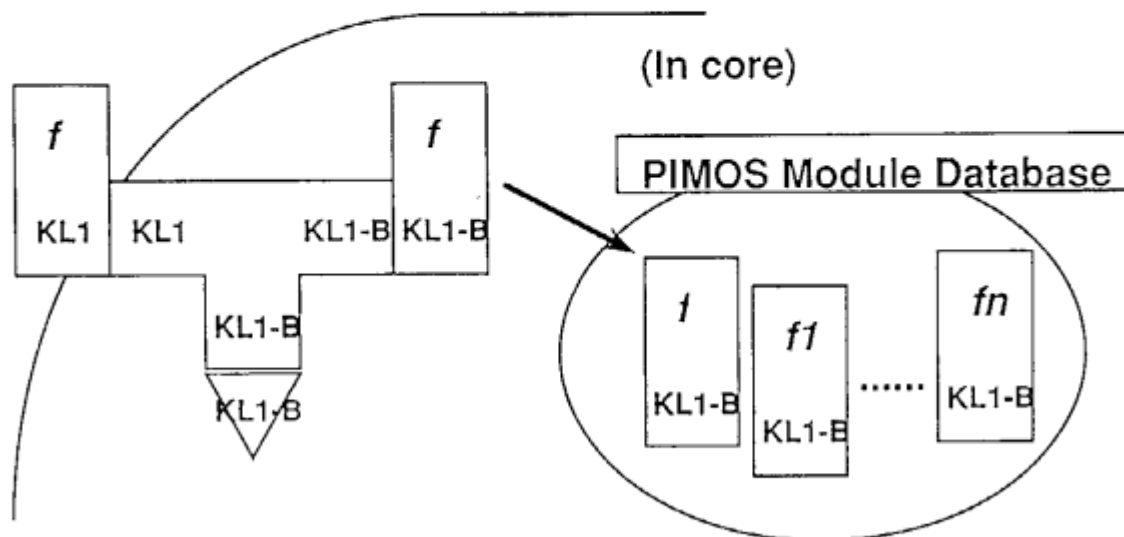


Compiler(1)

.translating from KL1 to KL1-B*

(*Abstract Machine Code)

Registration to PIMOS Module Database (In core)



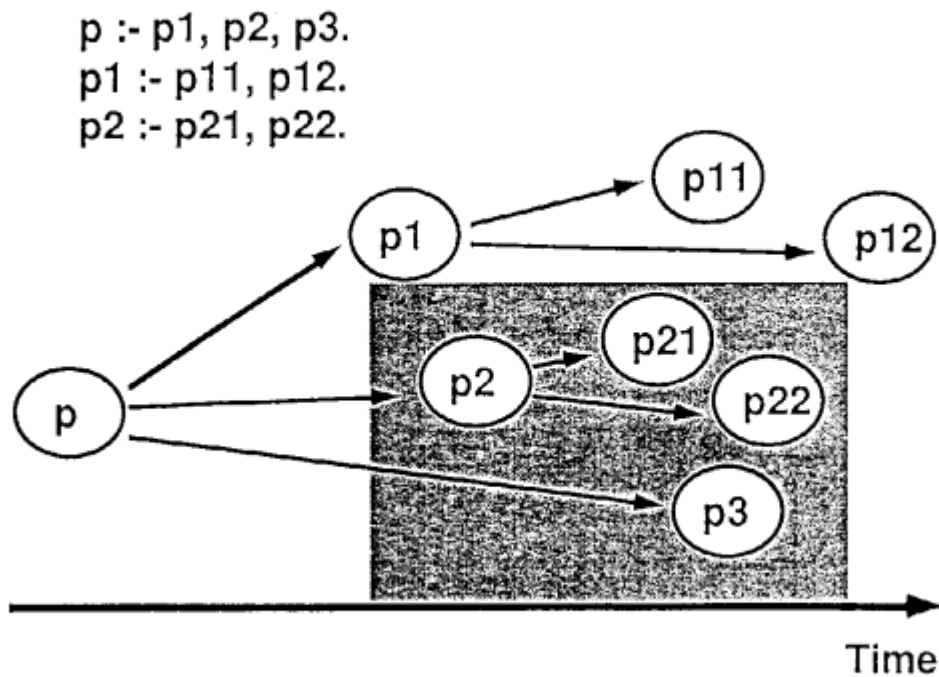
Listener

. KL1 language-oriented debugger.

Functions:

- Goal Execution
- Interactive Tracing
- Execution Profiler
- Deadlock Detection
- Inspecting and Monitoring Variables

KL1 Goal Execution

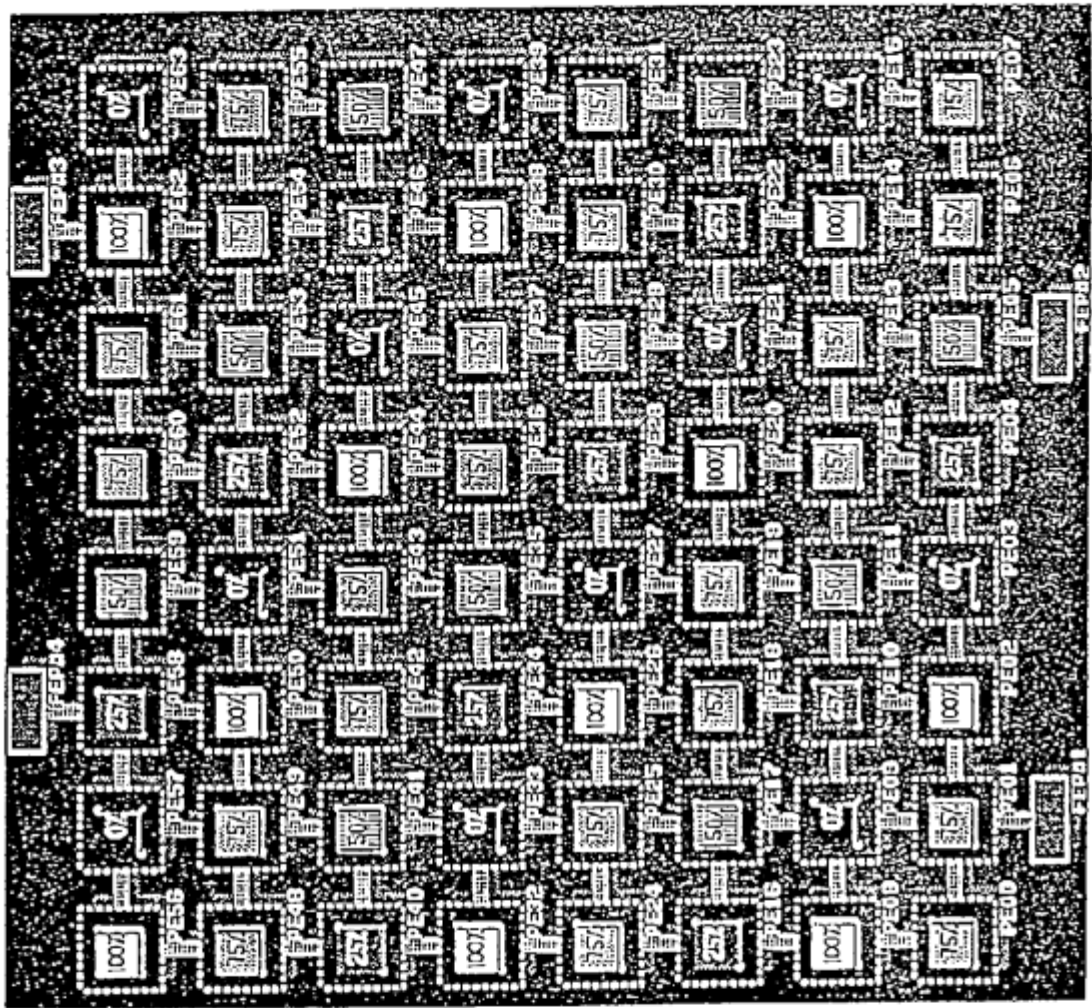


Inspector

Examiner for nested data structures.

- Examination of the arguments of the traced goals.
- The specified part of the inspected structure can be kept in named variable.
- Data-driven monitoring process can be initiated.

Analyzer(Performance Meter)



Conclusions

An overview of the KL1 programming environment provided by PIMOS has been described.

- Application programs and PIMOS itself have been developed on this system since 1988.
- We also have prepared the simulator of the Multi-PSI on personal WS.

Future plans:

1. Processor Profiler
examines the behavior of each processor
(e.g. frequency of communication with other processor)
2. Static Dataflow Analysis

PROGRESS IN THE DEVELOPMENT OF THE DATA DIFFUSION MACHINE

David H. D. Warren
Department of Computer Science
University of Bristol

The Data Diffusion Machine (DDM) is a novel scalable multiprocessor architecture. It has the coherent shared address space of a shared-memory machine in combination with the distributed physical memory of a message-passing machine. The location of a datum in the machine is completely decoupled from its address. In particular, there is no distinguished home location where a datum must normally reside. Instead, data migrate automatically to where they are needed, reducing access times and traffic.

The DDM consists of a hierarchy of buses (or networks), with a data directory at each level in the hierarchy. At the tips of the hierarchy are the processors, each having a large set-associative memory storing data and tag bits.

This talk will describe current progress in developing the architecture, which is being carried out by the Swedish Institute of Computer Science (SICS) and the University of Bristol in association with Meiko Limited as part of the Esprit project PEPMA.

A detailed simulator of the DDM data access protocol has been implemented by SICS in C++. This has verified the protocol on simulated traces of data accesses. A full-scale emulation of the architecture on transputers is being implemented by Bristol on the Meiko Computing Surface. The parallel Prolog system Aurora is being ported to this emulator to serve as a test vehicle. In parallel with these activities, SICS has started building a first hardware prototype of a 1-level DDM using commercial boards based on the 88,000 processor. A first working version is scheduled for late 1990.

Progress in the Development of the

Data Diffusion Machine

(a scalable "shared data" multiprocessor)

→ David HD Warren
Sanjay Raina*

Seif Haridi
Erik Hagersten
Anders Larzdin

University of Bristol

Swedish Institute of
Computer Science (SICS)

* Employee of Meiko

Topics

- Background
- Overview of the DDM concept
- DDM hardware prototype (SICS)
- Simulator of prototype hardware (SICS)
- Emulator of DDM concept on transputers (Bristol)
- Summary

ESPRIT Projects at Bristol in Parallel Logic Programming Systems

- PEPMA (2471)

3 years 680,000 ecu

Belgian Inst. of Management (BIM)
Meiko Limited
Swedish Inst. of Computer Science (SICS)
+ others

- EDS (2025)

4 years 640,000 ecu

European Computer Industry Research Centre (ECRC)
[ICL, Siemens, Bull, + others]

What :

Parallel execution of logic programs
on current and future multiprocessors.

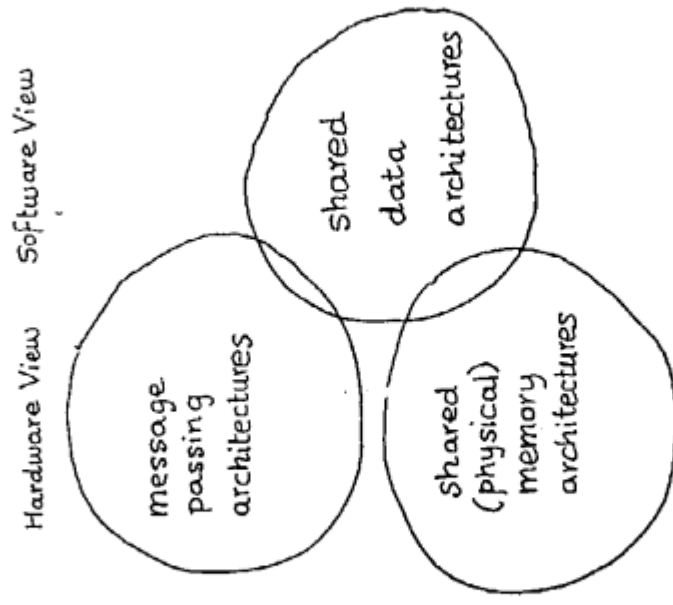
Why

To speed up programs in a way
that is largely transparent to
the programmer
(especially for advanced applications)

Main Developments

- Aurora
or-parallelism
- Andorra
or-parallelism + and-parallelism
- Data Diffusion Machine
shared data
without shared physical memory
(scalable)

Shared Data Architectures

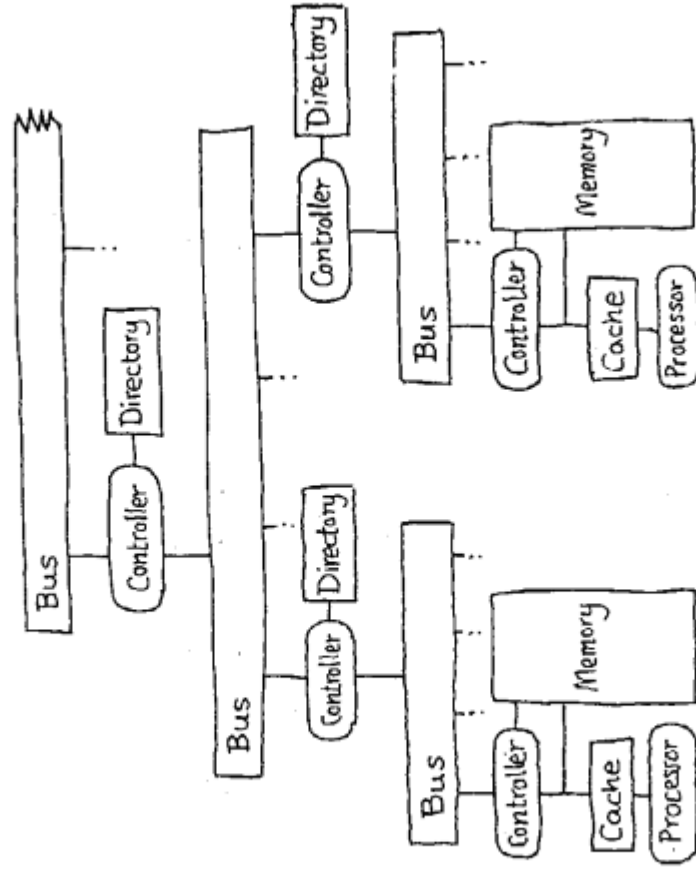


"Shared data" need not imply "Shared memory"

Data Diffusion Machine - Philosophy

- Processors share data rather than memory.
- A datum has "no fixed abode".
- A datum is identified by its virtual address.
- A datum may reside anywhere, and may be duplicated.
- Data diffuses to where it is needed.
- Most of a processor's data accesses are satisfied locally.
- Architecture should be general purpose.
- Architecture should take advantage of the properties of LP execution models: most data is write once then read only.

Hardware Organisation



Local and Non-Local Memory Accesses

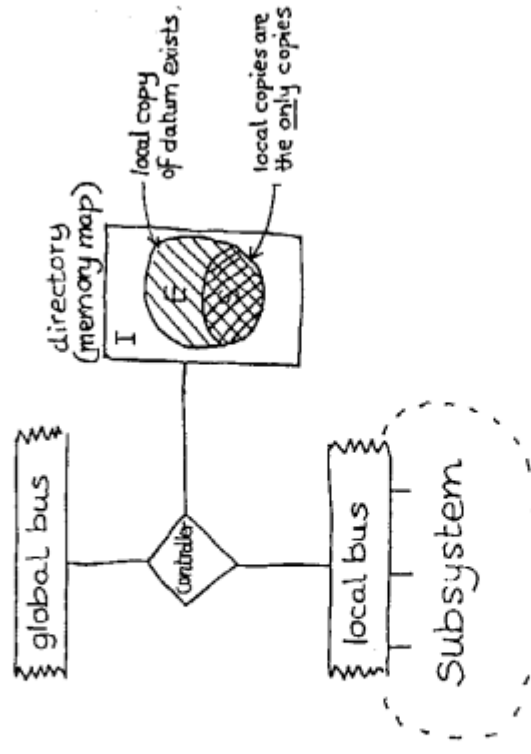
Local Communication

- read a local datum
- write an unshared datum

Non-Local Communication

- read a non-local datum
 - broadcast to nearest copy
 - mark as shared
- write a shared datum
 - erase all other copies
 - mark as unshared

The Data Controller



Possible states of a datum.

- Invalid (I) : does not exist within the subsystem
- Exclusive (E) : exists within the subsystem but not outside
- Shared (S) : exists within the subsystem and also outside

Summary of Controller's Actions

<u>Input Request</u>	<u>Condition</u>	<u>Output Request</u>	<u>Eventual State</u>
below: read(X)	invalid (X)	above: read(X)	shared (X)
below: write(X)	shared (X)	above: erase(X)	exclusive (X)
below: erase(X)	shared (X)	above: erase(X)	exclusive (X)
above: read(X)	valid (X)	[below: read(X)] [*]	shared (X)
above: erase(X)	valid (X)	below: erase(X)	invalid (X)

* Only one controller will be selected to respond.

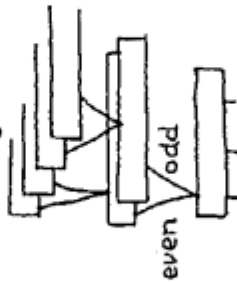
Machine Characteristics

- Data automatically resides where it was created or last accessed
- No need for repeated remote access to data that is initially remote
- Remote transactions are localised within the subsystem concerned
- In an N-level machine:
 - a read takes at most $4N-2$ bus cycles;
 - an erase takes at most $2N$ bus cycles
- There is a combining read effect when there are multiple requests for the same datum at more-or-less the same time

DDM - Other Issues

- Sparse Arrays and Null Values
Only addresses in actual use occupy storage.
- Locking.
Protocol extends to support locking.
"Spinning" on a lock can be performed without generating bus traffic.

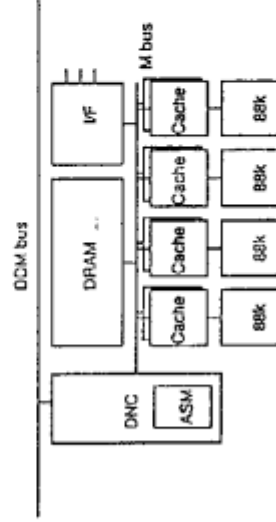
- Broadening Higher Buses to relieve bottlenecks



- Offsetting Latency of remote data access
Multiple lightweight processes per processor.
cf. Transputer, Denelcor HEP

DDM Hardware Prototype (SICS)

- Tadpole board 4 x Motorola 88,000 with cache + 8 M bytes (50 ns, 12 mips)
- DDM node controller (by gate array)
Split transaction bus \Rightarrow pipelined (50 ns, 40-80 M byte/sec)
- 20-processor (5-node) system by early 1991?
- Mach operating system
- Expected performance:
 - Cache access \sim 50 ns
 - Local mem. access \sim 500 ns
 - Remote mem. access \sim 5 μ s



Memory Overhead

AM = 2 Way, 8 Mbyte
item = 128 bits

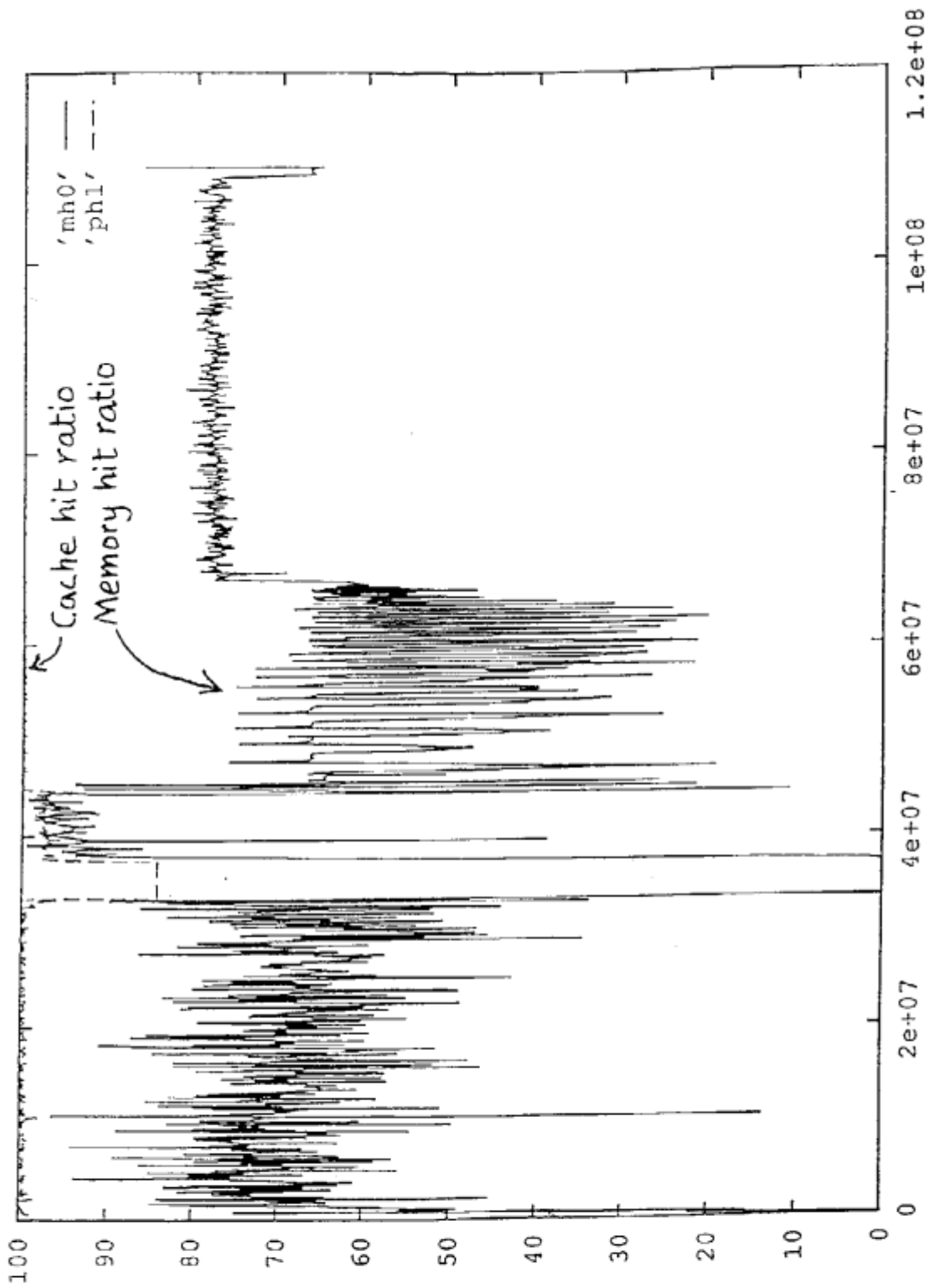
16-32 Processors, 6%:
AM: 4+4 bits,

128-256 Processors, 16%:
AM: 7+4 bits
DIR: 5+4 bits

Simulator of Prototype Hardware (SICS)

- Runs on Sun-4 (SPARC)
- Models timing of actual hardware
- Linked to application processes by Unix pipes (buffering \Rightarrow approx. to real timing)
- Simulation time $\approx 1000 N \times$ real time (where $N =$ no. of processors)
- Currently linked to Muse (or-parallel Prolog) Will be interfaced to other parallel applications (non-LP) and to Aurora and Andorra-I.

DDM Simulation Run



Emulator of DDM Concept on Transputers (U. of Bristol / Meiko)

Objective: To implement the DDM concept as efficiently as possible on transputers (Meiko Computing Surface)

Why?

- A scalable model of a full-scale (multi-level) DDM
- Running full-scale applications
- Memory expandable
- Quite fast
 - processor slowdown = $\times 4$ (\times row speed)
 - communications slowdown = 20-50 ?
- Can study e.g. traffic on higher level buses
- Gives insight into alternative hardware implementations of DDM, e.g. using networks or links instead of buses
- Can be slowed down and instrumented to act as a simulator of a specific hardware implementation

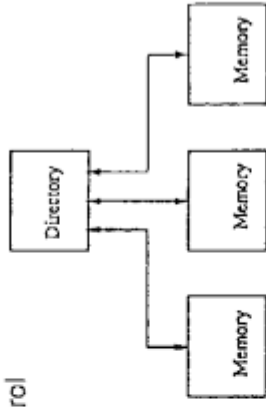
Main Problem

How to model the DDM bus with only transputer links?

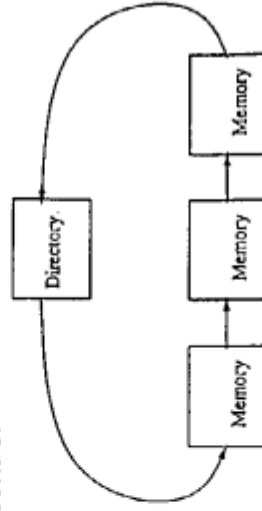
Implementing the DDM bus

Initial attempts - two solutions:

1. Centralised Bus Control



2. Distributed Bus control



Third solution: Modify the protocol.

Original protocol - a *snoopy* protocol

- relies on the assumption of a bus.

But, snoopy protocols - not a good idea for point-to-point interconnect based systems.

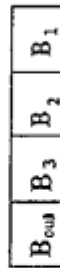
A link-based protocol for the DDM

Features:

- transactions are *directed* rather than *broadcast*.
- directories contain enough state information in order to direct transactions to one of the processors below or above.
- retains most of the functionality of the original protocol, only the mechanics are different.
- state information and protocol behaviour at the memory (leaf) processors unchanged.

Mechanics:

- Directories contain two types of state information.
 1. *Stable state information* associated with each item.
 2. *Transient state information* (cf. original protocol) associated with each outstanding request.
- state information is stored in the form of a *item presence bit-vector*.



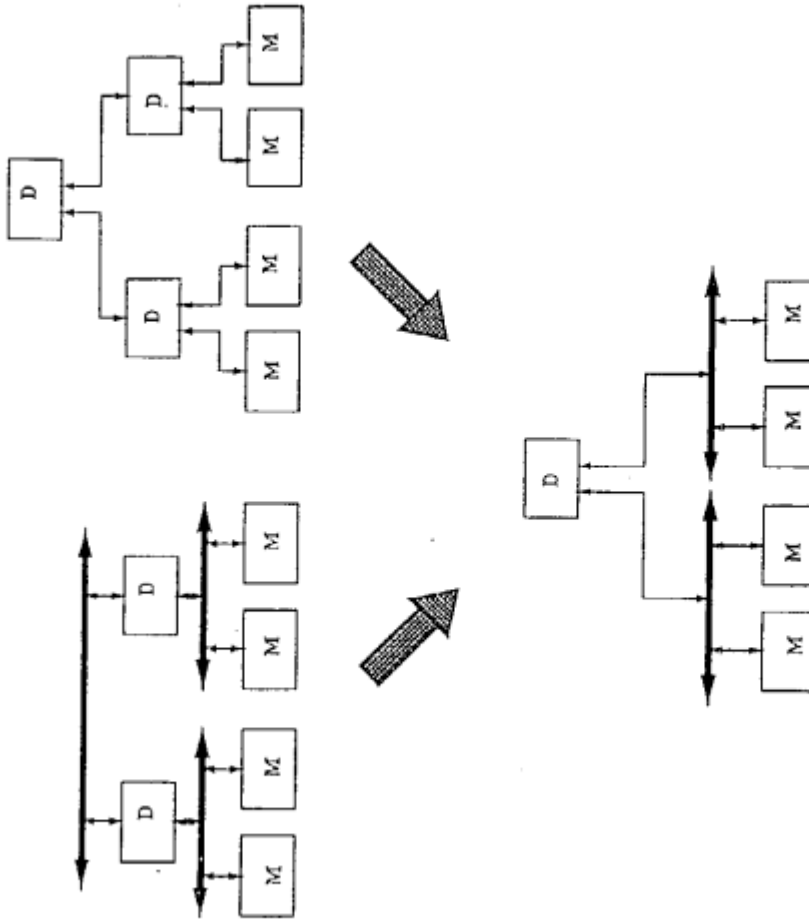
Stable state bit-vector: If B_n is set then item exists in subsystem r below.

If B_{out} is set then item exists outside this subsystem.

Transient state bit-vector: If B_n is set the response must be forwarded to subsystem n below on its way back.

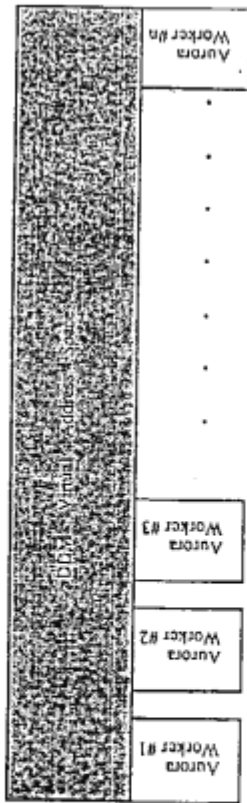
Note: All the states of the original protocol can be derived from the above information.

Possibility of a combined bus + link based DDM



- One of the levels of Directory nodes disappears

Porting Aurora (and friends) to the DDM



- Each DDM leaf processor runs an Aurora worker and treats the DDM address space as (virtual) shared memory.

Problem: To trap shared memory references from Aurora and divert them to the DDM emulator.

Solution: Compile Aurora source code into Transputer assembler.
 Only two transputer instructions *ldnl* and *stnl* access non-local memory.

Transform the assembler code by replacing *ldnl* and *stnl* instructions by DDM-trap code.

Example:

Before transformation:

```

.....
call 4
and (nonlocal)
word 4
ldnl 4
ldnl 5
.....

```

After transformation:

```

.....
call 4
and (nonlocal)
word 4
ldnl 4
ldnl 5
.....
-- begin ddm trap code
stl 2
stl 1
stl 0
ldnl 5
ldnl 4
ldnl 3
ldnl 2
ldnl 1
ldnl 0
call ddm_trap code
stl 0
stl 1
stl 2
stl 3
stl 4
stl 5
-- end ddm trap code
.....

```

Sicstus and single worker Aurora on a Transputer

benchmark	Sicstus			Aurora (worker)	
	Sun 3/50	Sun 3/50 (NoReg)	T800	Symmetry	T800
ahl-mustard	1.060	1.840	1.448	1.119	1.581
zebra	6.039	6.740	6.920	5.780	8.533
link	17.820	31.240	24.718	18.610	33.005

Table 1: Performance of Sicstus (Sun 3/50 vs T800) and single worker Aurora (Sequent Symmetry vs T800)

Benchmark	Total refs		Refs to Problog Stacks
	Without DDM	With DDM	
zebra	8.533	30.888	3,628,140
link	33,005	115,711	20,752,038

Table 2: Single worker Aurora with and without the DDM emulator

Data Diffusion Machine - Summary

Current status and further work

- a version of the emulator currently operational.
- porting of multiple worker Aurora under way.
- plan to port other shared memory applications.
- extensive analysis of the emulator running different applications.

- Shared address space (ie. shared data) but no shared physical memory
- Scalable
- Data migrates automatically to minimise remote accesses
- Physical location of a datum is completely decoupled from its virtual address
- Architecture is completely general purpose.
- Software compatible with current shared memory machines eg. Sequent
- Hardware prototype and simulator will demonstrate viability of at least a 1-level DDM
- Emulator will demonstrate scalability to a multi-level DDM

A Programming Environment for Parallel MIMD Machines

Patrick Evans, MEIKO

Meiko Scientific has been selling the Computing Surface, a distributed memory MIMD machine, since 1986. Some 300 systems are now in use in the field, applied to a wide variety of problem areas. The Computing Surface began as an entirely transputer-based machine and continues to use transputers for communications infrastructure. The majority of systems now sold, however, use Intel i860 or SPARC processors as primary computational resource. An occam-based software development system served for a number of years for parallel program development. More recently this has been superceeded by CS Tools, a cross development toolkit able to target a wide variety of processors from a number of different hosting environments.

Meiko's experience in the conceptual design and the software development process of parallel programs exceeds, perhaps, that of any other commercial organisation today. CS Tools, and the ideas behind it, represent the embodiment of much of that experience.

CS Tools supports the parallel programming of applications in two ways: firstly by providing a high productivity software development environment, secondly by supporting the notion of well defined abstract parallel machines.

Abstract machines are an essential aspect of modern software development - operating systems and high level languages, for example, cloak a variety of hardware with a consistent, easy to use interfac. Parallel programming introduces new issues including interprocessor communication, program configuration and load balancing, which impose new requirements for usable, hardware-independent abstractions.

CS Tools provides a general-purpose, low-level abstract parallel machine. This has been designed to support the creation of application-specific, higher level abstractions.

CS Tools also provides a high productivity software development system within a normal commercial context. It supports the cross-development of parallel programs from conventional hosting systems - Unix and VMS for example. It allows the majority of program coding to remain within widely used sequential languages such as C and FORTRAN.

This presentation overviews CS Tools and the use of CS Tools-based abstract machines in parallel applications.

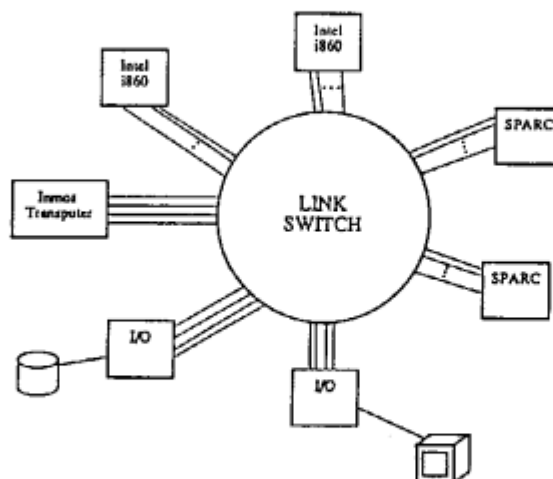
The Computing Surface and C S Tools

Patrick Evans, Meiko

© 1990

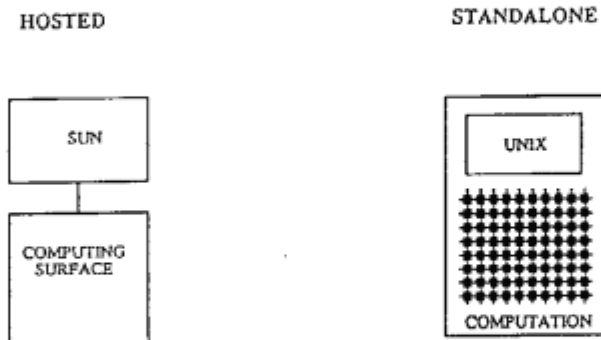
0

The Computing Surface: Distributed Memory
 MIMD
 Heterogenous



© 1990

THE COMPUTING SURFACE



©1990

CS Tools, communicating sequential tools, is a program development toolset for multi-processor computer systems.

It supports the development of single and multi-processor applications using familiar 'industry standard' development environments and languages.

CS Tools consists of cross-development tools, such as compilers and configuration systems, along with run-time support facilities such as high-level communication services and symbolic debuggers.

CS Tools is NOT a new "parallel operating system". What it provides instead is a set of tools and services which facilitate the cross-development of code for parallel machines.

CS Tools: a cross development approach

Development in a familiar software environment - vi, emacs, sccs etc.
- team working

Exploits existing investment in hardware and software

Minimises learning curve

©1990

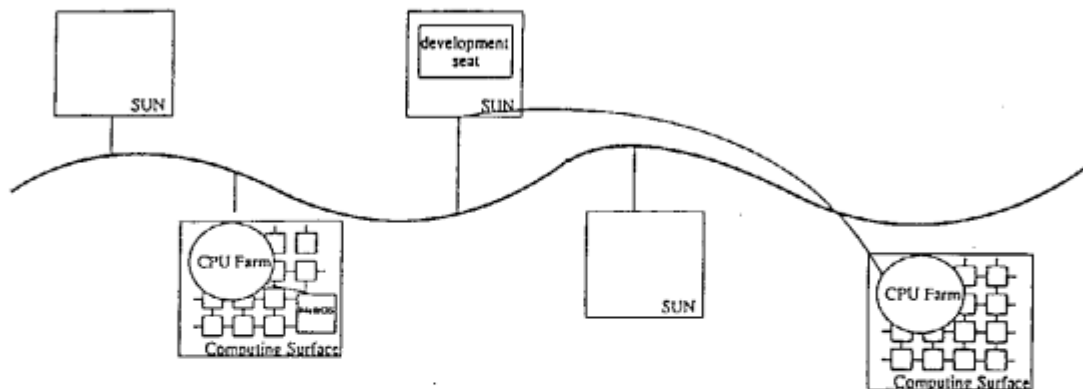
Three key aspects of the design philosophy embodied in CS Tools are, firstly

application development in familiar environments using tools and operating systems that developers are already skilled in using;

secondly, the exploitation of existing hardware and software resources generally;

and thirdly (arising naturally from the first two) a minimisation of the learning curve that developers must follow in order successfully to develop parallel applications.

CS Tools: a cross-development toolset



Programming a 'cpu farm' from a 'development seat'

Tools and run time facilities specifically for parallel programming

CS Tools - Sun, CS Tools - MeikOS, CS Tools - XXX

© 1990

Because it is designed as a cross-development toolset, CS Tools provides a clear distinction between the idea of a development seat and a parallel processing resource.

The parallel processing resource, the Computing Surface, can be viewed simply as a flexible, general computing resource, the power of which can be exploited efficiently using the CS Tools toolset.

The development seat takes the form of a hardware and software environment already familiar to the programmer; examples are Sun and VAX.

The development seat may also take the form of a MeikOS operating system running on 'standalone' Meiko hardware.

All development work takes place on the development seat, and, hence, existing software and hardware resources and skills are utilised.

Parallel programming in CS Tools is based on structuring a single application as a set of ordinary sequential code modules or processes.

Each of these processes will perform some of the processing required by the application as a whole, and will communicate with other processes to receive or provide information.

The modules will be written in standard high-level languages such as 'C' or Fortran (or a mixture) and may be allocated to specific processors as required.

CS Tools is designed to produce portable applications which may run on a variety of different hardware using exactly the same techniques and software.

meiko

CStools: Distributed Communicating Processes

- 1) Message passing facilities
- 2) 'O/S services'
- 3) Program configuration tools
- 4) Run time development tools - debuggers

In order to support this 'distributed communicating process' model of parallel processing, four distinct things must be provided:

Firstly, message passing facilities, to allow individual processes in an application to communicate efficiently;

Secondly, operating system services, to allow processes access both to local services such as memory allocation, and remote services such as file servers;

Thirdly, program configuration tools, allowing applications to be mapped onto appropriate hardware;

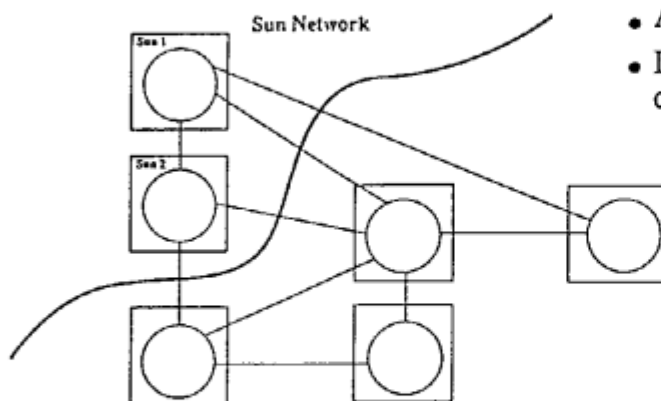
And, lastly, run-time development tools such as symbolic debuggers.

meiko

CS Tools: 1) message passing facilities

Design Aims:

- A 'fully connected' view of the hardware
- Identical functionality on the development seat



©1990

The major design aim for the first of these, message passing, is to provide the user with a "fully connected" view of the hardware so that the user does not need to consider the physical interconnectivity capabilities of the processors being used. -the system must provide simple logical connections between any one process and any other process so that the user may assume that everything is capable of communicating with everything else.

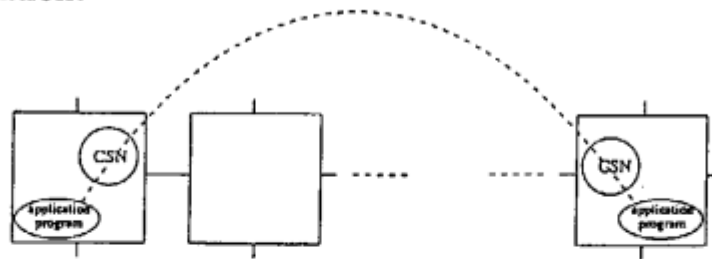
In order that applications may be developed wholly on the development seat, even though they may run on a separate parallel processing resource, the system also needs to provide identical functionality on the development seat as will be available on the target computing resource.

Notice that the middle processor in the diagram, which is a transputer, the process running on that processor is allowed to communicate with 5 other processors even though the transputer has only 4 physical links.

meiko

CS Tools: 1) Message passing facilities

Implementation:



```
csn_tx ( Transport, BLK, netid, message, mess_size)
```

```
csn_rx ( Transport, sender_id, buffer, mess_size)
```

©1990

CS Tools communication services provide cushioning between the programmer and the hardware, and present a clean, high-level model of the underlying parallel architecture.

The programmer's interface to these services is via a number of library function calls. Identical calls are provided on every processor type: i860, transputer etc. All CS Tools communications are made through the CSN, which takes the form of a series of background processes each of which is located on a specific processor.

These CSN processes are placed automatically by the CS Tools configuration and loading facilities.

Simple CSN interfaces provide general purpose communication functionality and are designed to be implemented efficiently on a variety of hardware.

This is a key aspect of the portability of the design of CS Tools.

In this particular example, two application processes which are "far apart" in terms of the processors which they inhabit, need to communicate.

The sender simply needs to call a CSN function, `csn_tx` or `transmit`, with parameters defining the communication channel, mode of communication (here, `BLOCKED`), the address of the receiver, the data and its size.

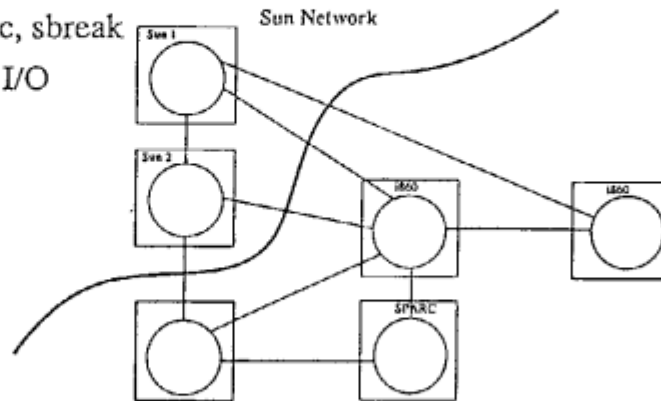
Similarly, the receiving process performs its part of the communication by calling a function `csn_rx`, or `receive` with parameters to define the communication channel, the buffer for receiving data and its size.

Optionally, the receiver may accept other information such as the identity of the sending process (`sender_id`).

CS Tools: 2) Operating System Services

Design Aims:

- A unix-subset runtime environment
- Local servicing - malloc, sbreak
- Remote servicing - file I/O



©1990

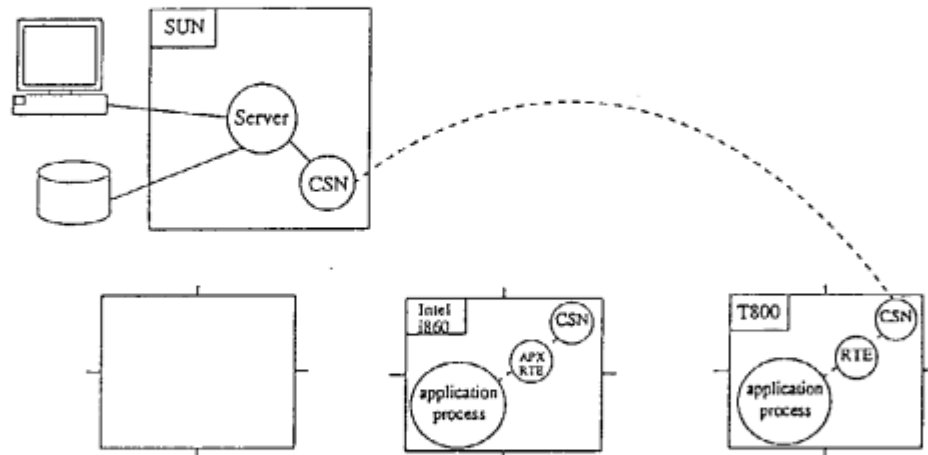
Design aims for operating system services are to provide each application process with facilities for both local and remote operating system calls.

These may range from allowing a process to print text to a remote screen to providing memory allocation and file servicing facilities across the CSN network.

CS Tools: 2) 'Operating System Services'

Implementation:

RTE + CSN: Local o/s facilities and a transparent bridge to remote services



© 1990

In this example, then, let's say that the application process wishes to access a file on the Sun's fileserver.

The process would make its service request to its associated RTE - the actual request will be unmodified from the programmer's viewpoint.

The RTE will then communicate this request to the server via the CSN, and the service request will be satisfied exactly as if the application process was running on the Sun itself.

Often services offered by traditional operating systems are required by processes in a parallel application.

If a process requiring such a service is not located within a conventional operating environment, CS Tools automatically assesses its requirements and puts alongside it a tailor-made RTE or RUN TIME EXECUTIVE.

The first function of the RTE is to provide local servicing for requirements which can be accommodated locally, such as memory allocation.

The RTE's second function is to provide services which can not be satisfied locally, and which require communication with a remote process or device, such as a file server.

The overall objective is to provide a complete run-time environment tailored to the requirements of each individual process and processor in a parallel application.

meiko

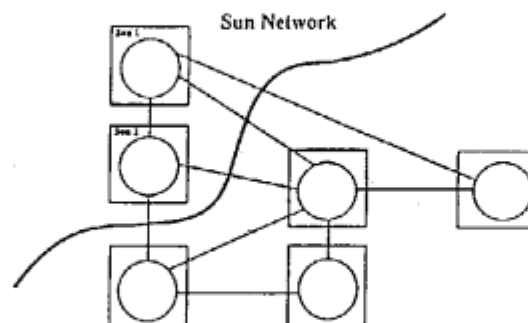
CS Tools: 3) Configuration and loading

Tools for:

Describing process distribution across processors

Describing communications topology

Loading and running



Configuration and loading facilities in CS Tools are used for describing the way in which individual parts of an application are allocated to processors along with information describing the required communication topology.

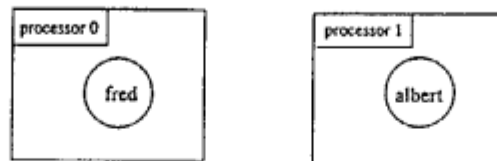
Configuration may happen in one of two ways, using either the Meiko par file loader or CS Build configuration tools - the base technology for all configuration mechanisms.

Both approaches automatically load and run an application once it has been configured.

meiko

CS Tools: 3) Configuration and loading

Simplicity: the .par file loader



```
mrunc example.par
```

```
example.par
```

```
par
  processor 0 fred
  processor 1 albert
endpar
```


The simplest of the 2 configuration mechanisms is the par file loader, MRUN.

The configuration, in this case simply two processes which need to run on 2 separate processors, is described in a par (-allel) file, here called example.par.

This file specifies that one processor, which we shall refer to as processor 0, will execute the process called fred, whilst the other processor, processor 1 will execute the process called albert.

The MRUN utility will interpret the par file, obtain resources (here processors) from the O/S, configure the processors (ie wire them together) and then load and run the processes.

Par files may be considerably more sophisticated, giving the user control over, for instance, exactly which processors are chosen, how they are connected, and various other characteristics of how the application is to be configured.

meiko

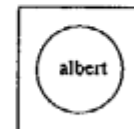
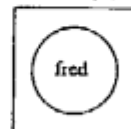
CS Tools: 3) Configuration and loading

Power and generality: `cs_build`

A library of C configuration and loading routines

User-visible and the base technology for parfile, application specific loaders and future developments - graphical design

```
main()
{
  GROUP fredGRP = cs_group(NULL, "fredGRP");
  GROUP albertGRP = cs_group(NULL, "albertGRP");
  :
  cs_exe (fredGRP, "my_fred", "fred");
  cs_exe (albertGRP, "my_albert", "albert");
  :
  cs_load ();
}
```



©1990

MRUN is a utility written using CS Build routines.
These are available directly, callable from 'C'*.

Developers may construct tailor-made configurer/loaders
for their specific applications

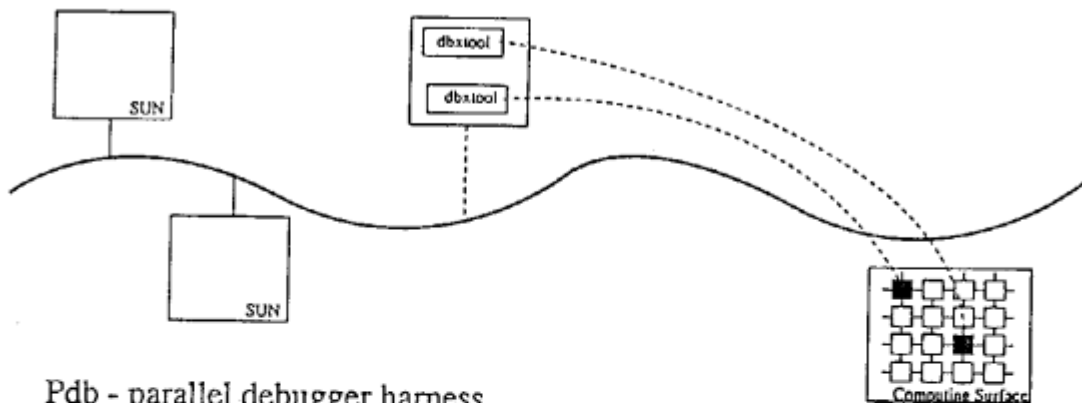
The configuration routines allow the user to define "groups" of
processes which may be run on one or many processors with
identical results (except for speed!) and to control the exact
configuration to be used at run time.

This example shows the CS Build version of the previous par file example.
2 CS Build GROUPS are defined, then 2 CS Build executable processes are created,
one belonging to each group. Each group and hence each process is, by default
loaded onto a separate processor by the call to `cs_load`;
The application will then be executed automatically.

* and, eventually, Fortran.

meiko

CS Tools: 4) Debugging



Pdb - parallel debugger harness

- + 'dbx-style' guest debuggers for i860, SPARC, transputers
- + 'dbxtool' window based source display

Debugging is obviously a key part of developing parallel applications.

CS Tools supports a full-function symbolic debugger, TDB based on the UNIX DBX debugger.

DBXTOOL, a windowed version of TDB, is also supported, giving full symbolic source-level debugging capabilities.

meiko

CS Tools: Abstract Parallel Machines

- **Base Level: Communications Calls
Configuration
Resource Management**
- **Application Specific Abstract Machines**

CS Tools: Summary

A Cross-Development approach to providing a full parallel programming environment.

Development in a familiar hardware and software environment exploiting existing skills and resources

'Abstract machine' interface cushions users from underlying hardware

DECOMPOSITION OF PARALLEL APPLICATIONS FOR SIMD MACHINES

1. COMPOSITION OF PRACTICAL SIMD SYSTEMS

SIMD systems obey a single instruction stream but each processor operates on its own independent data. All processors are lock stepped. Such systems are easy to program and easy to debug. Providing the problem tackled is appropriate they scale naturally. A solution developed for a given number of processors will improve pro-rata in performance as the number of processors is increased.

Processors can be very simple since they can be controlled by a single external control unit managing all the processors in the system.

In practice SIMD systems have been built from a very large number of very simple processors. The systems are massively parallel usually containing at least 1,000 processors. They are a very good fit to the properties of modern VLSI, which is at its best when replicating very simple units in large numbers at low cost and high reliability. Systems are produced using CMOS technology with processors usually provided in a customer designed chip. Typical numbers are 16 - 64 processors in a single chip usually operating at modest speeds 5 - 20 MHz. The processors undertake limited precision arithmetic, 1 - 8 bit are typical values. Systems are also provided with extremely good communication between processors.

2. SOME PROPERTIES OF SIMD SYSTEMS

There are a number of properties of today's SIMD systems that are not directly a consequence of the fact that they are SIMD but are by-products of the way SIMD systems are built in practice. These include:

- The systems are not tied to fixed word length. The system software provides a range of integer and floating point precisions and lower precision arithmetic is executed more rapidly than high precision.
- Internal bandwidth between memory and processors is very high and increases as the number of processors is increased. A by-product of this is that such systems can provide very high I/O capability with very low demand on the processors.
- Data communication and data movement are excellent, particularly for regular transformations.
- Efficient use of such systems requires that applications and algorithms maintain a high average utilization of all of the processors. Even a low percentage of the task requiring scalar arithmetic can limit the overall system performance.

3. COMPOSITION OF COMPUTE INTENSIVE TASKS

A problem that runs for minutes or hours on a fast modern serial computer system executes at least some part of the application code millions or billions of times. Compute intensive tasks are therefore always parallel. Only a minority of these tasks are inherently unsuitable for SIMD systems.

Much of modern computing practice and design has been determined by the requirements of numeric, scientific and engineering problems. This has set the practice of 32 bit floating point arithmetic or if this provides insufficient precision, 64 bit floating point.

Over the last 35 years our computer languages, libraries and tools and many of our algorithms have also been structured to meet these requirements plus the needs of serial architectures.

Despite the above a large fraction of today's compute intensive tasks are the processing of massive amounts of data, much of it non-numerical, and the majority of it limited precision. There is no general case for computer systems to be built for 32 bit or 64 bit floating point arithmetic.

Any substantial advance in computer performance must be based upon parallel systems with a large number of processors working efficiently on a single problem. For appropriate problems SIMD systems offer an easy route to massive parallelism and hence very substantial development scope.

Examples of such appropriate problems include:

- * Image Processing - a TV system provides 10^7 pixels per second with 8-12 bit data.
- * Signal Processing - Radar, seismic, sonar and many such sensor systems provide a continuous fast stream of data of limited precision.
- * Data Base Searching and Sorting requires the processing of massive amounts of data, much of which is of limited precision, eg. alpha-numeric.
- * Simulation and modeling often requires high precision numeric data but there are also many compute intensive tasks that process data of limited precision sometimes even Boolean. Some examples are; fault simulation of complex chips, cellular automata techniques, neural network simulation, etc.

4. MAPPING PROBLEMS ONTO SIMD SYSTEMS

4.1 Data Storage

The basic requirement to obtain high efficiency from a massively parallel SIMD system is to keep the average utilization of the processors high. Two extremes are very large problems, often referred to as "oversize problems" or the processing of a massive number of very simple problems in parallel, often referred to as "outer loop parallelism". As an example, for matrix inversion the former would be a matrix where the number of elements is much larger than the number of processors and where the single matrix is handled by the whole array. The latter would be the simultaneous inversion of a number of small matrices where the number greatly exceeds the number of processors. In this case each processor is used to tackle a complete matrix inversion and a N processor system completes N inversions at a time.

The design of correct algorithms demands consideration of the mapping of the total task to the SIMD system. Processing is not the sole requirement. Data must be sorted and moved in an optimum way. For real time applications latency may provide additional constraints.

The bad news is that there is no automatic way of converting serial code. The good news is that once the initial barriers are overcome the extensions provided to languages to make them suitable for parallel machines greatly ease the programming task. A Fortran-plus code is typically about one third the length of an equivalent Fortran code. AMT provides tools and libraries that together with the Fortran-plus compiler greatly aid the production of code. As an example most image processing tasks are a number of calls to routines provided in the image processing library.

The first requirement when tackling an application is to analyze the parallelism in the problem. When possible outer loop parallelism should be used since this reduces the requirement for data movement - if all computation is done in one processor all data for the calculation is provided in the data store for that processor. Often data provided from some practical system will be blocked up until enough is available to fill the computer system so that all processors are kept busy. Systems with a larger number of processors therefore require larger blocks and hence have a larger latency. This can sometimes provide a constraint on the size of system used.

If the problem is oversized, (ie. with more data points than the number of processors) thought must be given to the data mapping. Two general methods are used, "sheet mapping" and "crinkle mapping". Sheet mapping takes a local area of N^2 points, where N^2 is the number of processors and stores these as a sheet with one data point per processor and then stores successive sheets down the memory. Crinkle mapping folds the oversized problem so that it matches the array size. As an example a crinkle mapping of a 512×1024 size problem on to a 32×32 array will store local areas of $512/32 \times 1024/32$ (ie. 16×32) in the memory of each processor thus compressing the total set of data into the array. Each processor then deals with that local set of data - ie. $16 \times 32 = 512$ data points in the example. Crinkle mapping has the same advantages as outer loop parallelism in that it requires less data movement. AMT's compilers can automatically map oversized problems onto the physical array size and hence relieve the user of the task. The same source code will run on any array size and only a recompilation is needed.

Different mappings for the data are often required for different stages of a computation. Hence data has to be moved in store from one regular structure to a different regular structure. An example could be the transpose of a matrix or a perfect shuffle on a data set. These transformations can be complex to program but a solution is provided by the system software. AMT has some very elegant software called "parallel data transforms" (PDT) that handles these tasks automatically and very efficiently.

4.2 Arithmetic Precision

Since SIMD systems in practice are built from simple processors which execute limited precision arithmetic the systems run faster when lower precision is used. A 1024 processor DAP system working at 10 MHz will execute 10,000 million Boolean instructions per second.

Users can get code working using floating point arithmetic and can then experiment with lower precision to provide optimum performance.

The system software can be "clever" and use variable precision arithmetic for successive iterations or even tapered arithmetic increasing the precision one bit a time to match the increased requirements as the calculation progresses. Such attention to detail is not required by the average programmer but they can benefit by improved performance because the system software employs such procedures.

In some cases algorithms can be selected that use Boolean data to get improved performance. An example is the comparison of two images stored as a Boolean sets reduced from images of greater precision. A second example is track following when elements to be fitted to a track are stored as a Boolean map.

4.3 Algorithm Selection

In many problems the algorithms normally used have been selected to optimize performance on a serial machine. Often a more direct and far simpler solution has been rejected to gain small advances in performance. Often these "obvious" procedures are better for parallel machines. There is no short cut to the selection of suitable algorithms and no replacement for thought and clear thinking.

As an example finding the largest number from 4096 on a serial machine is completed by comparing successive values and selecting the largest. An obvious option for a parallel machine is to compare 2048 pairs, and then 1024 pairs etc. But for the DAP no arithmetic is used at all. The most significant bit plane is selected and in two cycles the DAP can decide if all elements are zero. If it is, then the next significant bit plane is studied until some elements are non-zero. Each of the processors with non-zero values is turned on and the others off. The process is continued thus finding the largest number. This procedure is extremely fast. I use it as an illustration that lateral thought often provides handsome rewards on massively parallel SIMD systems. It also illustrates the use of the activity control available on the DAP. This is an important design aspect in that it allows the system to be used efficiently on conditional code.

Space does not permit an in-depth treatment of algorithm selection. Even were the space available I would not wish to undertake it because it is an area where substantial research is required.

5. HARDWARE DESIGN FEATURES IN A SIMD SYSTEM

The architectural design of a SIMD system is a balance between system simplicity to keep the cost down so that massively parallel systems are affordable, and increased performance by adding additional features. As usual it is the balance of the system that is important.

AMT has provided hardware to provide performance on operations frequently required. Software to undertake more complex tasks infrequently required is provided instead of expensive hardware.

An example is data movement where AMT provides nearest neighbor, row and column connectivity in hardware and PDT software for less frequently used longer range transformations.

The average user should not be over impressed with hardware manufacturers claims for the advantages of their specific features. The only relevant parameters to the user are:

- * The performance on his task.
- * The ease of programming.
- * The price-performance provided.

Benchmarking for parallel machines is an important research topic that needs serious and urgent attention.



AMT

DECOMPOSITION OF PARALLEL APPLICATIONS FOR SIMD MACHINES

PROF G MANNING
CHAIRMAN ACTIVE MEMORY TECHNOLOGY

- * PROPERTIES OF SIMD SYSTEMS
- * PROPERTIES OF DAP
- * COMPOSITION OF COMPUTE INTENSIVE TASKS
- * MAPPING PROBLEMS ONTO SIMD SYSTEMS
- * HARDWARE FEATURES IN A SIMD SYSTEM
- * PERFORMANCE FIGURES FOR DAP




AMT

PROPERTIES OF SIMD SYSTEMS (inherent)

- * SINGLE INSTRUCTION STREAM
(one control unit)
- * MULTIPLE DATA STREAMS
(each processor has its own memory)
- * ALL PROCESSORS ARE LOCKSTEPPED


AVCOI-902



PROPERTIES OF SIMD SYSTEMS
(practical implementations)

- * **MASSIVELY PARALLEL** \geq 1000 PROCESSORS
- * **BUILT FROM SIMPLE PROCESSORS.**
eg 1 bit, 4 bit, 1 bit + 8 bit, 1 bit + $\frac{1}{32}$ 32 bit FP
- * **EXCELLENT CONNECTIVITY PROVIDED**
nearest neighbour
row and column
general

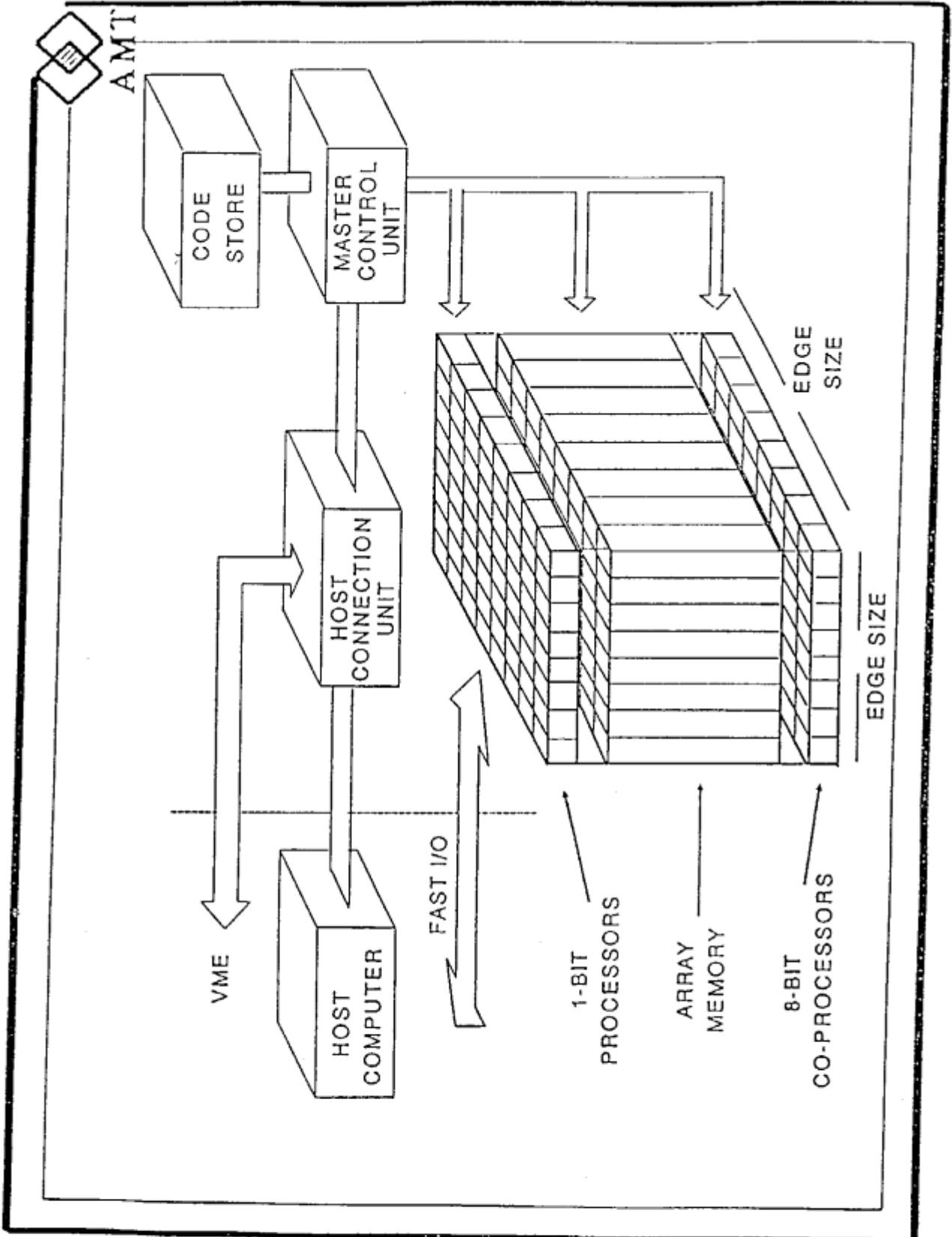
AMT
AVICOM 903



**RESULTING PROPERTIES
OF SIMD SYSTEMS**

- * no fixed word length - variable precision
- lower precision executed more rapidly
- * internal bandwidth outstanding and increases as number of processors increases
- * excellent input/output capability
- * excellent data movement and data communication
- * easy to program for appropriate problems
- * automatic conversion of serial code is not practical

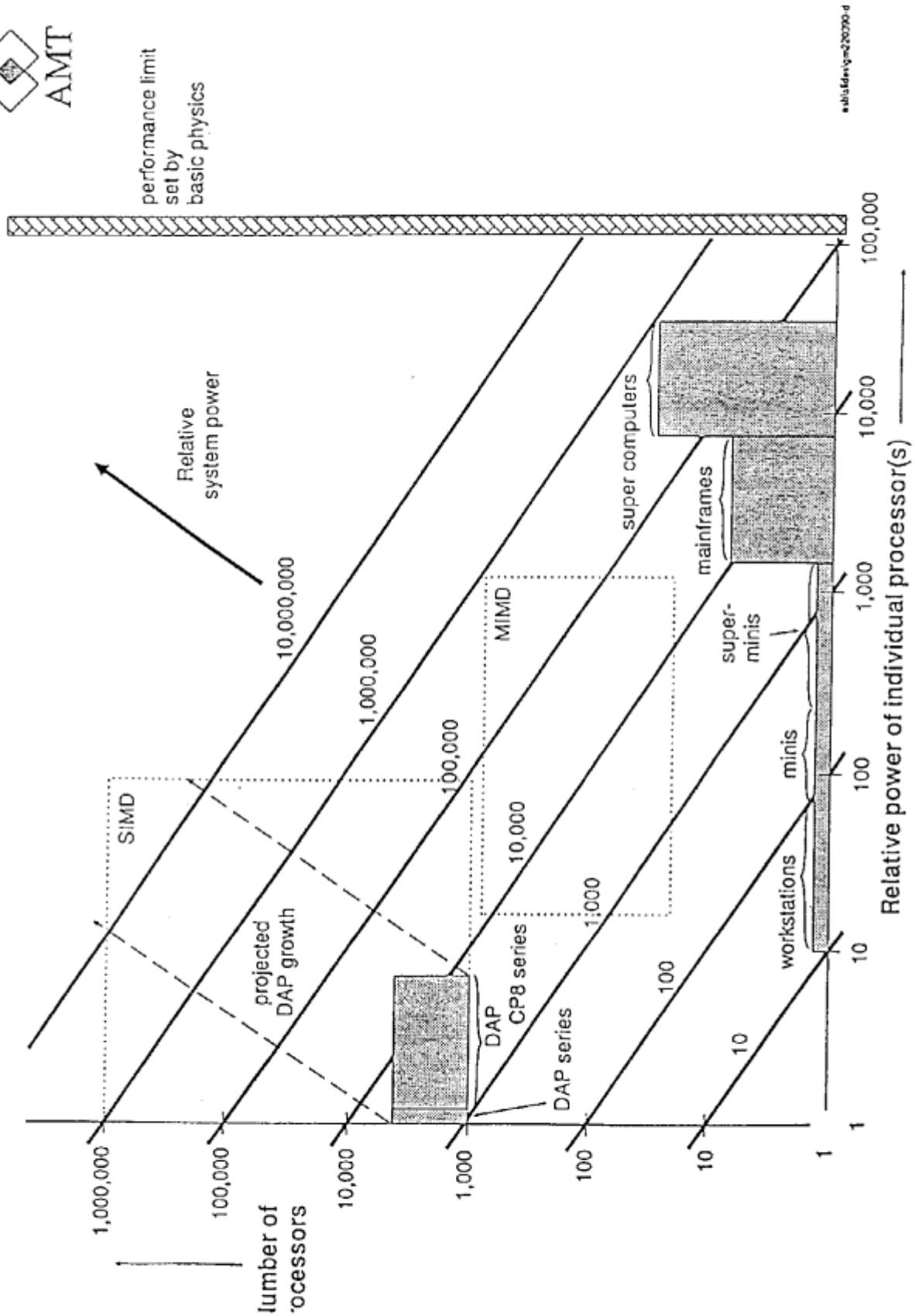
AMT
AVICOM 904






SUMMARY OF DAP PERFORMANCE

DAP	510	610	510C	610C
Speed	10 MHZ	10 MHZ	10 MHZ	10 MHZ
Processors				
1 bit	1024	4096	1024	4096
8 bit			1024	4096
Peak Performance				
1 bit MIPS	10,000	40,000	10,000	40,000
8 bit MIPS	400	1,600	5,000	20,000
32 bit MFLOPS	15	60	140	560
I/O M Bytes/s	40+40	80+80	40+40	80+80
Processor use	3%	1.5%	3%	1.5%
Relative Power	1	4	10	40
Price Performance				
\$/MFLOP	8000	5000	1150	700



AMT/610C/20790-d


Figure 3 The figure shows the position of the current DAP range and that of the new DAP/CP8 range. The 610C model has a performance in the super computer range.



COMPOSITION OF COMPUTE INTENSIVE TASKS

- * COMPUTE INTENSIVE TASK
 - runs for say 1 hour on modern serial system
 - real time requirementor
 - eg processing of data from some experimental system
- * All compute intensive tasks are parallel application code is executed millions or billions of times
- * Very few compute intensive tasks are inherently unsuitable for SIMD systems

A:\ICOT-90 8



EXAMPLES OF COMPUTE INTENSIVE TASKS

- * IMAGE PROCESSING
 - eg TV system provides 10^7 pixels/sec
 - 8-12 bit data
- * SIGNAL PROCESSING
 - eg radar, sonar, seismic
 - stream of data of limited precision
- * DATABASE SEARCHING AND SORTING
 - eg key word searching
 - DNA sequence matching
 - tabulation or histogramming
 - G Bytes of data of limited precision

A:\ICOT-90 9

AMT

CONCLUSIONS ON COMPUTE INTENSIVE TASKS

- * ARE INHERENTLY PARALLEL
- * A LARGE FRACTION ARE DATA PARALLEL
- * A LARGE FRACTION ARE PROCESSING DATA OF LIMITED PRECISION
- * A LARGE FRACTION FIT ONTO SIMD SYSTEMS
- * THERE IS NO CASE FOR FLOATING POINT PROCESSORS FOR ALL COMPUTE INTENSIVE TASKS
- * A VARIABLE PRECISION CAPABILITY IS AN ADVANTAGE

A:\ICOT\90 11

AMT

SIMULATION AND MODELLING OF PRACTICAL SYSTEMS

(an extremely broad area)

- sometimes requires high precision floating point arithmetic to get required accuracy
- sometimes very limited precision is adequate
eg fault simulation of complex chips
cellular automata techniques
neural networks

A:\ICOT\90 11

MAPPING PROBLEMS ONTO SIMD SYSTEMS

- * AIM IS TO KEEP AVERAGE UTILISATION OF PROCESSORS HIGH
- * THERE ARE THREE ITEMS TO BE CONSIDERED
 - algorithm selection
 - data storage
 - arithmetic precision
- * EXECUTION TIME INVOLVES COMPUTATION AND DATA MOVEMENT both are important and must be considered
- * FOR SOME TASKS "LATENCY" (ie the delay before a result is available) IS VERY IMPORTANT AND MUST BE CONSIDERED

ALGORITHM SELECTION

- * there is no automatic conversion of serial code
- * algorithms used on serial machines have been selected to optimise performance on serial machines and are not the obvious procedure to use
- * often there are more direct, more obvious and simpler algorithms that will map better onto a massively parallel SIMD system
- * look at inherent parallelism in the problem
- * consider how this maps onto the physical connectivity of the system to be used
- * attempt to minimise data movement
- * if latency is not a problem it often pays to collect data to make the problem large



Example find the largest of 4096 numbers

- * Serial machine
 - compare largest and next value
 - store largest
 - loop
 Takes 4096 steps
- * first thoughts for a parallel machine
 - compare 2048 pairs
 - store largest
 - compare 1024 pairs
 - etc
 Takes 12 steps but lots of data movement

- * DAP procedure
 - look at most significant bit plane and ask if any non-zero (2 cycles)
 - continue down planes until some non-zero
 - turn off all processors where bit is zero
 - continue to least significant bit
 - no arithmetic, no data movement, very fast

THERE IS NO SUBSTITUTE FOR CLEAR THOUGHT



TWO MAIN FORMS OF MAPPING COMPUTE INTENSIVE TASKS ONTO A SIMD SYSTEM

OVERSIZE PROBLEMS

- * Problem is inherently very large with the number of data points \gg number of processors
- * eg processing of images of 1024 x 1024 pixels
- * problem is tackled by complete array in a series of steps

OUTER LOOP parallelism

- * Repetitive processing of very large number of simple tasks
- * eg Search a very large text database for a keyword
- * each processor solves a task and N tasks solved at a time where $N =$ number of processors

IN BOTH CASES

Number of steps = Number of data points

Number of processors

but mapping of data into memory
can be very different

AMT

A:\COT-9016

DATA STORAGE

* THE MAPPING OF THE DATA INTO MEMORY
IS DEPENDENT UPON THE ARCHITECTURE
OF THE SIMD SYSTEM USED

* FOR THE DAP

THERE ARE TWO MAIN DATA MAPPINGS USED

- SHEET MAPPING
- CRINKLE MAPPING

AMT

A:\COT-9017

SHEET MAPPING

NUMBER OF DATA POINTS = D

NUMBER OF PROCESSORS = N

PRECISION OF DATA = n bits

- * first N data points stored in the processors with one data point per processor
 - this is a "sheet" of data
 - occupies n planes of memory
- * successive N data points stored as successive sheets in successive sets of n planes of memory
- * total data set stored as
 - D/N sheets
 - $\frac{nD}{N}$ planes of memory

CRINKLE MAPPING

THE DATA POINTS ARE COMPRESSED SO THAT THEY MATCH THE ARRAY SIZE

EXAMPLE

- * an image of 512 x 4096 pixels is to be mapped onto a 32 x 32 array of processors
- * the image is broken up into 32 x 32 elements with one element stored in one processor
- * each element has $\frac{512}{32} \times \frac{4096}{32}$ pixels
 - ie. 16 x 128 pixels
- * planes used are 16 x 128 x n

OUTERLOOP PARALLELISM

- * where possible use this form of parallelism
- * store data for each task in the memory of the processor undertaking that task
- * hence no data movement is required between processors
- * providing number of tasks \geq number of processors and all tasks require identical processing efficiency is 100%

AMT

A:\ICOT-90 20

OVERSIZE PROBLEMS

- * Crinkle mapping stores data for a local area in one processor
- * Hence if processing requires connectivity between local data points this mapping is better because there is less data movement
- * If there is little requirement for connectivity between local data points then sheet mapping is simpler

AMT

A:\ICOT-90 21

Note

While the mapping of data is important and requires experience, in practice it is made easier because the tools and libraries do a great deal of the work

- * DAP compilers map arbitrary size problems into actual array size automatically
- * libraries do a great deal of the work
- * Parallel Data Transforms (PDT) automatically move data between one mapping and another

A:\ICOT-90 22

ARITHMETIC PRECISION

- * SPEED INCREASES AS PRECISION DECREASES
 - HENCE 610 HAS 40,000 MIPS FOR 1 BIT ARITHMETIC
- * WHERE POSSIBLE USE BOOLEAN ARRAYS
- * CAN GET PROBLEM WORKING WITH FLOATING POINT ARITHMETIC
- * THEN INVESTIGATE NECESSARY PRECISION TO OPTIMISE PERFORMANCE
- * SYSTEM SOFTWARE USES VARIABLE PRECISION
 - eg for square roots
 - different precision for successive iterations
- eg tapered arithmetic for fourier transforms

A:\ICOT-90 23

HARDWARE FEATURES OF A SIMD SYSTEM

- * Architecture design is a compromise between improved efficiency provided by additional hardware and the cost of providing the hardware
- * Best solution is to provide hardware for frequently used operations and provide easy to use software for more complex infrequently used operations
eg Connectivity
DAP provides hardware for frequently used:
nearest neighbours
row and column highways
but PDT software for more general movements that are infrequently used

AMT

- * Different manufacturers have selected different "optimal" solutions
- * They all claim theirs are best
- * Don't believe them
- * Don't trust your intuition
- * Benchmark your problem
- * Investigate ease of programming
- * Look at price-performance

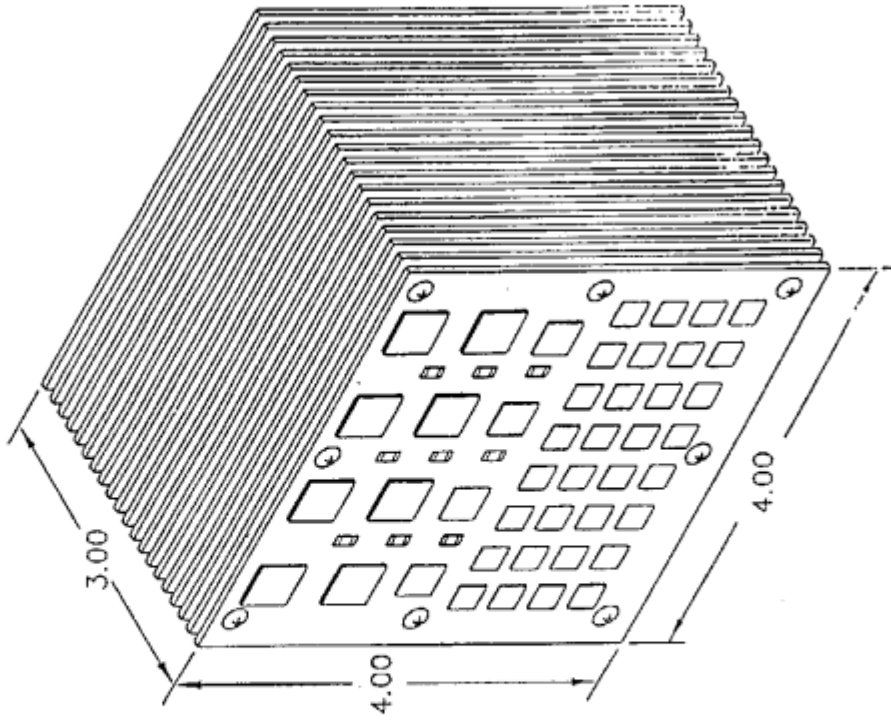
AMT

AVICOT-90 25

AVICOT-90 24



ULTRADAP / STARLIGHT



**SIMD SYSTEMS
CAN GIVE REMARKABLE
PERFORMANCE FIGURES**

**OPTIMAL FIGURES FOR DAP
FOR DEPLOYABLE SYSTEMS**

- 30 MFLOPS per cubic inch**
- 7 MFLOPS per Watt**
- \$250 per MFLOP (32 bit)**
- \$10 per MIP (8 bit)**
- Rugged**
- Easy to use**
- Wide range of applications**

A:\V00T-90 26

Parallel Application Program Research at ICOT

Nobuyuki Ichiyoshi
The Seventh Research Laboratory, ICOT

This extended abstract overviews parallel application program research and development at ICOT since the prototype parallel inference machine Multi-PSI became operational in late 1988.

1 Multi-PSI

The Multi-PSI is a prototype parallel inference machine (PIM), which became operational in late 1988. It is a distributed memory multiprocessor with an 8×8 mesh network (cut-through routing). The concurrent logic language KLI is implemented in microcode, and all programs including the operating system PIMOS is written in KLI. Several new implementation techniques were tested on the machine, and will be incorporated in the pilot PIM machines. The Multi-PSI now serves as a workbench for research and development of medium- to large- scale parallel programs.

2 Early Benchmark Programs

The first programs to run on the Multi-PSI were ones to solve simple problems. The development began in middle 1988, and some of them continued to be improved over a year or more. Among those programs were Pentomino and Bestpath programs.

Packing Piece Puzzle (Pentomino)

A rectangular box and a collection of pieces with various shapes are given. The goal is to find all possible ways to pack the pieces into the box. The puzzle is also known as the Pentomino puzzle, when the pieces are all made up of 5 squares. The program does a top-down OR-parallel all solution search.

Shortest Path Problem

Given a graph, where each edge has an associated nonnegative cost, and a start node in the graph, the problem is to find a shortest path to every node in the graph from the start node (single-source shortest path problem). The program performs a distributed graph algorithm. We used a 200×200 grid graph with randomly generated edge costs.

We developed a multi-level load balancing scheme (all written in KLI) in the Pentomino program, and attained a 50-fold speedup using 64 processors. The load balancing mechanism was extracted from the program, and was released as a first utility in the KLI algorithm library.

In the Bestpath program, several graph-to-processor mapping schemes were tested, and it was shown that the performance could change dramatically simply by changing the mapping schemes.

Other than the Pentomino and Bestpath, PAX, a syntax analyzer for English sentences, which is an implementation of a chart parsing algorithm in KLI, and a Tsumego solver that does parallel alpha-beta search in solving a Tsumego problem (counterpart of checkmate problem in the game of Go) were developed.

3 Recent Application-Oriented Programs

More recently, more programs have been written, which are more application-oriented. Those programs include the following.

- VLSI-CAD Programs

An LSI chip router and a logic simulator have been written, and are under improvement.

- Genetic Information Processing Programs

Two different algorithms for multiple sequence alignment are being written. Development of more knowledge intensive programs are also planned.

- Legal Reasoning Program

A prototype legal reasoning program that generates arguments, based on precedents, for both the plaintiff and the defendant in a law suit was developed. It employs a case-based reasoning mechanism.

- Go-playing Program

A sequential Go-playing program (GOG) has been developed since 1985, and is now being translated in KL1. The parallel version will have an enforced search task for determining the best move and incorporate new Go strategies to make the system stronger.

The purpose of developing those programs is not so much to test clear-cut parallel algorithms as to evaluate the utility of KL1 in developing parallel programs to solve non-toy problems. The details of the programs will be presented in the workshop sessions and the demonstration session.

4 Performance Analysis and Monitoring

The target machines of parallel program research at ICOT are highly parallel computers with 1,000 or more processors. This means we must not tailor the programs to the 64-processor Multi-PSI (or, for that matter, even to a 512-processor PIM). Raw performance figures on the Multi-PSI could be misleading in designing and tuning parallel algorithms and mapping schemes. The algorithm designers have to not only *measure* performance but *analyze and understand* it.

We started a *scalability analysis* of the multi-level load balancing scheme devised for the Pentomino program, and some interesting results have been obtained. This research area should be much more strengthened. Research in parallel algorithms and data structures is also pursued.

Performance monitoring capabilities help a great deal in getting the idea of the runtime characteristics of programs. Until recently, the only performance measurement/monitoring tools on the Multi-PSI were the timer and the performance meter which shows how busy the individual processors have been for the past two seconds. The latter helped very much in pin-pointing problems in load balance (e.g., finding the bottleneck in dynamic load balancing schemes) and make changes. The Multi-PSI system has recently added a task-wise profiler (which records what predicates have been executed how many times on which processors) and a processor-wise profiler (which records the idling periods, the number of messages sent and received, etc.). Visual tools to display the monitoring results postmortem are under development.

5 Conclusions

I have overviewed parallel applications research at ICOT in the past couple of years. The first programs to run on the prototype parallel inference machine Multi-PSI were simple programs to solve simple well-defined problems. More recently, programs are being developed which are more application-oriented. Along with writing and testing programs, research in performance analysis is conducted. Sophisticated performance measuring/monitoring tools to help understand the runtime behavior of programs are becoming available. We hope to lay a solid foundation of the principles and practice of programs on a highly parallel computers in the rest of the FGCS project.

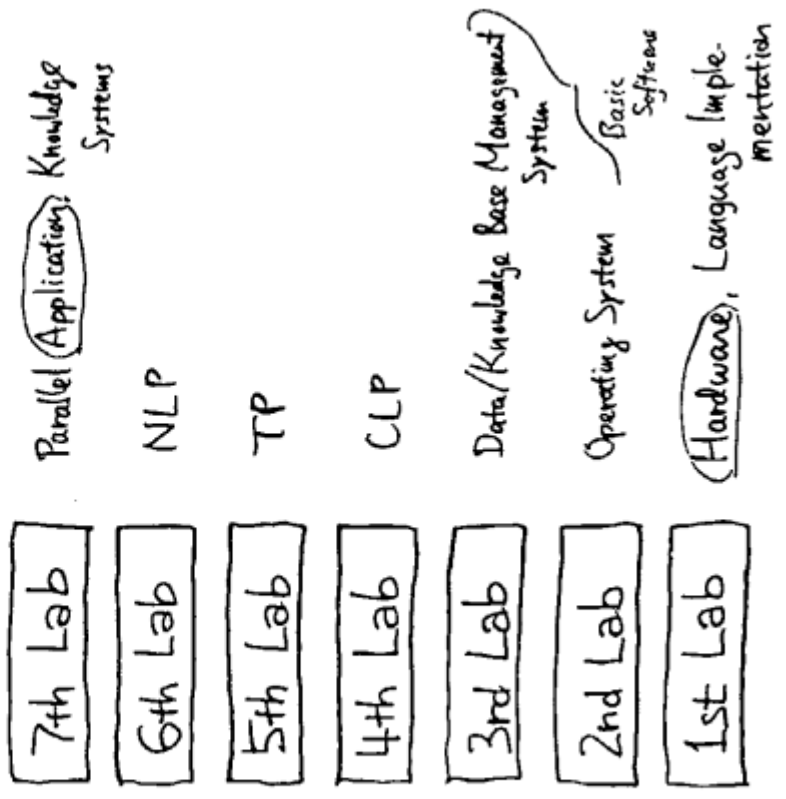
Meiko
AMT

"Real" Applications

Parallel Application
Program Research at
ICOT

Nobuyuki Ichiyoshi

Institute for New Generation
Computer Technology (ICOT)
Seventh Laboratory



Multi-PSI/V2

1988



16 PE_s ≈ 2 MLIPS / tatami-mat



PIM/m

1991



64 PE_s ≈ 20 MLIPS / tatami-mat

- Sequential systems**
- 1984 **PSI-I, KLO** 37K LIPS(KLO)
 - ESP, SIMPOS
 - 1986 **PSI-II** 330K LIPS(KLO)
 - (KLO append)
- Parallel systems**
- 1985 —GHC
 - 1986 **Multi-PSI/V1, FGHC** 1K LIPS×6PE(FGHC)
 - Small sample programs
 - 1988 **Multi-PSI/V2, KLI** 150K LIPS×64PE(KLI)
 - (KLI append)
 - PIMOS/V1, Demonstration programs
 - Now → 1990 **PIM/p** 600K LIPS×128PE(KLI)
 - PIMOS/V2, Application programs
 - 1992 **Final PIM system**
 - Final PIMOS, Large application programs

(LIPS : Logical Inferences per Second
 KLO, KLI, : Kernel languages
 GHC, FGHC
 SIMPOS, PIMOS : Operating Systems

Early Benchmark Programs (1)

- Simple programs to solve well-defined problems:

Packing Piece Puzzle (Pentomino)

Exhaustive search of a tree

Shortest Path Problem (Bestpath)

Graph algorithm for a sparse graph

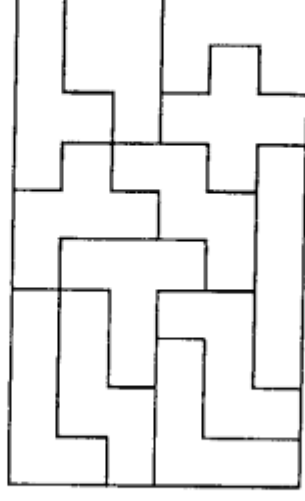
Game-Tree Search (Tsumego)

Alpha-beta pruning

Natural Language Parser (PAX)

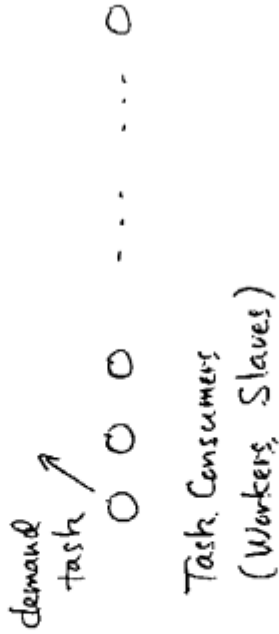
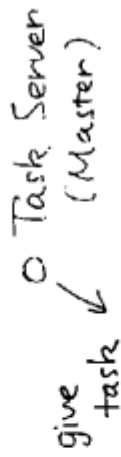
Chart parsing (dynamic programming)

Packing Piece Puzzle (Pentomino)



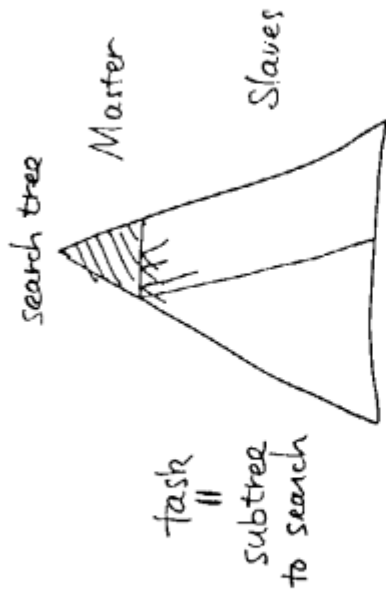
Dynamic Load Balancing in

the Pentomino Program

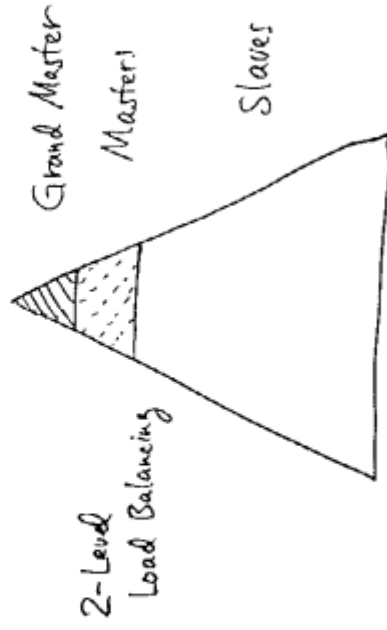


* All written in software using the priority scheduling mechanism and task dispatch mechanism

Multi-Level Load Balancing

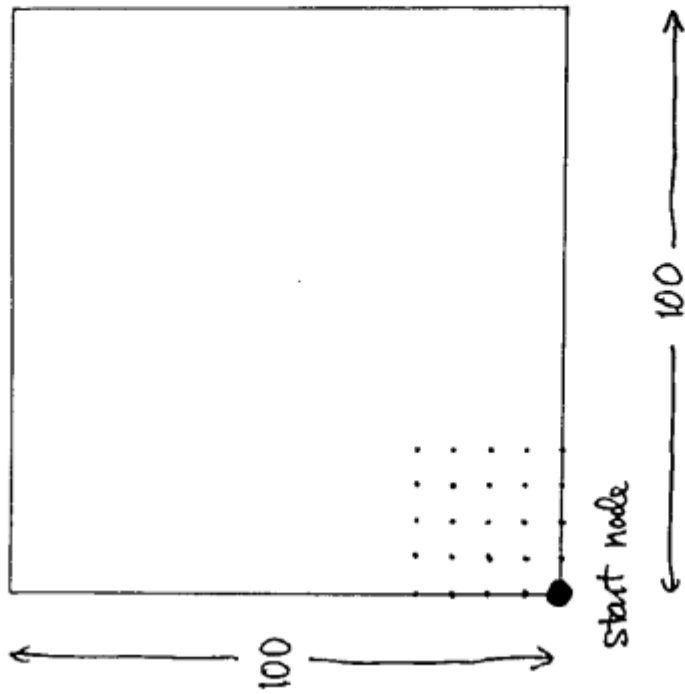


Master becomes bottleneck as #Slaves → large.



Bottleneck removed by hierarchical organisation.

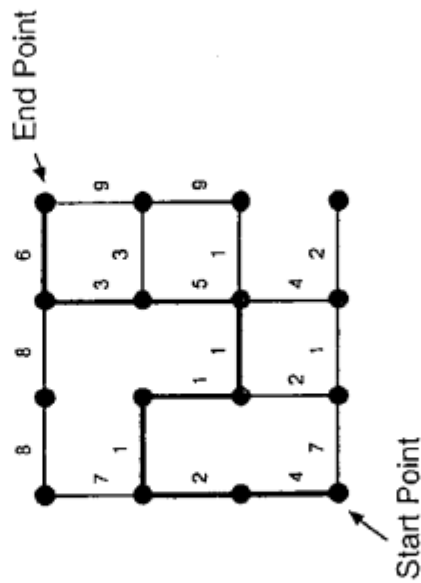
Mapping for Shortest Path



Problem

- 10,000 nodes (100×100)
- edges costs generated by random number generator
- Start node at bottom left

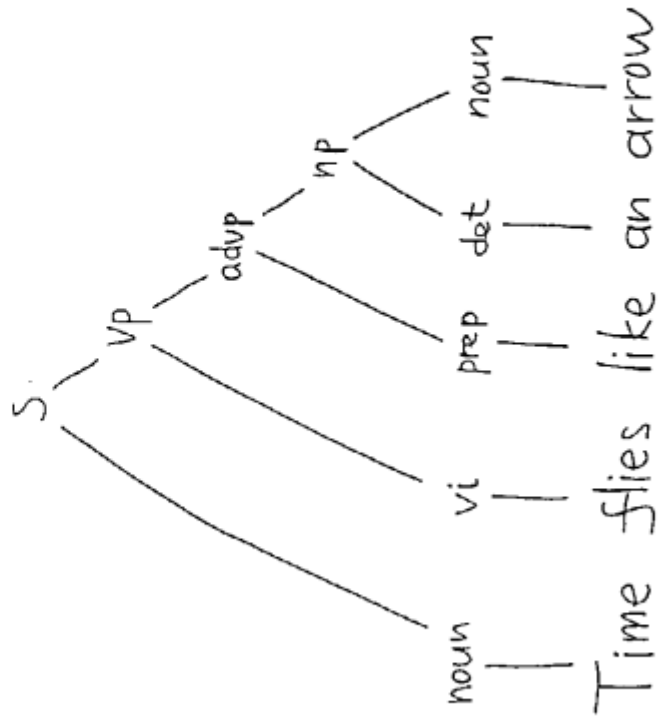
Shortest Path Problem



Vertex-to-Processor Mapping Schemes in the Bestpath Program

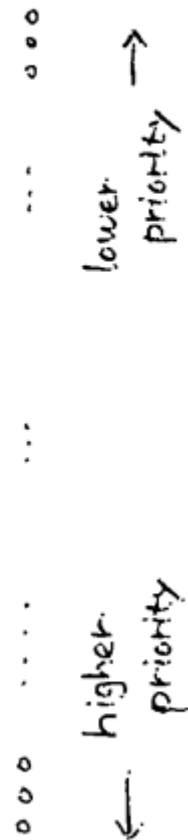
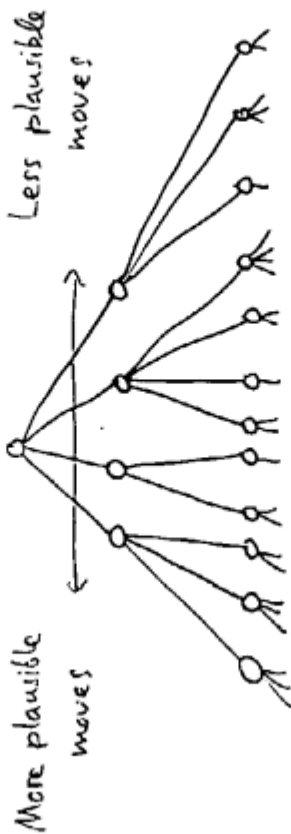
- Trade-off between
 - Communication Localization
- and
- Processor Utilisation

Natural Language Parser (PAX)

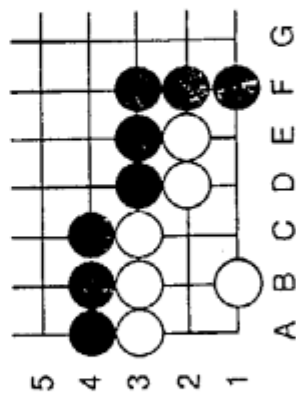


Priority Assignment in the Tsumego Program

Game Tree



Tsumego



More Recent Programs

- More application-oriented
- Larger scale (3,000~7,000 lines)

VLSI-CAD Programs

LSI chip routing program, Logic-level simulator

Genome Analysis Programs

Two algorithms for multiple sequence analysis

Legal Reasoning Program

Case-based reasoning

Go-playing Program

A "real time" program consisting of a number of situation-trigger problem solving routines

Early Benchmark Programs (2)

- Major issues in parallel programs on medium-scale multiprocessors were experienced:

Load balancing

Dynamic mapping, static mapping

Communication

Overhead, bottleneck

Speculative computation

Use of high-resolution priority for process scheduling

Performance Measurement/
Monitoring Tools

Performance meter

Real-time indicator of processor utilization.
Simple, but very useful in load distribution tuning.

Program profilers

- Which procedures were executed how many times on which processors
(Task-wise profiling)
- Log of processor idle times, message handling, user-defined events, etc.
(Processor-wise profiling)

Postmortem visualization tools are under development.

To Better Understand
Program Behavior
and Performance Debug

- Performance Measurement/
Monitoring Tools

Look ...

- Performance Analysis

and Think!

Performance Analysis

- is important, when you are dealing with a class of problems on a class of machines.

In particular,

Scalability analysis

- is important, when you are using (or, plan to use) scalable multiprocessors.
- Questions addressed by scalability analysis:
 - * "What will happen to your program when the number of processors doubles, triples, etc.?"
 - * "Will it be realistic to run your application on a 1,000 processor machine?"

Iso-efficiency

For a fixed problem class and a parallel algorithm:

$T(p, n)$... execution time to solve a problem of size n using p processors

$$\frac{T(1, n)}{T(p, n)} = S(p, n) \quad \text{Speedup}$$

$$\frac{S(p, n)}{p} \quad \text{efficiency}$$

★ In many cases, efficiency increases for a fixed p

as n increases.

i.e. Larger problems are easier to speedup.

Iso-efficiency (continued)

* For a fixed n , efficiency decreases as p increases.

Def. Isoefficiency function is the rate at which n must increase as p increases, in order to keep efficiency constant.
(due to V. Kumar)

Iso-efficiency (example)

One-Level Load Balancing



Master: creation/dispatch of p subtasks in $1/p$ of time

$$\Rightarrow \text{iso. eff.} = O(p^2)$$

Two-Level Load Balancing
 $O(p^{3/2})$... Better

N-Level
 $O(p^{N/N})$

Conclusions

We need to understand program performance better!

- Questions to ask yourself:
 - “Do the processors work most of the time?”
 - “What is the communication overhead?”
 - “Are there bottlenecks (sequential bottlenecks, communication hotspots, etc.)?”
 - “In short, does my program run just as I expected?”

i.e. Are there performance bugs?

⇒ Performance monitoring

- More questions to ask:
 - “How much speculative computation?”
 - “What will the speedup be, if I increase the number of processors twice?”
 - “Am I better off with larger problems?”
 - “What if I run this program on different architectures?”

⇒ Scalability analysis

MGTP: A Hyper-matching Model-Generation Theorem Prover with Ramified Stacks

Ryuzo HASEGAWA, Tadashi KAWAMURA, Masayuki FUJITA

Fifth Laboratory, ICOT

Hiroshi FUJITA

Mitsubishi Electric Corporation

Miyuki KOSHIMURA

JBA Co., Ltd.

October 1, 1990

The research on theorem proving in the FGCS project aims to develop a parallel automated reasoning system applicable to various fields such as natural language processing, intelligent database designing, and automated programming on the Parallel Inference Machine (PIM).

In this paper, we will present a parallel theorem prover for first-order logic implemented in KL1, and the KL1 implementation techniques which are useful for other related areas, such as truth maintenance systems and intelligent database systems.

The MGTP prover, which has already being developed, adopts a model generation method, as used in SATCHMO presented by Manthey and Bry. SATCHMO tries to find ground models for the given set of clauses that satisfies a condition called *range-restrictedness*. The condition imposed on the clause set allows us to use only matching rather than unification during the proving process.

This property is also favorable in implementing a prover in KL1 since matching is easily realized with head unification, and hence allows us to implement a model-generation based prover in KL1 very easily yet effectively.

To improve the efficiency of the MGTP prover, we developed an algorithm with *ramified stacks* for removing redundancy in matching literals in a clause against elements of a model candidate.

Experimental results show that the enhanced MGTP prover can achieve orders of magnitude speedup compared to the naive version without using the ramified stacks in the pseudo parallel environment on the PSI-II machine. Moreover it is expected to attain further speedup by giving priority to negative clauses and by employing a pruning mechanism like relevancy checking. Several experiments are being carried out in order to verify the effectiveness of MGTP in a parallel environment on the MULTI-PSI system.

With MGTP, we can solve a large class of problems very efficiently. To deal with more difficult problems hard to solve with this type of prover, such as the Lukasiewicz' problem, MGTP is extended by incorporating unification with occurs check, weighting heuristics, and other deletion strategies while still keeping a model generation paradigm.

Model Generation

MGTP:

A Hyper-Matching Model-Generation

Theorem Prover with Ramified Stacks

- To construct a model for a given clause set

$$A_1, \dots, A_m \rightarrow C_1; \dots; C_n$$

true $\rightarrow C_1; \dots; C_n$ (positive clause)

$A_1, \dots, A_m \rightarrow$ false (negative clause)

Extension Rule

If $A\sigma$ in a clause $(A \rightarrow C)$ is satisfied but not $C\sigma$ in a model M , then extend M with $C\sigma$

Rejection Rule

If $A\sigma$ in a negative clause $(A \rightarrow \text{false})$ is satisfied in a model M , then reject M .

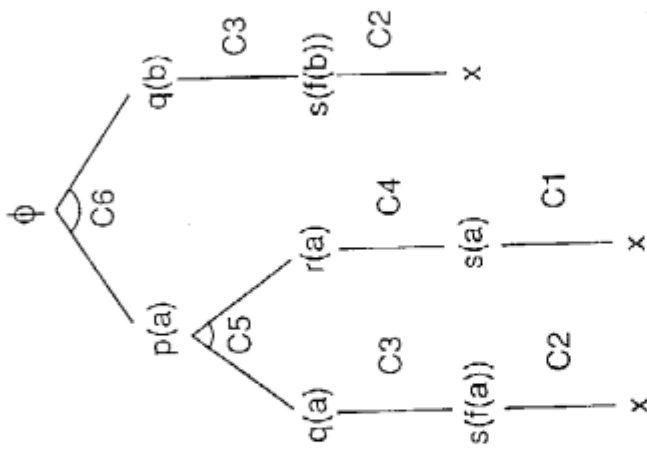
Ryuzo HASEGAWA
Tadashi KAWAMURA
Masayuki FUJITA

Institute for New Generation Computer Technology

Hiroshi FUJITA
Mitsubishi Electric Corporation

Miyuki KOSHIMURA
JBA Co., Ltd.

S1 Problem



C1 : $p(X), s(X) \longrightarrow \text{false.}$

C2 : $q(X), s(Y) \longrightarrow \text{false.}$

C3 : $q(X) \longrightarrow s(f(X)).$

C4 : $r(X) \longrightarrow s(X).$

C5 : $p(X) \longrightarrow q(X) ; r(X).$

C6 : $\text{true} \longrightarrow p(a) ; q(b).$

Implementation Problems

How to handle object-level variables

(1) KL1 ground terms

- Clear semantics
- Needs to write variable management routines
- Makes programs complex and inefficient

\longrightarrow Partial Evaluation

(2) KL1 variables

- Ambiguous semantics
 - Utilizes built-ins offered by the underlying system
 - Makes programs simple and efficient
 - KL1 built-in unification is restricted
- \longrightarrow Model generation (SATCHMO)

SATCHMO

Merits:

- Treats ground models
- Unification is not necessary (needs only matching)

KL1 implementation

- KL1 variables can be used for representing object-level variables
- Head unification can be used for matching

Techniques for Implementing MGTP in KL1

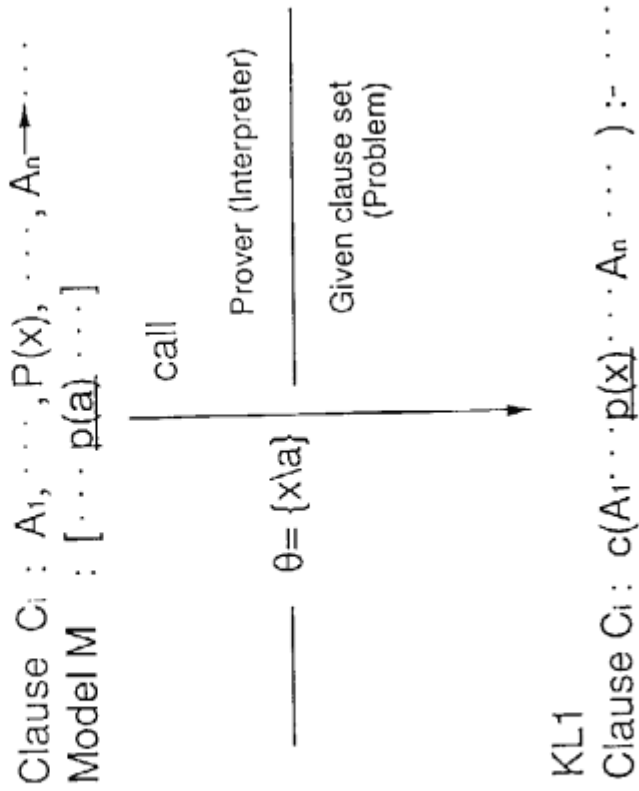
- Translate a given set of clauses into a corresponding set of KL1 clauses.
- Use head unification to realize conjunctive matching in model generation
- Obtain fresh variables by calling a KL1 clause.



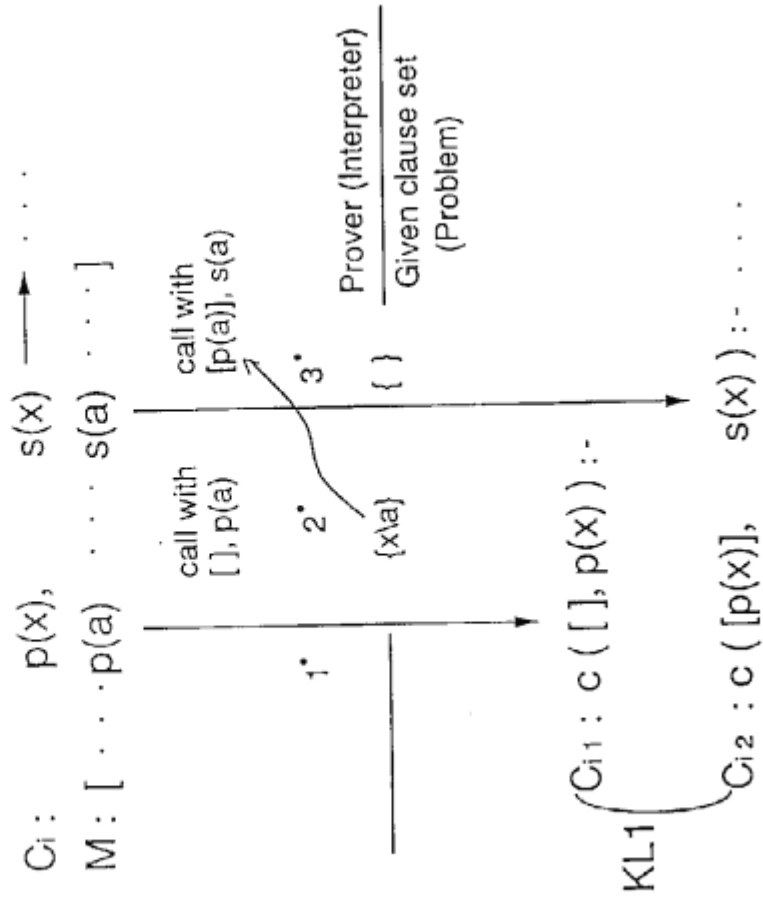
Simple yet Efficient
MGTP Interpreter

Matching a Literal against a Model Element

- Can be realized with head unification -



Passing a Substitution when Variables Shared among Literals



KL1 Representation for S1 Problem

```
:- module mgtp _problem.

:- public model / 1, nc / 1, c / 4.

model(M) :- true | M = [].

nc(NC) :- true | NC = 6.

c(1, p(X), [], R) :- true | R = cont.
% C1 : p(X), s(X) → false.

c(1, s(X), [p(X)], R) :- true | R = false.

c(2, q(X), [], R) :- true | R = cont.
% C2 : q(X), s(Y) → false.

c(2, s(Y), [q(X)], R) :- true | R = false.
% C3 : q(X) → s(f(X)).

c(3, q(X), [], R) :- true | R = [s(f(X))].
% C4 : r(X) → s(X).

c(4, r(X), [], R) :- true | R = [s(X)].
% C5 : p(X) → q(X) ; r(X).

c(5, p(X), [], R) :- true | R = [q(X), r(X)].
% C6 : true → p(a) ; q(b).

c(6, true, [], R) :- true | R = [p(a), q(b)].

otherwise.

c(_ _ _ _ R) :- true | R = fail.
```

```

:-module mgtp. :-public do/1.

do(A):-true|
    mgtp_problem:model(M),
    mgtp_problem:nc(N),
    satisfy_clauses(O,N,M,A,_).

satisfy_clauses(_,_,_,_,_unsat):-true|true. alternatively.
satisfy_clauses(J,N,M,A,B):-J<N,J1:=J+1|
    satisfy_ante(J,□,[true|M],M,A1,B),
    and_sat(A1,A2,A,B),
    satisfy_clauses(J1,N,M,A2,B).
satisfy_clauses(N,N,_,_A):-true|A=sat.

and_sat(sat,sat,A,_):-true|A=sat.
and_sat(unsat,_A,B):-true|A=unsat,B=unsat.
and_sat(_unsat,A,B):-true|A=unsat,B=unsat.

satisfy_ante(_,_,_,_,_unsat):-true|true. alternatively.
satisfy_ante(J,S,[P|M2],M,A,B):-true|
    mgtp_problem:c(J,P,S,R),
    satisfy_ante1(J,R,P,S,M2,M,A,B).
satisfy_ante(_,_□,_A):-true|A=sat.

satisfy_ante1(J,fail,P,S,M2,M,A,B):-true|
    satisfy_ante(J,S,M2,M,A,B).
satisfy_ante1(J,cont,P,S,M2,M,A,B):-true|
    satisfy_ante(J,[P|S],M,M,A1,B),
    and_sat(A1,A2,A,B),
    satisfy_ante(J,S,M2,M,A2,B).
satisfy_ante1(J,false,P,S,M2,M,A,B):-true|B=A,A=unsat.
satisfy_ante1(J,F,P,S,M2,M,A,B):-list(F)|
    satisfy_cnsq(F,F,M,A1,B),
    and_sat(A1,A2,A,B),
    satisfy_ante(J,S,M2,M,A2,B).

satisfy_cnsq(_,_,_,_,_unsat):-true|true. alternatively.
satisfy_cnsq([D1|Ds],F,M,A,B):-true|satisfy_cnsq1(D1,Ds,F,M,M,A,B).
satisfy_cnsq(□,F,M,A,_):-true|
    mgtp_problem:n(N),
    extend_model(F,M,N,A,_).

satisfy_cnsq1(D,Ds,F,[D|M2],M,A,_):-true|A=sat.
satisfy_cnsq1(D,Ds,F,□,M,A,B):-true|satisfy_cnsq(Ds,F,M,A,B). otherwise.
satisfy_cnsq1(D,Ds,F,[_M2],M,A,B):-true|satisfy_cnsq1(D,Ds,F,M2,M,A,B).

extend_model(_,_,_,_sat):-true|true. alternatively.
extend_model([D|Ds],M,N,A,B):-true|
    satisfy_clauses(O,N,[D|M],A1,_),
    and_unsat(A1,A2,A,B),
    extend_model(Ds,M,N,A2,B).
extend_model(□,_,_A):-true|A=unsat.

and_unsat(unsat,unsat,A,_):-true|A=unsat.
and_unsat(sat,_A,B):-true|A=sat,B=sat.
and_unsat(_sat,A,B):-true|A=sat,B=sat.

```

A MGTP interpreter in KL1

Performance Comparison for Ground Models

Problem	S1	S2*	S3**
PtTP†	86 [ms]	24 [s]	>30 [min]
SATCHMO†	16 [ms]	68 [ms]	6.3 [s]
MGTP-R‡	3 [ms]	66 [ms]	319 [ms]
(Number of reductions)	(210)	(4,021)	(21,743)

† Sicstus Prolog V0.6 on SUN3/260

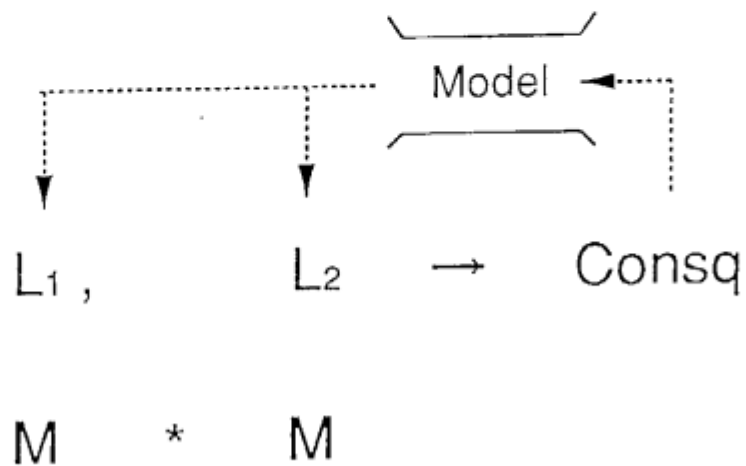
‡ Pseudo-Multi-PSI on PSI-II (1PE)

* S2: Schubert's Steamroller problem

** S3: Pelletier and Rudnicki's problem

Avoiding Redundancy

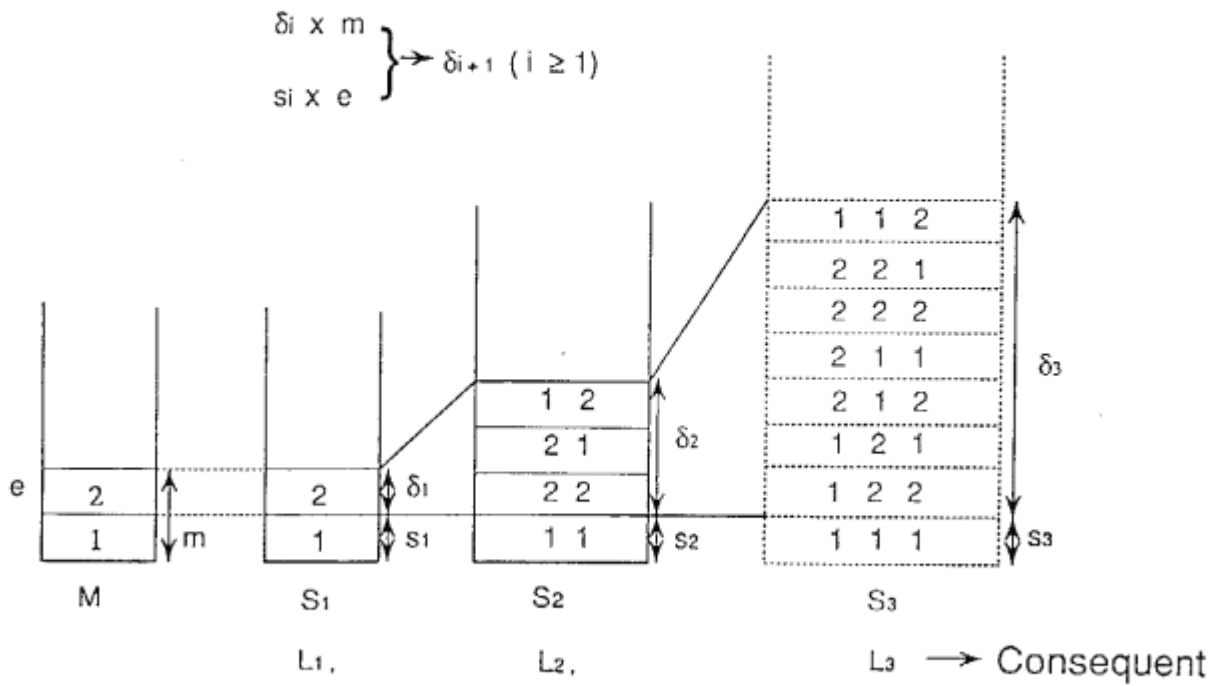
Redundancy in conjunctive-matching



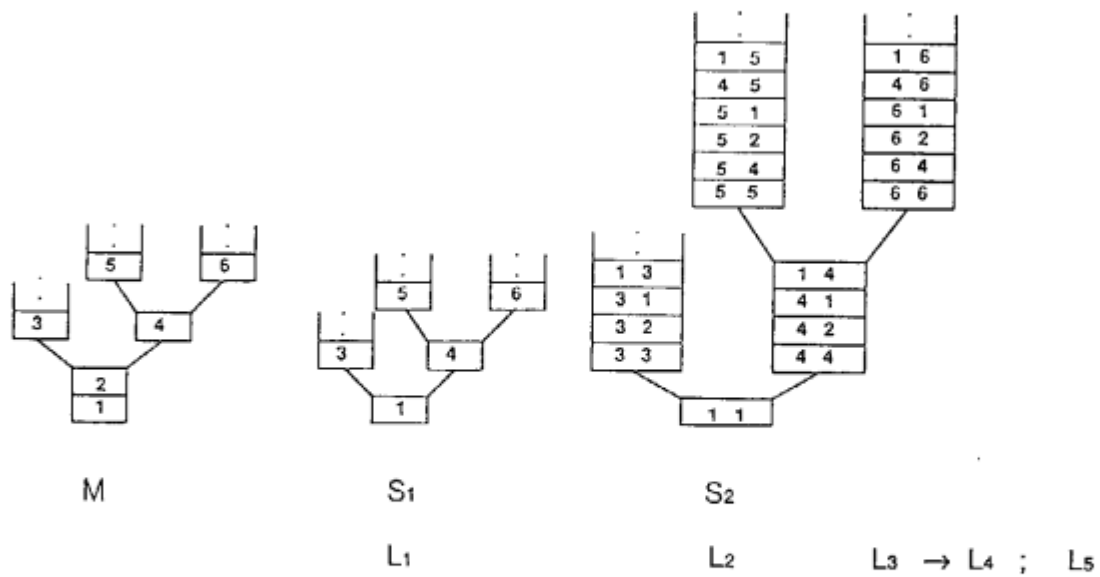
stage n $(M + \Delta) * (M + \Delta)$

stage n+1 $= \underline{M * M} + \Delta * M + M * \Delta + \Delta * \Delta$
 redundant

Literal Instance Stacks



Literal Instance Stacks (Non-Horn Case)



Performance Improvement
with a Ramified-Stack Algorithm

Parallel MGTP

Program	SATCHMO	mgtp-N (Naive)	mgtp-R (Ramified -stack)	mgtp-N / mgtp-R
4 Queens*	[ms]	[ms]	[ms]	
One	1519	436	22	19.8
All	3480	849	40	21.2
5 Queens*	[ms]	[ms]	[ms]	
One	5300	1800	14	129.0
All	50300	13654	195	70.
6 Queens*	[ms]	[ms]	[ms]	
One	92060	92631	153	605.0
All	899400	253418	650	390.0
8 Queens*			[ms]	
One	—	—	901	—
All	—	—	12538	—

- Sources of Parallelism

- multiple model candidates
- multiple clauses
- multiple literal instances

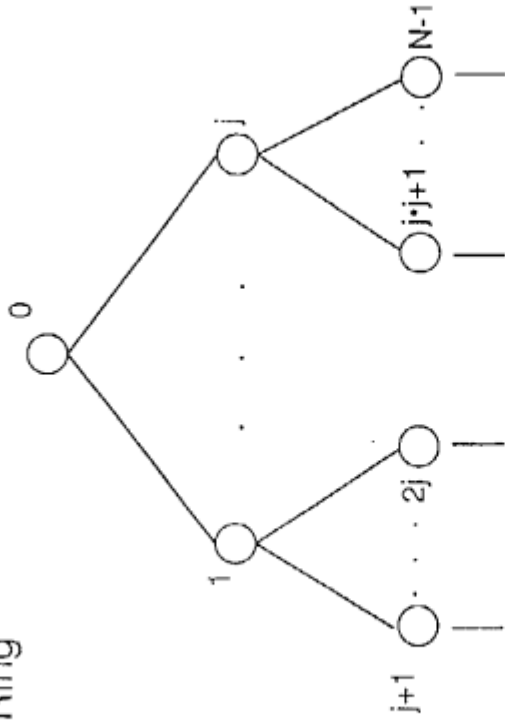
- MGTP can exploit OR-parallelism

considerably well for range-restricted
non-Horn problems.

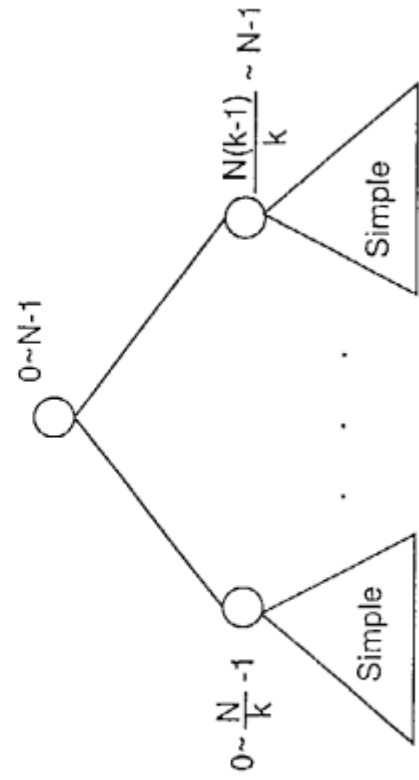
* Representation 1

PE Allocation(Continued)
(Bounded-OR Parallelism)

2. Ring

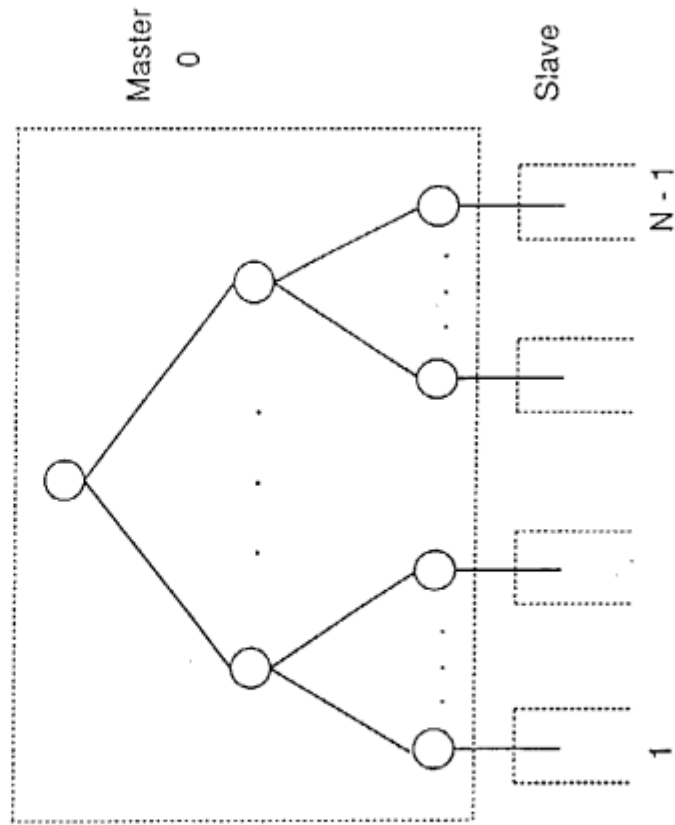


3. Group



PE Allocation
(Bounded-OR Parallelism)

1. Simple (Single Master PE)



Performance of MGTP-R on Multi-PSI
(Pigeon Hole)

Pigeon Hole Problem

		hole		
		1	2	3
1		p (1, 1)	p (1, 2)	p (1, 3)
2		p (2, 1)	p (2, 2)	p (2, 3)
3		p (3, 1)	p (3, 2)	p (3, 3)
4		p (4, 1)	p (4, 2)	p (4, 3)
	pigeon			

(N = 3)

true \rightarrow p(1, 1) ; p(1, 2) ; ... ; p(l, N)

...

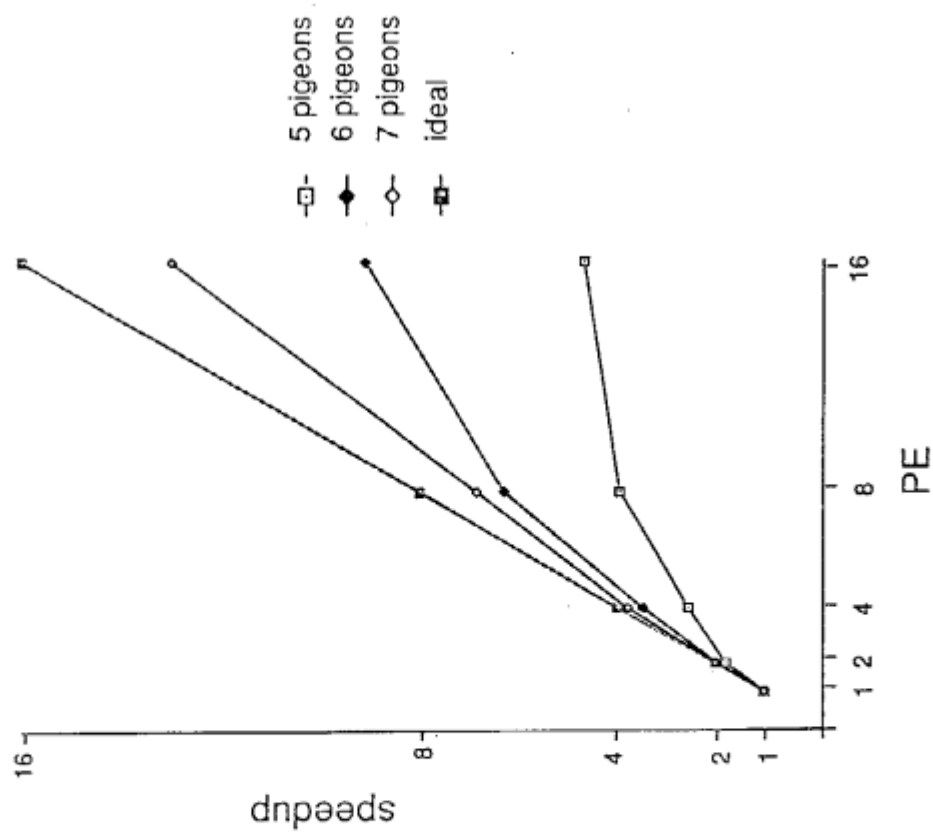
true \rightarrow p(N+1, 1) ; p(N+1, 2) ; ... ; p(N+1, N)

p(X₁, Y), p(X₂, Y), X₁ \neq X₂ \rightarrow false

Program	Number of processors					
	1	2	4	8	16	
5 pigeons						
time [ms]	1173	659	464	298	251	
speedup	1.00	1.78	2.53	3.94	4.67	
Kred	46	46	46	46	46	
6 pigeons						
time [ms]	8964	4624	2578	1429	988	
speedup	1.00	1.94	3.48	6.27	9.07	
Kred	356	356	356	356	356	
7 pigeons						
time [ms]	79238	39668	20957	11617	6054	
speedup	1.00	2.00	3.78	6.82	13.0	
Kred	3110	3110	3110	3100	3100	

Speedup Ratio of MGTP-R (Pigeon Hole)

N - queens Problem

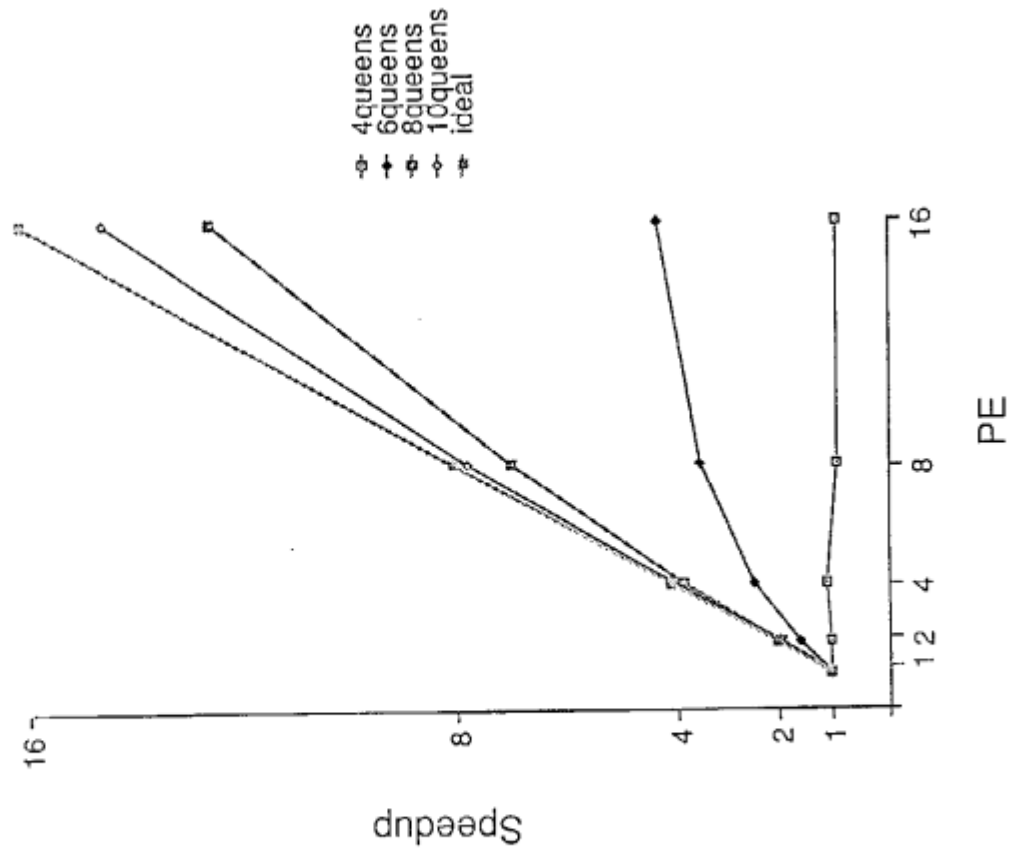


true → p(1,1); p(1,2); ... ; p(1,N)
 .
 .
 .
 true → p(N,1); p(N,2); ... ; p(N,N)
 P (X₁, Y₁) , p (X₂, Y₂) , not safe (X₁, X₂, Y₁, Y₂)
 → false

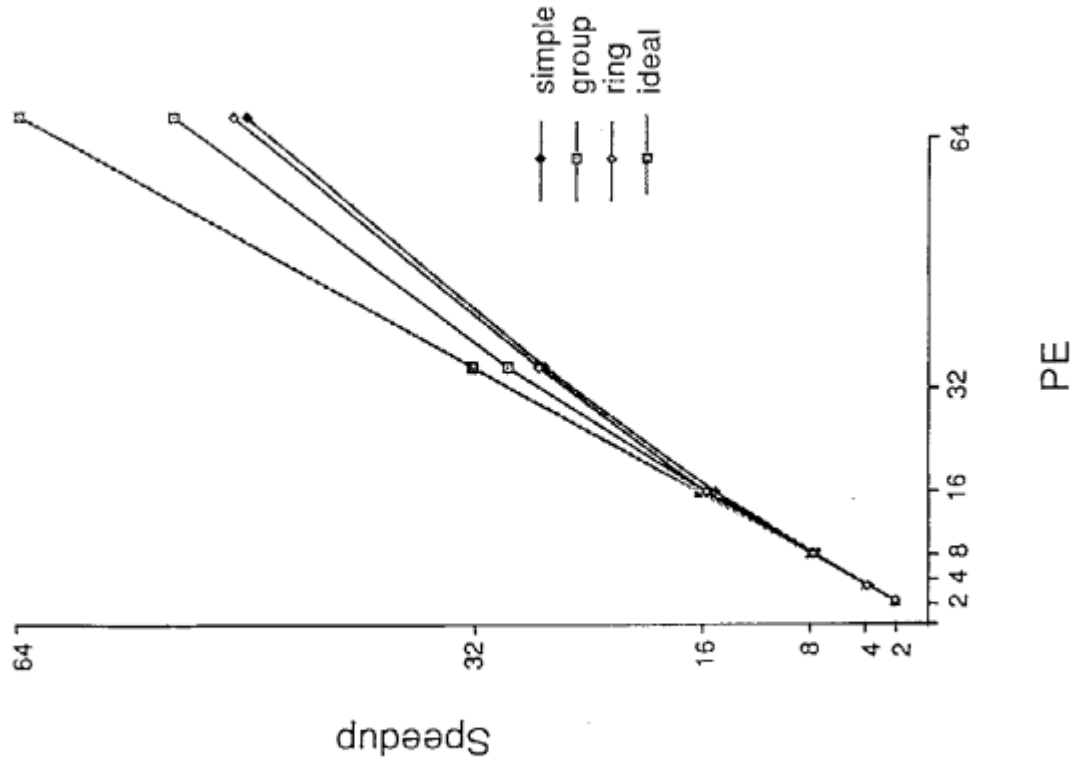
Performance of MGTP-R on Multi-PSI
(N Queens)

Program	Number of processors				
	1	2	4	8	16
4 Queens					
time [ms]	40	40	39	44	44
speedup	1.00	1.00	1.02	0.90	0.90
Kred	1.45	1.47	1.48	1.50	1.50
6 Queens					
time [ms]	650	407	266	189	154
speedup	1.00	1.59	2.44	3.44	4.22
Kred	23.7	23.7	23.7	23.8	23.8
8 Queens					
time [ms]	12538	6425	3336	1815	1005
speedup	1.00	1.95	3.76	6.91	12.5
Kred	460	460	460	460	460
10 Queens					
time [ms]	315498	159881	79921	40852	21820
speedup	1.00	1.97	3.94	7.72	14.50
Kred	11117	11117	11117	11117	11117

Speedup Ratio of MGTP-R
(N Queens)



Speedup Ratio for 12 Queens Problem



Conclusion

- Model-generation provers can be implemented in KL1 very simply yet effectively.
- A Rete-like algorithm enables us to avoid redundant hyper-matching in the model-generation provers.
- Parallel model-generation provers are considerably efficient in solving range-restricted non-Horn problems.

Parallel Programming in LSI CAD Systems

Kazuo Taki

e-mail address: taki@icot.or.jp

Institute for New Generation Computer Technology

1-4-28 Mita, Minato-ku, Tokyo 108, Japan

1. Focus of the talk

This talk reports the recent R & D status of parallel LSI CAD programs in ICOT. They are a router [1] and a logic simulator [2]. The talk will include two major aspects. One is the R & D status reports of practical application programs on our parallel machines, on the Multi-PSI and the PIM. The programs are written in our concurrent logic language, KL1. The other is to feature KL1 programming such as making a model of a problem with concurrent objects, its distributed implementation, separation of problem solving algorithm and load mapping onto physical processors, etc. They are shown along with examples of the developed software. Preliminary performance measurements will be also included.

You can find a description of the software, including problem descriptions, algorithms, program structures, and load mapping schemes, in the résumé of the Multi-PSI demonstrations [1][2]. Please look at it with this abstract.

2. Two LSI CAD programs

Two LSI CAD programs have been developed, a router and a logic simulator. These problems are well known as the most time consuming applications in LSI CAD field. That is, speed up and applicability to practical large problems are very important. Using these problems, the real performance and applicability of our machines, and easiness of writing in KL1 language are under examination.

The routing program is based on a line search method [3]. The search problem includes dynamic characteristics in nature, which fits the KL1 programming. New formulation of routing problems was taken for higher concurrency (described in next section). Both good absolute speed and speed up ratio are expected. The second version of the program has just been working, and will be shown in the demonstration. A real routing problem will be used for it.

The logic simulation program is based on the event simulation and the virtual time mechanism. It simulates the behavior of logic circuits described in the logic-gate level, taking delay time of each gate into account. The virtual time mechanism realizes the local time management in each processor although the logic circuit contains loop structures. An event history is recorded in each gate. When a time inconsistency occurs, the event history is rolled back and the simulation is replayed. Logic simulation programs in KL1 language are not easy to get good absolute performance. Because the KL1 language function is sometimes too high-level to write simple computation of a logic gate behavior, that is, basic implementation cost of the language tends to reduce the absolute performance. Virtual time mechanism is a novel trial in logic simulators, which evaluation will be an interest. Good speed up ratio is also expected. The second version of the program has just been working. It will be shown in the demonstration. Published benchmark problems will be used for it.

3. Feature of KL1 programming with examples

One of the most general programming styles in KL1 is to describe concurrent objects that communicate each other through streams. Formulation of the routing problem is leaded naively from the programming style of KL1 based on concurrent objects.

Formulation of a routing problem

The routing problem is to connect terminal pairs using two wiring layers without path conflict. A line search method, which uses a virtual grid, was assumed as the basic routing method for efficiency and general applicability. In the line search method, lines must be drawn along with the grid lines. The routing problem was modeled as follows. Any lines, already drawn lines, empty lines or grid lines, are modeled as independent objects. Line objects, crossing each other, have communication streams corresponding to cross points. The line search to find a good path for a terminal pair is executed by these line objects with exchanging search messages. The formulation can have much concurrency because each line objects can work independently according to received messages. Since the objects correspond to lines, the formulation can keep efficiency of the line search method. Two different types of concurrency is expected. One is found in the line search algorithm to connect a terminal pair. The other is in concurrent routing of different terminal pairs.

Process-oriented implementation of a router

These formulation can be implemented directly in KL1 with small runtime cost. Line objects are implemented as independent processes connecting each other with

communication streams. Status of each line, occupied (drawn line), empty, or under searching, is kept in a corresponding line process. Processes execute the routing with exchanging messages. Line drawing and rip-up correspond to dynamic split and joint of these line processes with changing the process status. Programmers can separate the writings of these problem solving algorithm from the load distribution code. Debug of the problem solving code is usually done before attaching load distribution code in KL1 programming. That is, the logical correctness of the problem solving algorithm is tested independently from load mapping.

Process allocation in a router

After that, annotations for process allocation to physical processors are attached. The annotation is called pragma. Communication and synchronization across the processor boundary are automatically maintained by KL1 language system. Programmers only have to concentrate on the process allocation. Communication locality and load balance are taken into account in this step.

Communication cost across the processor boundary is approximately twenty times higher than a inter-process communication within a processor. So, a ratio of the inter-processor communication cost and average computational amount in processes corresponding to each message is very important to estimate the inter-processor communication overhead. For the better load balance, line processes should be distributed among processors at random. However, a random distribution arises much inter-processor communication. When a expected communication overhead is not low enough, locally communicating processes should be grouped and mapped onto a same processor in order to reduce the inter-processor communication overhead. When the load balance or communication overhead is not good enough, pragmas are changed to improve them without changing main body of the program. These are the features of KL1 programming.

References

- 1 "Parallel LSI-CAD demonstration program (1) - LSI router", in a résumé of Guide to the Japanese Demonstrations, in JOINT ICOT/DTI-SERC WORKSHOP, Oct.15-17, 1990
- 2 "Logic-level simulator of LSI circuits: A parallel application program in LSI CAD", in a résumé of Guide to the Japanese Demonstrations, in JOINT ICOT/DTI-SERC WORKSHOP, Oct.15-17, 1990
- 3 Kitazawa,H. and Ueda,K., "A Look-ahead Line Search Algorithm With High Wireability for Custom VLSI Design", proc. of ISCAS 85, pp1035

Parallel Programming in LSI CAD Systems

Two LSI CAD Programs

1. Parallel LSI Router
2. Parallel Logic Simulator
 - Time consuming CAD applicatios

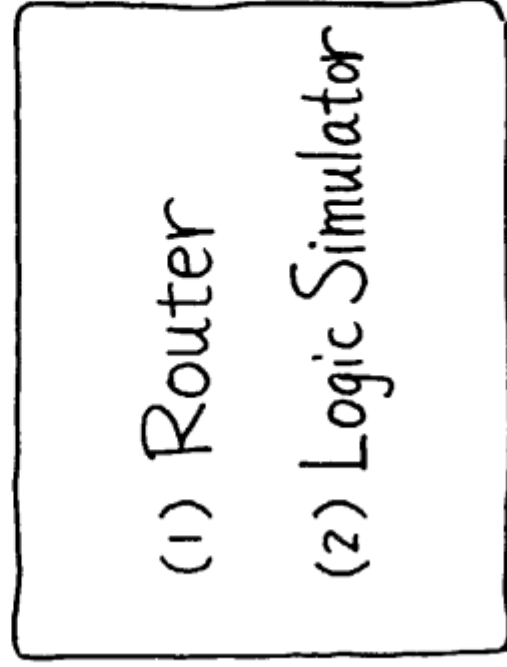
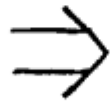
Purpose of R & D

- Realize high speed CAD tools
- Design efficient concurrent algorithms and mapping schemes
- Evaluate our concurrent language and parallel processing systems

KAZUO TAKI

ICOT 7th Lab.

KL1 Programming,
Process of modeling,
implementing, and mapping



Problem
explanation

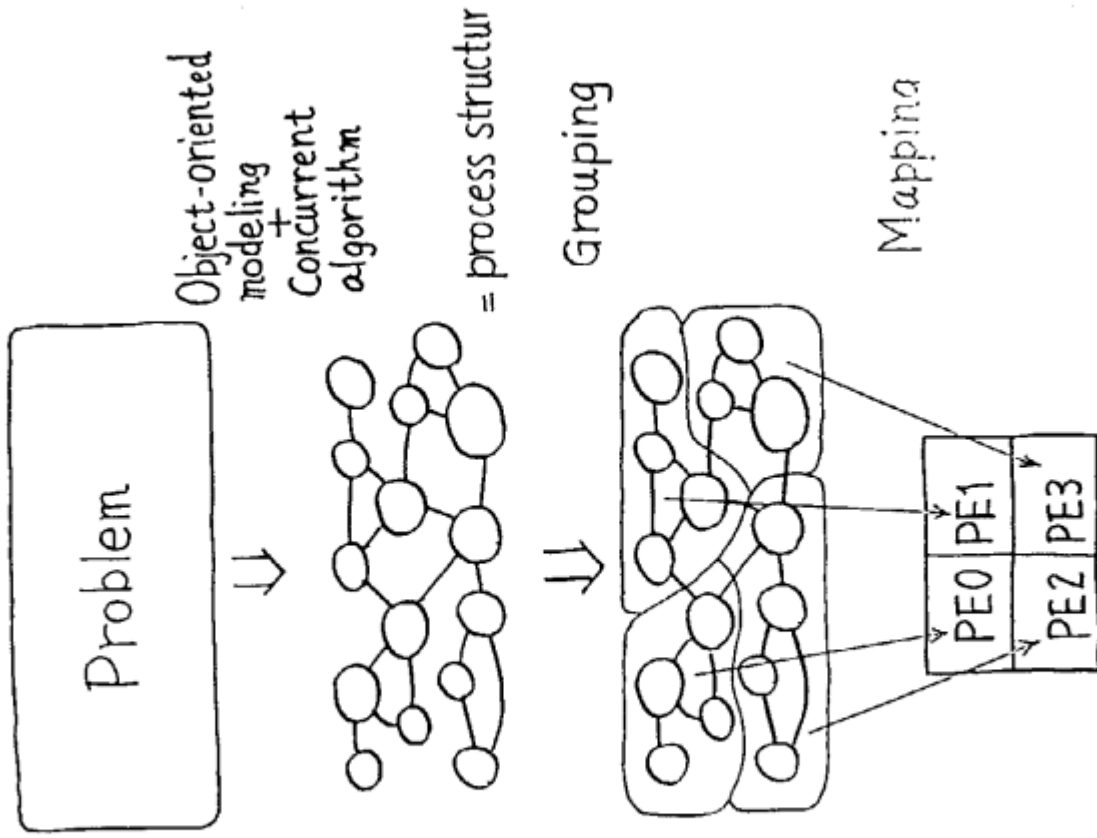


Algorithm
Implementation
Performance

Think in KLI

A typical way

1. Formalize a problem as concurrent objects communicating each other
 - Design a concurrent algorithm on it
2. Implement them in communicating processes with message streams
 - Locally communicating processes are grouped and mapped on a same processor
3. Map these processes on processors
 - Locally communicating processes are grouped and mapped on a same processor



`compute(I, 0) :- true |`

`compute1(I, S1) ,
compute2(S1, S2) ,
compute3(S2, 0) .`

Process of KL1 Programming

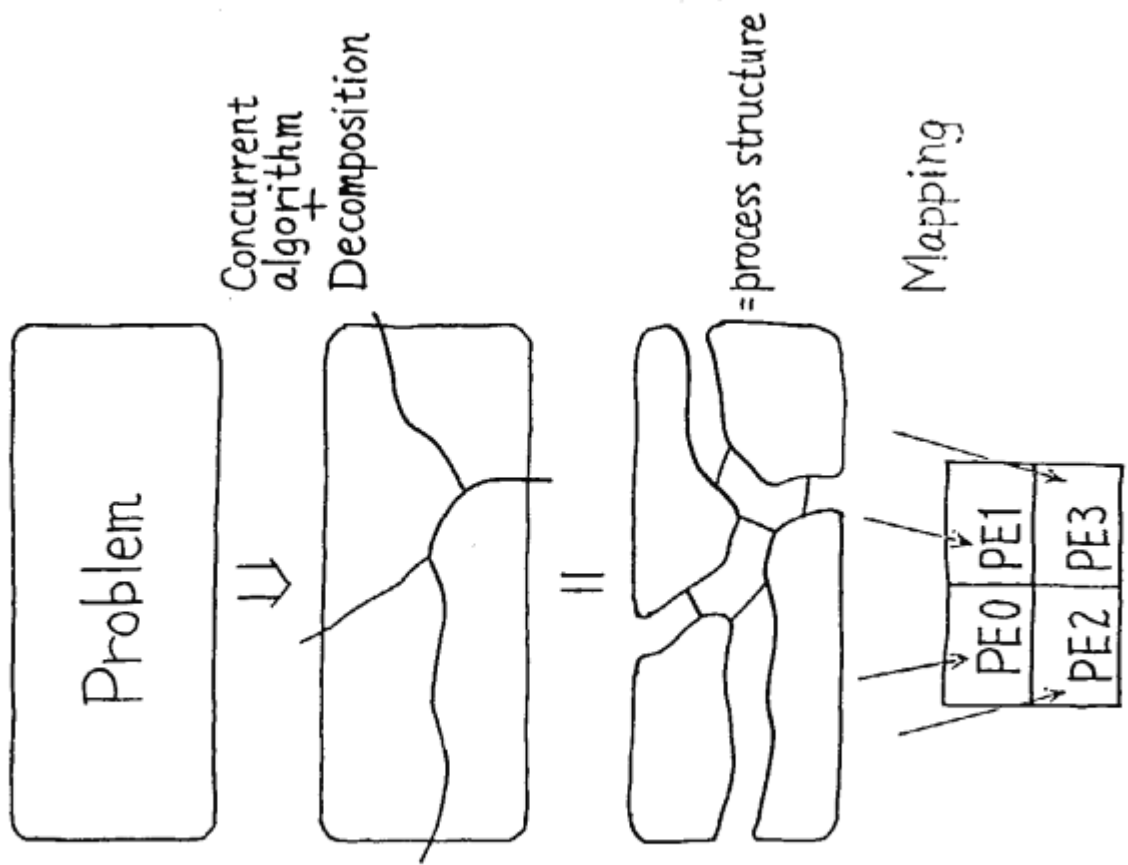
1. Analyze and formalize a problem
2. Design a concurrent algorithm
 - Low computational complexity
 - Communication locality
3. Implement it in communicating processes
without mapping and scheduling code
= without "pragma"
4. Debug it (debug the problem solving part of the program)
5. Design and implement the mapping and scheduling code
= attach the pragma
6. Test on a real machine

```

compute(I, 0) :- true |
  compute_mapping(I, X, Y),
  compute1(I, S1),
  compute2(S1, S2)@node(X),
  compute3(S2, 0)@node(Y).

```

Problem decomposition - based implementation



Routing problem

Decides connection paths between terminals of circuit blocks on a LSI surface

Several routing methods

Maze routing, Line search, Channel routing, etc. are well-known.

Basic algorithm in our program

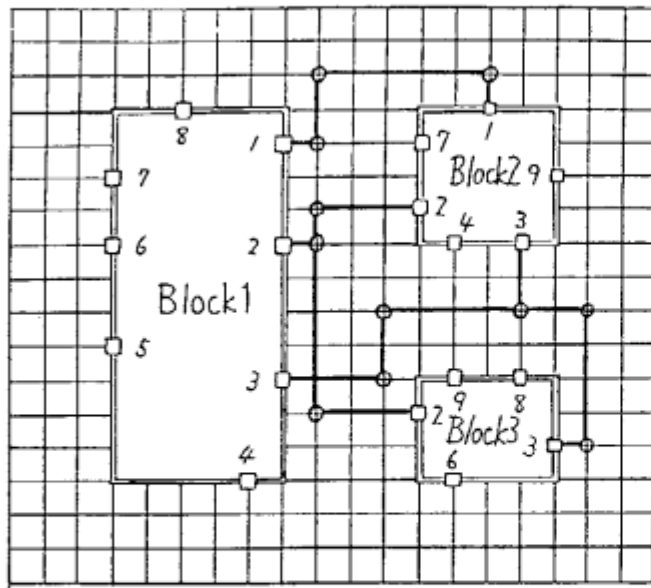
Look-ahead line search

- Efficient and widely applicable

⇒ Parallelize it for faster routing

Parallel LSI Router

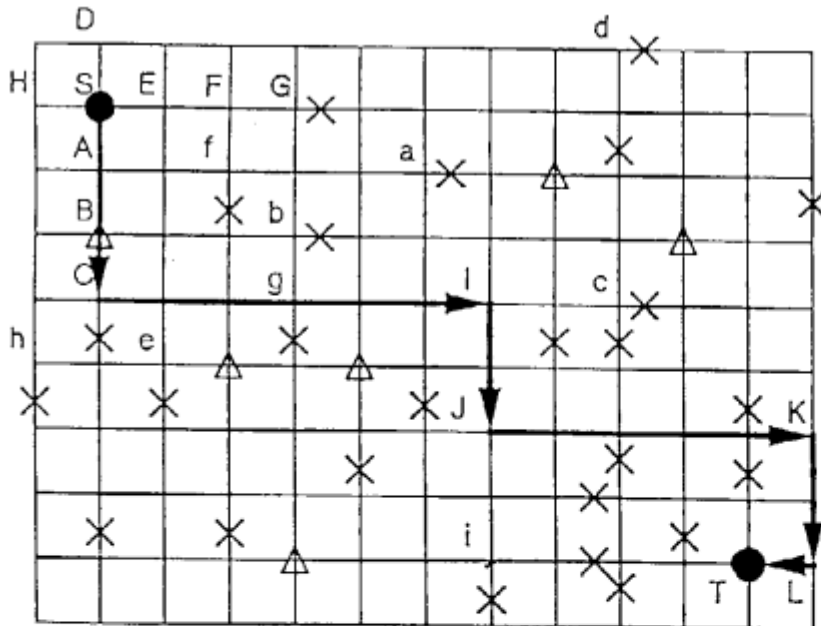
An example of "thinking in KLI"



- virtual grid
- two layers
- via hole
- block
- terminal

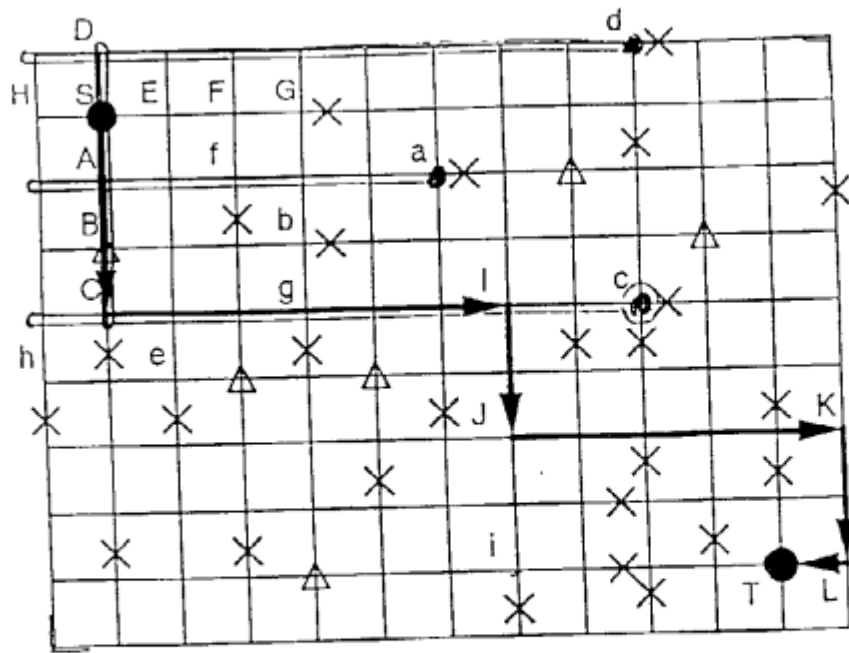
LSI surface

Routing problem



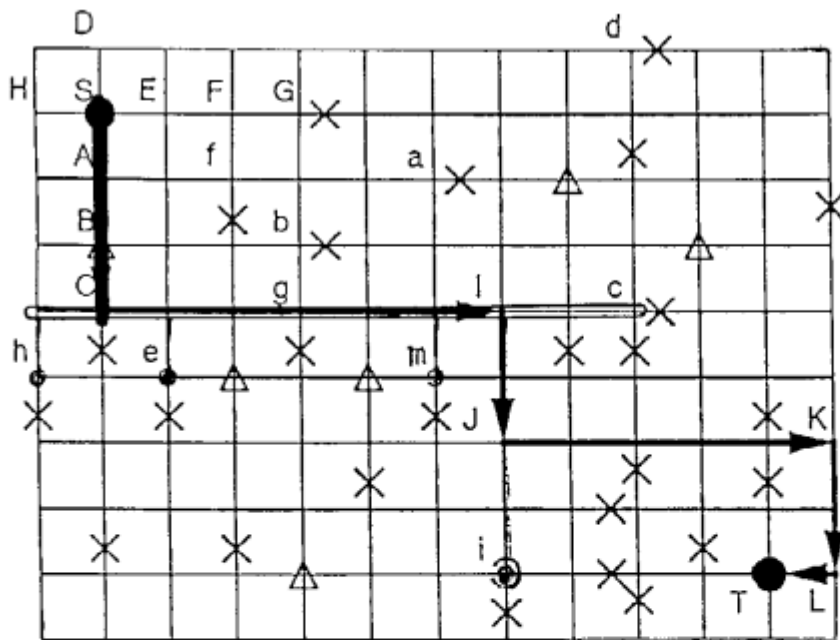
- × pass through inhibition
- Δ via hole inhibition
- S start
- T target

Look-ahead line search method



- X pass through inhibition
- △ via hole inhibition
- S start
- T target

Look-ahead line search method



- X pass through inhibition
- △ via hole inhibition
- S start
- T target

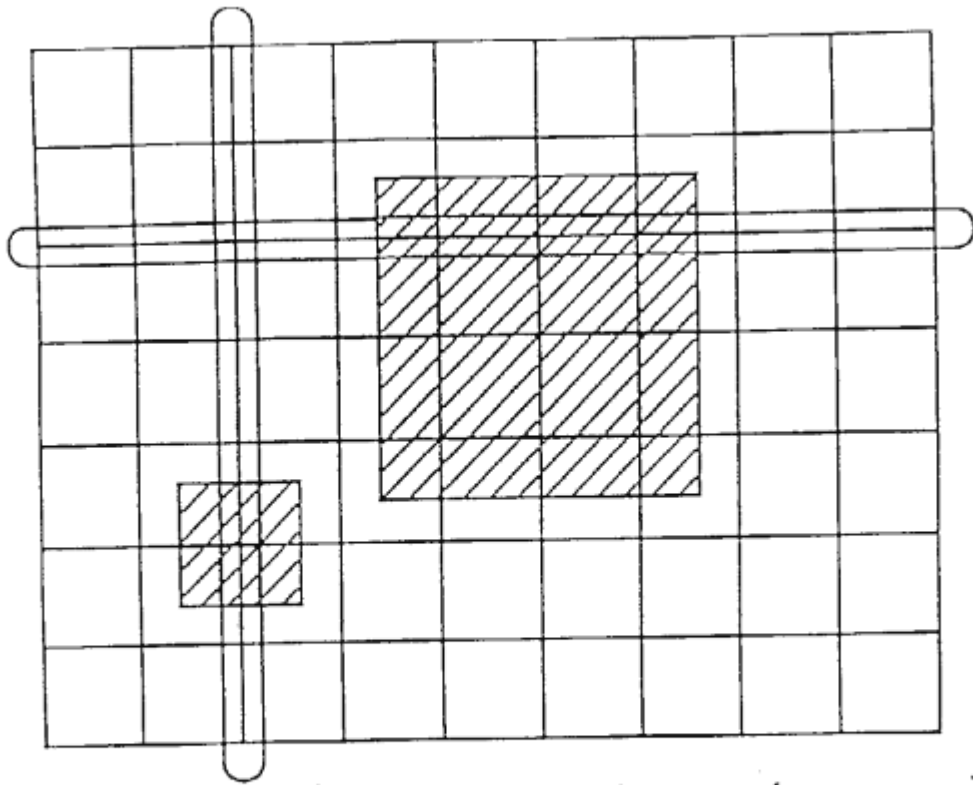
Look-ahead line search method

New formalism of the line search

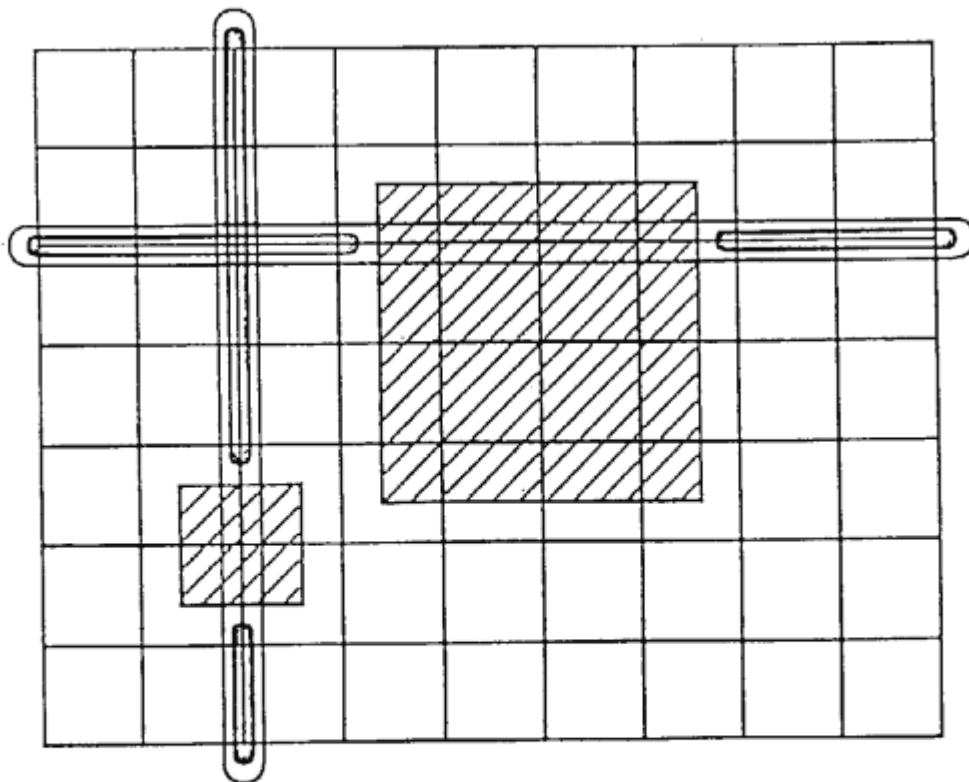
- Formalize every lines as objects
- These line objects communicate each other to search paths.
 - based on look-ahead line search algorithm

Concurrency

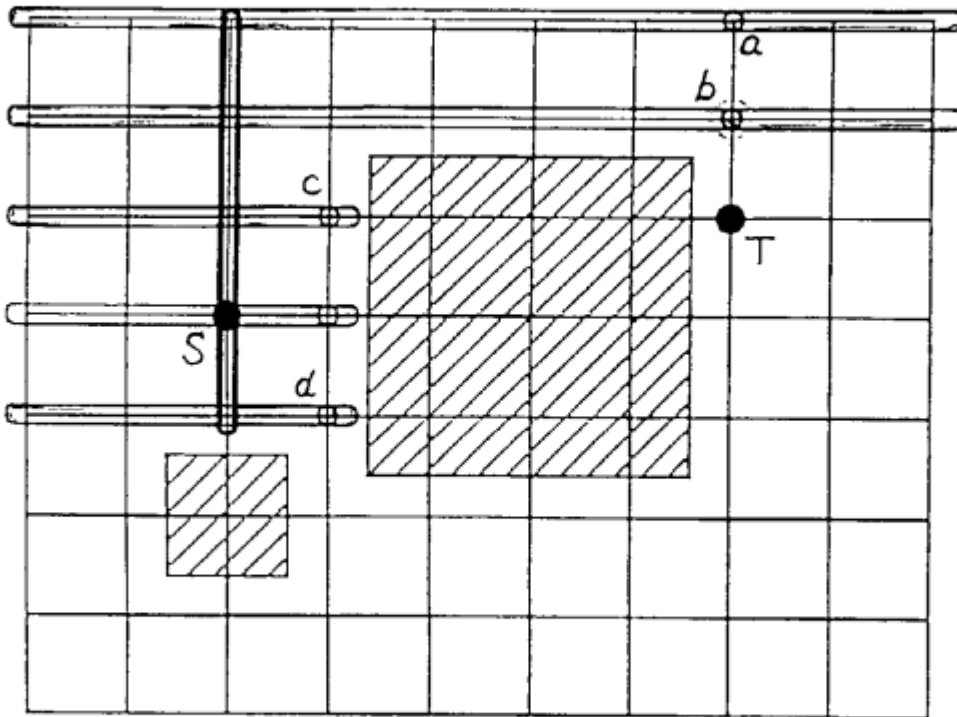
- Concurrent computation in “looking-ahead”
 - = calculation of expectation points
- Concurrent routing of different nets
 - When conflicts, first reaching net prior.



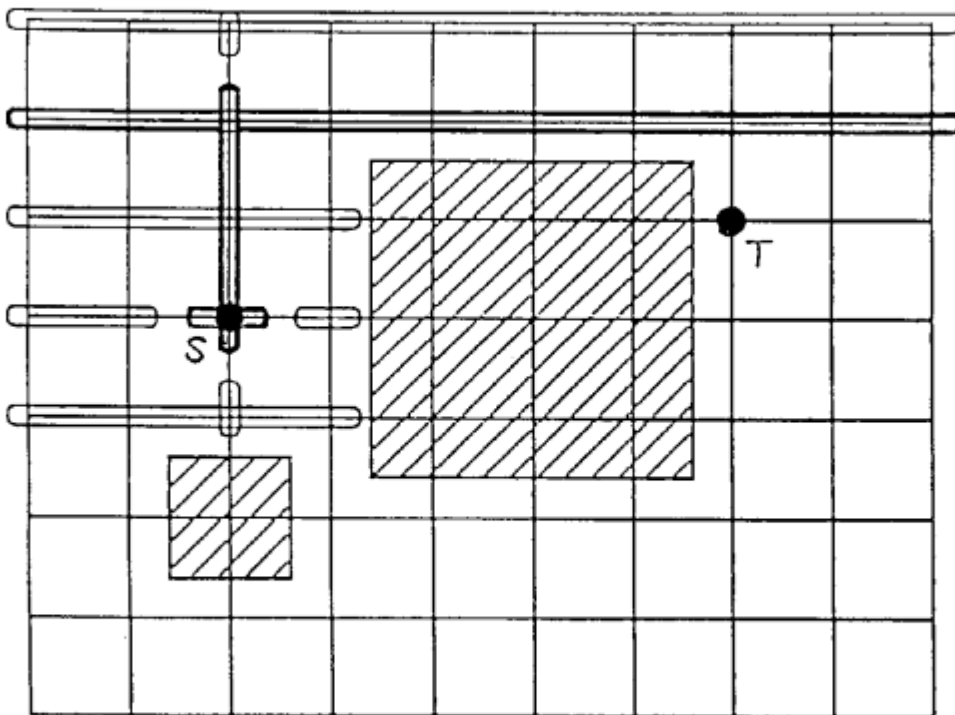
Master line objects (processes)

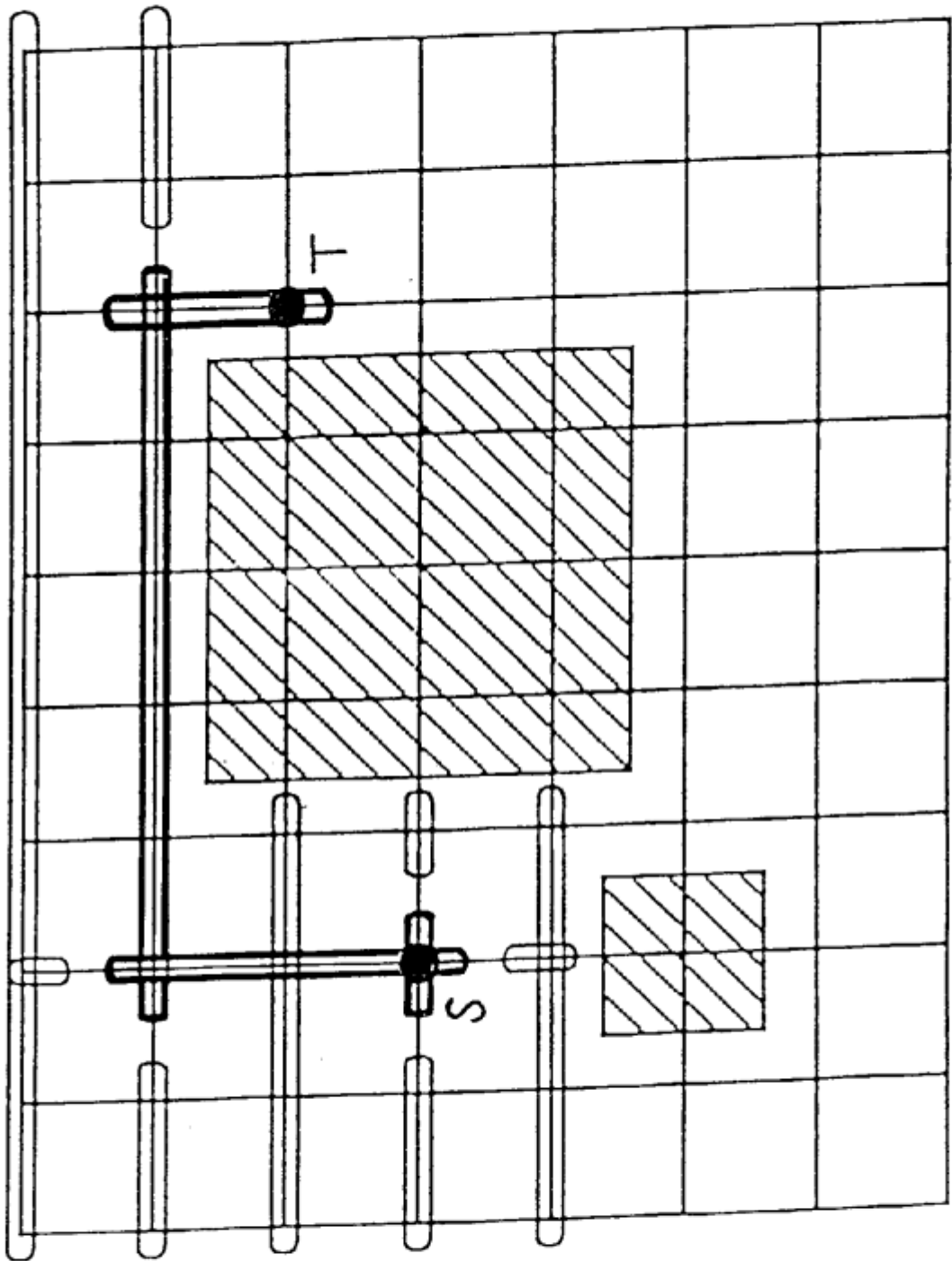


Line objects (processes)



Concurrent computation in "looking-ahead"





Implementation

Objects = processes

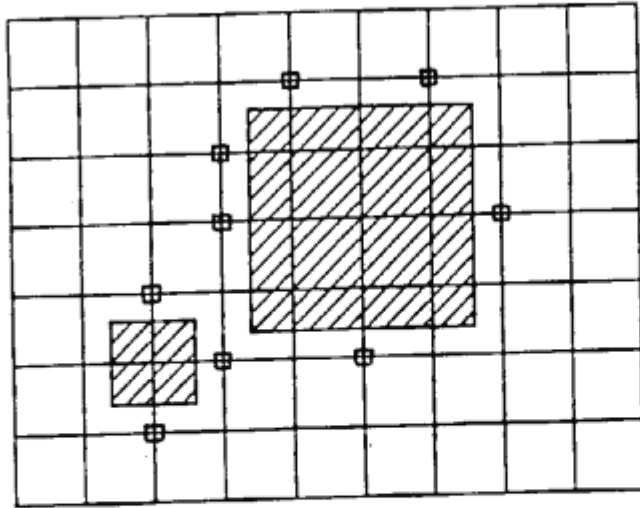
Communication path = message stream

– very naive

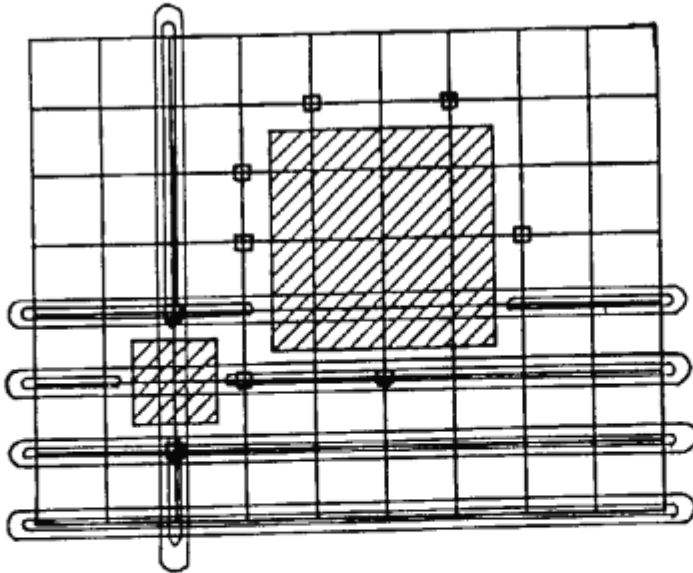
Grouping and Mapping

- A master line process and processes on it are grouped.
 - for communication locality
- Groups are mapped on processors at random.
 - for load balancing

Problem

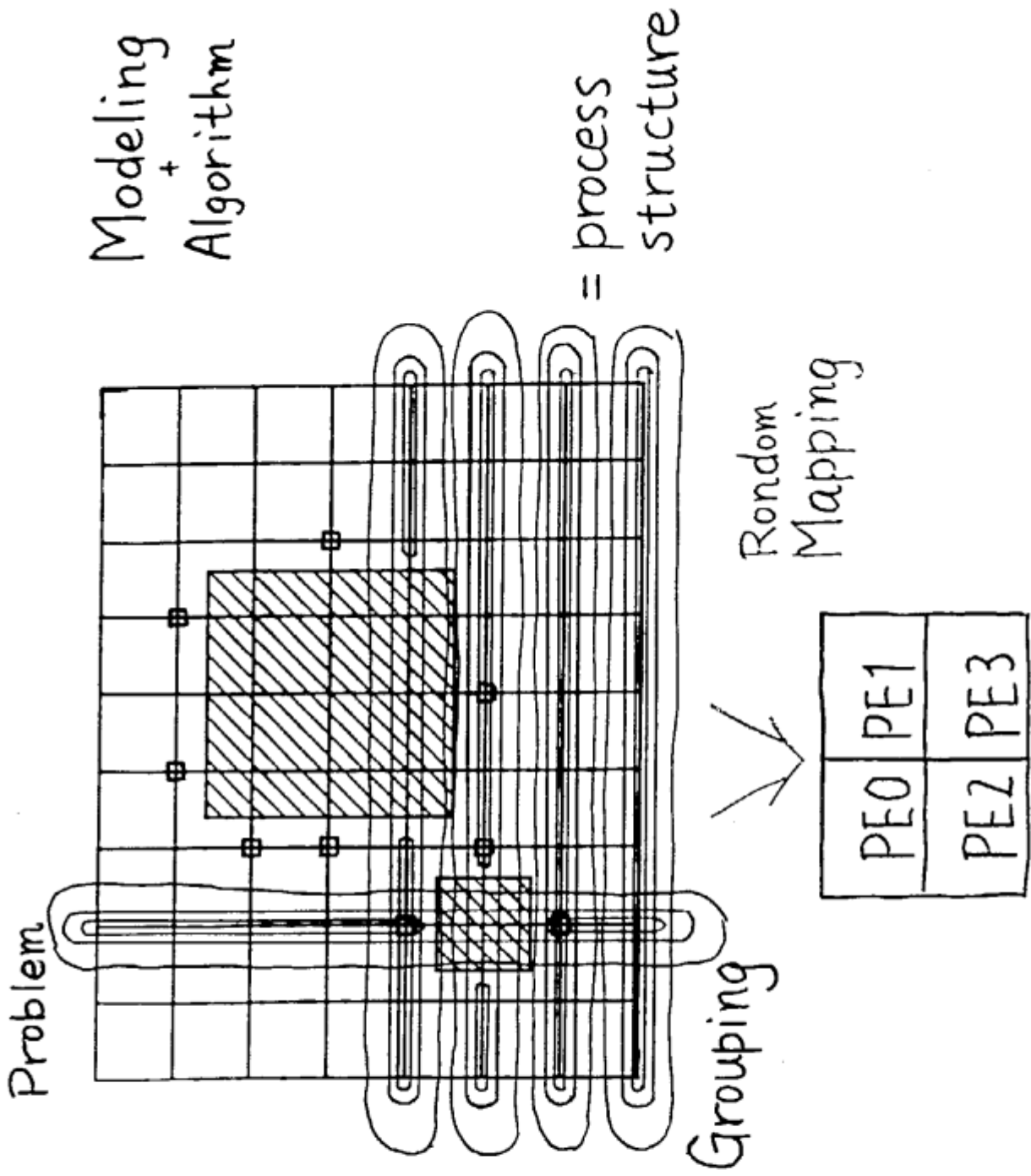


Problem



Modeling
+
Algorithm

= process
structure



Guideline for grain size control and grouping

- In a processor

Synchronization cost ≈ 1 LI
(Process switching cost)
V.S. ^{logical} inference

Grain size of a process

- Across the processor boundary

Communication cost ≈ 20 LI
V.S.

Grain size of a group

Development

1989 June - Basic idea
September - Design and coding

1990 May - Ver. 0
October - Ver. 0.7 (by 1 person)

3 K KL1 source lines

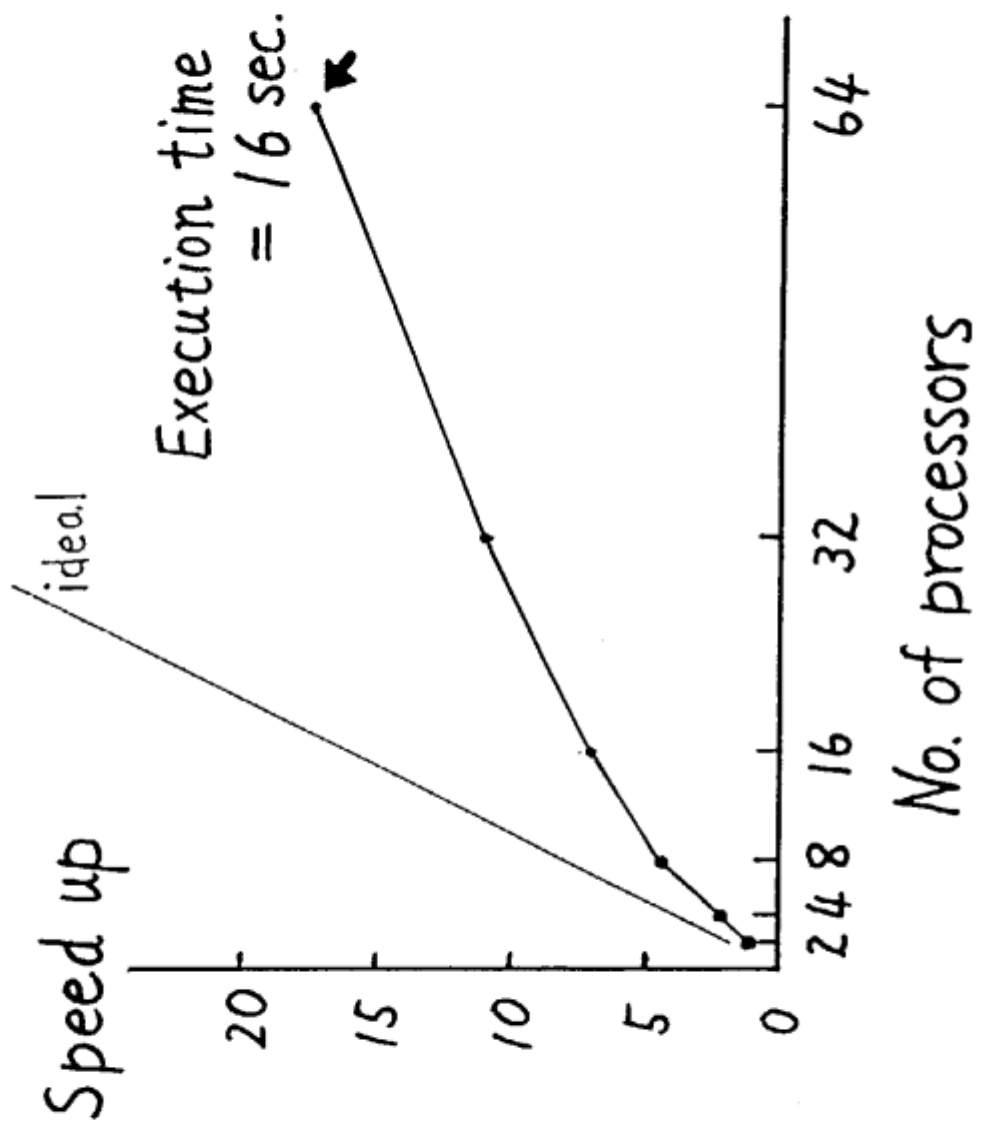
Measurements

Sample problem:

260 x 100 grid lines
136 nets
 ≈ 300 terminals

Results \rightarrow next page

Measurements on Router Program

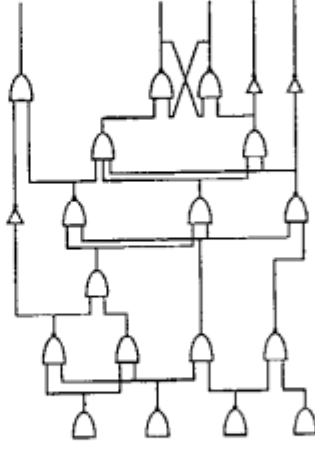


Logic-level simulation

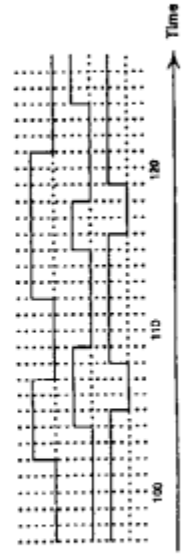
- purpose
 - To simulate behavior of circuits
 - * To verify the logical specification and signal propagation delays

Parallel Logic Simulator

An example of a circuit described in logic level



Timing chart



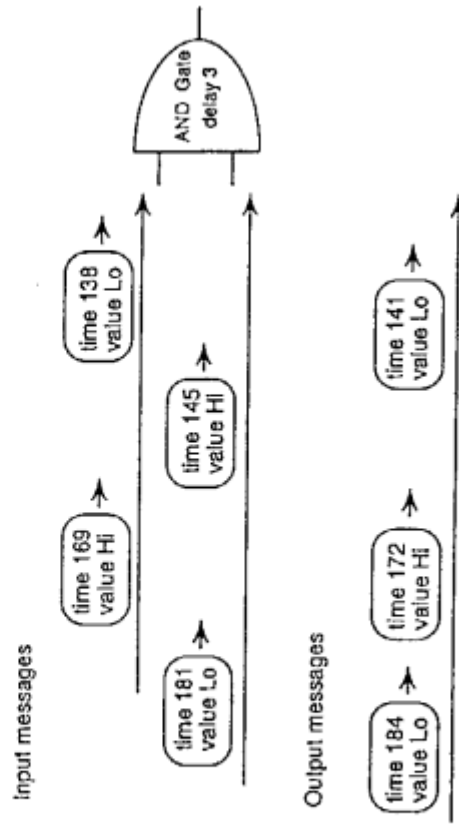
Outline of system

- Target circuits
 - Described at the gate level
- Available signal values
 - Hi,Lo,X(unknown) 3 values
- Available delay values
 - Multiples of a unit time

Simulation mechanism

- Event-driven simulation
 - Computes only when signals change at the gate inputs
 - Messages should arrive in correct time order

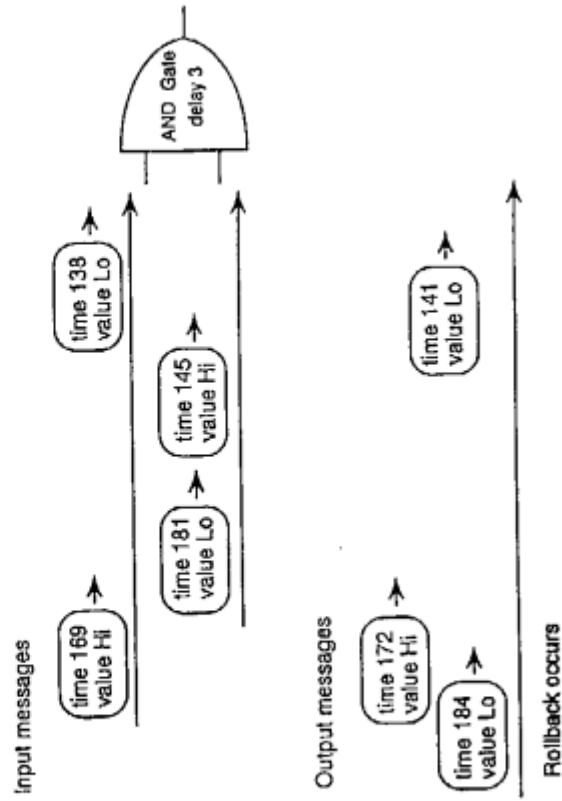
Event driven simulation



- Virtual time mechanism

- A parallel control mechanism using local synchronization
- “Rollback” occurs when a message arrives in incorrect order

Virtual time mechanism



Implementation; Which is better ?

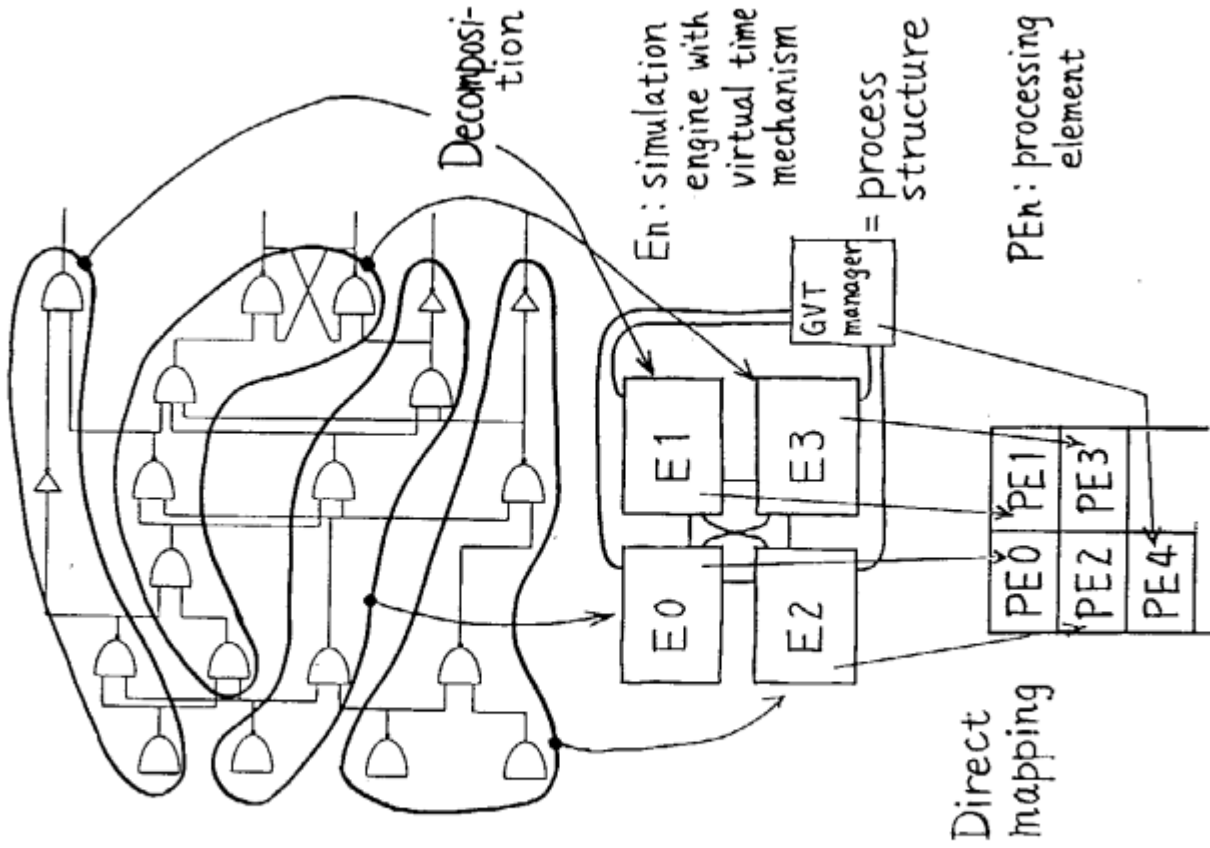
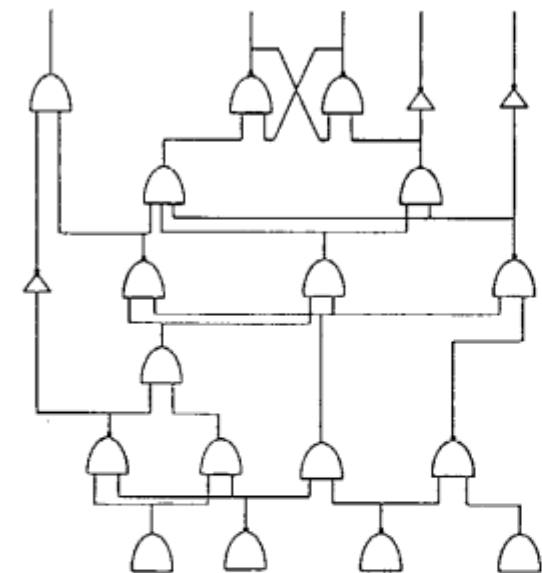
Gate = process

or

- Simulation engine = process,
gate = data

Problem decomposition

- Distribute gates equally to each processor
- or
- Reduce inter-processor communication
- Make roll-back frequency low



Development

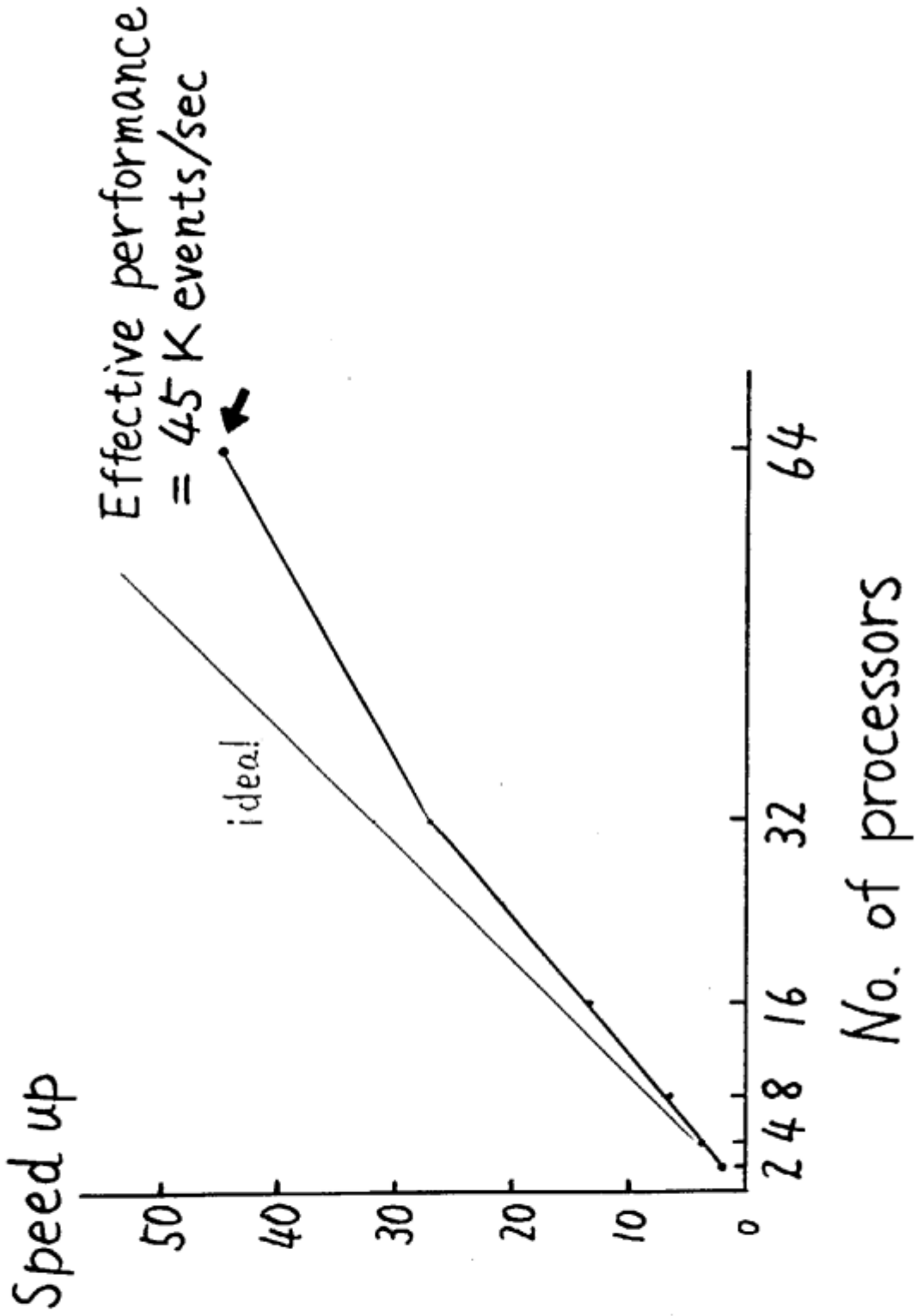
1990 March - Design and coding
May - V0
October - V0.7 (by 1 person)

7 K KL1 source lines

Target circuit

- A sequential circuit , published by ISCAS
s13207.bench
- Number of gates in the circuit — 13,000 gates
 - # 31 inputs
 - # 121 outputs
 - # 669 D-type flipflops
 - # 5378 inverters
 - # 2573 gates (1114 ANDs + 849 NANDs
+ 512 ORs + 98 NORs)
- Input sequences — randomly generated except clock lines

Measurements on Logic Simulation Program



Conclusion

1. Two parallel LSI CAD programs have been developed, as practical large application programs.
2. Parallel LSI router
 - Object-oriented formalism
—suitable to implement in KL1
 - Concurrent line search algorithm
 - Good absolute speed
3. Parallel logic simulator
 - Virtual time mechanism
 - Problem decomposing implementation
 - Good speed up
4. KL1 and our parallel processing systems are fairly efficient and applicable for these problems.

Parallel Programming in Genome Analysis System

Katsuni Nitta, Masato Ishikawa, Masaki Hoshida,
Makoto Hirosawa and Tomoyuki Toya

Institute for New Generation Computer Technology

1 Introduction

Molecular biology is advancing rapidly, and its progress have affected society significantly. As the amount of DNA/protein sequence data continues to grow, parallel computers and knowledge processing techniques to analyze them are becoming increasingly important. As ICOT has developed both parallel inference machines and knowledge processing techniques, we have suitable research environment to develop a genome analysis system. This system consists of several parallel programs for genome analysis. In this paper, overview of our parallel programs which align multiple sequences and extract motifs are introduced.

2 A Biological Introduction to DNA, RNA and Protein

2.1 DNA, RNA and Protein

The bodies of living things consists of cells. In each cell, a DNA molecule exists. A DNA is a sequence of 4 kinds of nucleic acids (A,C,G and T), and some parts of DNA have genetic information. In the case of human beings, the number of nucleic acids of a DNA is about 4,000,000,000 and it is expected that 50,000 genes exist in a DNA. The information of gene is transcribed into mRNA which is a sequence of 4 kinds of nucleic acids (A,C,G and U). The information of mRNA is translated into a protein by a ribosome. A protein is a sequence of 20 kinds of amino acids (Leu, Val, Ala, Met, ...) (Fig.1). Translation occurs in every 3 nucleic acids of mRNA according to the coding table. For example, ATG is translated to Met, GCT is translated to Ala, etc.. This coding table is common to all living things. A sequence of amino acids is folded and makes complex structures such as secondary structure, super secondary structure, tertiary structure, etc.. The function of a protein is closely related to its structure.

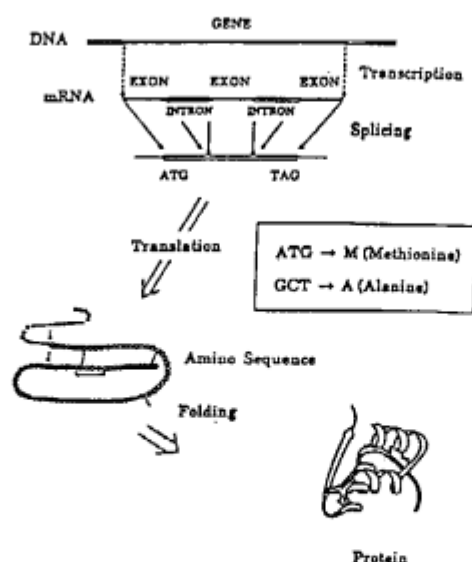


Fig.1 DNA, RNA and Protein

2.2 Databanks of DNA and Proteins

Sequence data of DNA/protein appeared in journals are put into databases. For example, GenBank contains DNA sequence data. PIR contains protein sequences and PDB contains three dimensional structure of proteins. In GenBank and PIR, not only sequence data but additional information is accommodated. For example, the name of features and specification of the residues comprising them are included in a feature table.

To make use of these databases, several programs have been developed by biologists. They are classified into two categories - database searching and sequence analysis. Database searching programs are used to select sequences for comparison with the test sequence. Once, sequences are selected from a database, they are compared and more information is extracted by analyzing them. For example, proteins contained in PIR are classified into groups from view point of biological evolution. These groups are called *super families*. Functions and structures of proteins which belong to the same superfamily show close similarity. Therefore, if we extract specific pattern (*motif*) to a super family, we can use it to predict features of unknown proteins.

2.3 Research Activities of ICOT

The 3rd research laboratory and the 7th research laboratory are developing programs related to genetic information processing.

- A unified database :

The 3rd laboratory has developed a database management system named *Kappa*, which employs the nested relational model, and is now developing a knowledge representation language named *Quixote*, which is a deductive and object oriented language and is good at representing complex knowledges, such as the ones of molecular biology. As for the genetic information processing, GenBank was stored in Kappa in 1989 to research what kinds of databases are really needed by the molecular biologists. Now the research focused to build an integrated database, which have data of GenBank, PIR, and other databases of molecular biology. As the first step, GenBank and PIR are stored together in Kappa, to develop a suitable tools for building useful secondary databases. As another step, a protein function database is written in Quixote, to complement the integrated molecular biological database.

- Parallel programs for analyzing protein sequence :

The 7th laboratory has developed basic application software for parallel inference machines. As for the genetic information processing, a KL1 program which classify protein sequences into super families was developed last year. Our next target is to develop following genome analysis tools with a biological knowledge base. We use *motifs* as a clue for analysis.

1. Searching homology.
2. Extracting *motifs* by comparing sequences.
3. Predicting the structure and functions of proteins.

3 Multiple Sequence Alignment and Motif Extraction

To align sequences is one of the most basic techniques to analyze sequences. It is used to find differences between sequences, to find motifs of a set of proteins and to search similar sequences from database.

An alignment is realized by lining the sequences with corresponding characters directly above one another. To make corresponding characters line up, we can insert dashes (*indel*) into the sequences. For example, if two sequences

```
S1: ACTGTTACTA
S2: ACCATATA
```

are given, we can align them as follows.

```
S1': ACTGTTACTA
S2': ACC-ATA-TA
```

From given sequences, we can create more than one alignments. Therefore, to evaluate the quality of an alignment, we need to define an alignment score.

A difference score *dif* is one of typical scores for an alignment.

$$dif = m \cdot \alpha + n \cdot \beta$$

In this formula, m is the number of unmatched (substitutions) in the alignment and n is the number of indels. α is a penalty for an unmatched, and β is a penalty of an indel. Optimal alignment is constructed by minimizing the difference score.

To align two sequences, Needleman and Wunsch developed a dynamic programming algorithm (*DP-matching*) [1]. The basic idea of their algorithm is as follows. Let input sequences be $S1 (=a1.a2....am)$ and $S2 (=b1.b2....bn)$. For all i,j such that $0 \leq i \leq m, 0 \leq j \leq n$, distance score can be calculated by the following formula.

```

score(i,0) = i*beta
score(0,j) = j*beta
for i,j such that 0<i<m, 0<j<n
  if ai=bj then
    score(i,j) = score(i-1,j-1)
  else
    score(i,j) = min { score(i-1,j)  + beta,
                      score(i,j-1)  + beta,
                      score(i-1,j-1) + alpha }

```

$Score(i,j)$ corresponds to the difference score of optimal alignment of sequences $S1'(=a1.a2....ai)$ and $S2'(=b1.b2....bj)$. Complete alignment of $S1$ and $S2$ can be constructed by tracing path from $(0,0)$ to (m,n) . For example, if $S1(=ADHE)$ and $S2(=AHIE)$ are given, the path which minimize $score(4,4)$ is $(0,0) \rightarrow (1,1) \rightarrow (2,1) \rightarrow (3,2) \rightarrow (4,4)$ (Fig.2).

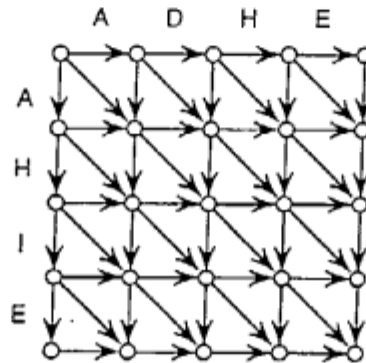


Fig.2 DP matching

From this path, we can construct the following alignment.

```

ADH-E
A-HIE

```

Aligning several sequences is more difficult than aligning just two sequences. If Needleman and Wunsch's algorithm is extended to align n sequences, execution time grows exponentially with n .

We are developing KLI programs which align multiple sequences and extract motifs.

3.1 Alignment by three dimensional DP matching

First one is based on Needleman and Wunsch's algorithm. We extended their algorithm to align three sequences as follows.

Let input sequences be $S1 (=a1.a2....am)$ and $S2 (=b1.b2....bn)$ and $S3 (=c1.c2....cp)$. We can align these sequences by constructing a three dimensional rectangular prism (Fig.3) and by finding a path from $(0,0,0)$ to (m,n,p) which minimize the scoring function $score2$.

$$score2(i,j,k) = score(i,j) + score(j,k) + score(k,i)$$

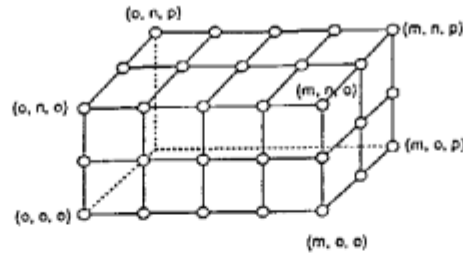


Fig.3 Three dimensional DP-matching

In our program, each node in the prism is represented as a KLI process. When a process(i,j,k) receives score data from adjacent processes such as process(i-1,j,k), process(i,j-1,k), process(i,j,k-1), process(i-1,j-1,k), process(i-1,j,k-1), process(i,j-1,k-1) and process(i-1,j-1,k-1), then this process can calculate its score. Therefore, the data dependencies form *wavefronts* diagonally across the prism from (0,0,0) to (m,n,p) and this program contains a lot of parallelism. However, if we align just three sequences once, each process becomes idle most time because a process must wait a long time until it receives all score data from adjacent processes and because a process become needless after it sends its score to neighbors. To make each process work effectively, we construct the prism (network) before sequences are given. After the network is constructed, if we input a set of triplets of protein sequences [(S1, S2, S3), (S4, S5, S6)...], our program aligns (S1,S2,S3) at first, and then aligns (S4,S5,S6), etc.. Consequently, each process continues to calculate its score until all triplets are aligned.

3.2 Alignment by Simulated Annealing

Simulated annealing is a technique to get optimum solutions for combinatorial problems. This procedure starts with initial solution candidate X0 and initial temperature T0, and tries to find the solution X which minimizes an energy function E(X) as follows (Fig.4).

Let current solution candidate be Xn. At first, this procedure generates another candidate Xn', and compares E(Xn) to E(Xn'). If E(Xn') is less than E(Xn), then Xn' becomes a new candidate. If E(Xn') isn't less than E(Xn), Xn remains as a candidate. To prevent E(Xn) falls into local minimum, occasionally, Xn' becomes a candidate instead of Xn even if E(Xn') is greater than E(Xn). The probability that the substitution occurs is controlled by temperature Tn. Usually, as n becomes larger, Tn is scheduled to become smaller and the probability for substitution becomes lower.

```

begin
  X0 := Initial solution ;
  {Tn}n=0..N-1 := Temperature (Cooling schedule);
  for n := 0 to N-1 do
    begin
      X'n := Some random neighboring solution of Xn;
      ΔE := E(X'n) - E(Xn);
      if ΔE < 0 then
        Xn+1 := X'n
      else
        if exp(-ΔE/Tn) ≥ random(0,1) then
          Xn+1 := X'n
        else
          Xn+1 := Xn
    end;
  Output XN;
end;

```

Fig.4 Simulated Annealing

Professor Kanehisa of Kyoto University developed an algorithm to apply this technique to a multiple alignment. The basic idea is as follows. Let input sequences be S1 (=a1,a2,...am), S2 (=b1,b2,...bn)... Sk (=c1,c2,...cp). At first, we construct initial alignment by adding indels to end of each sequence.

```

S1 : a1,a2,a3,.....,am,-,-,-
S2 : b1,b2,b3,.....,bn,-,-,-,-
.
.
Sk : c1,c2,c3,.....,cp,-

```

To modify the alignment, we can insert an indel to any place in the sequence and delete another indel. If the difference score becomes smaller by this operation, the new alignment becomes a candidate for solution. By repeating this operation, we can obtain the optimal alignment.

One of difficult problem to apply simulated annealing to actual world is making scheduling of temperature ($\{T_0, T_1, \dots, T_Y\}$). If the scheduling is not suitable, we cannot obtain good solution because it falls down to local minimum. Therefore, we must decide cooling scheduling carefully. To make scheduling problem easy, Kimura of ICOT developed a parallel simulated annealing program [2]. In his algorithm, each processor maintains one solution and performs the annealing process concurrently under a constant temperature. The solutions obtained by the processors are exchanged occasionally. To each processor, different temperature is allocated. Therefore, data exchange corresponds to the change of temperature. Consequently, we can avoid the task of deciding the cooling schedule.

We developed a KL1 program which apply Kimura's algorithm to align multiple sequences.

3.3 Motif extraction by comparing pairwise sequences

This program is based on the algorithm developed by R. Smith and T. Smith [3]. When protein sequences are given, this algorithm extracts the conserved pattern to input sequences. As it takes much time to compare all sequences at once, it calculates difference scores for all pairs of sequences. For example, if n sequences are given, $n \times (n - 1) / 2$ pairs are aligned and difference scores are calculated in parallel. After difference scores are calculated, a pair whose score is the lowest is selected. Two sequences of the pair are merged into one sequence using an amino acid class hierarchy. For example, if two sequences are selected and aligned as follows

```

S1 : ADDEK
S2 : A-DDP

```

then they are merged into the following sequence.

```

S0 : AgDdX

```

where g means a gap, d is the amino acid class to which D and E belong, and X means a wild card character. S_0 is the generalized pattern of S_1 and S_2 . This process is repeated until all input sequences are unified into one sequence. The final sequence contains a motif information of input sequences.

4 Conclusion

We introduced three KL1 programs which analyze the DNA/protein sequences. The purposes of developing these programs are to investigate the efficiency of parallel programs and to extract characteristic patterns.

To improve the quality of alignments or motifs, we have to use biological knowledge. Though we have not adopted knowledge based approach for alignment and motif extraction, we are investigating knowledge which biologists use to align sequences. The extracted information by these programs can be used to develop the biological knowledge base. Our next plan is to improve these programs by using the knowledge base.

REFERENCES

- [1] Needleman and Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins", *Journal of Molecular Biology*, (1970).
- [2] Kimura, "On a Time-homologous Parallel Simulated Annealing (in Japanese)", TR-565(1990).
- [3] R.Smith and T.Smith, "Automatic Generation of Primary Sequence Patterns from Sets of related Protein Sequences", *Biochemistry*, (1990).

Parallel Programming in Genome Analysis System

Katsumi Nitta

ICOT

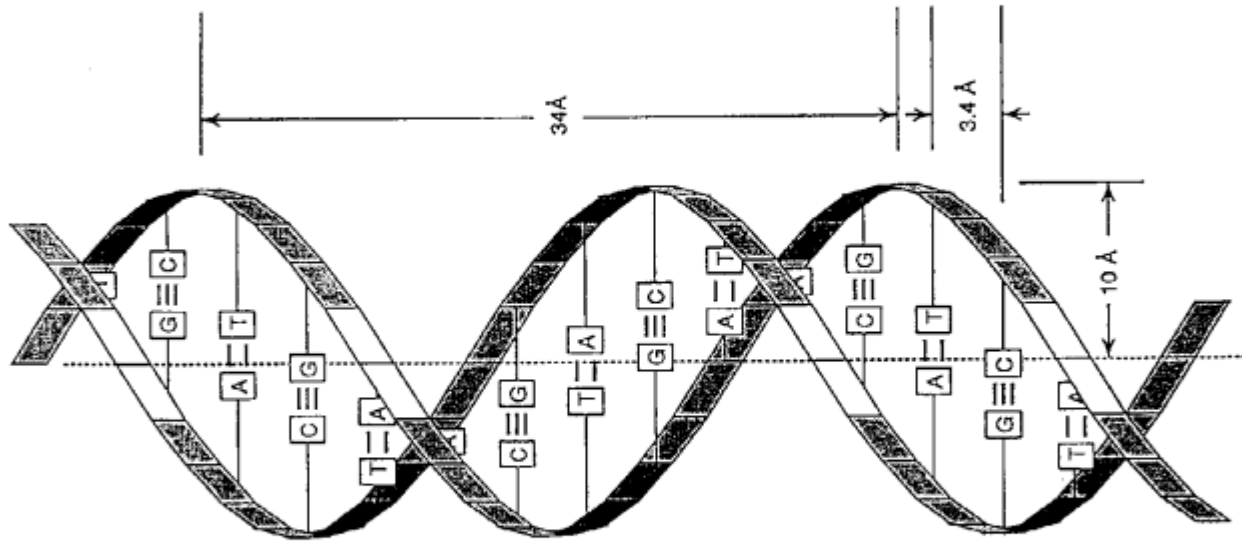


Application Software

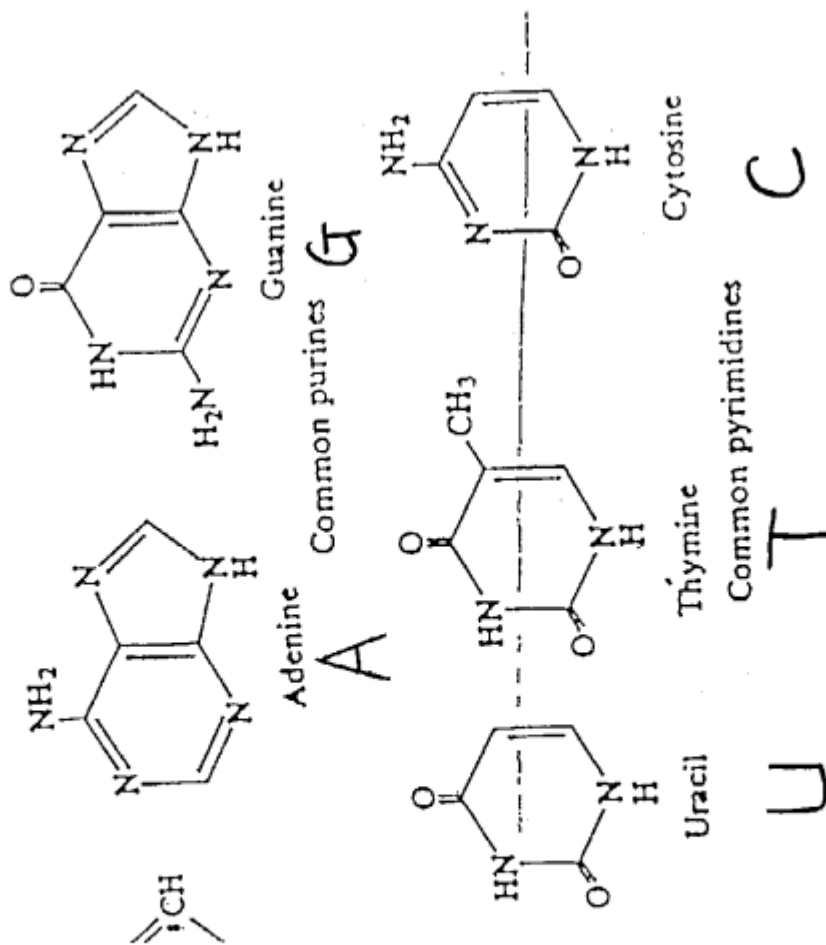
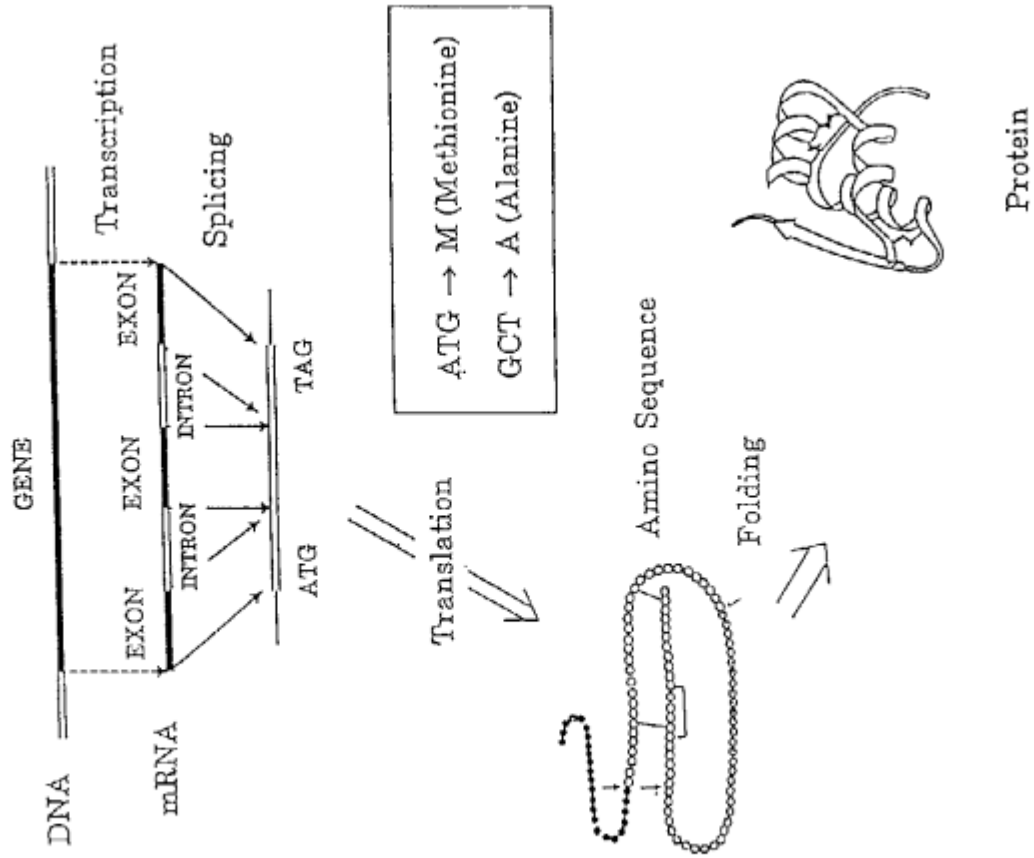
- VLSI CAD Program
- Intelligent Blackboard System
- Go Playing Game
- Legal Reasoning Experimental System
- Genome Analysis System

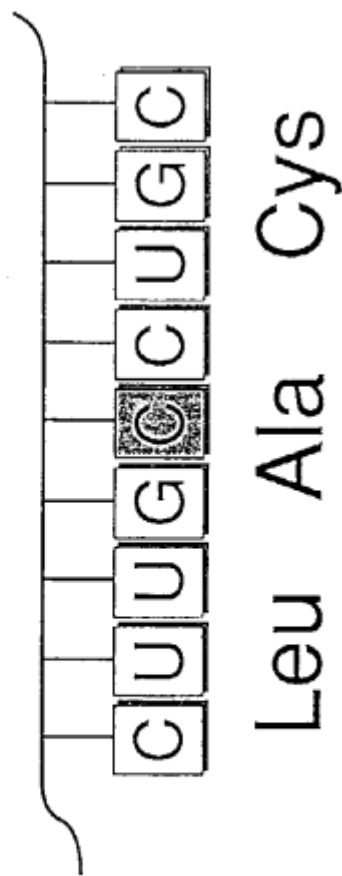
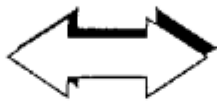
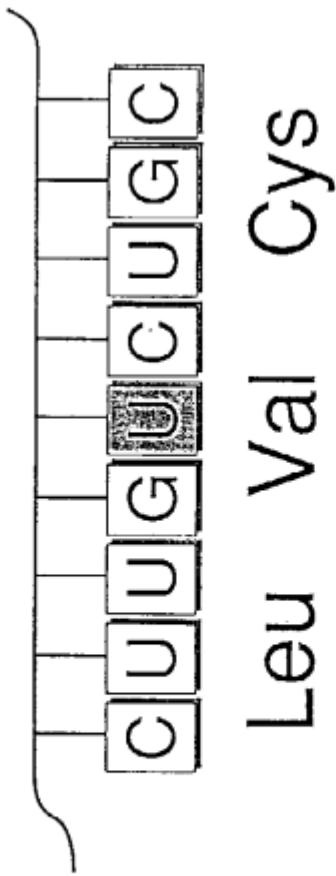
Subjects

- Introduction to Molecular Biology
- Research Activities in ICOT
 - Sequence Searching
 - Sequence Analysis System
- * Parallel Programs for Multiple Alignment



Genetic Information Processing





First position (5' end)	Second position				Third position (3' end)
U	U	C	A	G	U C A G
	Phe	Ser	Tyr	Cys	U C A G
	Phe	Ser	Tyr	Cys	U C A G
	Phe	Ser	Stop	Stop	U C A G
C	Leu	Pro	His	Arg	U C A G
	Leu	Pro	His	Arg	U C A G
	Leu	Pro	Gln	Arg	U C A G
	Leu	Pro	Gln	Arg	U C A G
A	Ile	Thr	Asn	Ser	U C A G
	Ile	Thr	Asn	Ser	U C A G
	Ile	Thr	Lys	Arg	U C A G
	Met	Thr	Lys	Arg	U C A G
G	Val	Ala	Asp	Gly	U C A G
	Val	Ala	Asp	Gly	U C A G
	Val	Ala	Glu	Gly	U C A G
	Val	Ala	Glu	Gly	U C A G

Structure of Proteins

- 1st
Val Leu Ala Ser Gly . . .

- 2ndary
alpha helics, beta strand, turn, coil

- tertiary



Name	3-letter code	1-letter code
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic acid	Asp	D
Cysteine	Cys	C
Glutamic acid	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

Table 4.1: The twenty amino acids and t

¹ Homologies often refer to evolutionary similarities resulting from in this thesis refers to any similarity, regardless of its cause.

Databases of DNA and Proteins

- DNA

- GenBank: Nucleic Acid Sequence

- EMBL: Nucleic Acid Sequence

- DDBJ: Nucleic Acid Sequence

- Proteins

- PIR: Amino Acid Sequence

- PDB: 3D Structure

```

///
ENTRY      #Type Protein
TITLE      Cytochrome c - Chimpanzee (tentative sequence)
DATE       17-Mar-1987 #Sequence 17-Mar-1987 #Text 17-Mar-1987
PLACEMENT  1.0 1.0 1.0 1.0 2.0
SOURCE     Pan troglodytes.#Common-name chimpanzee
ACCESSION  A00002
REFERENCE  (Compositions of chymotryptic peptides)
           Needleman S.B., Margoliash E.
           unpublished results, 1966, cited by Margoliash, E.,
           and Fitch, W.M., Ann. N.Y. Acad. Sci. 151,
           359-381, 1968.

REFERENCE  Needleman S.B.
           submitted to the Atlas, October 1968
           Chimpanzee cytochrome c appears to be identical with
           human.

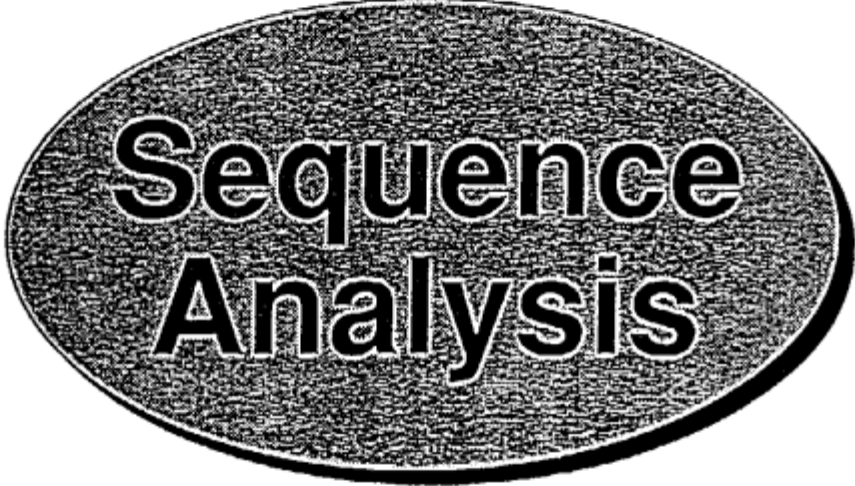
FEATURE
1          #Modified-site acetylated amino end\
14,17     #Binding-site heme (covalent)\
18,80     #Binding-site heme iron (axial ligands)

SUPERFAMILY #Name cytochrome c
KEYWORDS    mitochondrion\ electron transport\ respiratory
           chain\ oxidative phosphorylation\ heme\
           acetylation

SUMMARY     #Molecular-weight 11617 #Length 104 #Checksum 9501
SEQUENCE
           5 10 15 20 25 30
           1 G D V E K G K K I F I M K C S Q C H T V E X G K H K T G P
           31 N L H G L F G R K T G Q A P G Y S Y T A A N K N K G I I W G
           61 E D T L M E Y L E N P K K Y I P G T K H I F V G I K K E E
           91 R A D L I A Y L K K A T N E
  
```

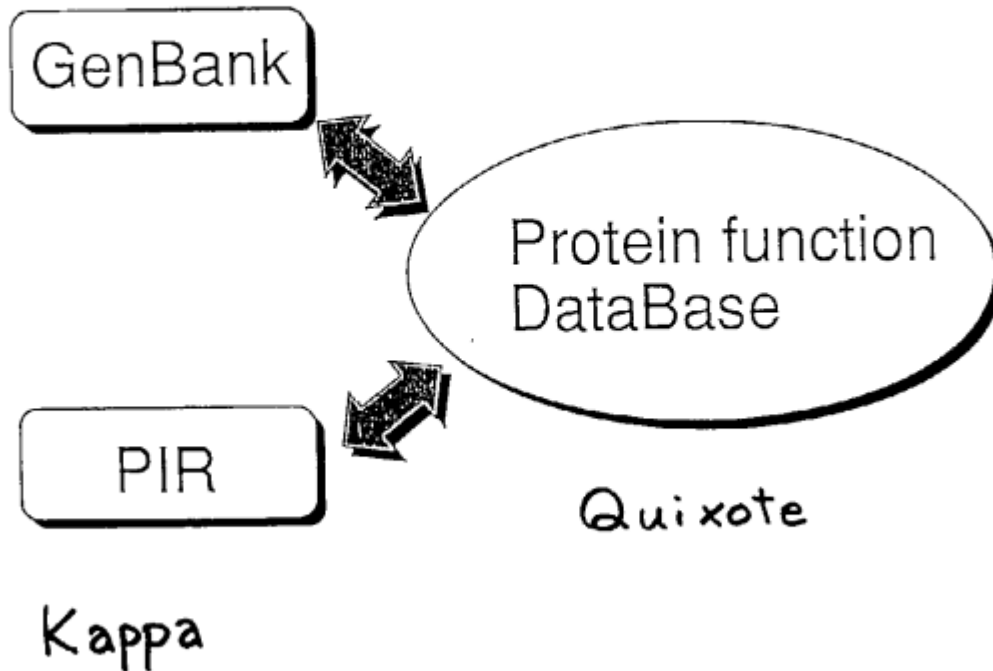


**Database
Searching**



**Sequence
Analysis**

Database Searching



Sequence Analysis

Motif extraction

Multiple Sequence Alignment

Structure Prediction

Motif (a)

HIV	PDIVIYQ	YMDD	LYVGS
HTLV-1	PQCTILO	YMDD	ILLAS
MMLV	PDLILLO	YVDD	LLLAA
RSV	PSLCMLH	YMDD	LLLAA

YXDD

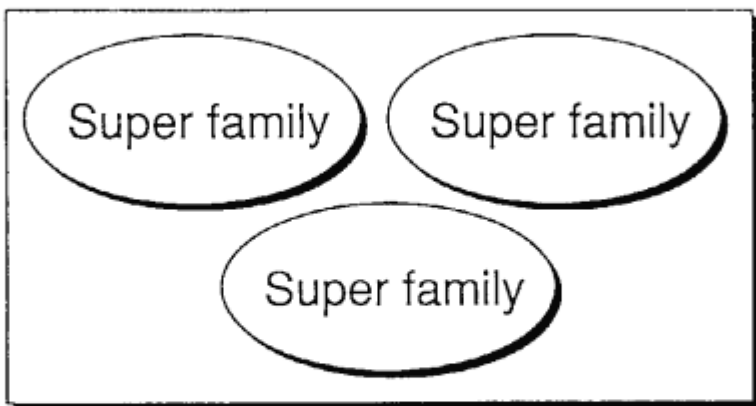
Motif (b)

1	YI	C	SFAD	C	GAAYNKNWKLQA	H	LC-K	H
2	FP	C	KEEG	C	EKGFTSLHHLTR	H	FL-T	H
3	FT	C	DSDF	C	DLRFTTKANMKK	H	FNRF	H
4	YV	C	HFEN	C	GKAFKKHNQLKV	H	QF-S	H

C (X₄) C (X₁₂) H (X₃) H

Motif Extraction

Motif, Template, Consensus, Pattern



Multiple Sequence Alignment

Purpose

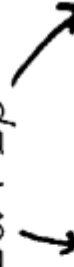
- Homology Search
- Extracting Motif
- Evolutional Tree

ACTGTTACTA
ACCATATA



ACTGTTACTA
ACC-ATA-TA

$$2\alpha + 2\beta$$



mismatch gap

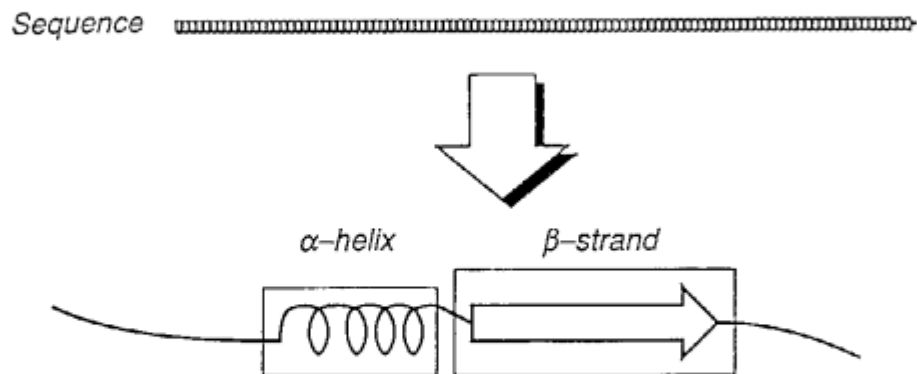


ACTG-TTACTA
ACC-AT-A-TA

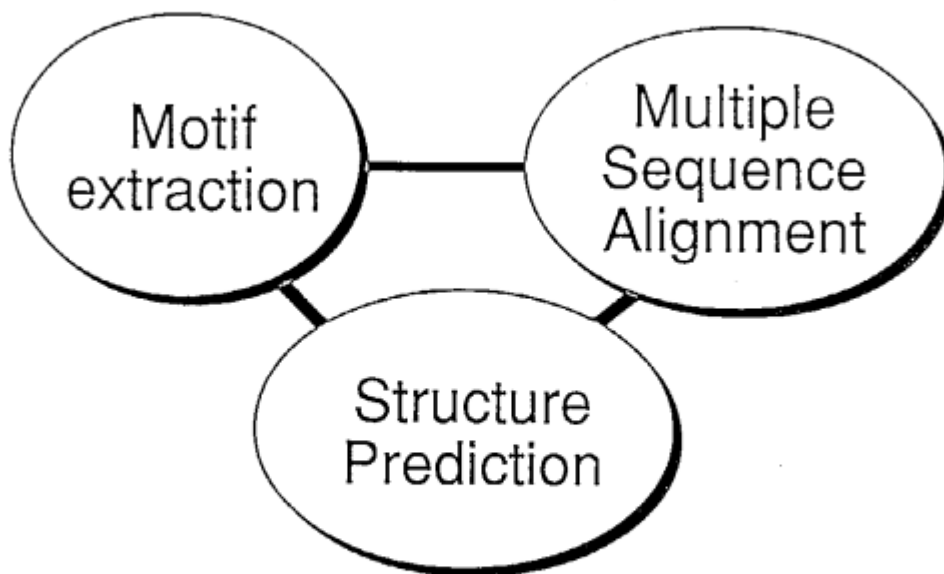
$$\alpha + 4\beta$$

Prediction of 2ndary Structure

Prediction of Secondary Structure



Sequence Analysis



Multiple Sequence Alignment

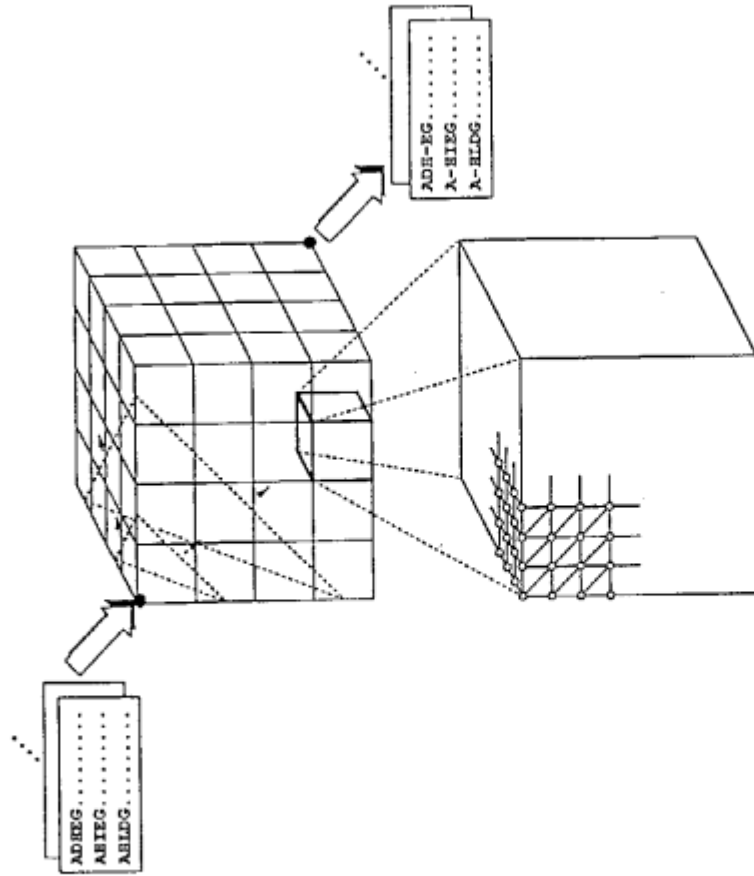
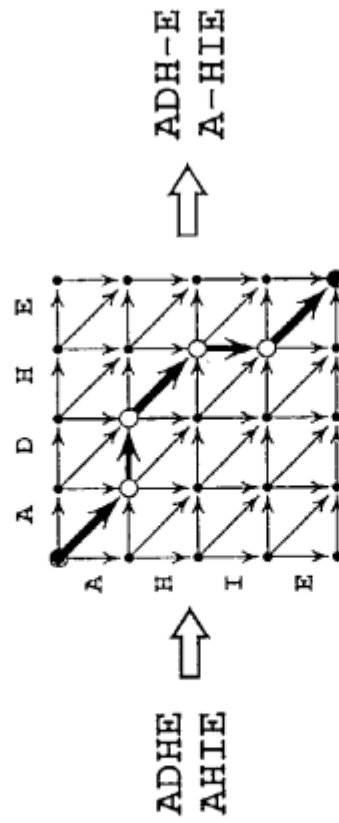
Parallel Program

Multiple Alignment by
DP-matching

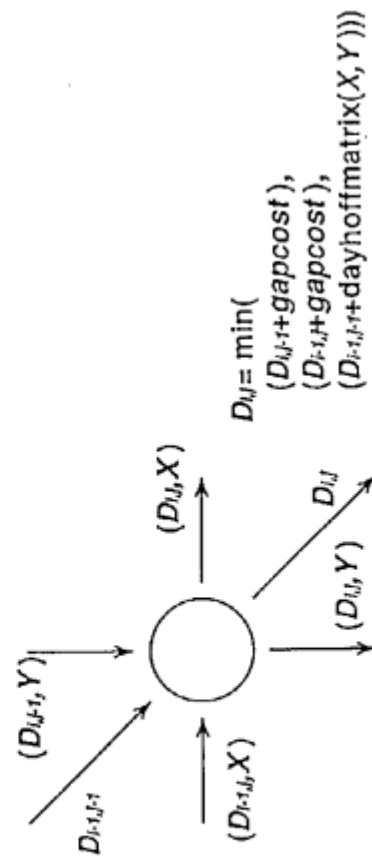
1. DP-matching

2. Simulated Annealing

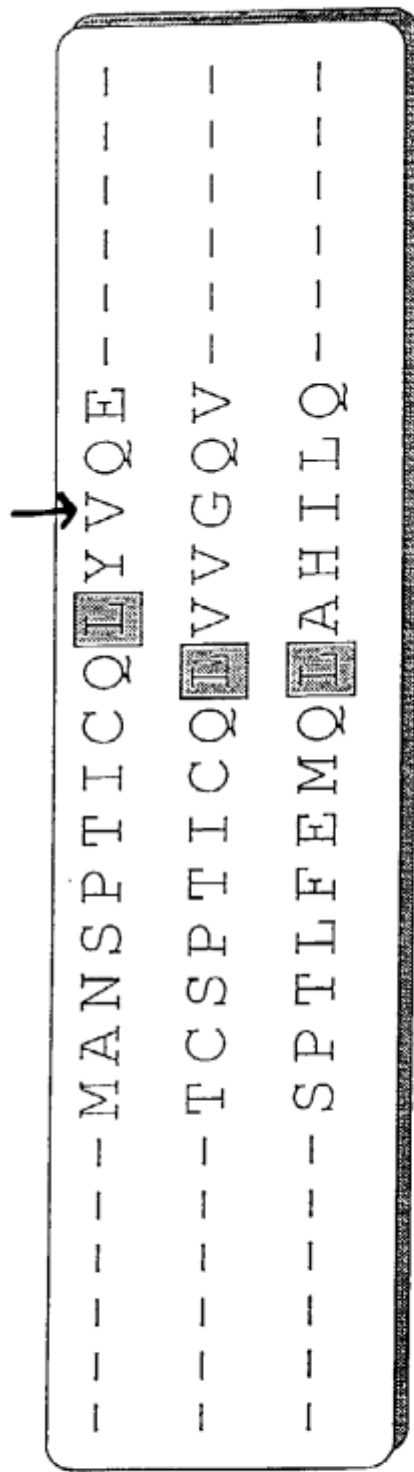
Sequence Alignment by Dynamic Programming



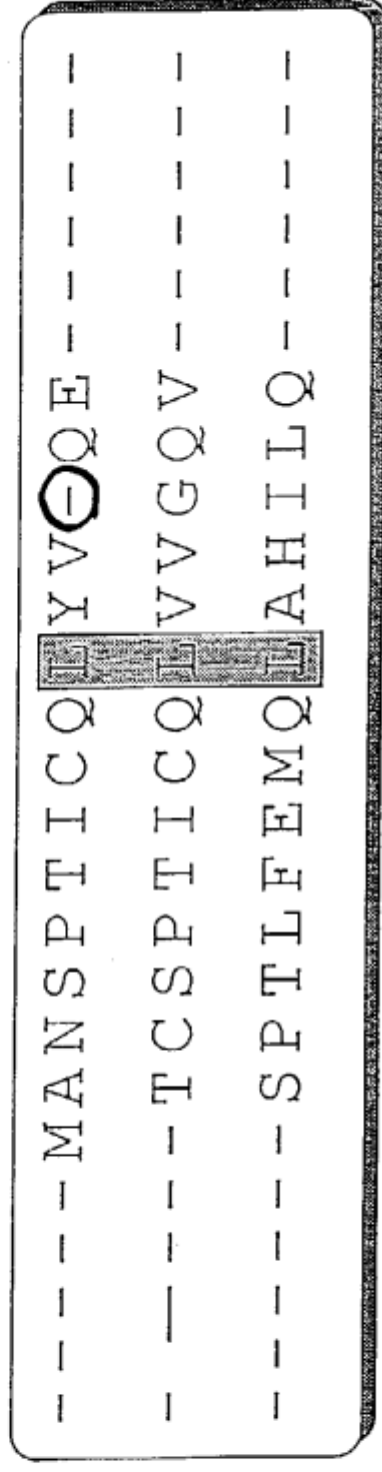
Parallel Pipeline Processing of 3D DP



Multiple Alignment by Simulated Annealing



An initial alignment



An alignment after the first move

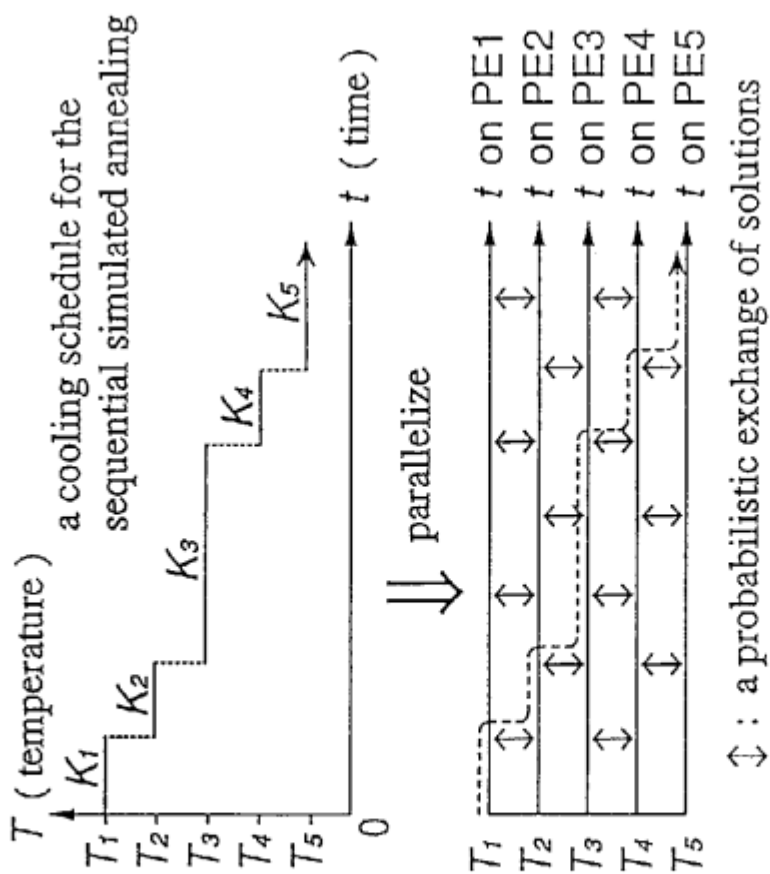
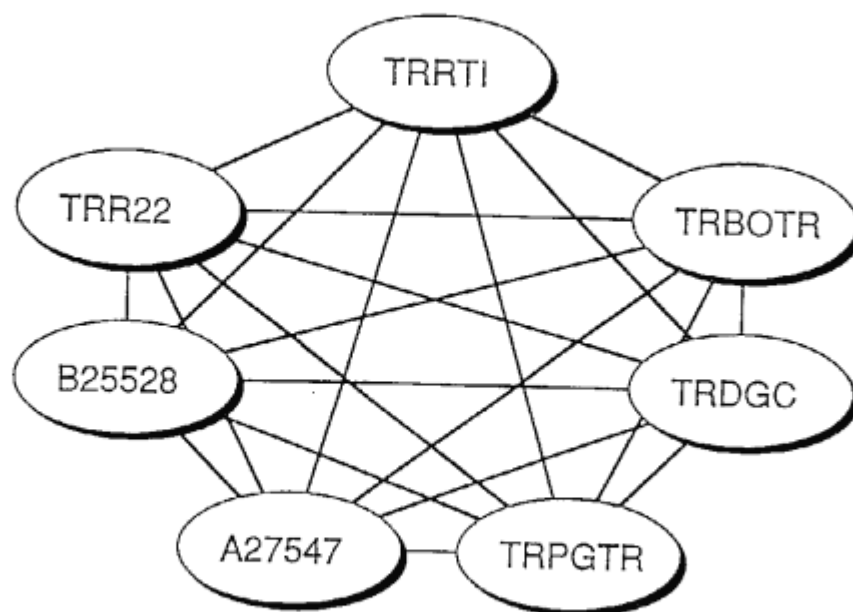
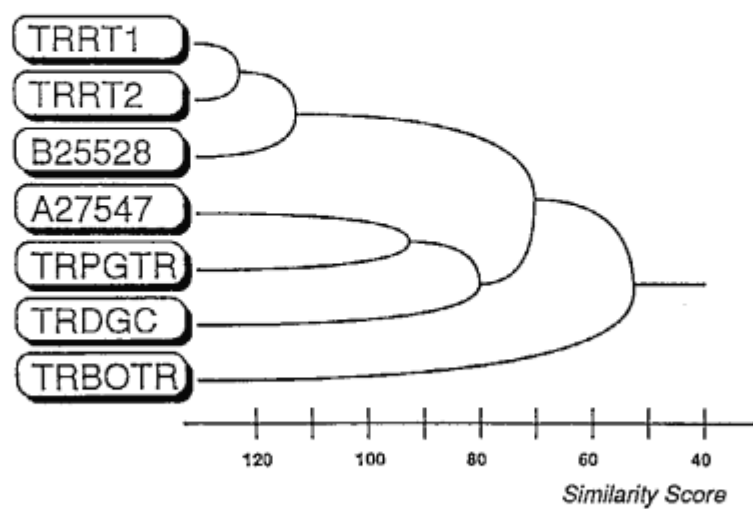


Fig. Scheduleless Parallel Simulated Annealing



Cluster 50 (Trypsinogen/Venom serine proteases)



Knowledge Based Approach

Problems

- Incomplete Domain Knowledge
- Data Contains Lots of Noise

Approach

- Statistics/ Probability
- Inductive Reasoning/ Analogical Reasoning / Case Based Reasoning

Conclusion

- Sequence Analysis
 - Motif Extraction
 - Multiple Sequence Alignment
 - Prediction of 2ndary Structure
- Motif + Biological Knowledge

Constraint Logic Programming and Its Parallel Implementation : Guarded Definite Clauses with Constraints Preliminary Report

David Hawley and Akira Aiba
Institute for New Generation Computer Technology (ICOT)
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

November 5, 1990

One major trend in the effort to extend logic programming languages is to include special handling for non-herbrand domains and certain predicates over these domains which are termed *constraints*. The component comprising the added functionality that is required to process constraints is called a *constraint solver*.¹ Incorporating non-herbrand domains increases the expressive power of logic programming, by allowing the natural expression of knowledge that is either difficult or impossible to represent in vanilla logic programming.

An important scheme which establishes the semantic foundations for constraint logic languages, is CLP(X), proposed by Jaffar and Lassez [JaL87]. Instances of this scheme include CLP(\mathbb{R}), which handles linear (in)equations over real numbers, and a number of other languages [DiH88, Wal89, BeP89]. A slightly different scheme is represented in the CAL family of languages at ICOT, which variously handle rational polynomials [SaA89], boolean equations [SaS88], and equations on finite/cofinite sets [Sat90].

Recent work in parallel logic programming research has included formulations based on constraints. To do this, the concepts of unification and matching are respectively general-

ized to constraint *satisfiability* and *entailment* [Mah87]. This idea is picked up in [Sar89], which proposes the Concurrent Constraint (cc) family of languages, which models concurrent computation as the interaction of multiple cooperative agents through the asserting and querying of a shared repository of information seen as constraints. Concretely, this scheme can be realized as a guarded (conditional) reduction system, where the guards contain the queries and assertions. Control is achieved by requiring that the queries in a guard are true (entailed), and that the assertions are consistent (satisfiable). We introduce Guarded Definite Clauses with Constraints (GDCC), an experimental instance of the cc scheme, which supports an unstructured user-specified set of sorts and constraint symbols in a committed-choice framework, and is intended to be used as a research tool for investigating issue of constraint solving in concurrent programming languages, such as concurrent constraint solvers, problem decomposition, use of multiple solvers and hybrid techniques, management of semi-decidable solution methods, debugging techniques, etc. GDCC is implemented in the KL1 committed-choice logic language, on the MultiPsi parallel logic machine.

In the CAL group, we are interested in symbolic algebra based techniques for solving constraints, specifically methods built around a canonical simplifier called a Gröbner Base, which is calculated using an algorithm due

¹Actual systems may use *freeze* techniques in conjunction with constraint solving. This is done either to restrict the application of the comparatively heavy constraint solver, or to shield solvers that use incomplete methods.

REFERENCES

to Buchberger [Buc83]. These methods were shown to be compatible with the CLP(X) scheme for sequential constraint logic programming [SaA89], and have the advantage of being complete with respect to deciding satisfiability and almost complete for entailment. Specifically, Gröbner Base solvers can handle non-linear polynomials in contrast to solvers based on Gaussian elimination such as the one used in CLP(\mathbb{R}). The Buchberger Algorithm is NP-hard, but it exhibits a high degree of apparent parallelism. On the other hand, its performance is extremely sensitive to the order in which constraints are encountered, and to the scheduling of its subcomputations. We discuss the use in GDCC of a Buchberger Algorithm/Gröbner Base constraint solver for rational polynomials, and our attempts to parallelize it.

References

- [BeP89] H. Beringer and F. Porcher. A Relevant Scheme for Prolog Extensions: CLP (Conceptual Theory). In *6th International Conference on Logic Programming*, pages 131-148, 1989.
- [Buc83] B. Buchberger. Gröbner bases: An Algorithmic Method in Polynomial Ideal Theory. Technical report, CAMP-LINZ, 1983.
- [DiH88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertheir. The Constraint Logic Programming Language CHIP. In *Proceedings FGCS-88*, 1988.
- [JaL87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, Munich, January 1987.
- [Mah87] Michael J. Maher. Logic Semantics for a Class of Committed-choice Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 858-876, Melbourne, May 1987.
- [SaA89] K. Sakai and A. Aiba. CAL: A Theoretical Background of Constraint Logic Programming and Its Applications. *Journal of Symbolic Computation*, 8:589-603, 1989.
- [Sar89] V. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.
- [SaS88] Y. Sato and K. Sakai. Boolean Gröbner Base, February 1988. LA-Symposium in winter, RIMS, Kyoto University.
- [Sat90] Y. Sato. Extension of SetCAL, September 1990. Joint American-Japanese Workshop on Parallel Knowledge System and Logic Programming, ICOT, Tokyo.
- [Wal89] C. Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *6th International Conference on Logic Programming*, pages 181-198, 1989.

Introduction to CLP

- Natural extension of LP
- Powerful language for imbedding constraints
- More expressive
- Ex: CLP(\mathcal{R}), Prolog II/III, CHIP, CAL

Constraint Logic Programming and
Its Parallel Implementation

David Hawley@ICOT 4th Laboratory

October, 1990.

- Introduction to CLP
- Parallel Language Model
- Parallel Solver
- Further Work

Example: Mortgage

```
mortgage(P, Time, I, MP, B) :-  
  0 <= Time, Time <= 3,  
  Total_Int = Time * (P*I/1200),  
  B = P + Total_Int - (Time*MP).  
mortgage(P, Time, I, MP, B) :-  
  Time > 3,  
  Q_Int = 3*(P*I/1200),  
  mortgage(P+Q_Int-3*MP, Time-3, I, MP, B).  
?- mortgage(100000,6,10,2000,B).  
  B = 92912.50  
?- mortgage(P,24,12,MP,P/2).  
  MP = 0.0284*P
```

CLP(X) Scheme

CAL

- Domain/Constraint Prerequisites:
 - Satisfaction Complete
 - Solution Compact
 - Ex: Finite domains, real linear eqns, boolean
- Results (extended SLDNF)
 - lfp = least model = success set
 - $\overline{\text{ogfp}}$ = greatest model = finite fail set
- Domain/Constraint Prerequisites:
 - Satisfaction Complete
 - Canonical form
 - Ex: rational non-linear, boolean, (co)finite sets
- Results (extended SLD)
 - lfp = least model = success set

CLP Research at ICOT

- To Date
 - Algebraic CAL
 - Boolean CAL
 - Set CAL
- In Progress
 - Hierarchical constraints
 - Dependency analysis
 - Combining constraints/solvers/domains
 - (Co)finite sets with variable elements
 - Parallel constraint solvers
 - Parallel constraint logic languages

Motivation for Parallelization

- Parallelism for solvers
 - Control of computation
 - applicability of clause
 - flow of "information"
 - control constraint order
- Increasing $\left\{ \begin{array}{l} X=4, X+Y=7, X-Y=1. \\ X+Y=7, X=4, X-Y=1. \\ X+Y=7, X-Y=1, X=4. \end{array} \right.$
Difficulty
- More concurrency in operational semantics

Config: $(a_1 \cdots a_i \cdots a_n, S)$

Clause: $h :- c, b.$

Reduce by 1. $S' = \text{solve}(S \cup c \cup a_i = h)$

2. Config: $(a_1 \cdots b \cdots a_n, S')$

Concurrent Constraint Languages

- Model computation as cooperation between agents via a constraint store
- “Telling” and “Asking”

Config: $(a_1 \cdots a_i \cdots a_n, S)$

Clause q : $h :- g_A ? g_T \mid c_T, b$.

Reduction of a_i by candidate q

Config: $(a_1 \cdots b \cdots a_n, S \cup g_T \cup c_T \cup a_i = h)$

Defn - q is a candidate for a_i

when $S, a_i = h$ entails g_A

and $S, a_i = h$ answers g_T .

GDCC

- Concurrent constraint language
- Don't care commit
- KLL1 implementation (parallel)
- Plug-in solvers for multiple domains

. :- domains alg(=).

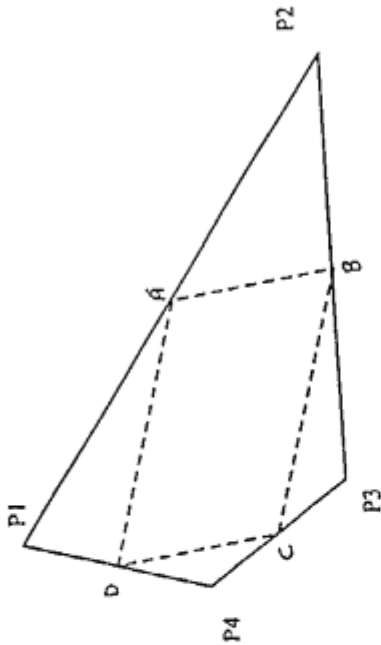
```
{create alg Ax,Ay,Bx,By,Cx,Cy,Dx,Dy}
start(P1,P2,P3,P4) :- true |
  check_para(A,B,C,D),
  mid(P1,P2,{Ax,Ay}), mid(P2,P3,{Bx,By}),
  mid(P3,P4,{Cx,Cy}), mid(P4,P1,{Dx,Dy}).
```

```
{input alg X1,Y1,X2,Y2,X3,Y3,X4,Y4}
check_para({X1,Y1},{X2,Y2},{X3,Y3},{X4,Y4})
  alg:(X1-X2)*(Y3-Y4)=(Y1-Y2)*(X3-X4),
  alg:(X1-X4)*(Y2-Y3)=(Y1-Y4)*(X2-X3) |
  true.
```

```
{input alg X1,Y1,X2,Y2,X3,Y3}
mid({X1,Y1},{X2,Y2},{X3,Y3}) :- true |
  alg:2*X3=X1+X2, alg:2*Y3=Y1+Y2.
```

```
?- para:start({X1,Y1},{X2,Y2},{X3,Y3},{X4,Y4}),
  create_alg(X1,X2,X3,X4,Y1,Y2,Y3,Y4).
```

GDCC



`:- domains alg(=).`

```
{create alg Ax,Ay,Bx,By,Cx,Cy,Dx,Dy}
start(P1,P2,P3,P4) :- true |
  check_para(A,B,C,D),
  mid(P1,P2,{Ax,Ay}), mid(P2,P3,{Bx,By}),
  mid(P3,P4,{Cx,Cy}), mid(P4,P1,{Dx,Dy}).
```

```
{input alg X1,Y1,X2,Y2,X3,Y3,X4,Y4}
check_para({X1,Y1},{X2,Y2},{X3,Y3},{X4,Y4}) :-
  alg:(X1-X2)*(Y3-Y4)=(Y1-Y2)*(X3-X4),
  alg:(X1-X4)*(Y2-Y3)=(Y1-Y4)*(X2-X3) |
  true.
```

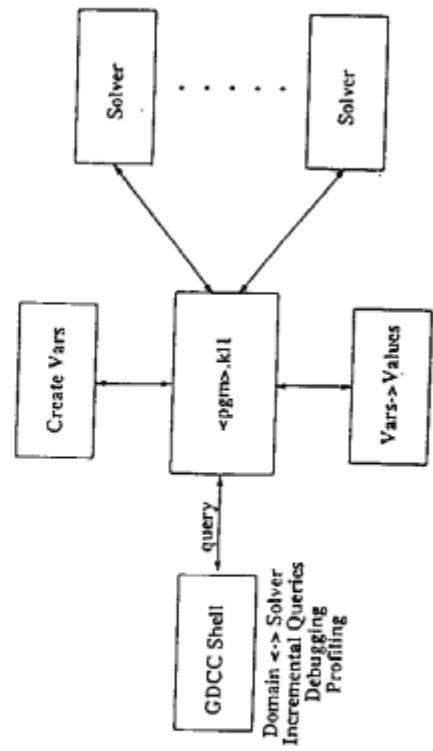
```
{input alg X1,Y1,X2,Y2,X3,Y3}
mid({X1,Y1},{X2,Y2},{X3,Y3}) :- true |
  alg:2*X3=X1+X2, alg:2*Y3=Y1+Y2.
```

```
?- para:start({X1,Y1},{X2,Y2},{X3,Y3},{X4,Y4}),
  create_alg(X1,X2,X3,X4,Y1,Y2,Y3,Y4).
```

GDCC Implementation

- Merge-tree stream to each solver
- Requests
ask(Symbol,Expr,CPU,S/F,Abort)
atell(Symbol,Expr,CPU,R/F,Commit,Abort)
tell(Symbol,Expr)

- Or-parallel guard test \rightarrow And-parallel test
- `r(Vars)` are object-level atoms



Example Domain & Solver

Domain: Algebraic Numbers

Constraints: Polynomials over Q

Solver: Grobner Base type

Grobner Base =_{def} confluent polys/rewrite rules.

Thm

Every soln of $p_1 = \dots = p_n = 0$ is a soln of p
 iff $\exists n \in \mathbf{N}$ s.t. $p^n \in I(\{p_1, \dots, p_n\})$

Corollary

$p_1 = \dots = p_n = 0$ has no soln
 iff $1 \in I(\{p_1, \dots, p_n\})$

Use Grobner Base to solve Ideal membership:

$p \rightarrow_{\text{GB}} 0$ iff $p \in I(\text{GB})$

But:

- Incomplete entailment
- Satisfiability test modifies GB

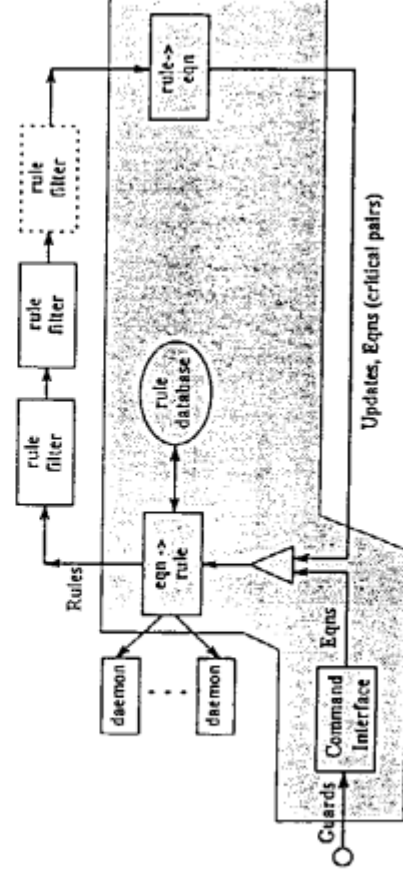
Buc^hberger Algorithm Version 1

Let P be a set of rules

Let $T = \{(p_j, p_k) \mid p_j, p_k \in P, j \neq k\}$

Forall $(L_j \rightarrow R_j, L_k \rightarrow R_k) \in T$
 if $\frac{L_j R_k}{\text{gcd}(L_j, L_k)} - \frac{L_k R_j}{\text{gcd}(L_k, L_j)} = S \neq 0$
 then

orient S and add to P .



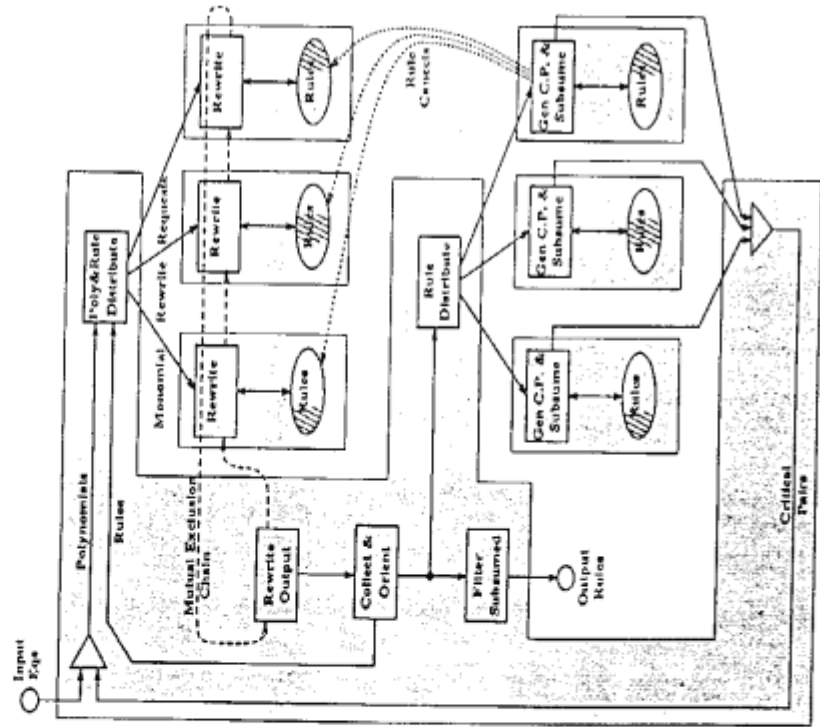
Evaluation: Version 2

Faster on simple problems

Slower on larger problems

Why? Rewrite/CP using unreduced rules

Bad information/parallelism tradeoff



Summary & Further Work

- CLP: unification \rightarrow constraint solving
- GDCC: cc language imbedded in KL1
 - More concurrent host \leftrightarrow solver interface.
 - Parallelism for solvers

Parallelizing Grobner Base is difficult.

- Control of computation
- Need additional control inside solvers.

Further Work:

- Algorithms for complete entailment, varieties of parameterized queries
- Other domains
- Other control structures: Blocks, Andorra principle, etc.

Mapping Applications onto Various Parallel
Architectures using Functional Language
and Program Transformation

Professor John Darlington
Department of Computing
Imperial College, London

Abstract

We are addressing the general problem of how applications can be mapped systematically onto a variety of parallel architectures in a manner that reconciles the need for efficient execution with the maintenance of desirable software characteristics such as comprehensibility, modifiability and portability.

The approach we are adopting is to use functional programming to serve as a general model of parallel computation and identify the characteristics of particular parallel machines with particular restricted subsets of the language and to develop program transformation techniques to convert general programs into the required forms.

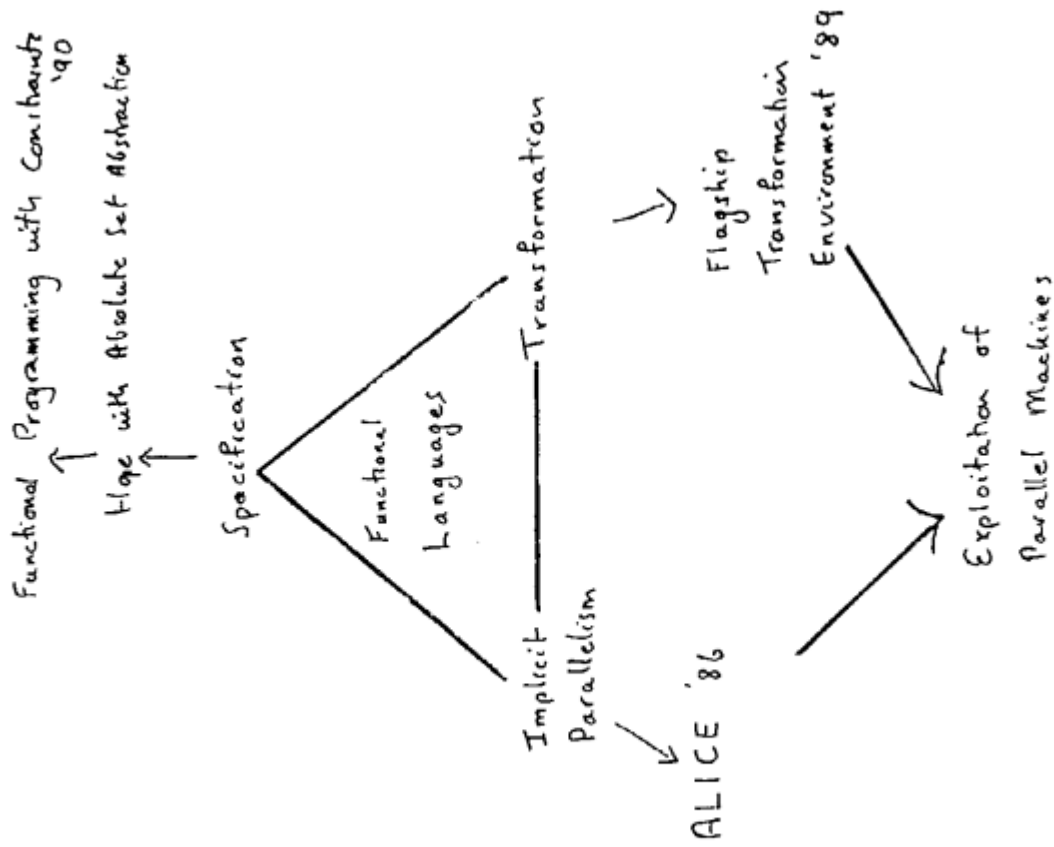
The appropriate program forms are expressed as skeletons, or general purpose higher order functions, which serve as algorithm building blocks, abstracting away many of the troublesome features of parallel machines.

Sequential implementations of the skeletons establish their declarative semantics and optimised parallel implementations are provided on suitable machines. Expression in a skeleton, however, is often not sufficient to achieve optimal performance and other, pragmatic, information is often required. We are investigating methods whereby such information can be expressed abstractly, as a requirement the system should attempt to satisfy, rather than explicitly, as a prescriptive way of meeting the requirement. Thus the compiler/loader/run time system is left with as much freedom as possible to satisfy these requirements with benefits to comprehensibility, adaptability and portability.

We are currently investigating dynamic MIMD machines, static MIMD machines and SIMD and systolic machines. Programs are expressed in the functional language Hope+, transformations are implemented on the Hope+ Transformation Programming Environment and the various parallel implementations are accomplished via the Hope+ to C Compiler.

Mapping Functional Programs onto Parallel Machines

John Darlington



Application to Parallel Exploitation

Functional language/transformation approach

- identify subsets of functional program forms that correspond to efficient forms for the machine type of interest
- use transformation to convert general program/specification into appropriate form and manipulate to fit machines physical characteristics

Diversity of parallel machine types

No common language/software methodology

Explicit control of parallelism disastrous for software productivity/reliability/portability
but necessary for performance?

GOAL

- a uniform software methodology applicable to all types of parallel machines
- systematic
- producing efficient implementations

SKELETONS

Objectives:

- Provide a set of algorithmic forms which implicitly specify process-granularity and interconnectivity.
- Implement skeletons for a wide class of parallel machines: closely-coupled to loosely-coupled to distributed networks of parallel machines (heterogeneous architectures).
- Use transformation technology to derive skeletal implementations of programs from initial high-level functional-language form.

Advantages of Skeletal Forms:

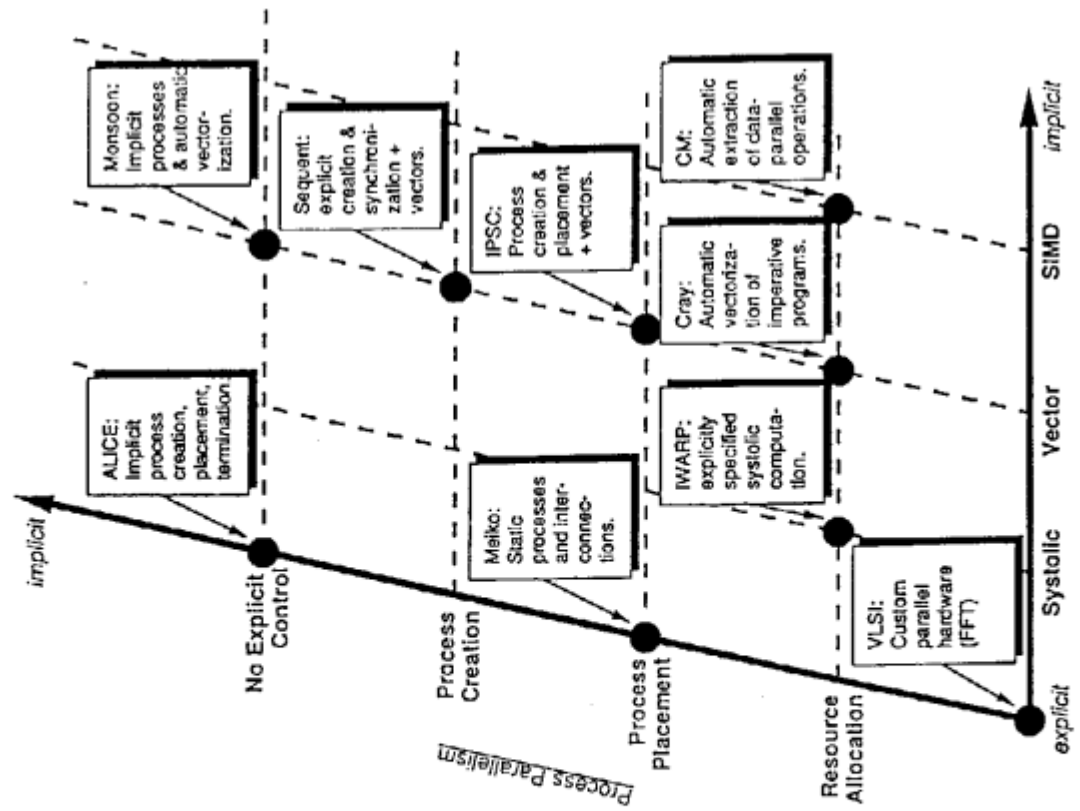
- Abstract architectural features.
- Abstract software constructs associated with conventional parallel programming: process description, placement, interaction.
- Re-usable forms which are easy to use & understand.
- Provide target forms for machine-assisted transformation environments.
- Enable highly optimised implementations.
- Structure transformation development.

Advantages of Functional Skeletons:

- Higher-order functions provide ready mechanism for delining skeletal forms.
- Inter-process communication elegantly modelled by streams.
- Parallel programs are deterministic and side-effect free.
- Sequential implementation establishes semantics, variety of equivalent parallel forms possible.

A Skeletal Approach to Parallelism

Parallel Machines & Parallelism



Data Parallelism

3 Types of Parallel Architectures:

- *MIMD* (e.g. Meiko, Sequent, distributed Suns).
- *SIMD* (e.g. MasPar, Dap, Connection-Machine).
- *Heterogeneous* (interconnected nodes of any of the above).

MIMD Skeletal Forms:

- *Static:* processes are fixed (in number as well as location).
- *Dynamic:* processes created at run-time when and where necessary.

Modelling Inter-process Communication:

- *Streams:* lazy-lists constructed with head-strict constructors. Simulates fixed process interconnectivity.
- *Bags:* non-deterministically-accessed structures which model the run-time dependent arrival of messages. Simulates general process interconnectivity through a switching-network.

MIMD: Static Skeletons:

- *Pipeline:* linear series of processes.
- *Mesh:* 2-D generalization of Pipeline.
- *Cooperating Specialists:* fully general static process network.

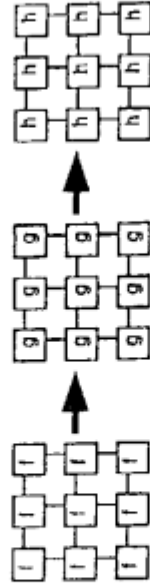
A Skeletal Approach to Parallelism

Pipeline:



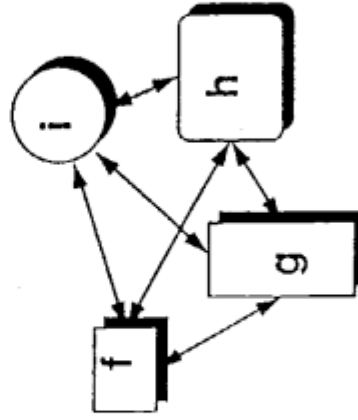
Pipe : $\text{seq}(\text{list}(_) \rightarrow \text{list}(_) \times \text{list}(\alpha) \rightarrow \text{list}(\beta))$

Mesh:



Mesh: $\text{seq}((\text{Nat} \times \text{Nat}) \times (_ \times _ \times _ \times _) \times _ \rightarrow _) \times (\text{Nat} \times \text{Nat}) \times \text{list}(\alpha) \rightarrow \text{list}(\beta)$

Cooperating Specialists:



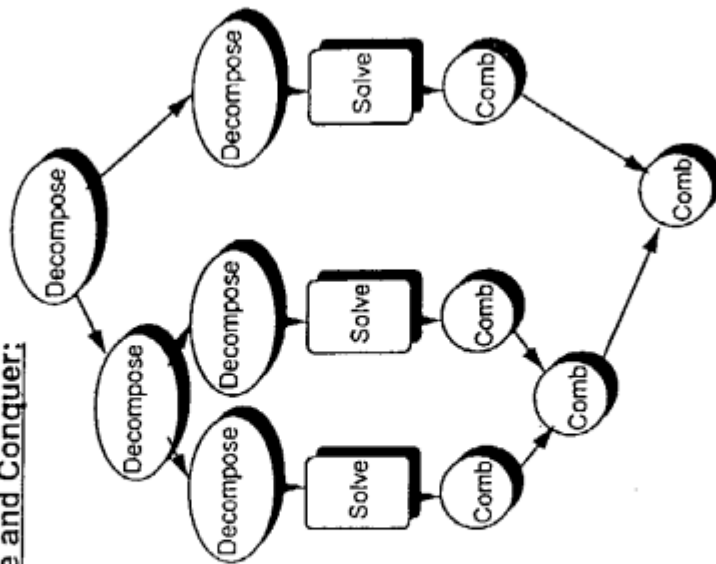
CS : $\text{seq}(\text{bag}(_) \times _ \rightarrow \text{bag}(\text{Nat} \times _) \times _) \rightarrow \text{seq}(_)$

A Skeletal Approach to Parallelism

MIMD: Dynamic Skeletons:

- *Divide-and-Conquer*. Classic problem decomposition skeleton.
- *Master-Slave*: General model for parallel exploitation of tasks.

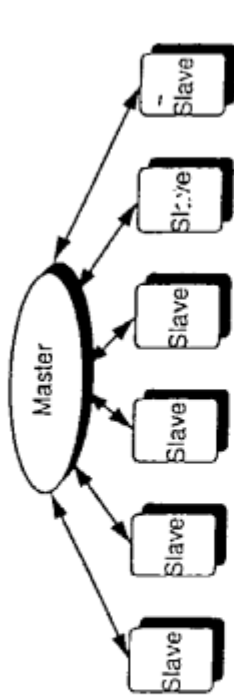
Divide and Conquer:



Solve $(\alpha \rightarrow \beta) \times (\text{list}(\beta) \rightarrow \beta) \times (\alpha \rightarrow \text{list}(\alpha)) \times (\alpha \rightarrow \text{Bool}) \times \alpha \rightarrow \beta$
 Decompose $(\alpha \rightarrow \beta) \times \gamma \rightarrow (\text{bag}(\alpha) \times \gamma)$
 Comb $(\alpha \rightarrow \beta) \times \gamma \rightarrow (\text{bag}(\alpha) \times \gamma)$
 Trivial $\text{Prob} \rightarrow \text{Result}$

A Skeletal Approach to Parallelism

Master-Slave:



Master $(\text{bag}(\beta) \times \gamma \rightarrow (\text{bag}(\alpha) \times \gamma)) \times (\alpha \rightarrow \beta) \times \text{bag}(\alpha) \times \gamma \rightarrow \gamma$
 Slave $\text{Initial State} \rightarrow \text{Result}$

SIMD Skeletons:

- Exploit *data-parallelism*: parallelism derived from applying a single operation concurrently to elements of a common data structure.
- Operate on array data structures distributed across SIMD plane of processors.

Why Data-Parallelism?

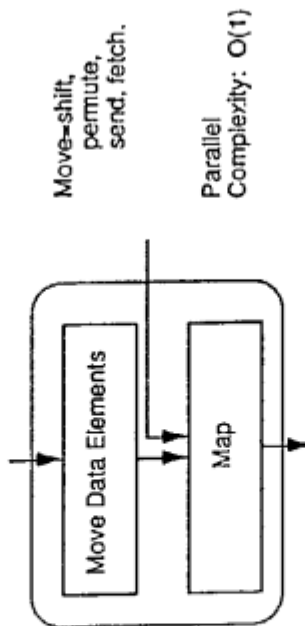
- Extremely low overhead & efficient (single instruction decoded for massive parallelism).
- Allows for highly-efficient use of array data-structures in functional languages.
- Implicit form of parallelism extremely suitable for applications in natural and applied sciences.
- Fine-grained approach to parallelism extraction often easier than coarse process-oriented (MIMD) approach.

A Skeletal Approach to Parallelism

SIMD Architecture Characteristics:

- Thousands of simple processing elements synchronously executing the same globally-broadcast instruction.
- Each processor is locally interconnected.
- General interconnection network for non-local communication may be available.

General Data-Parallel Operation



Functional Representation

- Functional languages naturally regard arrays as monolithic structures and data-parallelism is exploited through composition of operations on arrays.
- Functional representations exist for instruction broadcasting (map) and data-movement (shift, send, fetch).
- Resulting data-parallel programs are amenable to transformation: derivation of algorithms from high-level specifications or optimizations (e.g systolic algorithms).

A Skeletal Approach to Parallelism

Primitive Operations:

newarray: $\text{Nat} \times \alpha \rightarrow \text{array}(\alpha)$
 bound: $\text{array}(\alpha) \rightarrow \text{Nat}$
 select: $\text{Nat} \times \text{array}(\alpha) \rightarrow \alpha$
 imap: $(\text{Nat} \times \alpha \rightarrow \beta) \times \text{array}(\alpha) \rightarrow \text{array}(\beta)$
 zip: $\text{array}(\alpha) \times \text{array}(\beta) \rightarrow \text{array}(\alpha \times \beta)$
 shift: $\text{Nat} \times \alpha \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$
 send: $\text{array}(\text{Nat}) \times \text{array}(\alpha) \rightarrow \text{array}(\text{list}(\alpha))$
 fetch: $\text{array}(\text{Nat}) \times \text{array}(\alpha) \rightarrow \text{array}(\text{list}(\alpha))$
 extend: $\text{Nat} \times \alpha \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$
 trunc: $\text{Nat} \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$

Derived Operations:

map: $(\alpha \rightarrow \beta) \times \text{array}(\alpha) \rightarrow \text{array}(\beta)$
 map2: $(\alpha \times \beta \rightarrow \gamma) \times \text{array}(\alpha) \times \text{array}(\beta) \rightarrow \text{array}(\gamma)$
 update: $\text{Nat} \times \alpha \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$
 rotate: $\text{Nat} \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$
 scan: $(\alpha \times \alpha \rightarrow \alpha) \times \alpha \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$
 reduce: $(\alpha \times \alpha \rightarrow \alpha) \times \alpha \times \text{array}(\alpha) \rightarrow \alpha$
 permute: $\text{array}(\text{Nat}) \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$
 scatter: $(\alpha \times \alpha \rightarrow \alpha) \times \alpha \times \text{array}(\text{Nat}) \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$
 gather: $(\alpha \times \alpha \rightarrow \alpha) \times \alpha \times \text{array}(\text{Nat}) \times \text{array}(\alpha) \rightarrow \text{array}(\alpha)$

A Skeletal Approach to Parallelism

Transformation Algebra:

(map f) (map g) = map ($f \circ g$)

(shift i) (shift j) = (shift $i+j$)

(map f) (shift i) = (shift i) (map f)

(map f) (permute ns) = (permute ns) (map f)

(reduce f) (map g) = g (reduce f) iff g dist. through f

(map f) (scatter g b ns) = (scatter g b ns) (map f)

(map f) (gather g b ns) = (gather g b ns) (map f)

Iteration promotion (enables map with tail-recursive functions)

Image Averaging Example:

A graphics screen is represented by a 2-D array with a value at each element corresponding to the intensity of the corresponding pixel on the screen. We wish to yield an average of the pixel values as follows:

	($i-1, j$)	
($i, j-1$)	(i, j)	($i, j+1$)
	($i+1, j$)	

$$x(i, j) \leftarrow \frac{x(i, j) + x(i-1, j) + x(i+1, j) + x(i, j-1) + x(i, j+1)}{5}$$

The corresponding program using higher-order functions:

```
North = (-1, 0), East = (0, 1),
West = (0, -1), South = (1, 0)
```

```
Average(xs) = map (/5),
              fold(map2 (+), [shift(North, xs),
                              shift(East, xs),
                              shift(West, xs),
                              shift(South, xs),
                              xs])
```

The function Average can calculate the average of an image in 9 steps. The complexity of Average on a SIMD machine is $O(1)$.
(End of Example)

THE DECLARATIVE EXPRESSION OF

OPERATIONAL CONCERNS

Systolic Algorithms:

- A restricted class of data-parallel algorithms where all communication is nearest-neighbour and data-invariant.
- Highly efficient systolic algorithms exist for a wide variety of numerical problems (e.g. matrix-multiplication, LU-decomposition, etc.)
- Functional-language data-parallel programs can be transformed into systolic equivalents via guaranteed meaning-preserving transformations.

Derivation of Systolic Algorithms:

- Data-parallel operations can be divided into systolic & non-systolic.
- Original problem specified as high-level specification.
- Initial algorithm may use both systolic and non-systolic operations.
- Transformation is used to change algorithm into form using only systolic operations. Systolic nature of an algorithm is therefore a static property of the program!

Systolic Skeleton:

SYSTOL : $(list(\alpha), (Nat \times Nat) \times (list(\alpha) \times (Nat \times Nat) \times array(\alpha) \times (\alpha \times \alpha \times \alpha \rightarrow \alpha) \rightarrow array(\alpha))$

Specifies input streams & direction (North, East, West, South) & function applied at each processor.

A Skeletal Approach to Parallelism

- Expression in a skeleton is not always sufficient to ensure optimal implementation
- Compiler/loader/run-time system needs to be given help to optimise run-time behaviour
 - e.g.
 - Process locality (minimise communication)
 - Grain size
 - Type of processor required (vector, array, SPARC)
- Can we state the appropriate information, abstractly, as a requirement to be satisfied rather than explicitly as a prescription for (one particular) solution
 - Annotations - Properties of Programs, Execution Requirements
 - Meaning Preserving
 - Declarative - What not How
 - Designed to interface with high-level system interface capable of satisfying the requirements


COMPILATION ROUTE

Benefits			
<ul style="list-style-type: none"> • Focus on essential properties • Separation of declarative and operational concerns • Flexibility of implementation • Comprehensibility, adaptability, portability 	<ul style="list-style-type: none"> • Hope+ to C compiler • New back-end to Hope+ FPM compiler <ul style="list-style-type: none"> • Via macro set • Flexibility, portability • Skeletons implemented with <ul style="list-style-type: none"> • Magic functions • Via communication / load balancing primitives expressed as Hope+ continuations 	<ul style="list-style-type: none"> • CS - tools • Functional bodies → C code • Communications → C library calls • Annotations → C library run time / load balancing calls 	
e.g.			
Skeleton	Property	Annotation	
Divide & Conquer Farm	Locality	Near, Same, Don't Care	
Pipeline Mesh	Grain Size	Relative Complexity	
	Inter-Process Communications	Process Independence	
	Processor Allocation	Share of total Processors	
		Processor Requirement	

Transformations Studied

⇒ Coarse grained
Divide & Conquer

⇒ Pipeline
mesh

⇒ SIMD ⇒ Systolic


⇒ Message Passing

Second Joint ICOT/DTI-SERC Workshop

Title: Future Direction of Parallel Symbolic Processing

The aim of the panel: To identify the future direction of parallel symbolic processing from the aspects of architecture, programming languages, and application areas.

architecture: Do we need special architecture such as dataflow?

programming languages: Either conventional languages or high level languages?

application areas: Do any significant application areas exist which need parallel symbolic processing?

final question: Do you think that parallel symbolic processing will bring an innovative tool for human being and/or human society?

Guide to the Japan & UK Demonstrations

Second Joint ICOT/DTI-SERC Workshop

on

Decomposition of Parallel Applications

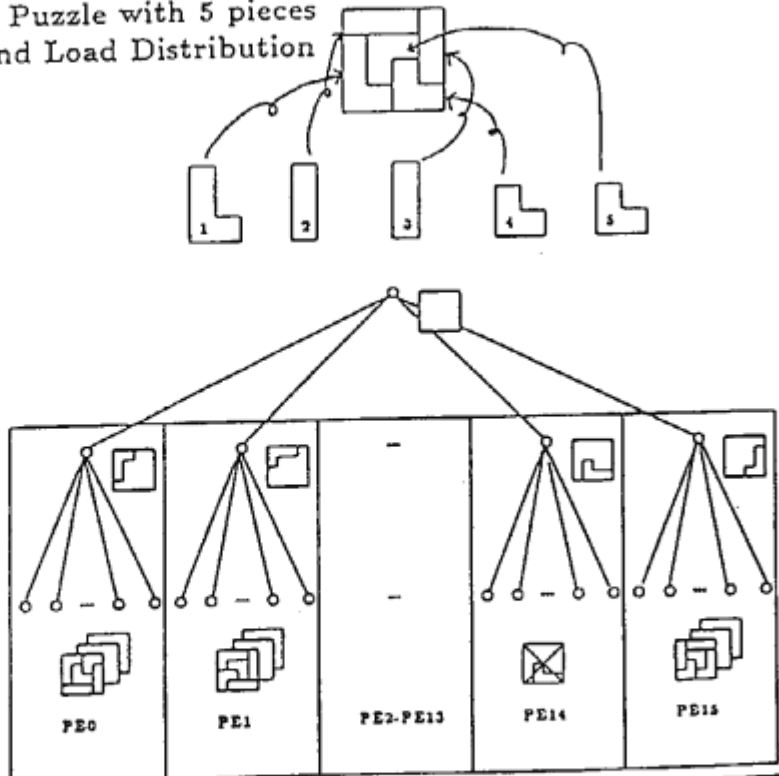
and

Benchmarking and Evaluation of Parallel Systems

Wednesday, October 17, 1990

A List of Japanese Demonstrations

1. Pentomino—Packing Piece Puzzle Solver
2. Bestpath—Shortest Path Problem Solver
3. Experimental system of parallel version of computer Go-playing program : GOG
4. Parallel LSI-CAD demonstration program (1) : LSI router
5. Logic-level simulator of LSI circuits : A parallel application program in LSI CAD
6. Experimental System of Parallel Legal Reasoning using Precedents
7. Genome Analysis Program (1) : Multiple Sequence Alignment by 3-Dimensional DP-matching
8. Genome Analysis Program (2) : Multiple Sequence Alignment by Parallel Simulated Annealing
9. Constraint Logic Programming Experimental System : CAL
10. Molecular Biological Database in Kappa

Title	Pentomino — Packing Piece Puzzle Solver
Purpose	Dynamic load balancing scheme for OR-parallel search programs is studied. Multi-level load balancing scheme is proposed, and evaluated by implementing all-solution exhaustive search Packing Piece Puzzle (Pentomino) solver program.
Outline & Features	<p>Packing Piece Puzzle is a puzzle, consisting of a rectangular box and a collection of pieces with various shapes. The problem is to find all possible ways to pack the pieces into the box. This puzzle is known as the Pentomino puzzle, when the pieces are all made up of 5 squares. This is a typical OR-parallel search program. A multi-level dynamic load balancing scheme is developed to highly utilize the processors.</p> <p>Program structure: An OR-parallel exhaustive search.</p> <p>Load distribution: Tasks are generated by a master processing elements (PE), and are distributed to idle PEs, in order to balance work loads. To overcome the task supply bottleneck at the master PE, multi-level load balancing is introduced.</p>
System Configuration	<p>Packing Piece Puzzle with 5 pieces Search Tree and Load Distribution</p> 

1 Overview

In the demonstration, packing piece puzzle with 10 pieces (Fig.1) is solved with increasing number of processing elements (PEs), and speedup figures are shown.

The demonstration is carried out as follows.

- Program is executed on 16 processors with simple load balancing scheme.
- Load balancing can be observed real-time in the performance meter window.
- Program is executed on 64 processors with simple load balancing scheme.
- Task supply bottleneck can be observed in the performance meter window.
- Program is executed on 64 processors with multi-level load balancing scheme.
- Near-linear speedup is obtained.

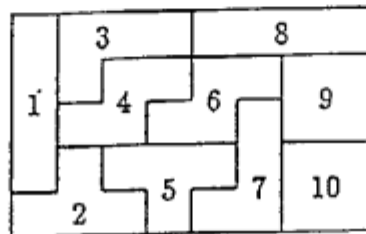


Figure 1: Packing Piece Puzzle

2 Description of the program

To solve this puzzle, the program starts with the empty box, and finds all possible placements of a piece to cover the square at the top left corner. Then, for each of those placement, it finds all possible placements of a piece (out of the remaining pieces) to cover the uncovered square which is the topmost leftmost, and so on until the box is completely filled. Each partly filled box defines an OR-node, where the possible placements of a piece to cover the uncovered topmost leftmost square define child nodes.

The program does a top-down exhaustive search of this OR-tree. Here, deepening the tree depth corresponds to place one piece in the box. The number of OR-nodes increases as the search level deepens, but since some OR-nodes are pruned when there are no more possible placements, number of OR-nodes decreases below a certain tree depth.

3 Load balancing scheme

Load balancing is done on master PE by partitioning a program into mutually independent subtasks (Subtask Generation), and by distributing subtasks to idle PEs so as to balance work loads (Subtask Allocation). To detect idle PEs, on-demand distribution method is

utilized. When a PE becomes idle, it sends a message to the master PE, requesting a new subtask. Subtask generation is done until the search reaches the certain depth in the tree. However, as the number of processors increases, the rate of subtask execution eventually becomes larger than the rate of subtask supply. In other words, subtask generation becomes a bottleneck.

To overcome this bottleneck, we have introduced multi-level load balancing scheme. Each subtask generator is in charge of a certain fixed number of processors, which form processor groups (PG). N processors are grouped into M processor groups, therefore, each PG is composed with $\frac{N}{M}$ PEs and a certain PE in a PG is called group master PE.

In Fig.2, two-level load balancing scheme is shown. At the first level distribution, super-subtasks are distributed to idle PEs to balance the loads of PGs. At the second level, subtasks are distributed to idle PEs to balance the loads of PEs which belong to a PG.

This scheme is scalable to any number of processors because of this multi-level structure.

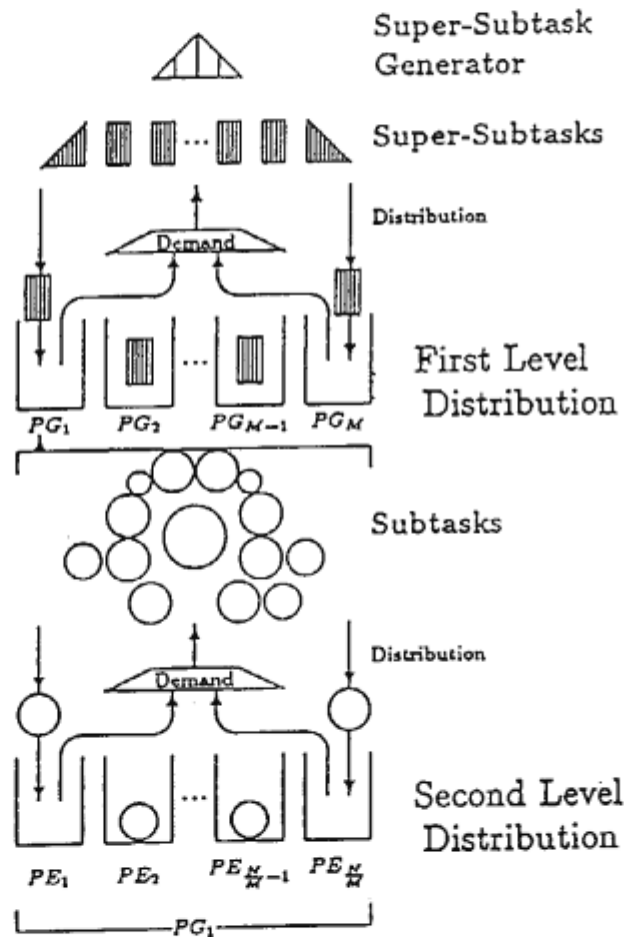


Figure 2: Structure of Multi-Level Load Balancing

4 Speedup Measurement

Execution times are measured for one-level load balancing and two-level load balancing. Speedup (S_N) is defined as the ratio of execution time on 1 PE (T_1) to N PEs (T_N), and calculated by $\frac{T_1}{T_N}$, and it is described in Figure 3.

Speedup of one-level load balancing becomes saturated because of the subtask generation bottleneck. However, it is improved by two-level load balancing, and near-linear speedups are obtained: 7.7 with 8 PEs, 15 with 16 PEs, 28.4 with 32 PEs, 50 with 64 PEs.

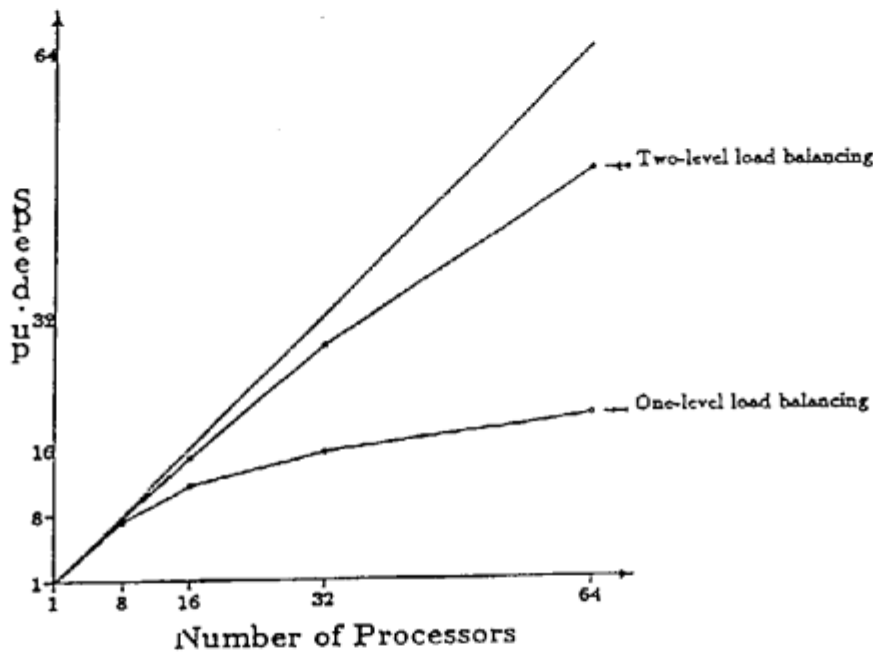


Figure 3: Speedups

5 Conclusion and Future Works

This scheme is efficient not only for OR-parallel search problems, but also applicable to general trees search problems including alpha-beta pruning problems, which does not involve frequent inter-processor communication.

This multi-level dynamic load balancing scheme is now available as a utility program to come with the operating system PIMOS.

Title	Bestpath — Shortest Path Problem Solver
Purpose	The problem of mapping intercommunicating processes on loosely-coupled multiprocessors is studied and evaluated by implementing a shortest path problem solver.
Outline & Features	<p>Problem : The single-source shortest path problem is to find the minimum cost paths between a given starting vertex and all other vertices of a graph in which each edge has a non-negative cost. Large-scale grid graphs with tens of thousands of vertices are used in the demonstration. Edge costs are given by random numbers as the test data.</p> <p>Algorithm : Processes corresponding to each vertex exchange messages with each other. Each message contains path and cost from the starting vertex. A priority is attached to each message so that a message with lower cost is sent earlier than one with higher cost. Each vertex remembers the shortest path notified so far by the messages and its cost.</p> <p>Load balancing : Three different static mapping strategies are tried to get high processor utilization with low interprocessor communication.</p>
System Configuration	<p>Example</p>

Outline

The single-source shortest path problem is to find the minimum cost paths between a given starting vertex and all other vertices of a graph in which each edge has a non-negative cost. In the demonstration, the grid graph consists of forty thousand vertices. Edge costs are given by random numbers.

In the demonstrated program, processes are generated for each vertex in a graph and computation is performed by exchanging messages between processes. This algorithm requires less computation than the algorithm in which processes are forked for each candidate path. Priority control efficiently prunes the search branches, so the algorithm is as computationally efficient as Dijkstra's algorithm.

Algorithm

A message contains the path from the starting vertex to the receiver vertex and its cost. The computation is initiated by sending a message with an empty path and zero cost to the starting vertex. All the vertices remember the minimum cost to reach the vertex notified by the messages received so far. Initially, the cost remembered by each vertex is infinite (Figure 1).

When a message arrives at a vertex and the cost notified by the message is smaller than the minimum cost remembered in the vertex, the new cost is remembered and messages with better paths and costs are sent further to the adjacent vertices (Figure 2). If the message has a larger cost value than the known minimum, it is simply discarded.

Since a message is represented by a process, sending a message means creating a message process, while receiving a message means executing a message process. Each message process has a priority decided by the cost: a message with a lower cost is received earlier than one with a higher cost.

When there are no messages left in the graph, each vertex has the shortest path from the starting vertex and its cost.

Load Balancing

The heaviest part of the processing is communication, which is initiated at the starting vertex and propagates gradually to the whole graph like wave propagation. So, the processor utilization is expected to be higher as the grid is divided into smaller blocks.

Conversely, when the grid is divided into blocks for mapping, interprocessor communication arises at the boundaries of the blocks. So the more the grid is divided into smaller blocks, the more interprocessor communication occurs.

The program tries to attain a good compromise between communication localization and processor utilization.

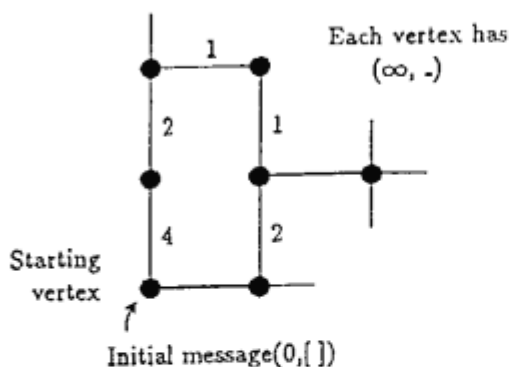
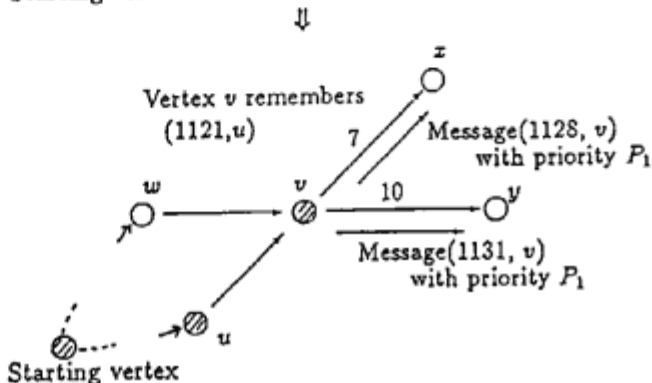
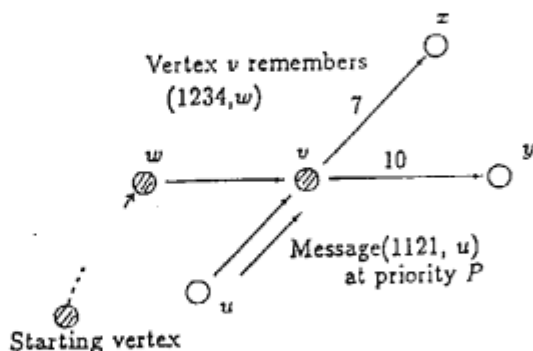


Figure 1: Initial state



Priority:
(higher) $P \geq P_1 \geq P_2$ (lower)

Figure 2: Message communication

Mapping Strategies

The following three mapping strategies are tried. In each mapping, $p = q^2$ processors are employed.

Two-Dimensional Simple Mapping

Divide the grid into $q \times q$ blocks and map each block onto the corresponding processor.

Two-Dimensional Multiple Mapping

Divide the grid into k super-blocks, each of which is again divided into p blocks just as in the two-dimensional simple mapping. Each processor is responsible for k blocks, one from each super-block.

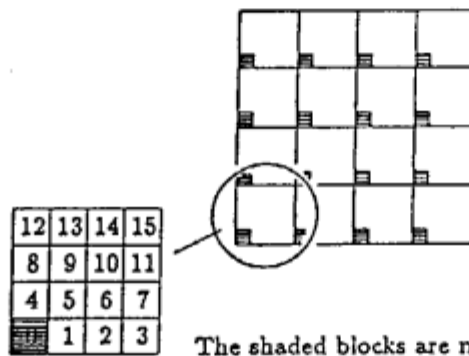
One-Dimensional Simple Mapping

Divide the grid simply as p narrow rectangular strips and map them onto the processors.

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

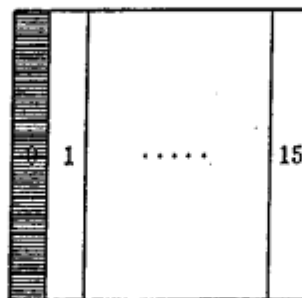
The shaded block is mapped onto processor 0.

Figure 3: Decomposition of a grid for two-dimensional simple mapping



The shaded blocks are mapped onto processor 0.

Figure 4: Decomposition of a grid for two-dimensional multiple mapping



The shaded block is mapped onto processor 0.

Figure 5: Decomposition of a grid for one-dimensional simple mapping

Performance Results for a 40,000-vertex grid

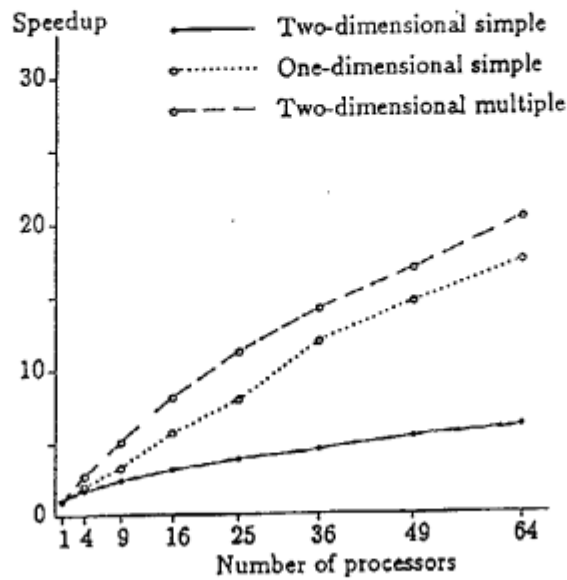


Figure 6: The speedup for various mappings and numbers of processors

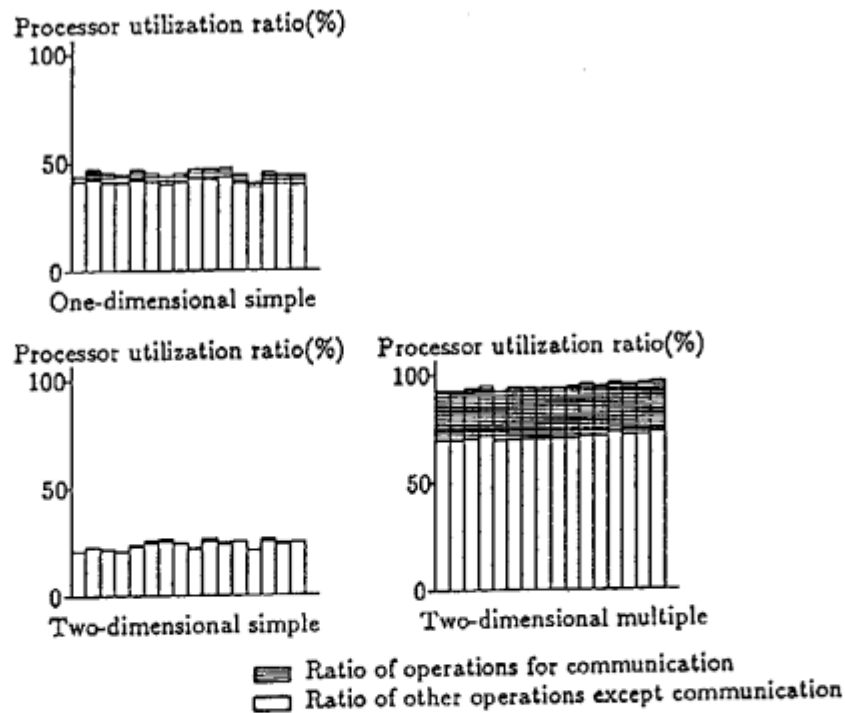
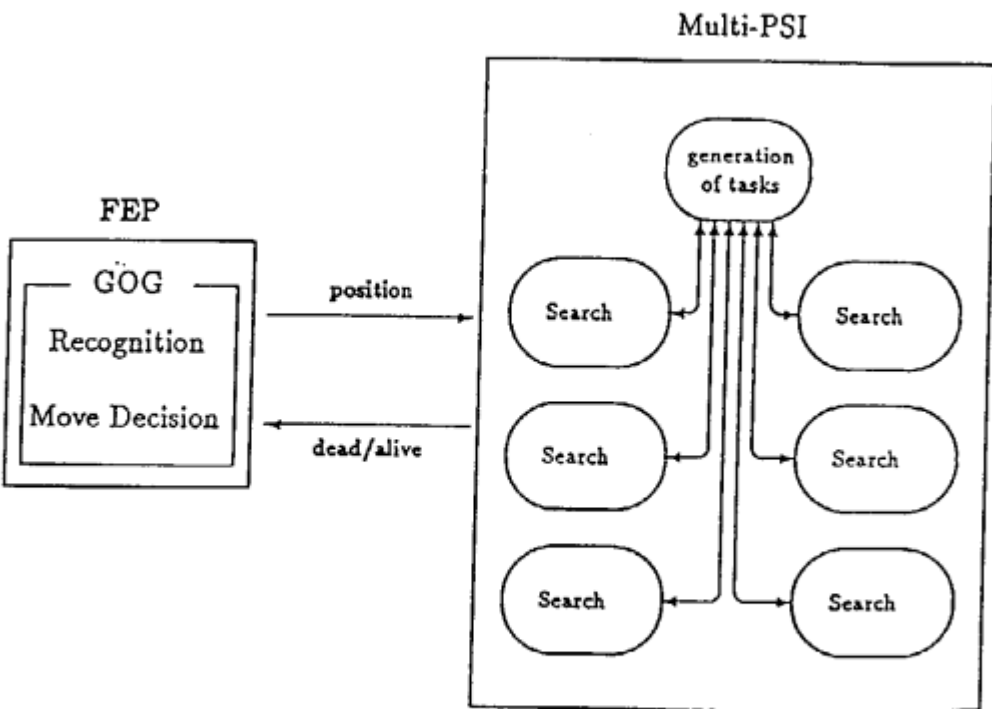


Figure 7: The processor utilization ratio for various mappings with 16 processors

Title	<p>Experimental system of parallel version of computer Go playing system "GOG"</p>
Purpose	<p>Go has been a difficult game for the computer to play. We are trying to build a strong Go program using the computer power of the parallel inference machines.</p>
Outline & Features	<p>[Outline]</p> <ul style="list-style-type: none"> • The intermediate results of parallel Go playing system "GOG" on parallel inference machine. • This research is being jointly developed with ETL. <p>[Feature]</p> <ol style="list-style-type: none"> 1. The tasks of small granularity tend to smooth out the load imbalance caused uneven tasks of large granularity. 2. The frontend sequential GOG system dispatches string search works to the Multi-PSI which works as a backend machine.
System Configuration	 <p>The diagram illustrates the system configuration. On the left is a box labeled 'FEP' containing a sub-box 'GOG' with 'Recognition' and 'Move Decision' components. An arrow labeled 'position' points from the 'GOG' box to the 'Multi-PSI' box on the right. Inside the 'Multi-PSI' box, there is a 'generation of tasks' oval at the top, which connects via six vertical lines to six 'Search' ovals arranged in two columns of three. An arrow labeled 'dead/alive' points from the 'Multi-PSI' box back to the 'FEP' box.</p>

Developing a computer Go playing system

We have been developing a sequential computer Go playing system called "GOG" on the sequential inference machine since 1985. The current system is stronger than a human Go beginner, but considerably weaker than a average-level amateur.

To make the present system stronger, GOG has to incorporate much more processing. But, if the system simply took in all those tasks, the execution time would be too long. In the parallel GOG system, we would like to incorporate more tasks and still keep the execution time relatively short.

A parallel version is now being designed to run on the parallel inference machines. Part of the system have already been parallelized. It is a intermediate result of computer Go system "GOG" of parallel inference machine.

The capture search

The outline of the process in which GOG determines its next moves is as follows.

- Recognition of board configuration
- Generation of candidates moves
- Decision of next move

In the recognition of board configuration, GOG determines which strings (connected stones) are in danger of being captured. And, for each such strings, it does a search to decide whether the string will be captured or not. This recognition is very important in this phase, because next move depends on the state of stones "dead or alive".

This system recognizes dangerous stones with DAME of 4 or less. Such a state (string with few DAMEs) occurred frequently and many dangerous stones are appeared at the same time. The capture search time accounts for roughly 40% of processing time of the board recognition phase. As the first step of parallelization, we wrote the capture search part in the concurrent logic language KL1.

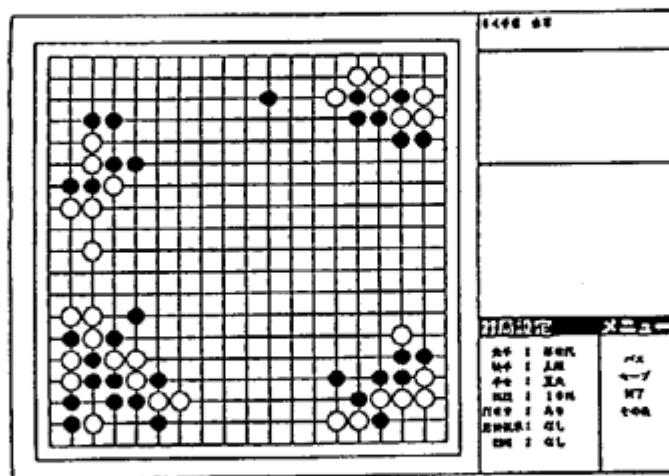


Figure1. There are many dangerous strings.

For efficient parallel execution

(1) We adopt a dynamic load balancing scheme to make the most of processors. There is a master processor to distribute search tasks to other processors. An idling processor requests work to the master processor, and receives a search task.

(2) All processors have a copy of the board, and updates it every time a move is made. This reduces interprocessor communication, since the master processor needs only to specify which string is the target of the search.

(3) After all dangerous cluster search tasks have been dispatched, a particular kind of plausible move generation tasks are dispatched. Those are "KESHI" candidate moves that may restrict the enemy's potential territories. Since KESHI candidate generation tasks are of smaller granularity than capture search tasks, they tend to smooth out the load imbalance due to search task size variance, thus making the processor utilization higher.

Front End Processor (FEP)

Multi-PSI

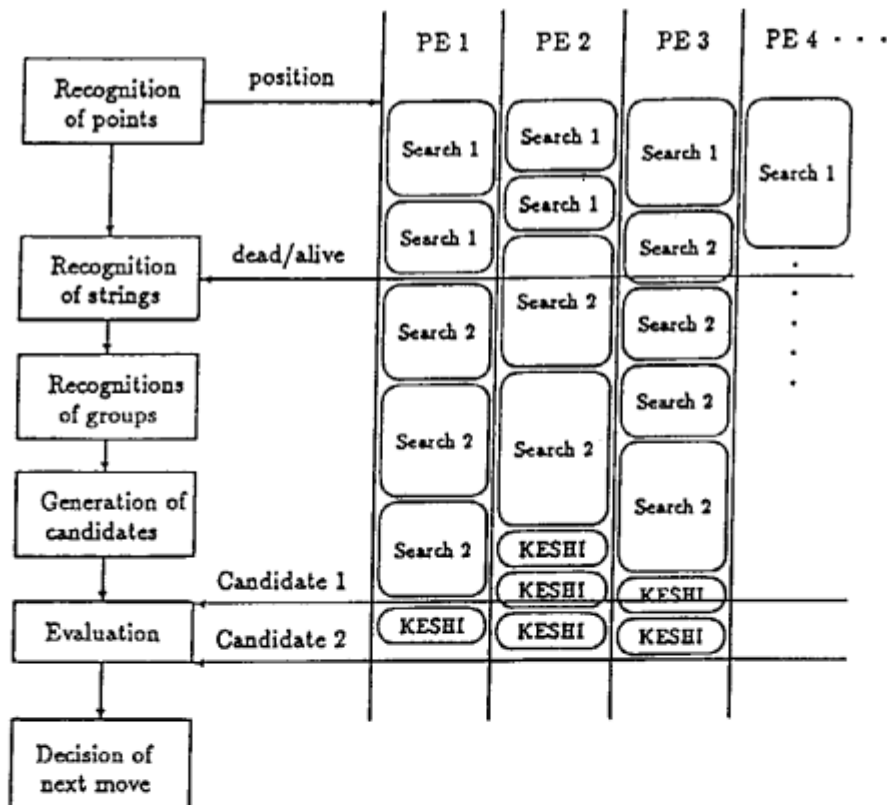


Figure 2. How the FEP and Multi-PSI processors cooperate to decide the next move

How to decide the next move

We have incorporated the parallel board recognition program into the sequential GOG system.

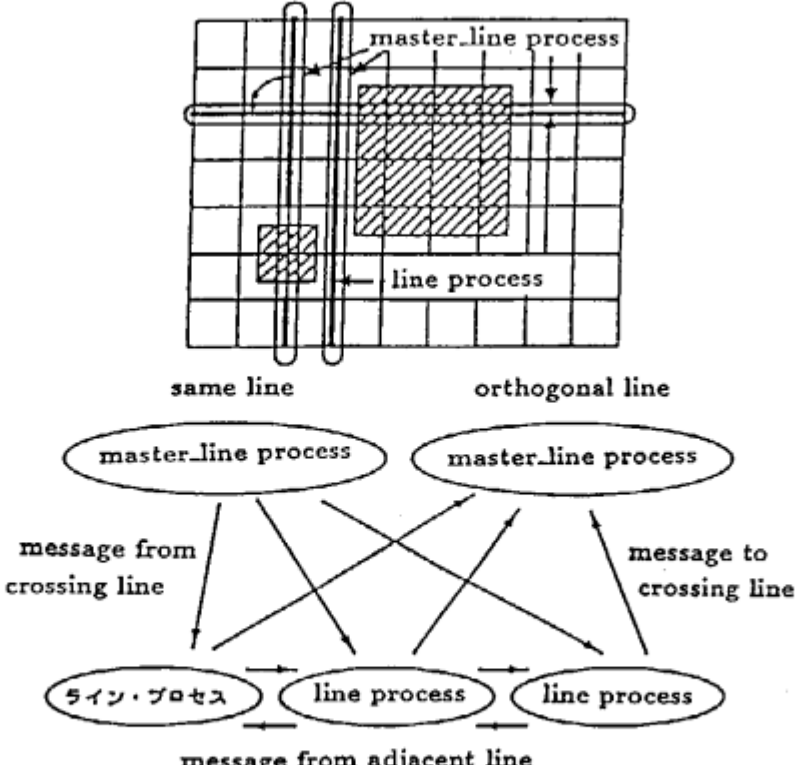
1. The frontend sequential GOG system notifies the enemy moves to the Multi-PSI master processor. Then, the master processor dispatches the capture search tasks and then KESHI candidate generation tasks to the other processors. The results are sent to the master processor and then to the frontend processor.

2. After the frontend sequential GOG system get the candidate moves from the Multi-PSI, it evaluates them with the candidates generated by itself. Then it decides the next move.

This experimental parallel GOG performs more processing than the sequential GOG system, but the processing time is kept roughly the same.

Demonstration

1. Recognition of the board configuration in parallel
2. Play with Multi-PSI

Title	<p style="text-align: center;">Parallel LSI-CAD demonstration program (1) LSI router</p>
Purpose	<p>A VLSI layout problem consists of several different problems that require massive computational power. Routing is one of those problems. Our aim is to study concurrent algorithms and load-balancing methodologies through design and development of parallel routing programs.</p>
Outline & Features	<p>[Abstract] This program executes routing between modules on an LSI chip, after the placement of each module has been fixed. It determines the connection paths between terminals of each module.</p> <p>[Concurrent algorithm] The Basic algorithm is a sequential line search, the look-ahead line search algorithm. It is expanded for parallel execution. Major parallelism is extracted from concurrent routing between nets.</p> <p>[Implementation] As this program is based on a kind of line search algorithm, processes are assigned to each line segment on each grid line as concurrent execution primitives. Intermediate results of routing are kept as inner statuses of each line process. Routing is executed by communication between these line processes. The master line processes stand for grid lines, and manage line processes on a corresponding grid, and relay those communication messages. Line drawing and rip-up correspond to dynamic split and joint of these line processes.</p>
System Configuration	 <p>The diagram illustrates the system configuration and communication flow. At the top, a grid shows a horizontal line and two vertical lines. A shaded rectangular area is labeled 'master_line process'. Below it, a smaller shaded area is labeled 'line process'. Labels 'same line' and 'orthogonal line' are placed below the grid. Below the grid, a flowchart shows two 'master_line process' ovals at the top and three 'line process' ovals at the bottom. The leftmost 'line process' is labeled 'ライン・プロセス' (Line Process). Arrows indicate communication: 'message from crossing line' points from the left 'line process' to the left 'master_line process'; 'message to crossing line' points from the right 'master_line process' to the right 'line process'; and 'message from adjacent line' points between the middle and right 'line processes'.</p>

[Routing problem]

Routing is one of the VLSI layout problems which determines connection paths between terminals of modules on an LSI chip. Routing is executed after placement of modules has been determined in an LSI design of gate arrays, standard cells, or building blocks. There are several well known algorithms for the problem such as maze routing, line search and channel routing. We assume two routing layers, one for vertical and the other for horizontal paths. We also assume that each connection must be routed on a virtual grid on a chip surface. The block and through hole inhibition conditions are also dealt with.

[Basic algorithm]

This program is based on a kind of line search algorithm, look-ahead line search. This algorithm calculates positions that are expected to lead to a good solution before routing each line segment. Figure 1 shows this process. Start point S and a target T are given. If a line drawn downward from S turns at A, then the reachable point that is closest to point T is point a. Similarly, if the line turns at point C, D then corresponding points are c and d for each. These points (a, c, d) are called expectation points for S. Note that as the through hole is inhibited at point B, so point b cannot be an expectation point for S. Of all these expectation points, point c is the closest one to point T, so point S and point C will be connected in this search step. Similar processes are followed and thus point S and point T will be connected. In addition to the above processes, this algorithm includes two more functions. One is to get out of local optimal point in expectation points calculation. The other is backtrack for escaping from a dead-end by removing the last-connected line and returning to the point visited last. Thus this algorithm guarantees the wireability between two terminals, if connection paths exist.

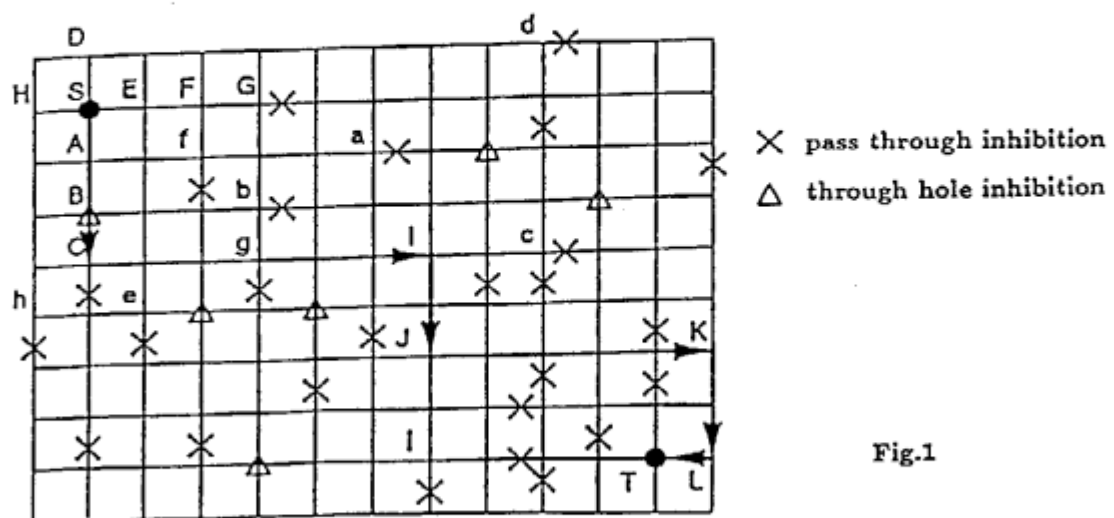


Fig.1

[Concurrent algorithm]

This program uses a parallelized version of the line search algorithm shown above. The program is designed to extract a parallelism of computation mainly from the concurrent search of multiple nets. On KL1 programming, the minimum execution unit is called process. We usually adopt an execution model in which the computation is executed by exchange of messages between these processes. This program also adopts this execution model. As our algorithm is based on the line search algorithm, so processes correspond to each lines on grid. Each line process maintains the corresponding line's status and at the same time the execution entity of search. As figure 2 shows, each process corresponds to each grid line and line segment on it. In this program, search and routing proceeds by the exchange of messages between these line processes. The routing process of one net is almost the same as that of the basic algorithm, but the computation of the expectation point, mentioned before, is parallelized. The computation of the best expectation point is executed in this program as follows. Request messages for calculation of expectation point are distributed from the line process now being searched to the line processes that cross it. Thus computation of expectation point is executed concurrently on each line process that received a calculation request message. Later, the result of each calculation will be returned from these line processes to the searching line process, then this line process aggregates these results and determines which is the best expectation point. When the best expectation point is found, the searching line is connected (fixed) to the crossing line that includes the best expectation point.

(Figure 3(a)-(d))

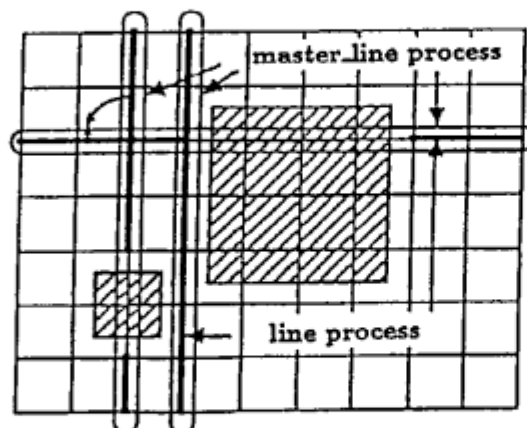


Fig.2

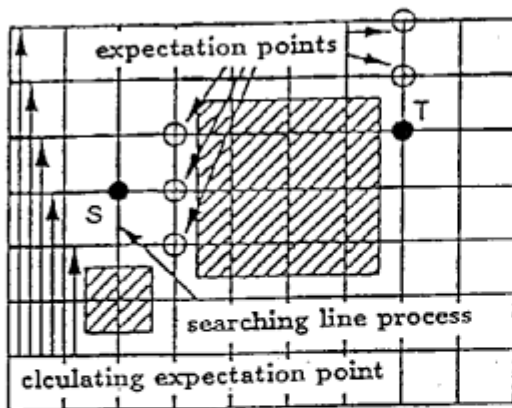


Fig.3(a)

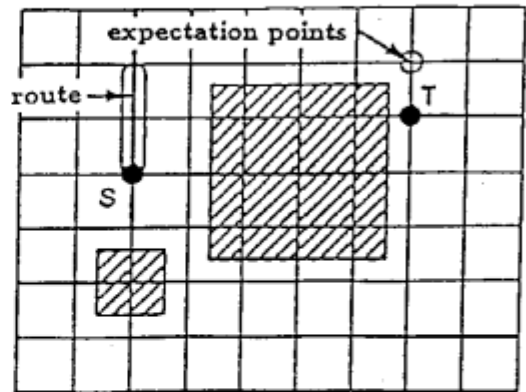


Fig.3(b)

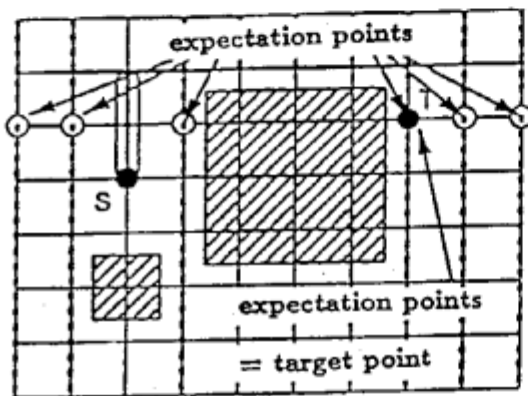


Fig.3(c)

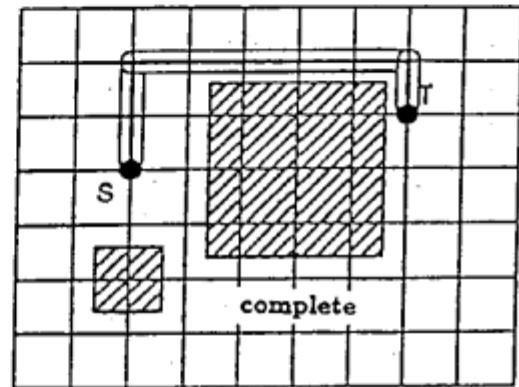


Fig.3(d)

Note again that the concurrency of computation is extracted mainly from routing of multiple nets, in other words, from parallel search for multiple nets. In this program, routing is scheduled to route nets in increasing order of their size, shortest net first and longest net last.

[demonstration]

The parallel routing program, written in KL1 on the Multi-PSI, executes routing of LSI chips of a practical size. Execution results will be shown on a display in real time.

[Kitazawa,H. and Ueda,K., "A LOOK-AHEAD LINE SEARCH ALGORITHM WITH HIGH WIREABILITY FOR CUSTOM VLSI DESIGN", proc. of ISCAS 85, pp1035]

Title	Logic-level simulator of LSI circuits : A parallel application program in LSI CAD
Purpose	<ul style="list-style-type: none"> • To construct a logic-level simulator in a concurrent logic language KL1 which can simulate practical-size logic circuits on the Multi-PSI. • To evaluate the virtual time mechanism, which is a parallel control mechanism for discrete event simulations.
Outline & Features	<p>The system simulates the behavior of logic circuits described at the logic-gate level, taking delay time of each gate into account. The virtual time mechanism is used in order to realize local time management in each processor. When a time reversal occurs, the rollback of event history is performed to maintain the correctness of the simulation.</p> <ol style="list-style-type: none"> 1. Event-driven simulation <ul style="list-style-type: none"> • To simulate the gate behavior only when its input signal changes (only when an event occurs) 2. Parallel control mechanism – virtual time mechanism <ul style="list-style-type: none"> • Local synchronization by passing messages between gate processes Each message has the information of new signal values and its changing time • Rollback will happen when a message arrives in incorrect order. (Rollback means rewinding the history of a gate process.) • Scheduling of message processing in one processor so as to decrease the frequency of rollback 3. Load balancing – static load balancing by preprocessing <ul style="list-style-type: none"> • To decrease the frequency of rollback • To decrease the inter-processor communications
System Configuration	<p><u>static load balancing by preprocessing</u></p> <pre> graph LR A(circuit data) --> B(dividing the circuit statically) B --> C(the divided circuit data) </pre> <p><u>parallel execution of simulation</u></p> <pre> graph LR D(the divided circuit data) --> E(simulation engine) F(input signal sequences) --> E E --> G(output signal sequences) </pre>

OUTLINE

Logic-level simulation of LSI circuits is one of the most significant stages in the LSI design process. The purpose of logic-level simulation is to verify the correctness of the logic circuit design from the viewpoints of its logical specification and signal propagation timings. Since simulations needs a long run time , high speed simulator is eagerly awaited.

We constructed a logic-level simulation system on the Multi-PSI which can deal with gate delays. The virtual time mechanism was adopted as parallel control mechanism, and we have just started evaluation of the efficiency of parallel processing.

APPLICATION

The system can simulate both combinatorial circuits and sequential circuits of practical size such as those consisting of over 10,000 gates. Circuits should be described at the gate level. Different delay time with a multiple of a unit value can be assign to each gate. The simulator handles three signal values, Hi Lo and X (not specified).

SIMULATION METHOD

We adopted an event simulation method, that is, the gate behavior is simulated only when its input signal changes. Each gate is implemented as a process. Events are propagated as messages with a signal value and time. When a message arrives at an input of a gate process, it calculates an output value. When the output value changes, that is, when an event occurs, a new message is generated and propagated to following gates.

PARALLEL CONTROL MECHANISM

We adopted the virtual time mechanism as a parallel control mechanism. When a gate process receives messages in correct time order, it calculates events recording its event history. However, when a message arrives in incorrect order, (a message arrives which has an earlier time tag than that of the last message) the gate process must rewind its history (this procedure is called "rollback") and simulate again from the time which the time-reversal message kept.

Opposite to our method, there is a well-known parallel simulation mechanism, called distributed simulation. In that mechanism, each gate does not start event calculation until messages arrive at all its inputs. This mechanism assures the correctness of time order of messages, but it does have the possibility of deadlock. Supplementary messages are used to avoid deadlock, which makes costs very high.

The remarkable merit of the virtual time mechanism is that there is no possibility of deadlock.

1. Local control mechanism

(a) When a message arrives in correct order

- Computes the new output signal value using previous state of the gate and the new input signal value
- Sends a new message to following gates if the output signal value changes

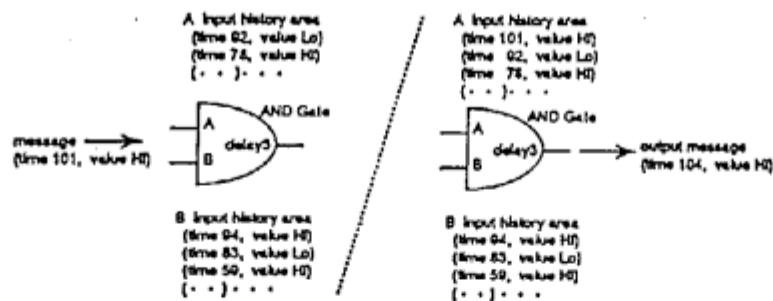
(b) When a message arrives in incorrect order

- Rewinds the history and restores the gate state for the time just before the message should have arrived, and resumes simulation from that time
- Cancels the invalid messages already sent

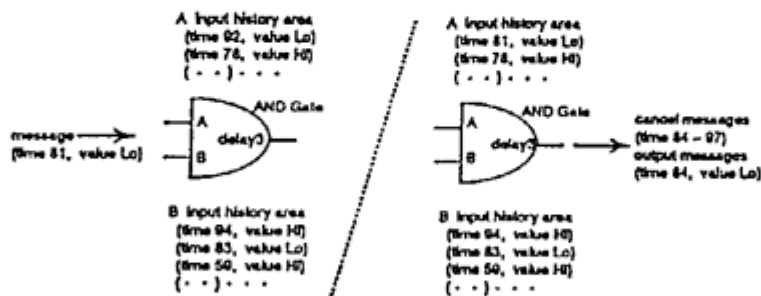
2. Global control mechanism

(a) Memory management: Sometimes computes global virtual time(GVT). GVT is updated equal to the earliest time kept between gates. GVT is used to release the event history. History records, corresponding to the time earlier than GVT, can be released.

(b) Termination detection: Simulation is regarded as finished when GVT exceeds the simulation finish time.



(a) When a message arrives in correct order



(b) When a message arrives in incorrect order

3. Scheduling

In one processor, we should schedule messages to arrive at their destinations in correct order (to keep rollback frequency low). We made a scheduler process in each processor to sort messages in the right sequence.

POINTS TO CONSIDER IN LOAD BALANCING

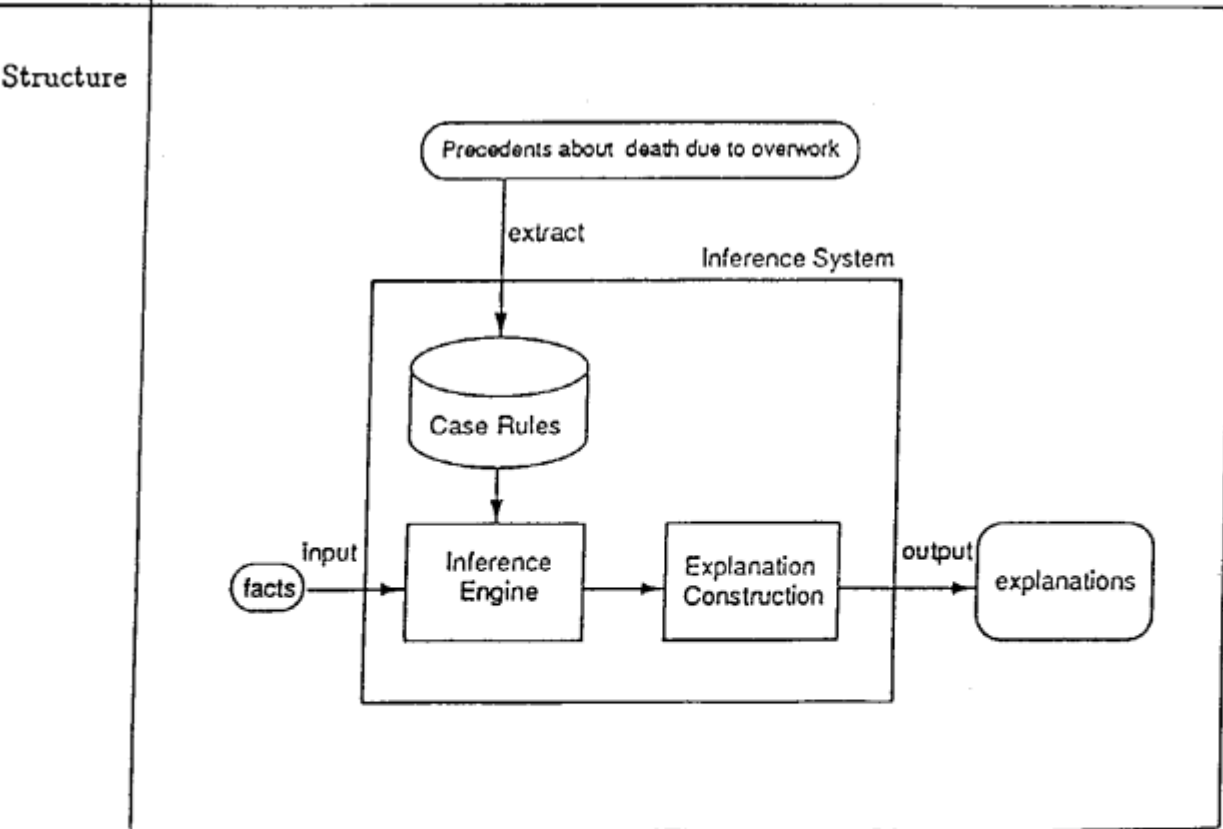
1. To make simulation proceed in each processor at the same pace
2. To make communication frequency between processors as low as possible

DEMONSTRATION

We simulate the sequential circuit which consists of about 13,000 gates. Input signal sequences are generated randomly except clock lines.

As the result of the simulation, besides the output signal sequences, we get several data for the performance evaluation. We can evaluate the elapsed time, and can compare the total number of messages with the number of rolled back messages.

Title	Experimental System of Parallel Legal Reasoning using Precedents
Objectives	The objectives are to develop a parallel inference engine and to investigate the computational model of legal reasoning.
Outline & Feature	<ul style="list-style-type: none"> • Case-based reasoning is reasoning from precedents, adapting old solutions to solve new problems. We evidenced that legal reasoning can be modeled as case-based reasoning. • Our system deals with precedents about death due to overwork. Legal decisions and arguments made by plaintiffs, defendants and judges in law courts are extracted from old cases, and represented in the form of case rule. When set of facts of a new case are input to the system, it creates all possible inference trees the roots of which are legal consequences. Our system is used to construct explanations to help the plaintiff or defendant make their assertion firm. • Condition part of each case rule is checked if it is similar to the new facts. The matching process can be independently performed for each case rule. We dispatch case rules to multiple processing elements (PEs) of Multi-PSI and the matching process are performed actually in parallel. We much improved the efficiency of the legal reasoning.



[Legal Reasoning Problem]

We might think that legal problem is solved by reasoning using syllogism based on legal rules. But this approach did not work well, because legal predicates appeared in legal rules are often ambiguous. In fact, a legal rule does not make it clear how to apply the law to a new case. In a lawsuit, both plaintiff and defendant asserts their interpretations using suitable facts.

When making interpretations, law experts use a kind of case-based reasoning. They refer precedents concerning old cases similar to the new one to help themselves make explanations. Then legal reasoning is a good example of case-based reasoning.

Our demonstration system deals with precedents about death due to overwork. When facts of a new case are input to the system, it retrieves cases similar to the new case from a case base, and outputs all possible inference trees to explain each consequence.

[Representation of Cases]

Each fact being involved in a new case is expressed in the form of 3-tuples, {*object, relation, value*}. A set of the 3-tuples construct a semantic network representing the relevant facts of the case.

Precedents contain reasoning process, which are legal decisions and arguments made by plaintiffs, defendants and judges in law courts. They reason some consequences based on the initial facts. The reasoning processes can be regarded as a set of rules. We call this type of rule a "case rule". The condition part of a case rule represents a situation of an old case, and its action part represents a result of inference.

A case rule is apparently similar to production rule, but is different because it is not a generalized rule. It represents just relations between concrete situations of the case. As a case rule is expressed in concrete level, it is particularly useful for solving problems in which knowledge is difficult to be represented as general rule. About 270 case rules are previously extracted from 25 precedents and stored in the case base.

[The Process of Generating Explanation]

When a new accident happens, we represent it as a semantic network. Facts of the new case are input into the system. The inference engine retrieves case rules each of which condition part matches to the new facts, and executes their condition parts. By matching-execution cycle, the reasoning proceeds with forward chaining. The explanations are constructed as chains of the inference.

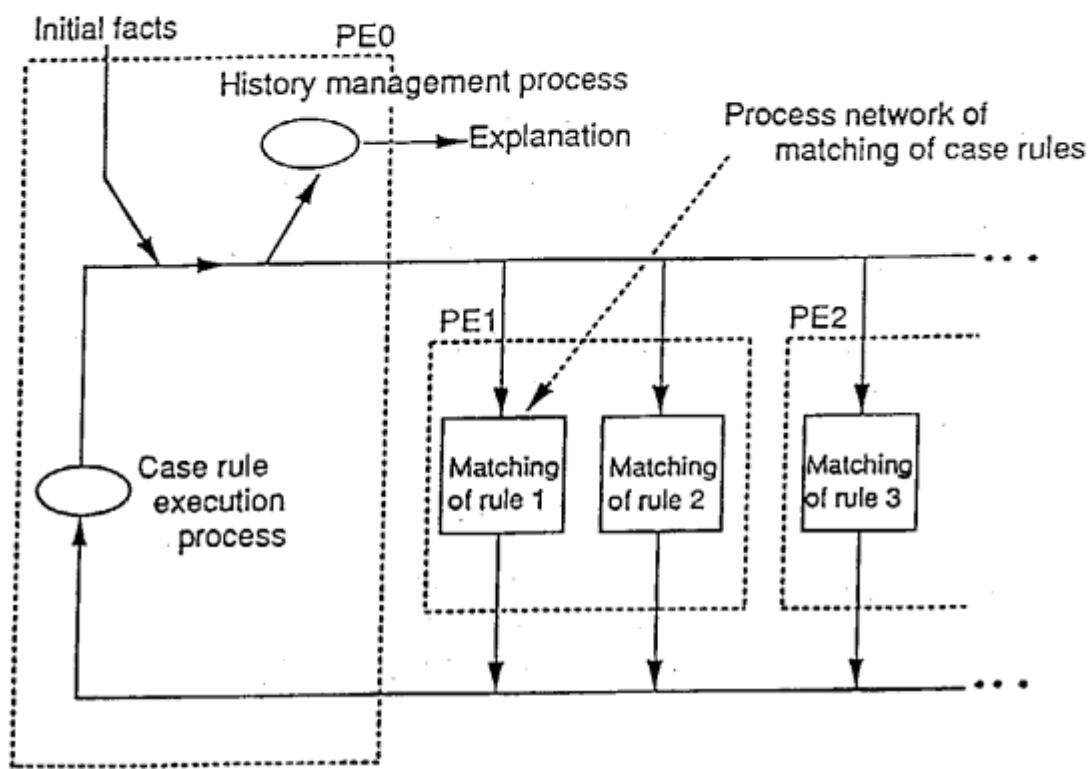
The matching process is based on similarity, and it has two characteristics. (1) If important facts of the condition part matches to the input facts, the rule can be fired even if other condition does not match. (2) Even if condition part does not match input facts, if they match at the abstract level, they are regarded to be matched. This is a kind of generalization of a case rule using *ISA* link of a semantic network.

[Inference Engine and Parallel Processing]

In the experimental system, as past precedents are stored in the form of 'case rule', the reasoning process is performed by a matching-execution cycle like an inference engine of a production system. The condition part of each case rule is converted to a process network of KL1 in advance.

When the facts of the new case are given to all of these process networks of matching, each process network checks whether the set of facts are similar to the condition part of the case rule by using similarity-based matching. If the sum of the similarity weights of all conditions in the condition part is greater than the threshold of the case rule, then this facts is regarded as matching the case rule. Then the action part of the case rule is transferred to the process network of execution, and the results of the matched case rules are given to all of the process networks of matching. Thus forward chaining of the reasoning is generated.

As mentioned above, the cost of the similarity-based matching is very high, because a case rule generally has many conditions and the similarity-based matching is heavy enough. Also the matching process for each case rule can be performed independently. If there are a large number of case rules stored, we dispatch case rules to multiple PEs and the process of matching condition parts are performed actually in parallel. Then we can improve the speed of the legal reasoning.



Parallel Inference Engine

Fact Window

60: (["accident#10", "accident"], "decision", ["labor accident"])
61: (["accident#10", "accident"], "decision", ["caused by the labor"])
62: (["accident#10", "accident"], "decision", ["non-labor accident"])
63: (["accident#10", "accident"], "decision", ["non-labor accident"])
64: (["accident#10", "accident"], "decision", ["non-labor accident"])
65: (["accident#10", "accident"], "decision", ["labor accident"])
66: (["accident#10", "accident"], "decision", ["labor accident"])
67: (["accident#10", "accident"], "decision", ["labor accident"])

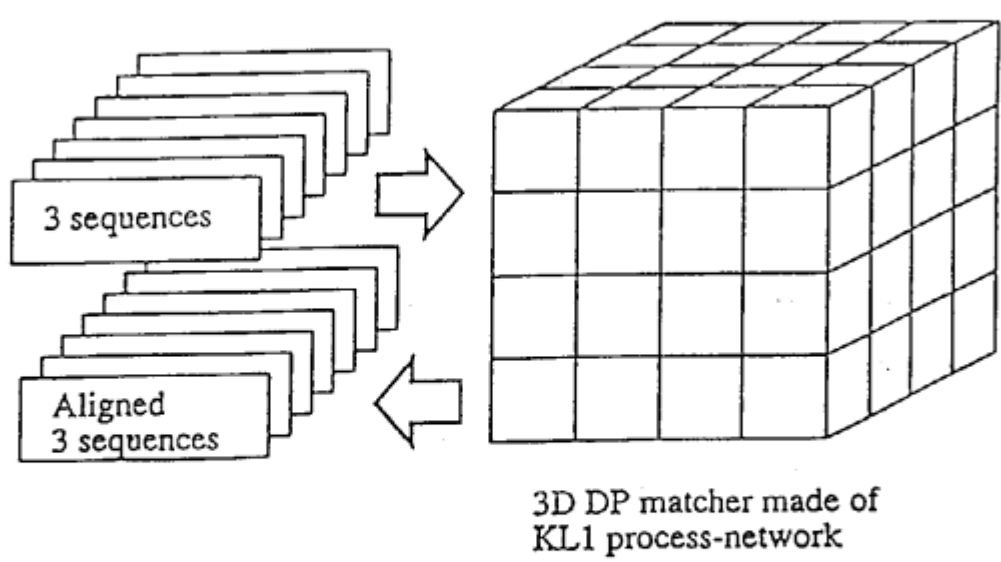
Explanation Window

FACT:67: (["accident#10", "accident"], "decision", ["labor accident"])
!--RULE:Id114
!--FACT:61: (["accident#10", "accident"], "decision", ["caused by the labor"])
!--FACT:63: (["accident#10", "accident"], "decision", ["occurred during laboring"])

Id114:
" If it is decided that the accident was caused by the labor
" and that it occurred during laboring, then it can be decided
" that the accident is a labor accident.

FACT:61: (["accident#10", "accident"], "decision", ["caused by the labor"])
!--RULE:Id112
!--FACT:1: (["employment#10", "employment"], "employee", ["Mr. Tegawa", "person"])
!--[nil]
!--FACT:54: (["employment#10", "employment"], "probable causality between accident and labor", ["accident#10", "accident"])

Id112:
" If there is a probable causality between the accident
" happening to Mr. Matsukawa and his labor, then it can be
" decided that the accident is caused by the labor.

Title	<p style="text-align: center;">Genome Analysis Program (1): Multiple Sequence Alignment by 3-Dimensional DP-matching</p>
Purpose	<ul style="list-style-type: none"> • Parallel programming on a large-scale problem in KL1. • The first step to genome analysis.
Outline & Features	<p>[Outline]</p> <p>The system solves three-sequence alignment problems by 3-dimensional DP-matching. The DP-matching is executed by a prism network of KL1 processes. The network works as a parallel pipeline.</p> <p>[Feature]</p> <ul style="list-style-type: none"> • Efficient DP-matching by parallel pipeline processing. • Quality improvement in three-sequence alignments.
System Configuration	 <p style="text-align: center;">3D DP matcher made of KL1 process-network</p>

1 What is multiple sequence alignment?

Biologists often align DNA and protein sequences in order to determine how similar they are. DNA is a chain of four kinds of nucleic acids and a protein is a chain of twenty kinds of amino acids, which are translated from a chain of nucleic acids. Strong similarities between sequences may result from a common evolutionary relationship, and these sequences may have almost same function.

Figure 1 shows a typical multiple sequence alignment. Twelve fractions of enzyme proteins are aligned. Each letter stands for an amino acid: D is aspartic acid, R is arginine, H is histidine, and P is proline. A good alignment has same or similar amino acids in each column. To make an alignment good, each sequence is shifted or gaps (dash characters) are inserted into the sequence.

```

---DRHP-IPHMDEILGKLGRC-NYFTTIDLAKGFHQIEMDPESYSKTAFS-----
---DAYN-LPHKDELLTLIRGK-KIFSSFDCKSGFWQYLLDQESRPLTAFT-----
---DIHPTVPHPYNLLSGLPPSHQWYTYL DLKDAFFCLRLHPTSQPLFAFEW-ROPEM
---L-FGPYQRGLPLLSALPQDWKLI-IIDIKDCFFSIPLYPRDRPRFAFTIPSLKHM
---P-FGAVQQGAPYLSALPRGWPLM-VLDLKDCFFSIPLAEQDREAFATLPSYHKQ
---DLSSSSPGPPDL-SSLPTTLAHLQTI DLDAFFQIPLPKQFQPYFAFTYPPQCHY
---TLTSPSPGPPDL-TSLPTALPHLQTI DLDAFFQIPLPKQYQPYFAFTIPQPCHY
---PIPALSPGPPDL-TAIPTHPHIICLDLXDAFFQIPYEDRFRSYLSFTLPSGGGL
---D-FWEYQLGIPHPAGLXXXXSVT-VLOYGDAYFSYPLQEDFRKYTAFTIPSINNE
YHWPXF-AYPNLQTLAHLSTDLQWL-SLDYSAAFYHIPISPAAYPHLLYG-----
YSWPXF-AYPNLQSLTNLLSSNLSWL-SLDYSAAFYHIPLHPAAMPHLLYG-----
MRFPRY-WSPHLSTLRRILPYGMPRI-SLDLSQAFYHLP LNPASSSRLAYS-----
    
```

Figure 1 Multiple sequence alignment

2 Dynamic programming on sequence matching

Dynamic programming (DP) is a basic method to find an optimal alignment. The method is regarded as the best path search in the N-dimensional network. In the method, for example, if two sequences, ADHE and AHIE are given, we form a 2-dimensional network that has 25 nodes connected by arrows. A cost is assigned to each arrow. We search a path from the top left node to the bottom right node, minimizing the total cost of arrows. In this case, the set of arrows that connect white circle nodes is the best path. This best path corresponds to the optimal alignment, ADH-E and A-HIE (Figure 2.1).

Costs on arrows should reflect similarity between compared characters. In the case of protein sequence alignment, Dayhoff's odds matrix (Figure 2.2) is the most popular way of obtaining the costs. The matrix was obtained by statistical analysis of mutation probability of amino acids.

Though DP-matching is an optimal method for alignment, it takes a lot of calculation time. DP-matching with more than three dimensions is too time-wasteful to be used for practical alignment. So DP-matching has been used for partial matching, when several sequences need to be aligned. For instance, we can produce all pairwise alignments of given sequences with 2-dimensional DP, then merge the alignments one by one.

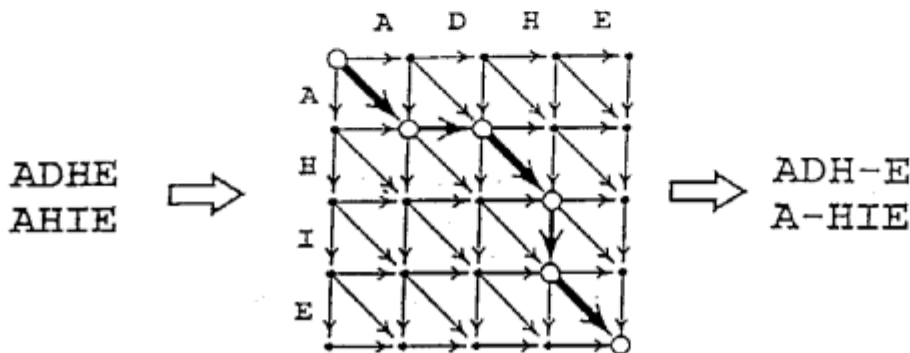


Figure 2.1 DP-matching method

	A	R	K	D	C	Q	E	G	H	I	L	K	F	P	S	T	V	Y	V	B	Z	I	
A	1	-2																					
R	2	-6																					
K	0	0	-2																				
D	0	1	-2	-4																			
C	2	4	4	5	-12																		
Q	0	-1	-1	-2	5	-4																	
E	0	1	-1	-3	5	-2	-4																
G	-1	3	0	-1	3	1	0	-5															
H	1	-2	-2	-1	3	-3	-1	2	-6														
I	1	2	2	2	2	2	2	3	2	-5													
L	2	3	3	4	6	2	3	4	2	-2	-6												
K	1	-3	-1	0	5	-1	0	2	0	2	3	-6											
X	1	0	2	3	5	1	2	3	2	-2	-4	0	-6										
F	4	4	4	6	4	5	5	5	2	-1	-2	5	0	-9									
P	-1	0	1	1	3	0	1	1	0	2	3	1	2	5	-6								
S	-1	0	-1	0	0	1	0	-1	1	1	3	0	2	3	-1	-2							
T	-1	1	0	0	2	1	0	0	1	0	2	0	1	3	0	-1	-3						
V	6	-2	4	7	8	5	7	7	3	5	2	3	4	0	6	2	5	-17					
Y	3	4	2	4	0	4	4	5	0	1	1	4	2	-7	5	3	3	0	-10				
V	0	2	2	2	2	2	2	1	2	-4	-2	2	-2	1	1	1	0	6	2	-4			
B	0	1	-2	-3	4	-1	-2	0	-1	2	3	-1	2	5	1	0	0	5	3	2	-2		
Z	0	0	-1	-3	5	-3	-3	1	-2	2	3	0	2	5	0	0	1	6	4	2	-2	-3	
I	0	1	0	1	3	1	1	1	1	1	1	1	1	2	1	0	0	4	2	1	0	1	1

Figure 2.2

Dayhoff's odds matrix

3 Parallel pipeline processing of 3-dimensional DP

If 3-dimensional DP can be executed rapidly, it is useful for partial matching because it tolerates noise better than 2-dimensional DP does. We have implemented 3-dimensional DP on the parallel machine, Multi-PSI, and improved the speed of three-sequence matching.

Our system constructs a 3-dimensional prism network with KL1 processes (Figure 3). The prism network is divided into 64 subprisms of equal volume and is mapped to 64 process elements (PEs). The KL1 is suitable for constructing such mesh-like process networks and the network can be used as data-flow pipeline easily.

If many different combinations of three-sequence alignments are available, we expect to merge whole sequences adequately for multiple alignment. This system provides optimal three-sequence alignments by parallel pipeline processing.

4 Demonstration

The demonstration system solves three-sequence alignment problems continuously by parallel pipeline processing. After several initial alignment data are fed to PE0, their optimal alignments come out from PE63 and are displayed at short intervals. During processing, the performance meter window shows that several wavefronts pack and propagate from PE0 to PE63 clearly.

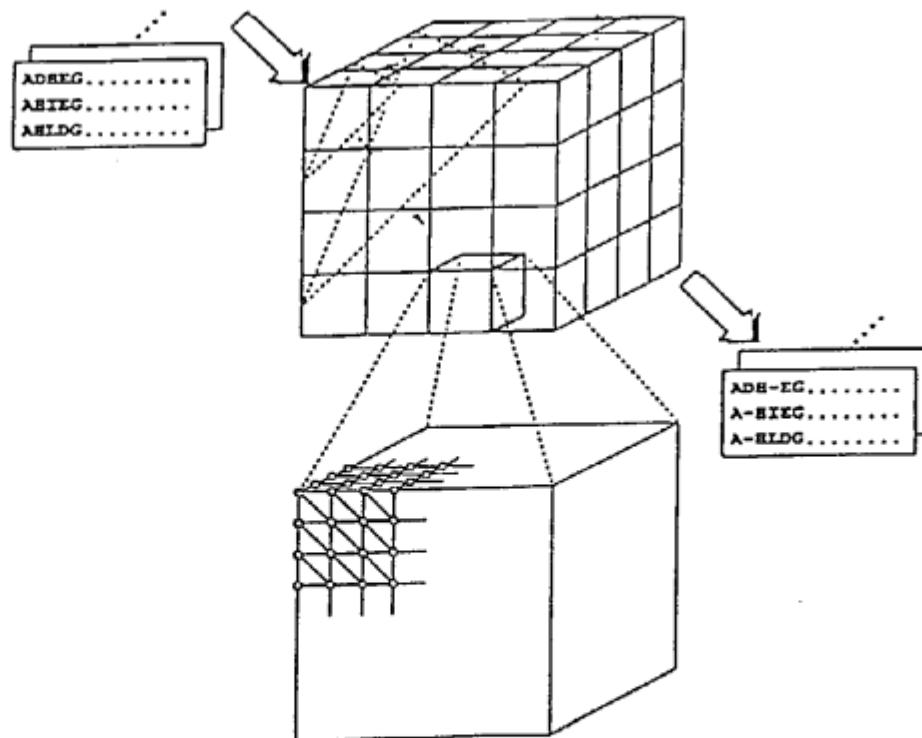


Figure 3 3-dimensional DP-matching

Title	<p style="text-align: center;">Genome Analysis Program (2)</p> <p style="text-align: center;">Multiple Sequence Alignment by Parallel Simulated Annealing</p>
Purpose	<ul style="list-style-type: none"> • Application of a parallel simulated annealing to a practical problem. • The first step to genome analysis.
Outline & Features	<p>[Outline]</p> <p>The system solves a multiple sequence alignment problem by scheduleless parallel simulated annealing. Each PE has a constant temperature and exchanges solutions with neighbor PEs in some probabilistic way.</p> <p>[Feature]</p> <ul style="list-style-type: none"> • Simulated annealing without designing a cooling schedule. • Generating various alignments in different local minima.
System Configuration	<div style="text-align: center;"> <p style="text-align: center;">Initial Sequences</p> <p style="text-align: center;">↓</p> <div style="display: flex; justify-content: center; align-items: center;"> <div style="margin-right: 10px;"> T_1 T_2 T_3 T_4 T_5 </div> <div style="border: 1px solid black; padding: 5px; margin: 0 10px;"> </div> <div style="margin-left: 10px;"> t on PE1 t on PE2 t on PE3 t on PE4 t on PE5 </div> </div> <p style="text-align: center;">↓</p> <p style="text-align: center;">Aligned Sequences</p> </div>

1 Simulated annealing algorithm

In many important practical problems, a solution is an arrangement of a set of discrete objects according to a given set of constraints. Such problems are typically known as combinatorial problems. The set of all solutions is referred to as the solution space and an energy function is defined for all solutions. To solve a combinatorial problem is to find a minimum-energy spot in the solution space.

A general strategy to search in the space is the method of 'iterative improvement'. The method requires a set of moves that can be used to modify a solution. One starts with an initial solution and examines its moves until a neighboring solution with a lower energy is discovered. The neighbor becomes the new solution and the process is continued to examine the neighbors of the new solution. This iteration terminates when it arrives at a spot that has locally minimum energy.

Simulated annealing algorithm is an extension of the method of iterative improvement based on an analogy between a combinatorial problem and the problem of determining the ground state of a physical system. To bring a fluid to a highly ordered state like a single crystal, a process called 'annealing' can be employed. We first melt the system by heating it to a high temperature, then cool it slowly, spending a long time at temperatures in the vicinity of the freezing point. Kirkpatrick et al suggested that better results to combinatorial problems can be obtained by simulating the annealing process of physical systems (Figure 1).

```

begin
   $X_0 :=$  Initial solution ;
   $(T_n)_{n=0 \dots N-1} :=$  Temperature (Cooling schedule);
  for  $n := 0$  to  $N-1$  do
    begin
       $X'_n :=$  Some random neighboring solution of  $X_n$ ;
       $\Delta E := E(X'_n) - E(X_n)$ ;
      if  $\Delta E < 0$  then
         $X_{n+1} := X'_n$ 
      else
        if  $\exp(-\Delta E/T_n) \geq \text{random}(0,1)$  then
           $X_{n+1} := X'_n$ 
        else
           $X_{n+1} := X_n$ 
    end;
  Output  $X_N$ ;
end;
```

Figure 1 Simulated annealing algorithm

2 Multiple alignment as a combinatorial problem

There may be some ways to formulate multiple sequence alignment as a combinatorial problem. Kanehisa, a professor at Kyoto university, developed an ingenious formulation in order to solve multiple alignment problems by simulated annealing algorithm. We adopt his formulation.

Kanehisa's idea is as follows. First, we make an initial alignment by adding a number of gaps to both head and tail of each sequence (Figure 2.1). To modify the alignment, we focus on one sequence in the alignment and select a gap and an amino acid randomly in that sequence. Moving the gap to the other side of the selected amino acid gives the modified alignment (Figure 2.2).

The energy of an alignment is calculated by summing up each correlation value of pairs of characters located in the same column. The correlation value comes from Dayhoff's odds matrix. If the energy of the modified alignment is lower than that of the previous one, the modified alignment is always regarded as a new alignment. If not, whether the modified one is regarded as a new alignment or not depends on the probability derived by temperature. The temperature is decided according to a cooling schedule. This annealing operation often brings good alignment (Figure 2.3).

```

"-----NAPATFQRCMNDILRPLL NKHCLVFSTSLD-----"
"-----LKQAPSIFQRHMDEAFRVFRKFCCVFSNNE-----"
"-----NSPTLFDEALHRDLADFRIQH PDLILLQAA-----"
"-----MANSPTICQLYVQEALEPIRKQFTSLIVIH-----"
"-----TCSPTICQLVVGQVLEPLRLKHPSLCMLHA-----"
"-----SPTLFEMQLAHILQPIRQAFPQCTILQASP-----"
    
```

Figure 2.1 An initial alignment

```

"-----NAPATFQRCMNDILRPLL NKHCLVFSTSLD-----"
"-----LKQAPSIFQRHMDEAFRVFRKFCCVFSNNE-----"
"-----NSPTLFDEALHRDLADFRIQH PDLILLQAA-----"
"-----MANSPTICQLYVQEALEPIRKQFTSLIVIH-----"
"-----TCSPTICQLVVGQVLEPLRLKHPSLCMLHA-----"
"-----SPTLFEMQLAHILQPIRQAFPQCTILQASP-----"
    
```

Figure 2.2 An alignment after the first move

```

"-----NAPATFQ--RCM-NDIL--RPLL NKHCLVFSTSLD----"
"-----LKQAPSIFQ--RHM--DEA-FRVF-RKFCCVFSNNE----"
"-----NSPTLFDEALH-R-DLADFRIQH-PDLILLQAA-----"
"-----MANSPTICQLYV-QEA-LEPIR-KQFTSLIVIH-----"
"-----TCSPTICQLVVGQ-V-LEPLRLKH-PSLCMLHA-----"
"-----SPTLF-EMQLAHÍ-LQPIRQA-FPQCTILQASP-----"
    
```

Figure 2.3 A good alignment

3 Scheduleless parallel simulated annealing

Designing a cooling schedule is troublesome because the optimal cooling schedule depends on the type and the scale of combinatorial problems. Without careful temperature reduction, a solution is trapped in a local minimum which has relatively high energy. Kimura, a member of ICOT, developed the method of parallel simulated annealing that makes it possible to avoid designing the cooling schedule.

In Kimura's method, each process element (PE) maintains one solution and performs the annealing operation concurrently under a constant temperature that differs from PE to PE. The solutions obtained by the PEs are occasionally exchanged between PEs that hold neighbor temperatures (Figure 3). This exchange of solutions is controlled in some probabilistic way. Kimura proposed a scheme of the probabilistic exchange, and justified it from the viewpoint of the probability theory. He applied his method to a graph-partitioning problem, one of the representative combinatorial problem. That proved his method to be efficient.

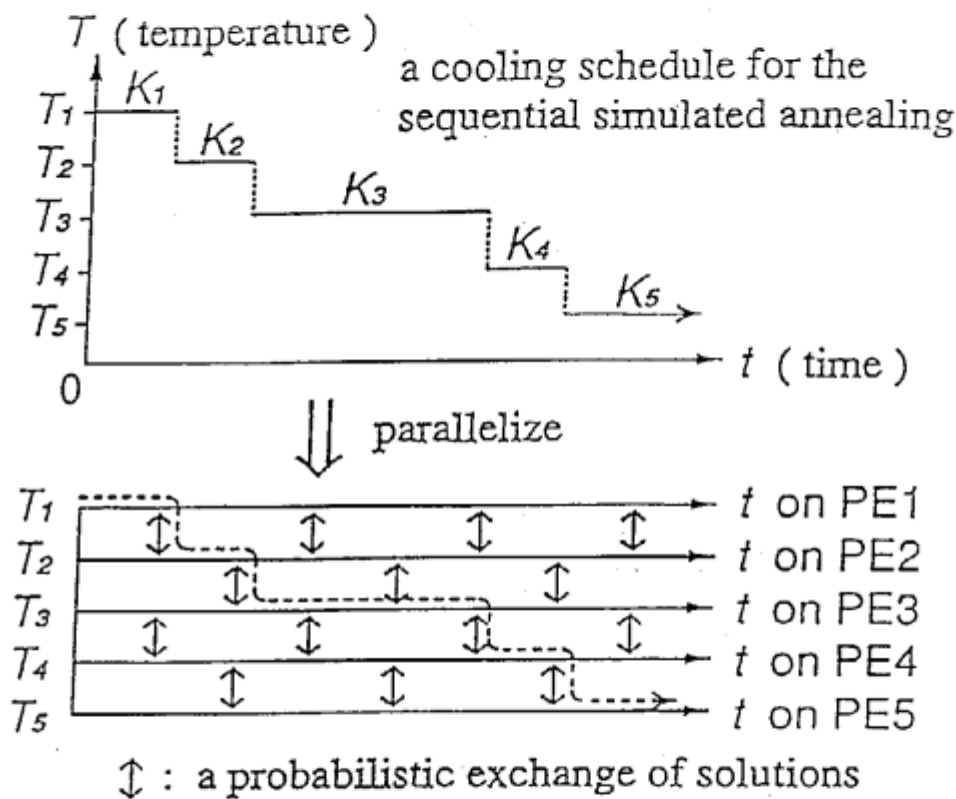
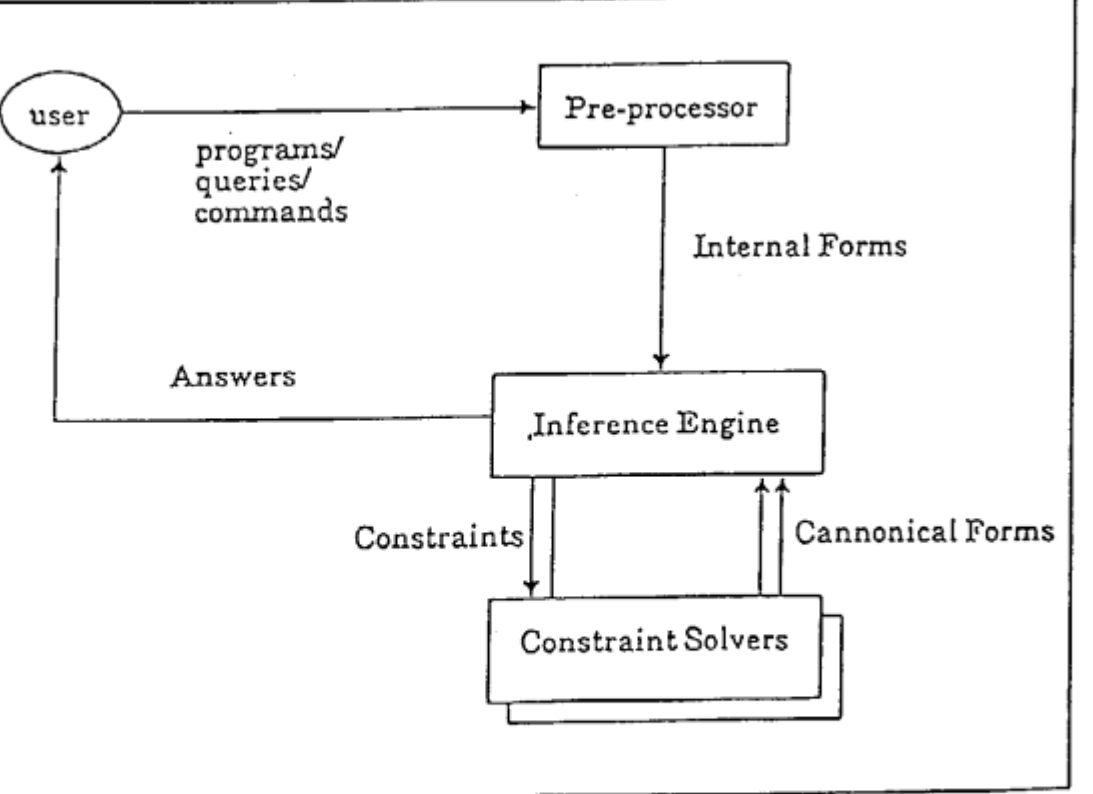


Figure 3 Scheduleless parallel simulated annealing

4 Demonstration

The demonstration system solves multiple sequence alignment problems by the parallel simulated annealing method. The multiple alignment problem is formulated as a combinatorial problem by Kanchisa's idea, and the simulated annealing operation is processed by Kimura's method.

Generally, it takes hundreds of hours for optimization by simulated annealing. The demonstration is a brief version of multiple alignment. It shows you gradual improvement of the alignment of some small protein sequences.

Title	<p style="text-align: center;">Constraint Logic Programming Experimental System CAL</p>
Purpose	<p>(1) Programs easier to write and read (2) Highly abstract programming (3) Research on efficient problem solving techniques</p>
Outline & Features	<p style="text-align: center;">CAL : Contrainte Avec Logique</p> <p style="text-align: center;"><i>Contrainte</i> – Constraint Programming + <i>Logique</i> – Logic Programming</p> <ul style="list-style-type: none"> • Very High-Level Declarative Programming Language • Powerful Problem Expression and Solving Ability • Can be parallelized similarly to Prolog – KL1
System Configuration	 <p>The diagram illustrates the system configuration. A 'user' (represented by an oval) sends 'programs/queries/commands' to a 'Pre-processor' (rectangle). The 'Pre-processor' sends 'Internal Forms' to an 'Inference Engine' (rectangle). The 'Inference Engine' sends 'Answers' back to the 'user'. The 'Inference Engine' sends 'Constraints' to 'Constraint Solvers' (rectangle), which in turn sends 'Canonical Forms' back to the 'Inference Engine'. The 'Constraint Solvers' box is shown with a double-line border, indicating it is a sub-component or library.</p> <pre> graph TD User((user)) -- "programs/queries/commands" --> PreProcessor[Pre-processor] PreProcessor -- "Internal Forms" --> InferenceEngine[Inference Engine] InferenceEngine -- "Answers" --> User InferenceEngine -- "Constraints" --> ConstraintSolvers[Constraint Solvers] ConstraintSolvers -- "Canonical Forms" --> InferenceEngine </pre>

★ Pony and Man Problem

- This problem consists of :

Number of Ponies,
 Number of Men,
 Total Number of Heads,
 Total Number of Legs.

- Two Relationships among them:

Number of Ponies + Number of Men = Total Number of Heads
 $4 \times$ Number of Ponies + $2 \times$ Number of Men = Total Number of Legs

pm(Ponies, Men, Heads, Legs) :-
 alg: Ponies + Men = Heads,
 alg: $4 * \text{Ponies} + 2 * \text{Men} = \text{Legs}$.

Now we can ask various questions :

1. Ponies = 2	\Rightarrow	Heads = 5
Men = 3		Legs = 14
2. Heads = 6	\Rightarrow	Ponies = 3
Legs = 18		Men = 3
:		:

If you use Prolog.....

```
pm(Ponies, Men, Heads, Legs) :- int(Ponies), int(Men), !,
    Heads is Ponies + Men,
    Legs is 4*Ponies + 2*Men.
```

```
pm(Ponies, Men, Heads, Legs) :- int(Ponies), int(Heads), !,
    Men is Heads - Ponies,
    Legs is 3*Ponies + Heads.
```

```
pm(Ponies, Men, Heads, Legs) :- int(Ponies), int(Legs), !,
    Men is Legs/2 - 2*Ponies,
    Heads is Legs/2 - Ponies.
```

```
pm(Ponies, Men, Heads, Legs) :- int(Men), int(Heads), !,
    Ponies is Heads - Men,
    Legs is 4*Heads - 2*Men.
```

```
pm(Ponies, Men, Heads, Legs) :- int(Men), int(Legs), !,
    Ponies is Legs/4 - Men/2,
    Heads is Legs/4 + Men/2.
```

```
pm(Ponies, Men, Heads, Legs) :- int(Heads), int(Legs), !,
    Ponies is Heads/3 - Legs/6,
    Men is 2*Heads + Legs/6.
```

```
pm(Ponies, Men, Heads, Legs) :- int(Ponies), !,
    Men = Heads - Ponies,
    Legs = 2*Ponies + 2*Heads.
```

```
pm(Ponies, Men, Heads, Legs) :- int(Men), !,
    Ponies = Heads - Men,
    Legs = 4*Heads - 2*Men.
```

```
pm(Ponies, Men, Heads, Legs) :- int(Heads), !,
    Ponies = Heads - Men,
    Legs = 4*Heads - 2*Men.
```

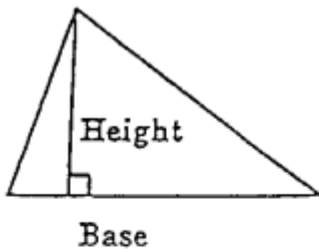
```
:
:
:
```


★ Heron's Formula

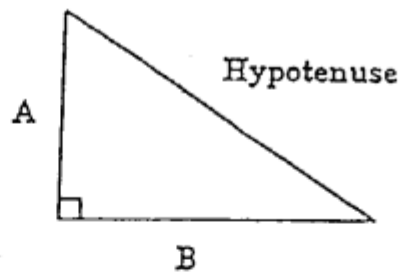
surface(Height, Base, Area) :-
 alg:Base*Height = 2*Area.
 pythagoras(A, B, Hypotenuse) :-
 $A^2 + B^2 = \text{Hypotenuse}^2$.

triangle(A, B, C, S) :-
 alg:C = CA + CB,
 pythagoras(CA, H, A),
 pythagoras(CB, H, B),
 surface(H, C, S).

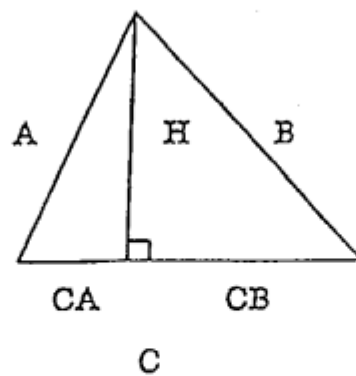
surface:



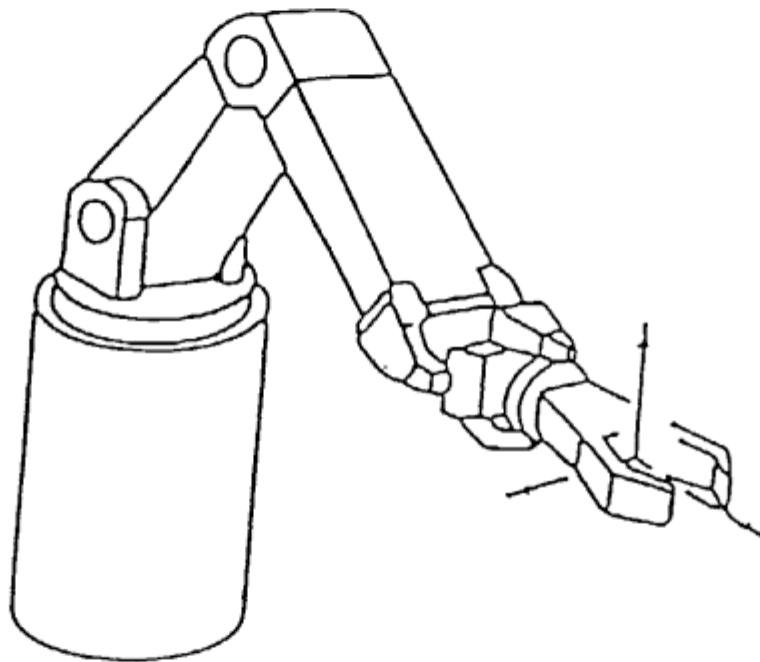
pythagoras:



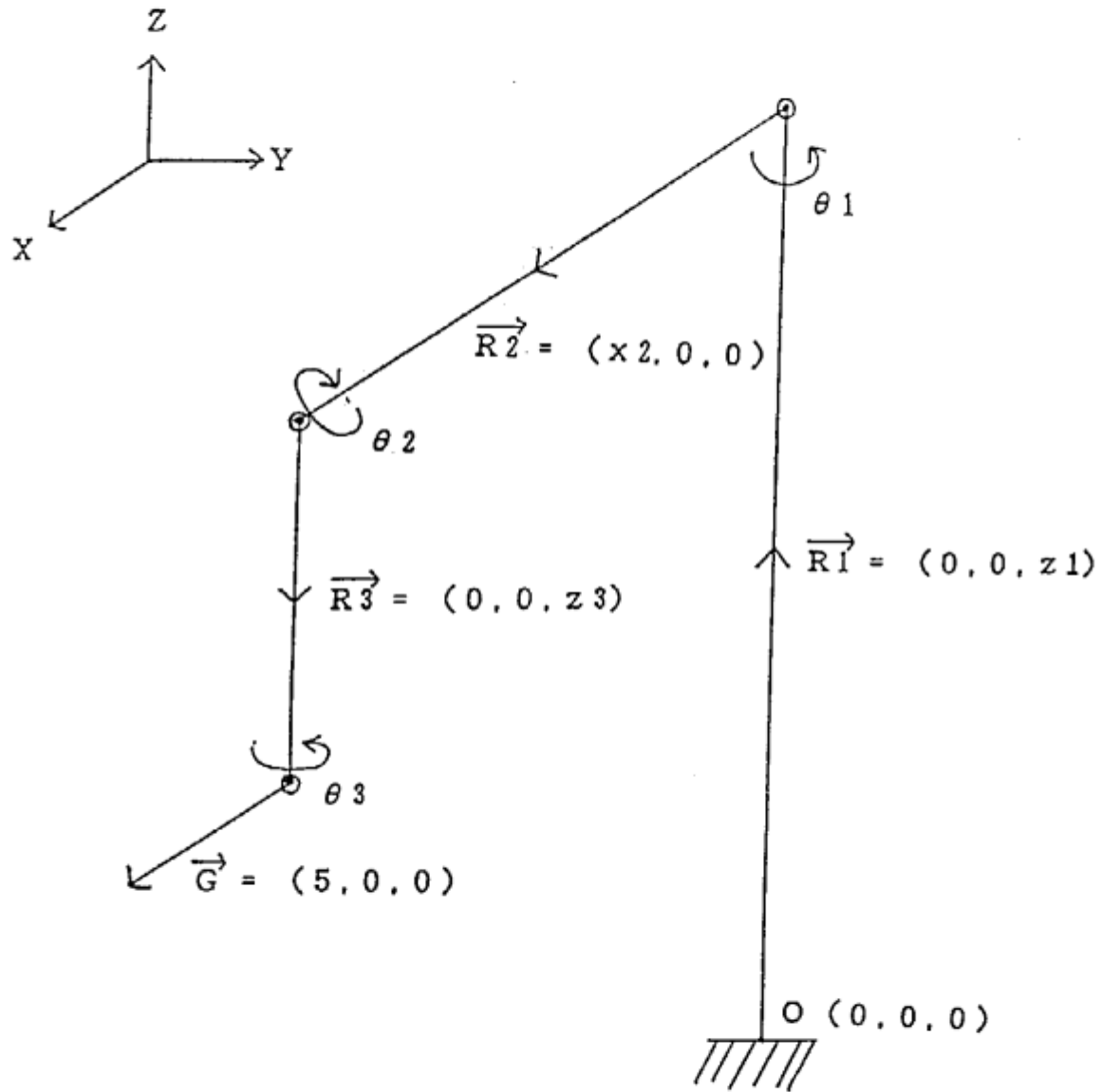
triangle:



Handling Robot Kinematics



ex. Vector Sketch of a Robot
(3 Joints and 6 freedoms)



AMF BERTHATRAN

?- robot3:

```
robot([[cos3, sin3, 0, 0, z3, 0, 0, 1],
      [cos2, sin2, x2, 0, 0, 1, 0, 0],
      [cos1, sin1, 0, 0, z1, 0, 0, 1]],
      5, 0, 0, 1, 0, 0, 0, 1, 0,
      px, py, pz, ax, ay, az, cx, cy, cz).
```

$$\sin 5^2 = 1 - \cos 5^2 .$$

$$\sin 4^2 = 1 - \cos 4^2 .$$

$$\sin 1^2 = 1 - \cos 1^2 .$$

$$px = \cos 1 * \cos 2 * z3 - 5 * \cos 1 * \sin 2 * \cos 3 + \sin 1 * x2 + 5 * \sin 1 * \sin 3$$

$$py = \cos 1 * x2 + 5 * \cos 1 * \sin 3 - \sin 1 * \cos 2 * z3 + 5 * \sin 1 * \sin 2 * \cos 3$$

$$pz = z1 + 5 * \cos 2 * \cos 3 + \sin 2 * z3 .$$

$$ax = -1 * \cos 1 * \sin 2 * \cos 3 + \sin 1 * \sin 3 .$$

$$ay = \cos 1 * \sin 3 + \sin 1 * \sin 2 * \cos 3 .$$

$$az = \cos 2 * \cos 3 .$$

$$cx = -1 * \cos 1 * \sin 2 * \sin 3 - \sin 1 * \cos 3 .$$

$$cy = -1 * \cos 1 * \cos 3 + \sin 1 * \sin 2 * \sin 3 .$$

$$cz = \cos 2 * \sin 3 .$$

yes

yes

?- robot3:

```
robot([[cos3, sin3, 0, 0, z3, 0, 0, 1],
      [cos2, sin2, x2, 0, 0, 1, 0, 0],
      [cos1, sin1, 0, 0, z1, 0, 0, 1]],
      5, 0, 0, 1, 0, 0, 0, 1, 0,
      40, -30, 20, 1/3, 2/3, -2/3, -2/3, 2/3, 1/3)
```

$$\sin1^2 = 1/5 .$$

$$\cos2 = -5/3*\sin1 .$$

$$\sin2 = 2/3 .$$

$$\cos1 = -2*\sin1 .$$

$$z1 = 6 .$$

$$\cos3 = 2*\sin1 .$$

$$\sin3 = -1*\sin1 .$$

$$z3 = 26 .$$

$$x2 = 105*\sin1 .$$

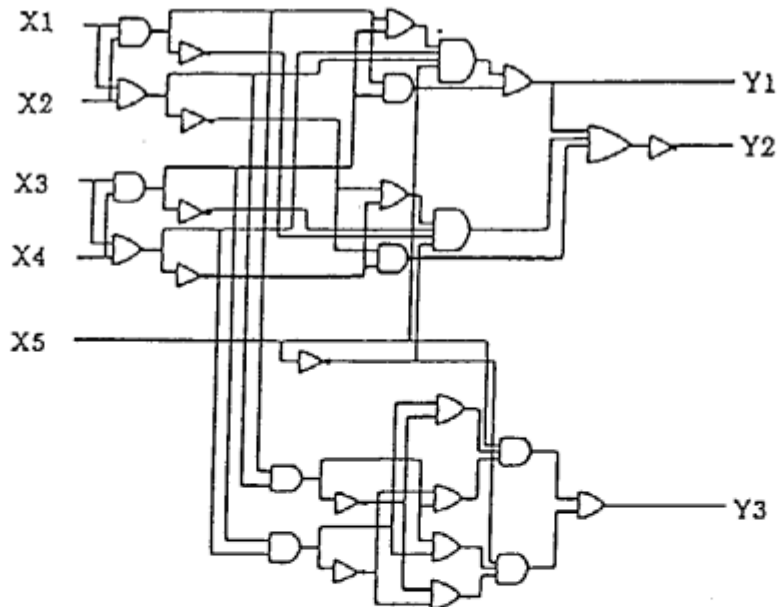
yes

?-

★ Count 1's: Boolean Constraints

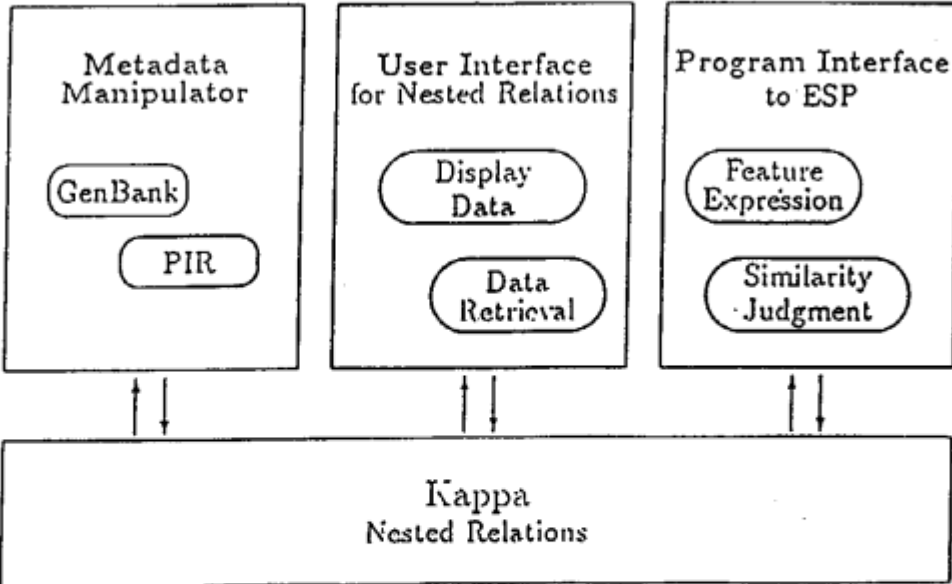
```

circuit(X1, X2, X3, X4, X5, Y1, Y2, Y3) :-
  bool:I1 = X1&X2, bool:I2 = X1vX2, bool:I3 = X3&X4,
  bool:I4 = X3vX4, bool:I5 = ~I1, bool:I6 = ~I2,
  bool:I7 = ~I3, bool:I8 = ~I4, bool:I9 = I1vI3,
  bool:I10 = I1&I3, bool:I11 = I6vI8, bool:I12 = I6&I8,
  bool:I13 = ~X5, bool:I14 = I5&I2, bool:I15 = I7&I4,
  bool:I16 = ~I14, bool:I17 = ~I15, bool:I18 = I15vI16,
  bool:I19 = I14vI17, bool:I20 = I14vI15, bool:I21 = I16vI17,
  bool:I22 = I9&I4&I2&X5, bool:I23 = I11&I7&I5&I13,
  bool:I24 = X5&I8&I9, bool:I25 = I13&I20&I21,
  bool:I26 = I22vI10, bool:I27 = I26vI23vI2,
  bool:Y1 = I26, bool:Y2 = ~I27, bool:Y3 = I24vI25.
  
```



★ Conclusion

- In Constraint Logic Programming, problems are described simply and straight forwardly
- The system will find a method to solve problems given their descriptions
- Constraint Logic Programming increases programmer productivity

Title	Molecular Biological Database in Kappa
Purpose	We are providing a new-concept DBMS, together with several tools, for the effective use of molecular biological data. We are planning to integrate molecular biological databases on our database management system. First we stored GenBank and PIR; now we are researching the best way to use them together.
Outline & Features	<p>Demonstration Outline:</p> <ol style="list-style-type: none"> (1) Nested relational schemas for molecular biological database <ul style="list-style-type: none"> - of GenBank - of PIR (2) Functions and performance of the user interface of Kappa <ul style="list-style-type: none"> - Display Data - Data Retrieval (3) Examples of tools, their simplicity and performances (Functions of the program interface of Kappa) <ul style="list-style-type: none"> - Feature Expression - Similarity Judgment
System Configuration	 <p>The diagram illustrates the system configuration. At the top, three boxes represent the main components: 'Metadata Manipulator', 'User Interface for Nested Relations', and 'Program Interface to ESP'. The 'Metadata Manipulator' box contains two sub-components: 'GenBank' and 'PIR'. The 'User Interface for Nested Relations' box contains two sub-components: 'Display Data' and 'Data Retrieval'. The 'Program Interface to ESP' box contains two sub-components: 'Feature Expression' and 'Similarity Judgment'. Below these three boxes is a larger box labeled 'Kappa Nested Relations'. Vertical double-headed arrows connect each of the three top boxes to the 'Kappa Nested Relations' box, indicating bidirectional communication.</p>

1 Preliminary

1.1 Kappa

Kappa (Knowledge APPLICATION-oriented Advanced database management system) is one of the ICOT KBMS projects, and aims to provide DBMS for Knowledge Information Processing Systems (KIPS). Kappa is the DBMS on PSI-II/SIMPOS, while Kappa-P, which we are now developing, is the parallel version of Kappa, on PIM/PIMOS.

Kappa is a DBMS with the following features:

- (1) Nested relational model is employed.
- (2) Large amounts of data are effectively accessed.
- (3) An user interface tuned for nested relational model is provided.
- (4) ESP program interface and extended relational algebra are provided.
- (5) Program interface can be customized for each application.

1.2 Nested Relational Model

The definition of the nested relational model (which is employed by Kappa) is intuitively as follows:

$$NR \subseteq E_1 \times \dots \times E_n$$

$$E_i ::= D \mid 2^{NR}$$

Compared to the relational model:

$$R \subseteq D_1 \times \dots \times D_n$$

where D_i is a domain, R is a relation and NR is a nested relation.

Relational:

Tour Schedule			Group	
Date	To	Group	Name	Member
90.6.4	ANL	Setting_G	Setting_G	Sugino
90.6.25	ANL	Lecture_G	Lecture_G	Ichiyoshi
			Lecture_G	Kondo
			Lecture_G	Susaki

Nested Relational:

Tour Schedule			
Date	To	Group	
		Name	Member
90.6.4	ANL	Setting_G	Sugino
90.6.25	ANL	Lecture_G	Ichiyoshi
			Kondo
			Susaki

1.3 Molecular Biology

DNA sequence: It can be considered as a string, whose length can be more than several hundred thousand. It consists of 4 characters, namely A,T,G,C.

amino acid sequence: Protein may also be considered as a string. It consists of 20 characters, which represent amino acids.

protein coding region: It is part of a DNA sequence. It is translated into amino acid sequences by a certain rule.

exon and intron: Protein coding regions of some organisms include sequences which are not translated into amino acid sequences. We call them introns, while exons are the translated parts.

genome: The 'full set' of DNA sequences of an organism. Notice that each human possesses two genomes in his chromosomes.

restriction enzyme: We use it to cut a DNA sequence to a modest length, to read DNA. The position it cuts is called restriction site.

1.4 Molecular Biological Databases

1.4.1 GenBank

There are two major databases of DNA sequences: GenBank¹ and EMBL². They and DDBJ³ have agreements on dividing tasks of data collection and exchanging data collected. The distribution forms of the data are flat files, in MT, FD, CD-ROM, or through networks. Some portion of distribution will remain in the flat file for the time being, while it will soon be something hierarchical, and exchanged between DBMS. GenBank began to manage their data on a relational database in 1989.

1.4.2 PIR

PIR⁴ is the representative database of protein sequence. It contains a DNA sequence database in a similar format. It also uses flat file distribution. It is distributed with access methods on VAX/VMS, called NAQ and PSQ.

1.4.3 Other Databases

Protein Structure PDB⁵ is the database of atomic coordinates of amino acids, namely 3-dimensional structures of proteins.

Maps of DNA We have maps of restriction sites, which are used as physical maps, and of relational distances between genes, which are used as genetic maps.

¹ Genetic Sequence Data Bank, IntelliGenetics Inc. and Los Alamos National Laboratory, US

² Nucleotide Sequence Data Library, European Molecular Biology Laboratory, EC

³ DNA Data Base of Japan

⁴ Protein Information Resource, National Biomedical Research Foundation, US

⁵ Protein Data Bank

2 Specifications

2.1 Schema of GenBank in Kappa

The schema based on the nested relational model for GenBank is shown in Fig. 1 in detail.

gene : main table which has locus name, definition, accession, keywords, identifiers to the other tables, and so on.

reference : table which has authors, titles, journals in which the paper was published, and so on.

feature : consists of a region of the sequence and its feature.

seqdata : sequence data represented in string form.

2.2 Schema of PIR in Kappa

pir_gen : main table which includes names, placements, sources and sequence data.

pir_ref : table which has authors, titles, journals and so on.

pir_fea : consists of a region of the sequence and its feature.

Schema for PIR is shown in Fig. 2.

2.3 Stored Data

GenBank	Release 60.0 (89.6.15) 26323 entries, 32 M bases
PIR	Release 21.0 (89.6.30) 6158 entries, 1.7 M residues

3 Demonstration

3.1 Display Tables : Kappa User Interface

3.2 Retrieve Data

Example: make a gene table which consists of entries (records) whose reference Dr. Woese wrote.

$$\pi_{\text{ref.id}}(\sigma_{\text{authors}='Woese*'}(\text{reference})) \bowtie \text{gene}$$

3.3 Feature Expression

3.4 Similarity Judgment

Flowchart is shown in Fig. 3.

Translation We select a DNA sequence in the feature table to translate into an amino acid sequence. The sequence is translated according to the table shown in Fig. 4.

DP matching We compare the '*translated*' sequence with another (selected and translated) sequence. The sequences are compared according to the table selected by the user shown in Fig. 5, for example.

D e t a i l s (5 / 6)

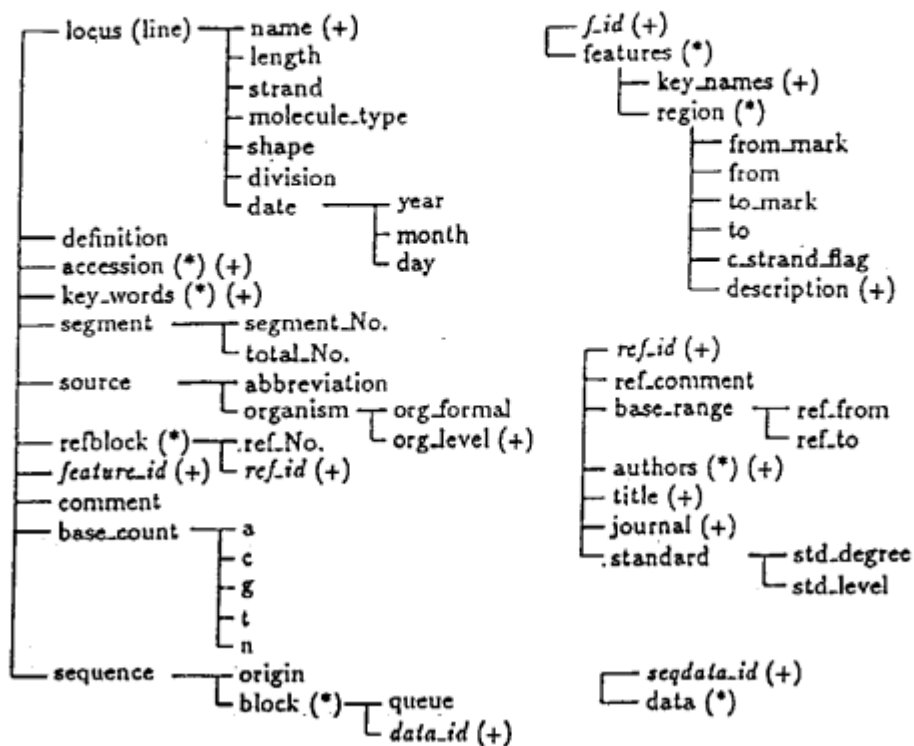


Fig. 1 Schema of GenBank in Kappa

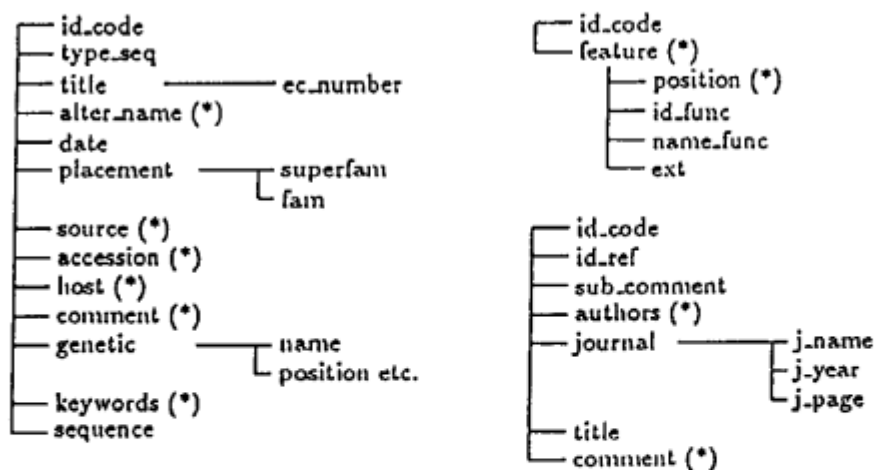


Fig. 2 Schema of PIR in Kappa (excerpt)

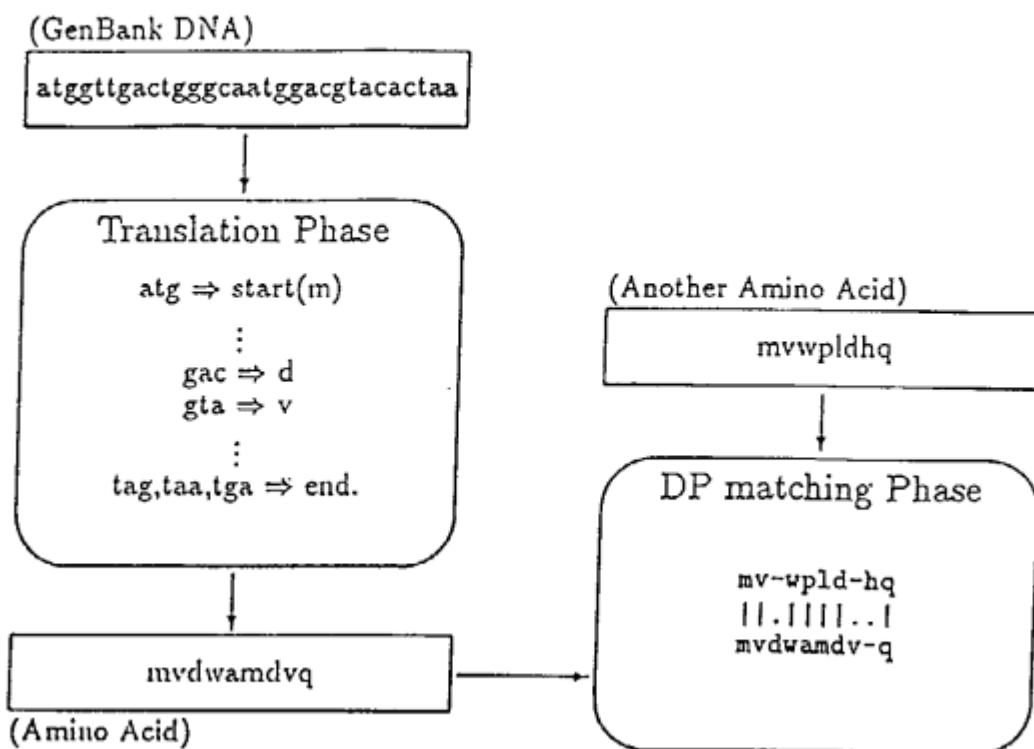


Fig. 3 Similarity Judgment between DNA's

Fig. 4 Translation Table

1st (5'end)	2nd				3rd (3'end)
	T	C	A	G	
T	l	s	y	c	T
	l		*	*	A
			*	w	G
C	l	p	h	r	T
					C
			q		A
				G	
A	i	l	n	s	T
					C
			k	r	A
	m			G	
G	v	a	d	g	T
					C
			e		A
				G	

(* terminate codon)

Fig. 5 Mutation Table (Feng)

Mutability	c	s	t	p	a	g	n	d	e
c: cysteine	6								
s: serine	4	6							
t: threonine	2	5	6						
p: proline	2	4	4	6					
a: alanine	2	5	5	5	6				
g: glycine	3	5	2	3	5	6			
n: asparagine	2	5	4	2	3	3	6		
d: aspartic acid	1	3	2	2	4	4	5	6	
e: glutamic acid	0	3	3	3	4	4	3	5	6
q: glutamine	1	3	3	3	3	2	3	4	4
h: histidine	2	3	2	3	2	1	4	3	2
r: arginine	2	3	3	3	2	3	2	2	2
k: lysine	0	3	4	2	3	2	4	3	4
m: methionine	2	1	3	2	2	1	1	0	1
i: isoleucine	2	2	3	2	2	2	2	1	1
l: leucine	2	2	2	3	2	2	1	1	1
v: valine	2	2	3	3	5	4	2	3	4
f: phenylalanine	3	3	1	2	2	1	1	1	0
y: tyrosine	3	3	2	2	2	2	3	2	1
w: tryptophan	3	2	1	2	2	3	0	0	1

A Functional Programming Environment Supporting Execution, Partial Execution and Transformation

John Darlington, Peter Harrison, Hessam Khoshnevisan, Lee McLoughlin, Nigel Perry,
Helen Pull, Mike Reeve, Keith Sephton, Lyndon While & Sue Wright

Department of Computing,
Imperial College,
London

Abstract

The Functional Programming Section in the Department of Computing at Imperial College has been conducting research aimed at making practical the theoretical benefits inherent in pure functional programming, i.e. more expressive programming languages, software development based on formal program derivation and parallel execution. This work has led to, amongst other things, language design and implementation, the development of some of the earliest program transformation systems and the construction of ALICE, a parallel graph reduction machine.

Recently some of this work has been carried forward within the Flagship project. Flagship is a collaborative project, funded under the United Kingdom's Alvey Programme and running from 1986 until early 1989. The partners involved are ICL (International Computers Limited), Imperial College and Manchester University. Flagship's overall goal is to develop an integrated application development and execution technology based on declarative programming languages (both functional and logic).

Imperial College's contribution involves language design and implementation and the construction of a program development technology and associated tools based on the idea of correctness-preserving program transformations. This has led to the development of a range of transformation technologies for functional programming languages and the implementation of a programming environment that supports these technologies. Details of the transformation technologies themselves have been published elsewhere and so they are only summarised here. In this paper we will concentrate on the integration of these technologies into the programming environment.

First we describe our functional language, Hope⁺, and introduce an extension to the language that allows the use of logical variables. We then briefly describe each of the transformation technologies that have been developed. Next we show how these are integrated into an environment that supports transformational programming. We give some examples of transformational developments conducted in the environment and show the preliminary results of experiments conducted in developing program forms suitable for efficient execution on ALICE and the Flagship parallel machine. We conclude with a review of the current state of our activities, a discussion of the implications of our work and an outline of our plans for the future.

Keywords: Functional programming environment, program transformation, partial evaluation, parallel execution.

The Functional Programming Section in the Department of Computing at Imperial College has been conducting research aimed at making practical the theoretical benefits inherent in pure functional programming, i.e. more expressive programming languages, software development based on formal program derivation and parallel execution. This work has led to, amongst other things, language design and implementation, the development of some of the earliest program transformation systems [BD77] and the construction of ALICE [CDF87][DaR81], a parallel graph reduction machine.

Recently some of this work has been carried forward within the Flagship project. Flagship is a collaborative project, funded under the United Kingdom's Alvey Programme and running from 1986 until early 1989. The partners involved are ICL (International Computers Limited), Imperial College and Manchester University. Flagship's overall goal is to develop an integrated application development and delivery technology based on the adoption of declarative programming languages (both functional and logic). Towards this goal Flagship is developing:

- a hardware emulator for a parallel graph reduction machine [WaW86][Tow87].
- a series of compilers for a range of declarative and symbolic programming languages onto the parallel machine via the intermediate graph rewrite language DACTL [GKS87].
- a system architecture and system software capable of supporting a range of relevant applications and interacting with other existing systems [Bro87].
- extensions of functional languages and efficient sequential implementations.
- a program development technology and associated tools based on the idea of correctness-preserving program transformations.

The last two tasks are the prime responsibility of Imperial College and have led to, amongst other things, the development of a range of transformation technologies for functional programming languages and the construction of a programming environment that supports these technologies. In this paper we would like to introduce these technologies and the environment.

The starting point for our current work was the language Hope developed at Edinburgh University [BMS80]. We have developed a compiler for Hope, the FP/M compiler, that on sequential machines gives performance comparable with conventional imperative languages. Within Flagship our first step was to extend Hope, in a conservative manner, to a language Hope⁺ that was used as the implementation language within Flagship. Hope⁺ differs from Hope in having:

- real numbers and vectors
- recursive *let* and *where*
- continuations [StW74][Per88] for all input/output including language interworking
- lazy data constructors
- best-fit pattern matching [FHW87][Whi88]

From this basis we have developed the language and transformation technologies and implemented them to provide an integrated programming environment that directly supports transformational programming. In

sections 2-4 below we introduce the language and transformation technologies. Section 5 discusses the aims of the environment and how these are realised. Section 6 briefly introduces the graphical interface provided by the environment. Section 7 describes how the environment is constructed. Section 8 gives an example transformational development carried out in the environment aimed at developing a program form suitable for efficient execution on both the ALICE and Flagship parallel machines. Preliminary results of experiments carried out on ALICE are presented. Finally section 9 reviews the current state of our activities and discusses the implications of our work and plans for the future.

2 Hope⁺ With Unification

Absolute Set Abstractions were introduced into Hope⁺ to increase the expressive power of the language [DFP86]. In any functional language, function definitions exhibit directionality, i.e. under normal execution mechanisms they accept input values in the domain of the function and return output values in the range of the function. We relax this, however, when a function application appears in an Absolute Set Abstraction (ASA), and have implemented an algorithm based on narrowing to execute this language construct (evaluation of all other constructs is by reduction). Thus, we provide a mechanism for writing executable high level specifications, and incorporate the full power of logic programming into Hope⁺.

An example of an ASA is the following function which returns the set of sublists of its argument.

```
dec sublists : list  $\alpha$  -> set ( list  $\alpha$  );
-- sublists l <- { v with u , v , w | u  $\diamond$  v  $\diamond$  w = l } ;
```

Here u, v and w are *logical variables* bound by the enclosing braces. Informally, the expression denotes the set of values obtained by evaluating the expression to the left of the with for each distinct tuple (u, v, w) which satisfies the constraint to the right of the |. A full, formal semantics is given in [Guo88][DaG88].

The algorithm for evaluating ASAs is based on lazy narrowing and executes a complete, fair search. We repeatedly select an equation from the body of the ASA, attempt to unify the equation and apply substitutions if successful. If unification is not possible then narrowing is performed until unification can proceed. Under lazy narrowing, this means that an expression is only narrowed until it has a constructor at the outermost level, allowing failure of unification to be established early. [Guo88][DaG88] show that this narrowing strategy is both correct and optimal for the solution of these equations.

Narrowing introduces all the potential inefficiencies of any non-deterministic programming language. Rather than demand that the programmer understand the intricacies of the implementation, and structure his program accordingly to cope with this (as is the case with Prolog), we provide a complete implementation of ASAs and wherever possible use transformation to convert ASAs into efficient equivalent deterministic programs which execute by reduction.

The fact that ASAs provide full support for logical variables means that symbolic execution, execution with non-ground values, is subsumed. Execution can be extended to provide a full transformation capability, equivalent to the classic unfold/fold system [BuD77], by the incorporation of a

few extra rules, e.g. the application of laws and folding. Transformation thus becomes a superset of execution and the same mechanism, lazy narrowing, can be used for its support. This simplification also brings with it extra power, for example the fact that narrowing (not pattern matching) is used for unfolding and folding means that the correct instantiations of the unfolded (folded) sub-expressions are often generated automatically.

We have also developed a mechanism that allows the execution of Hope⁺ programs to be controlled. A set of execution control primitives together with combining forms are provided as the Hope⁺ data type script. These are applied using the meta-programming facilities outlined in section 5. As transformation is now a superset of execution, the transformation control primitives are similarly a superset of the execution control primitives. Thus a uniform program development methodology is supported whereby efficient execution strategies for a program can be explored by developing appropriate scripts and applying them to specific executions of the program. Once an efficient execution strategy has been found a transformation script can often be developed by a systematic modification of the execution script. Applying the transformation script to the source program verifies and implements the transformation, returning a transformed program which will execute in the prescribed manner without further explicit control.

A fuller description of the unification of execution and transformation enabled by ASAs and the use of scripts can be found in [DP87].

3 Algebraic Transformations

3.1 Axiomatic Approach, FP Form

The unfold/fold transformation methodology is very general in that most known optimisations can be expressed as a sequence of its primitive steps, but its scope for automation is limited since the step sequences tend to be lengthy; this has led to the semi-automatic procedure using scripts. However, consideration of the object domain of a function need not be crucial to the analysis of the function itself, which is the real objective of our optimisation, and may well obscure it. The *algebraic* approach to transformation derives theorems which state generic identities between functions by establishing the corresponding equalities between function applications to objects in the underlying semantic domain. A transformation then becomes just an instance of an application of a theorem in a term-rewriting system. This approach has three main advantages:

- Reasoning at the 'function-level', we have no need to be concerned with the auxiliary domain of objects and the functional expressions acquire a simpler structure - we use the FP style of [Bac78].
- Powerful transformations can be derived relatively easily because of the simpler syntax of the functional expressions.
- The *grain-size* of the transformations is increased since the theorems equate whole function definitions rather than repeatedly making appropriate variable substitutions.

Based upon a sound semantic analysis, the theorems are adopted as the syntactic *axioms* of a term-rewriting system which applies them to function definitions with the appropriate form.

There are several transformation schemes based on the FP algebraic approach. One scheme addresses the transformation of a class of linear functions into imperative language loops, or equivalent tail recursive forms [HaK86]. There are also schemes that transform certain classes of non-linear functions into linear form, one of these using memoisation with dynamic organisation of the memo-table storage, as described in section 3.2. Using an extended algebra which includes axioms for many-valued functions, it is possible to synthesise mechanically *inverses* for a significant class of recursive functions and we discuss this in section 3.3.

These algebraic transformations have been developed with the support of the U.K. Science and Engineering Research Council and carried out in close collaboration with the Flagship project.

3.2 Memoisation

Memoisation is a route to the efficient implementation of non-linear functions. A memo-function, originally introduced by Michie [Mic68], is like an ordinary function except that it remembers all the arguments it has been applied to, together with the corresponding results computed from them. If a memo-function is ever re-applied to an argument it does not recompute the result, but just re-uses the result computed earlier. Thus

memoisation can be used to replace a potentially expensive computation by a simple table lookup. The classic example is the Fibonacci function:

```
dec fib : num -> num ;  
-- fib n <= if n <= 1 then 1 else fib ( n-1 ) + fib ( n-2 ) ;
```

Since each call to fib generates two recursive calls, the cost of computing fib(n) is exponential in n. However, memoised fib will execute in linear time, since for each value n, fib(n) is computed only once.

The difficulty traditionally associated with memo-functions is the issue of controlling the size of the memo-table. It is difficult to know when an entry for a particular argument can safely be deleted from the table and thus the table may grow continuously, interfering with garbage collection and also increasing the cost of the table lookup operation. We have developed a variant of memoisation where memo-tables manage their own storage by deleting (or reusing) entries when it is known that such entries will never be referenced again. The function definition is statically analysed and a *table-manager* function is generated. For a function f of type $\alpha \rightarrow \beta$, the table-manager tabf is of type $\alpha \rightarrow \text{list } \alpha$; given an element x in the domain of f , the expression $\text{tabf}(x)$ specifies which entries can safely be deleted from the table when an entry for x is added to the table. Furthermore, the size of the memo-table is guaranteed not to exceed a compile-time constant. For our fib example above, the table-manager function is $\lambda x.[x-2]$ and the transformed function fib' is:

```

dec fib' : num # table ( num # num ) -> num # table ( num # num ) ;
--- fib' ( n , tab0 ) <== if n == 1
    then ( 1 , tab0 )
    else let found res == lookup ( n , tab0 ) in ( res , tab0 )
        otherwise let ( r1 , tab1 ) == fib' ( n-1 , tab0 ) in
            let ( r2 , tab2 ) == fib' ( n-2 , tab1 ) in
                let res == r1 + r2 in
                    ( res , insert ( n , res , tab2 , λx.[ x-2 ] ) ) ;

```

This improvement in space usage is achieved by increasing the cost of the insert operation by a small amount, since insert now has to execute the table-manager each time an entry is added to the table. However, table-managers are guaranteed to be non-recursive and furthermore, no extra apparatus is needed to execute them since they can be expressed in the functional language itself.

The entire memoisation operation is automatic. The definition of the memoisable class of functions, the proofs of correctness, the synthesis of table-manager functions, details of variable-free analysis and further examples are given in [Kho87].

3.3 Function Inversion

Despite their advantages, functional programming languages lack the ability to use relations in several modes, as in logic languages. This ability has been incorporated into Hope⁺ by the means of Absolute Set Abstractions, but supporting unification within a language is less efficient than the reduction system used for implementing pure functional languages. The implementation of relations would be facilitated by *function inversion*, as a function together with its inverse constitutes a relation. Inverses can be synthesised from ASAs by a series of unfold/fold steps. The analysis outlined here provides a way of *automatically* generating the inverses of many first-order functions.

The analysis requires some extensions to the normal FP language; in particular, we extend the language to include logical variables and function-level unification which are needed to express the inverses of some of the FP constructs. Rules are defined to invert each of the FP constructs along with a number of standard Hope⁺ functions and the user is required to specify the inverses for any functions which are not built-in to the system. The resulting expressions are simplified using a term-rewriting system along with a set of axioms designed to perform transformation-time function-level unification, thus removing logical variables from the expression which are introduced by the inversion process.

Full details of the modified FP language along with the rules used for inversion and the term-rewriting system used can be found in [KhS89]. A more theoretical basis for the inversion process and axioms relating to function-level unification can be found in [HaK88].

An important use of function inversion is in the optimisation of data type representations. Given an abstract data type α , its concrete representation α' and a function abs of type $\alpha' \rightarrow \alpha$ that formalises the representation relationship, a function f working on the concrete data type can be synthesised from the definition of an f which works on the abstract data type by

$$f = \text{abs}^{-1} \circ f \circ \text{abs}$$

This relationship is shown in Figure 3.1.

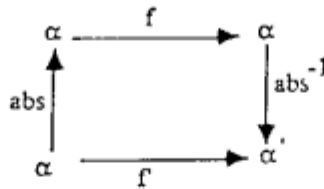


Figure 3.1: Commutative Square for Data Type Mappings

This idea, described in [BD77][HaK89], enables the programmer to work with problem-specific abstract types without worrying about efficiency as functions implemented on the (primed) concrete types can be generated automatically. An example of this use of inversion is given in section 8.4.

4 Temporal Synchronisation of Functional Programs

One of the intrinsic advantages of declarative languages is that they free the programmer from the need to specify any explicit ordering on the evaluation of his program; a program is simply a *mapping* from input values to output values. This in turn leads to such benefits as the formal manipulation of programs and parallel evaluation. However, for some classes of applications the behavioural component of a program is as important to its correctness as the value returned by its execution. For such applications the programmer needs some way of exerting control over the behaviour of his program, i.e. of (at least partially) controlling the order in which reductions occur during its execution. Typical examples of such applications are where the program has to communicate with the "real world", e.g. to control plant machinery, or in transaction processing systems where common resources have to be scheduled between multiple users.

Our approach to the problem of specifying temporal behaviours for declarative programs is to use a *temporal logic* meta-language, SIAN, to describe the required relative occurrences of critical events in a program's execution. SIAN is a standard discrete-time temporal logic [MaP81] augmented with the ability to define predicates in Hope⁺ over the numbers of events occurring during a program's execution. A *critical event* is defined to be the use of an identified rewrite rule in the program to reduce a redex in the program graph. Control is thus expressed at the level of a single function rewrite. The programmer constructs a SIAN expression describing the requirements on the relative occurrences of the set of critical events; an execution behaviour for a program is correct relative to a SIAN expression if the expression is true at every point in the execution, i.e. at every rewrite.

This specification is implemented using *program transformation* techniques, combining the original declarative program and the associated SIAN expression into a single (almost) declarative program which has the same meaning as the original program (i.e. it returns the same value, or more generally in the presence of non-determinism, a non-empty subset of the original set of possible values) but whose

behaviour is guaranteed to observe the requirements stated in the temporal logic part of the specification. The first stage of this transformation is to reduce the SIAN expression to its *normal form* representation, using a complete and confluent set of transformation rules. The normal form for a SIAN expression comprises a list of implications, the consequence of each being a conjunction of events indicating which events must *not* occur in the current reduction cycle. The importance of the normal form (and what distinguishes it from a general SIAN expression) is that it is defined in such a way that for each implication the condition is testable from what has happened in the *immediate past* and the consequence is enforceable simply by preventing certain events occurring in the *present*. The latter can be achieved via the normal rewrite-rule selection mechanism. Thus at each step of the execution an event which would cause any of the implications to become false can be disallowed. This means that a normal form expression can be expressed as a finite-state machine (FSM), the state of which contains all the information necessary to preserve the truth of the associated SIAN expression and the transitions of which represent the occurrences of the permitted critical events. The FSM can be represented as a data structure and incorporated into the original program being controlled. A rewrite rule for the FSM is derived for each critical event and the equation for each event is transformed to include a pattern on the state of the FSM and to update the FSM whenever it is used in a rewrite.

As a simple example of this technique, consider the following Hope⁺ function which merges two lists of objects non-deterministically, i.e. it does not specify which list should be given precedence when both are non-empty (this is in fact a slight generalisation of the normal Hope⁺ pattern-matching scheme).

```

dec merge : list α # list α -> list α ;
  --- merge ( nil , x ) <= x ;
  --- merge ( x , nil ) <= x ;
ML: --- merge ( x :: y , z ) <= x :: merge ( y , z ) ;
MR: --- merge ( y , x :: z ) <= x :: merge ( y , z ) ;
merge ( list1 , list2 ) ;

```

The event names ML and MR are associated with the two critical equations and can be used in a SIAN expression to refer to uses of those equations. This function could be constrained in many different ways according to the behaviour required by the particular application. Suppose the requirement is that items are merged from the two lists *alternately*. This can be expressed in SIAN with the expression:

$$ML \rightarrow T MR, MR \rightarrow T ML ;$$

The first implication states that if an item is merged from the first list in the present cycle, i.e. *today*, an item must be merged from the second list in the next cycle, i.e. *tomorrow* (denoted by the modal operator T). The second implication states the converse. The normal form corresponding to this expression is

$$Y ML \rightarrow \text{not } ML, Y MR \rightarrow \text{not } MR ;$$

where the Y operator denotes the previous cycle, i.e. *yesterday*, and is the inverse of T. This expression is represented as a finite-state machine and incorporated into the original function, giving as the final result:

```

data state == ST ( truval # truval ) ;
dec merge : state # list  $\alpha$  # list  $\alpha$  -> list  $\alpha$  ;
  --- merge ( _ , nil , x )          <= x ;
  --- merge ( _ , x , nil )         <= x ;
ML: --- merge ( ST ( false , _ ) , x :: y , z ) <= x :: merge ( ST ( true , false ) , y , z ) ;
MR: --- merge ( ST ( _ , false ) , y , x :: z ) <= x :: merge ( ST ( false , true ) , y , z ) ;
merge ( ST ( false , false ) , list1 , list2 ) ;

```

Each element of the state 'records' the current value of the condition of one of the implications in the normal form expression. The top-level call to merge will allow either equation to be used initially but it is easy to see that after the first rewrite ML and MR will be used alternately until one of them fails to match.

The final transformed program is non-functional in the sense that for the critical events to communicate their occurrences to each other, the data structure representing the state of the FSM is shared between them and updated *in-place* using destructive assignment. However, the whole transformation process from "program+SIAN expression" to the final program is totally automatic, so the programmer need not be concerned with the implementation details.

Full details of this approach and associated issues can be found in [DaW87][Whi88].

3 The User's View of the Environment; Meta-programming

The developments outlined above provide a potentially powerful battery of techniques to support functional programming and transformational developments. The task in designing the environment was to present this capability to the user in the most unified and simple way possible. The design criteria for the environment included:

- The environment should, as much as possible, follow the functional 'style'. That is simplicity should be favoured over complexity and powerful constructs should be built by the uniform application of a small number of simpler constructs. More technically a strong typing discipline should be enforced.
- "Normal" programming and transformational program development should be supported uniformly.
- Transformations should be checked automatically for correctness. It should be impossible for a user to construct and have implemented an incorrect transformation.
- Transformational developments should be represented by a concrete, analysable and storeable object.
- Transformations should be directed by and comprehensible to the user. We are not aiming for an entirely "automatic" programming system but one where the user is in strategic control and the system provides high level support for checking and implementing his plans.
- Completeness should be preferred to cleverness. We should aim, initially, for a system capable of supporting a wide range of interesting transformational developments, even at the cost of some effort from the user, rather than a system that is able to automatically accomplish a more limited range of transformations.

The environment should be extensible. An ordinary user should be able to define more powerful transformation capabilities and other environment enhancements with safety guaranteed at all times.

The technique we used to meet these design goals was to incorporate the transformation capability into the Hope+ with Unification (HwU) interpreter via *meta-programming*. In this way, normal (HwU) programming and transformational programming are the same, they involve defining and applying functions in the normal manner, it is just that the functions operate on objects of different types.

Thus the type *program* is available as a system-provided data type; an object of type *program* is a user program module, that is a set of functions and type definitions together with *import* and *export* declarations. Access to objects of type *program* is controlled to ensure safety. A set of *meta-functions* are provided that take an object of type *program* and return an object of type *program*. These correspond to the basic transformation operations.

The basic meta-functions currently provided are *apply_script*, *trace_search*, *build_tree*, *memoise*, *invert* and *synchronise*. Their types are overloaded to permit flexible use.

Let *f* denote a function identifier of type $\alpha \rightarrow \beta$
g denote the name of a new function (a list of characters)
m denote the name of an existing module
p denote an object of type *program*
s denote a script
t denote a SLAN temporal logic expression
n denote an expression of type *num*
e denote an expression whose type is relative to the type of the corresponding *f* in the following expressions

Then the following are all the valid forms of expressions of type *program*.

m
- modules can be named

apply_script (*s*, *p*)
- the script *s* is applied to (some function definitions from) the program *p*

trace_search (*s*, *p*)
- the script *s* is applied to the program *p* and the transformation is traced

build_tree (*s*, *p*)
- the script *s* is applied to the program *p* and a data structure representation of the transformation is returned

memoise (*f*, *p*)
- the function *f* is memoised. If the system cannot detect that the function is memoisable the program is returned unaltered

memoise (*f* using manager *e*, *p*)
- as above but with the table manager provided explicitly

memoise (*f* using reference *n*, *p*)
- as above but with the table reference count provided explicitly

- invert (f giving g , p)
 - a new function g is created which is the inverse of f
- invert (f given [f₁ inverts_to e₁ , ... , f_n inverts_to e_n] giving g , p)
 - as above but using the information that inverses exist for functions used in the definition of f
- synchronise (t , p)
 - the synchronisation constraints specified by t are incorporated into the appropriate function definitions in the program p

All of the above functions also simplify the module returned. Any existing equations that are completely subsumed by new equations produced by the transformations are discarded. All of the above meta-functions act as first class HwU functions so they can be composed in any type-correct manner and used in the bodies of other function definitions. Thus a transformation plan can be constructed simply as a HwU function definition. Applying the function to the appropriate program object implements the transformation, producing the transformed program object. Safety is guaranteed by the fact that the basic meta-functions are the only ones able to construct objects of type program and they are correct by construction and correctness (i.e. meaning preservation) is preserved by function composition. Transformation is thus uniform with execution, it is just the types that are different. Our environment thus uses an amalgam of ideas from LISP and Prolog environments [Gre84][CFL88] and the ML/LCF Theorem Proving System [GMW77].

The function definitions produced by the application of a meta-function are similarly first class citizens and can be used as normal for execution. Before this can happen the newly produced modules must be added to the environment's set of visible modules. To this end the user is provided with an environment updating infix operator := of the following form

m := p

In this case m must be a new name, not the name of any module currently visible. The way we like to think of := is as if it had the declaration

dec := : module_name # program # environment -> environment ;

where environment is a mapping from names to objects including environments and programs. If we enforced this and also ensured that all other 'system' functions, e.g. editors or program executors, had to be passed the environment explicitly we could claim that our whole environment was functional. However we concede that this would be tedious and accept for the moment a single updateable environment and use the normal 'module' and 'uses' commands of Hope+ to load definitions.

We have further enhanced the extensibility of the environment for the user programmer by providing him with access functions to the Hope+ data structure representing the abstract structure of function definitions. These allow the forms of function definitions to be analysed and more general transformation functions to be defined, corresponding to ML/LCF tacticals. These functions have type

program -> (program -> program)

It is by the definition of such functions that we envisage the general 'intelligence' of the environment being increased. Generally such functions will employ a searching strategy, guided by heuristics and

transformational knowledge, to attempt to produce a specific transformation applicable to the given program, from a generic class of transformations. Note that safety is still guaranteed by the fact that such functions only seek to affect the given program by using existing meta-functions and return a composition of these meta-functions.

With the apparatus outlined above we feel we have met most of our design criteria. The user can define and implement a transformation simply by writing a functional program which is, of course, a storable and re-useable object. Transformations of significant complexity can and have been defined using this apparatus. Some examples are given in section 8 and an initial set of tacticals have been written.

6 The Graphical Interface to the Environment

The graphics interface gives a visual representation to the objects in the underlying module system of Hope+. It was implemented in Hope+ using the Hope+ interface to the X Windows System [DGN86]. The Xray toolkit [HP86] was used to provide the higher-level graphics facilities.

On starting the environment the user is presented with two windows. The first is a terminal emulator (titled 'xterm' slave). The second is a window representing the "open" modules and is titled 'environment'. Figure 6.1 shows a view of a typical session, specifically associated with the transformational development example described in section 8.

The terminal window provides the keyboard/screen-based interaction with the environment. All of the prompts and messages from the environment are displayed in this window. Replies to the prompts, new scripts and Hope+ functions are all typed in here.

In the module window titled 'environment' each "open" module is represented by an icon. To look at the contents of one of these modules the mouse is clicked on its icon. Clicking the mouse on a module's icon causes a new module window to appear, titled with the name of the module, that contains an icon for each function or type defined in the module. All module windows, except the 'environment' window, can be removed from the screen by clicking on the "close box" beside its title.

The definition of a function or type can be viewed by clicking the mouse on its icon. This will cause a browser window to appear containing the code for the function (or type). The browser program can be chosen by the user before entering the environment.

There is no limit to the number of icons that may be in a module window so there are scrollbars to move the window over the icons. Similarly there are no limits to the number of modules or browsers that may be open at any one time. These windows are ordinary windows under the X Windows System and can be moved around by the window manager of the user's choice.

The transformation environment described in this paper was written in the language Hope⁺ and compiled to run on a SUN3 machine using a compiler which was also developed at Imperial College [PeS87]. The underlying system on top of which the transformation system is built was originally developed as a compiler for the language Hope⁺. The system consists of a number of individual building blocks such as a parser, type-checker, interpreter, transformation tactics, etc. which are linked together by a piece of "glue" which handles the interaction between the various parts. The separate modules communicate information about the program via a symbol table. Due to the modular nature of the existing compiler and the various other tools, it was possible to integrate the system with relative ease. Figure 7.1 shows an overview of the software architecture of the system.

7.1 Continuations

Continuations provide a way of performing I/O and other system related operations within a functional language without having to resort to functions which have side-effects [StW74][Per88]. The system provides a data type continuation which is used to give "instructions" to the underlying operating system.

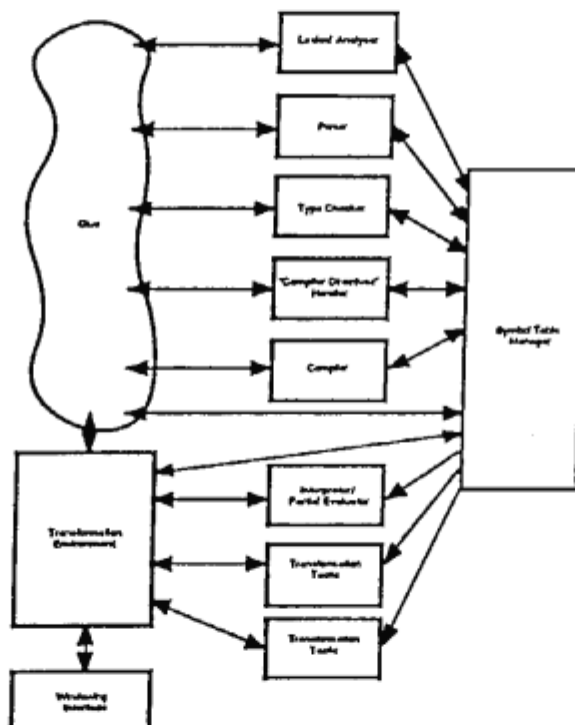


Figure 7.1: Software Architecture of the System

Each continuation includes a function which is to be called once the operating system task has been performed. For example, to read a line of text from the keyboard:

```
GetLine ( lambda ( input , status ) => ProcessInput input )
```

Within the environment, continuations are used for input/output to the terminal and individual files; testing whether files exist and are readable; detecting whether errors have occurred whilst performing I/O operations (this is particularly important in the case of saving programs, as failure to write a file might otherwise result in the user losing his program); handling signals so that the user may type Control-C to interrupt the interpreter; and interlanguage working to link up with the graphics package. All of these operations can be performed without having to worry about the system deadlocking (as can be the case with streams), or operations being performed in the wrong order (as can be the case with side-effect I/O).

A full description of the continuations supported by Hope⁺ can be found in [Per87].

8 A Simple Example of Parallelism Exploitation on a Flagship-style Machine

8.1 Example 1: Transformation and Optimisation

Many applications naturally involve the list data structure. However, the list is not a good data structure for parallel evaluation; its elements are most naturally accessed *sequentially* from the head. An alternative data structure is parlist, a tree whose tips contain lists:

```
data parlist  $\alpha$  ::= tip ( list  $\alpha$  ) ++ node ( parlist  $\alpha$  # parlist  $\alpha$  );
```

The relationship between parlists and lists can be formalised by defining a function abs:

```
dec abs : parlist  $\alpha$  -> list  $\alpha$  ;  
--- abs ( tip l )          <= l ;  
--- abs ( node ( t1 , t2 ) ) <= abs t1 <> abs t2 ;
```

where <> is the built-in Hope⁺ function which appends two lists together.

A sequential list can then be represented as a parlist by breaking it up and distributing it across the tips of the parlist. If the number of tips can be made equal to the number of processors available, the use of the parlist data structure will allow parallel processes to be executed on individual processors, requiring access only to the segment of the list stored locally. Such a style of execution ideally suits a machine of the flagship style which is a multi-processor, shared memory machine with closely coupled local store segments [WaW86][Tow87].

However, it would be awkward for programmers to have to directly code applications using the parlist structure, arising as it does purely from the machine's physical characteristics rather than any application considerations. Instead, we can use transformation techniques to allow the programmer to use the appropriate, application-oriented, data structures and then systematically convert such programs to versions that operate on the efficient, machine-oriented, structures.

For example, consider the function count, that counts the number of elements in a list:

```

dec count : list  $\alpha$  -> num ;
-- count nil      <= 0 ;
-- count ( x :: l ) <= 1 + count l ;

```

The definition of an analogous function on parlists, `countpar` say, is induced by the parlist representation of lists, `abs`, described earlier:

```

dec countpar : parlist  $\alpha$  -> num;
-- countpar pl <= count ( abs pl ) ;

```

Executing this definition directly would of course produce no benefit as it would convert the parlist to a sequential list before carrying out any operations on it. However we can use simple transformations to produce a direct definition of `countpar` operating solely on parlists:

```

countpar ( tip l )      = count ( abs ( tip l ) )      instantiation
                        = count l                    unfold abs

countpar ( node ( t1 , t2 ) ) = count ( abs ( node ( t1 , t2 ) ) ) instantiation
                              = count ( abs t1 <> abs t2 )      unfold abs
                              = count ( abs t1 ) + count ( abs t2 )
                              using lemma:
                              count ( t1 <> t2 ) = count t1 + count t2
                              = countpar t1 + countpar t2      fold countpar

```

Thus our definition of `countpar` becomes:

```

-- countpar ( tip l )      <= count l ;
-- countpar ( node ( t1 , t2 ) ) <= countpar t1 + countpar t2 ;

```

We can improve things even further by converting the call to `count` in the base case of `countpar` into a call to a tail recursive version of `count`. On ALICE these tail recursive calls can be compiled to imperative loops that run sequentially on one processor in fixed space.

Thus we have converted our program into a coarse-grained collection of sequential processes ideally suited to the target machine. The program can potentially make optimal use of the machine's resources, by executing each recursive invocation of `count` on a single processor and minimising communication, e.g. by executing the `+` on the same processor as one of the recursive calls to `count`. To ensure that this potential is realised, we can use meaning-preserving annotations to indicate the optimal positioning for the processes. The route used to produce the transformed version of `count` is illustrated in Figure 6.1.

The benefits gained by incorporating the above transformations and optimisations into the `count` program have been evaluated experimentally using ALICE [CDF87][DaR81]. Three cases were studied; the sequential `count` program, `countseq`, the transformed parallel `count` program, `countpar` and the optimised transformed parallel counting program, `while_countpar`.

8.2 The ALICE Machine

The ALICE machine logically consists of three types of units. These are processors (or packet rewrite agents), stores (or packet pool segments) and a switching network. ALICE was specifically designed to be a flexible experimental vehicle which can be configured in different ways for different execution modes. It can be parameterised at program invocation time to operate in one of several load-sharing modes. The ones used for the count example were uncoupled-random for the countseq program and close-coupled for the countpar and while_countpar programs. The uncoupled-random load-sharing mode involves running the program across all the agents and stores, with the agents looking for work and storing any new packets generated randomly across the stores. In close-coupled load-sharing mode each agent has a store logically associated with it, its local store, the number of agents and stores being equal. The agents then look for work and store any new packets in their local store only, unless explicitly directed to do otherwise by annotations appearing in the optimised transformed program.

The Flagship machine is itself designed around the notion of close-coupled processor-store pairs upon which programs annotated with the required load-sharing strategy can be run. Thus analysing the three versions of the count example on ALICE provides a way of predicting the potential benefits of the transformation and optimisation route adopted for the count program on the Flagship machine.

8.3 Experimental Details and Results

The current Imperial College ALICE machine has 16 agent boards and 26 store boards, thus in close-coupled mode the maximum number of agent-store pairs which the machine can be configured with is 16. Thus the countseq program was analysed with all agents and stores running whilst the countpar and while_countpar programs had 16 agents and 16 stores operating. The length of the list used as the argument to the count programs was 400. Therefore, the length of the list at each tip of the parlist for the countpar and while_countpar programs was 25.

The execution times for the three count programs when applied to a list with 400 elements are given in Table 8.1. These show an increase in efficiency of approximately 9 fold in moving from the countseq program to the countpar program and an increase of 13 fold from the countpar to the while_countpar programs. This produces a marked overall increase in efficiency of approximately 110 fold comparing the countseq program to the while_countpar program.

Program	Execution Time(ATUs*)
countseq	1789
countpar	214
while_countpar	16

* ATU = ALICE Time Unit = 20ms

Table 8.1: Execution Times for the Three Versions of the count Program

Also of interest is the percentage of local agent-to-store accesses with time for the countseq and countpar programs (the countpar and while_countpar being approximately the same). These statistics have been obtained and also show a significant improvement due to the transformation. The percentage of local accesses for the countseq program tends to lie in the 5-10% range whereas that for the countpar program tends to lie in the 80-90% range; lower figures are only found where work is being exported to build and collapse the parlist respectively. This marked increase in local accesses can be of great significance to machines configured in close-coupled mode, such as the Flagship machine, where the cost of accessing local memory is likely to be much less than that of accessing remote store.

These results clearly show the potential benefits of using program transformation techniques to optimise functional programs which are to be run on a Flagship-style parallel machine. These benefits are shown in terms of an increase in both efficiency and the number of local store accesses for the transformed and optimised versions of the count program compared to the original sequential version.

8.4 Example 2: Inversion of Data Types

Suppose we were to consider another function on lists, map:

```
dec map : (  $\alpha \rightarrow \beta$  ) # list  $\alpha \rightarrow$  list  $\beta$  ;
--- map ( f , nil ) <= nil ;
--- map ( f , x :: l ) <= f x :: map ( f , l ) ;
```

Rendering this function into parlists involves the inverse of the representation function abs, rep:

```
dec mappar : (  $\alpha \rightarrow \beta$  ) # parlist  $\alpha \rightarrow$  parlist  $\beta$  ;
--- mappar ( f , pl ) <= rep ( map ( f , abs pl ) ) ;
```

As previously, we could use unfold/fold transformations to generate a new version of mappar solely in terms of parlists. This example, however, can be transformed *automatically* using the inversion techniques of section 3.3 to generate the definition of rep. The first step is to invert the function \leftrightarrow in the definition of abs, generating the function split:

```
dec split : list  $\alpha \rightarrow$  set ( list  $\alpha$  # list  $\alpha$  ) ;
--- split nil <= { ( nil , nil ) } ;
--- split ( x :: l ) <= { ( nil , x :: l ) } U { ( x :: u , v ) | ( u , v ) in split l } ;
```

The same technique is then used to generate rep itself:

```
dec rep : list  $\alpha \rightarrow$  set ( parlist  $\alpha$  ) ;
--- rep l <= { tip l } U { node ( pu , pv ) | pu in rep u , pv in rep v , ( u , v ) in split l } ;
```

Note that rep is in fact set-valued; this corresponds to the fact that there are several concrete representations of any particular value in the abstract domain. It is in part the appropriate choice made between possible representations which gives the concrete representation a performance advantage over its abstract

counterpart. We represent this in the definition of `mappar` by adding a function `any`, which selects a single element from a set:

```
--- mappar ( f , pl ) <- any ( rep ( map ( f , abs pl ) ) ) ;
```

9 Conclusions

We have presented the technologies underlying the transformational development of programs being pursued by our group. These include a functional language that admits logical variables, fold/unfold transformations, algebraic transformations and a temporal logic system for controlling a program's behaviour. We showed how these are combined to provide a uniform environment that supports program execution, partial evaluation and transformation by the concept of meta-programming. Meta-functions operate on programs to produce new programs and control the evaluation strategy of the Hope⁺ with Unification interpreter itself. Finally we demonstrated how such an environment can be used to derive an efficient program for a parallel machine from an application oriented but sequential specification.

The above system has been implemented (in Hope⁺) and is in day-to-day use by our group. It has also been shipped to our Flagship partners for evaluation by their application writers. In addition, a number of joint projects with large commercial users who wish to quantify the leverage the technology will give to the development of "real world" applications are being established.

The group is continuing to do research at all levels: language design (syntax and semantics) and implementation; transformation technology; meta-programming; and parallel computer architecture. Much emphasis is being placed on gaining experience in using the transformational development process on large applications. Aside from highlighting any deficiencies in the technology, it is hoped that this will reveal a library of high-level strategies (heuristics) that will be widely applicable and can be captured by the composition of meta-functions.

References

- [Bac78] Backus, J. W., *Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs*, CACM, 21(8), August 1978, pp. 613-641.
- [Bac81] Backus, J. W., *The Algebra of Functional Programs*, Function Level Reasoning, Linear Equations and Extended Definitions, Springer-Verlag, Volume 107, 1981, pp. 1-43.
- [BMS80] Burstall, R. M., MacQueen, D. B. and Sannella, D. T., *HOPE: An Experimental Applicative Language*, Internal Report CSR-62-80, University of Edinburgh, U.K., 1980.
- [Bro87] Broughton, P., Thomson, C. M., Leunig, S. R. and Prior, S., *Designing System Software for Parallel Declarative Systems*, ICL Technical Journal 5(3), May 1987, pp. 541-554.
- [BuD77] Burstall, R. M. and Darlington, J., *A Transformation System for Developing Recursive Programs*, J. ACM, 24(1), January 1977, pp. 44-67.
- [CDF87] Cripps, M. D., Darlington, J., Field, A. J., Harrison, P. G. and Reeve, M. J., *The Design and Implementation of ALICE: A Parallel Graph Reduction Machine*, Selected Reprints on Dataflow and Reduction Architectures, ed. S. S. Thakkar, IEEE Computer Society Press, 1987.

- [CFL85] Coscia, P., Franceschi, P., Levi, G., Sardu, G. and Torre, L., *Meta-Level Definition and Compilation of Inference Engines in the Epsilon Logic Programming Environment*, Proceedings of the Fifth International Conference and Symposium on Logic Programming, Volume 1, Kowalski, R. A. and Bowen, K. A. (eds), The MIT Press, London, 1988.
- [Coh83] Cohen, N. H., *Eliminating Redundant Recursive Calls*, ACM Transactions on Programming Languages and Systems, 5(3), July 1983, pp. 265-299.
- [DaG88] Darlington, J. and Guo, Y. E., *Narrowing and Unification in Functional Programming*, Internal Report, Functional Programming Section, Department of Computing, Imperial College, London, October, 1988.
- [DaR81] Darlington, J. and Reeve, M. J., *ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages*, ACM/MIT Conference on Functional Programming Languages and Computer Architecture, 1981.
- [DaW87] Darlington, J. and While, R. L., *Controlling the Behaviour of Functional Language Systems*, Proceedings of the International Conference on Functional Programming and Computer Architecture, Portland, Oregon, 1987.
- [DFP86] Darlington, J., Field A. J., and Pull, H.M., *The Unification of Functional and Logic Languages*, Logic Programming: Functions, Relations and Equations, Degroot, D. and Lindstrom, G. (eds), Prentice-Hall, 1986.
- [DP87] Darlington, J. and Pull, H.M., *A Program Development Methodology Based on a Unified Approach to Execution*, Partial Evaluation and Mixed Computation, Proceedings of the IFIP Workshop on Partial Evaluation and Mixed Computation, October 1987, eds. Bjorner, D., Ershov, A. P. and Jones, N. D., North Holland.
- [DGN86] Della Fera, T., Gettys, J. and Newman, R., *Xlib - C Language X Interface Protocol Version 10*, Digital Equipment Corporation, MIT/Project Athena, January 1986.
- [FW87] Field, A. J., Hunt L.S. and While, R. L., *Best-fit Pattern-matching*, Internal Report, Functional Programming Section, Department of Computing, Imperial College, London, December 1987.
- [GKS87] Glauert, J. R. W., Kennaway, J. R., Sleep, M. R., Holt, N. P., Reeve, M. J., and Watson, I., *Specification of Core DACTL1*, Internal Report SYS-C87-09, University of East Anglia, U.K., 1987.
- [Gre84] Greenblatt, R. D., *The LISP Machine*, in Interactive Programming Environments, Barstow, D. R., Shrobe, H. E. and Sandewall, E. (eds), McGraw-Hill, 1984.
- [Guo88] Guo, Y. E., *An Execution Mechanism for a Functional Language with Unification*, BCS FACS Workshop on Term Rewriting, Bristol, September, 1988.
- [HaK86] Harrison, P. G. and Khoshnevisan, H., *Efficient Compilation of Linear Recursive Functions into Object-level Loops*, Proceedings 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, June 1986.
- [HaK88] Harrison, P. G. and Khoshnevisan, H., *On the Synthesis of Function Inverses*, Internal Report, Functional Programming Section, Department of Computing, Imperial College, London, April 1988.
- [HaK89] Harrison, P. G. and Khoshnevisan, H., *The Mechanical Transformation of Data Types*, Comp. J., 1989, to appear.
- [HP86] Hewlett-Packard, *Programming with the X-window System*, Internal report, November 1986.
- [GMW77] Gordon, M., Milner, R. and Wadsworth, C., *Edinburgh LCF*, Report CSR-11-77, Computer Science Department, Edinburgh University, 1977.
- [KeS81] Keller, R. M. and Sleep, M. R., *Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Applicative Languages*, ACM Conference on functional languages and computer architecture, Portsmouth, 1981, pp. 131-140.
- [Kho87] Khoshnevisan, H., *Automatic Transformation Systems Based on Functional-Level Reasoning*, PhD thesis, Functional Programming Section, Department of Computing, Imperial College, London, 1987.
- [KhS89] Khoshnevisan, H. and Sephton, K. M., *InvX: An Automatic Function Inverter*, Conference on Rewriting Techniques and Applications, Chapel Hill, North Carolina, April 1989.
- [MaP81] Manna, Z. and Pnueli, A., *Verification of Concurrent Programs: the Temporal Framework*, Computer Science Department, Stanford University, U.S.A, 1981.
- [Mic68] Michie, D. "Memo" Functions and Machine Learning, Nature, No. 218, 1968, pp. 19-22.
- [Per87] Perry, N., *Hope+C A Continuation Extension for Hope+*, IC/FPR/LANG/2.5.1/21, Internal Report, Functional Programming Section, Department of Computing, Imperial College, London, 1987.
- [Per88] Perry, N., *Functional Language I10*, IC/FPR/LANG/2.5.1/29, Internal Report, Functional Programming Section, Department of Computing, Imperial College, London, 1987.
- [PeS87] Perry, N. and Sephton, K. M., *Hope+ Compiler*, IC/FPR/LANG/2.5.1/14, Internal Report, Functional Programming Section, Department of Computing, Imperial College, London, 1987.
- [StW74] Strachey, C. and Wadsworth, C. P., *Continuations - A Mathematical Semantics for Handling Full Jumps*, PRG-11, Programming Research Group, University of Oxford, 1974.
- [Tow87] Townsend, P., *Flagship Hardware and Implementation*, ICL Technical Journal 5(3), May 1987, pp. 575-594.
- [WaW86] Watson, I. and Watson, P., *Graph Reduction in a Parallel Virtual Memory Environment*, Proceedings of the MCC Graph Reduction Workshop, Santa Fe, New Mexico, Springer-Verlag, 1986.
- [Whi88] While, R. L., *Behavioural Aspects of Term Rewriting Systems*, PhD thesis, Functional Programming Section, Department of Computing, Imperial College, London, 1988.

Report on the Demonstrations at the Second Joint ICOT/DTI-SERC Workshop

Tsutomu Yoshioka

Abstract

This report summarizes the demonstration session held on the last day (October 17, 1990) of the Second Joint ICOT/DTI-SERC Workshop.

1 Participants

Approx. 30 people attended.

2 Demonstrations

Though the demonstration session took rather long, the audience watched it closely and there were active discussions.

The summary of the demonstrations is as follows:

2.1 ICOT side demonstrations

ICOT showed six parallel programs on the parallel inference machine prototype, Multi-PSI, and two sequential programs on the PSI-II sequential inference machine. For details, please refer to 'Guide to the Japanese Demonstrations'.

- Pentomino — Packing Piece Puzzle Solver

This program solves a Packing Piece Puzzle, consisting of a rectangular box and a collection of pieces with various shapes. This is one of the parallel benchmarking programs running on Multi-PSI.

- Bestpath — Shortest Path Problem Solver

This program finds the shortest paths from a given starting vertex to all other vertices in a graph. This is one of the parallel benchmarking programs running on Multi-PSI.

- Go-playing program : GOG
This program is a parallel version of the Go-playing program having been developed on PSI-II.
- LSI CAD (Routing)
This program executes routing between modules on a LSI chip on Multi-PSI, after the placement of each modules has been fixed. It determines the connection paths between terminals of each module.
- LSI CAD (Logic Simulation)
This program simulates the behavior of logic circuits described at the logic-gate level on Multi-PSI, taking delay time of each gate into account, and evaluates the virtual time mechanism, one of parallel control mechanisms for discrete event simulations.
- Legal Reasoning program
This program deals with legal decisions by reasoning from precedents, adapting old solutions to solve new problems. We evidenced that legal reasoning can be modeled as case-based reasoning.
- Genome Analysis Programs
These programs solves multiple sequence alignment problems in genome analysis, by 3-dimensional DP-matching and by scheduleless parallel simulated annealing.
- Constraint logic programming language, CAL
This program is a constraint logic programming language system with examples about controlling robots and logic circuits, running on PSI-II.
- Molecular Biological Database in Kappa
This program is a database system with molecular biological data, adapted to a database management system, Kappa.

2.2 UK side demonstration

Prof. Darlington (Imperial College), "Functional Programming Environment"

Prof. Darlington brought the demonstration programs in a cartridge magnetic tape (CMT), and installed them on a Sun-3 at ICOT.

He showed a programming environment of the functional language Hope+.

Hope+ is an extension of the functional language Hope, designed at Edinburgh University, by adding real and vector as data type, a lazy data constructor, optimal pattern matching, etc.

The user interface is built on the X-window, and provides various facilities, including editing of functional definition files, compilation, program transformation, etc.

In the demonstration, a few instances of program transformation were shown.

Though program transformation itself is recognized to be useful, it is still very difficult to transform programs fully automatically.

On the other hand, manual transformation is very cumbersome and prone to errors.

One of the features of this system is that the user directs the system as to what transformation is to be done and the system carries out the actual transformation.

In the first example, a function 'front', which returns the header list of an input list was given. It was defined in a generate-and-test manner in a language with unification as in logic programming. Prof. Darlington had the system transform the function into a recursive definition in Hope+.

With the former definition, it took about 10 seconds to return a list of 3 atoms from a list of 5 atoms. The transformed version returned the result in a flash.

Next, he showed a transformation of a 'grep' program as one of bigger size examples.

One of the advantages of functional languages (or, declarative languages in general, including logic programming languages) is that the evaluation order of functions need not be overspecified. This makes those languages amenable to program transformation and parallel execution.

On the other hand, in application programs interacting with external world, the relative order of evaluations of some specific important events must satisfy some constraints.

In the programming environment of Hope+, the user can give a logical definition of a function in Hope+ and constraints on timing in a temporal logic meta language called SIAN, and the system can transform them into a program which is almost declarative but is capable of describing the order of events.

In the final example, Prof. Darlington showed how the system generated a program to do flow control (such as handling of Control-S/Control-Q) by transformation.