

TM-0891

Parallel Inference Machines and
Programming in the FGCS Project

by
K. Taki

July, 1990

©1990, ICOT

ICOT

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

(03) 456-3191-5
Telex ICOT J32964

Institute for New Generation Computer Technology

Contents

Part 1.

Machine Architecture and System Overview	----- 2
--	---------

Part 2.

KL1 Programming, Examples and Performance Measurements	----- 39
--	----------

Appendix A.

Parallel Software Development System and Application Programs	----- A-0
---	-----------

Appendix B.

General Information for Parallel Inference Systems Research in FGCS Project –The FGCS Computing Architecture–	----- B-0
---	-----------

Part 1.

Part 1.

Machine Architecture and System Overview

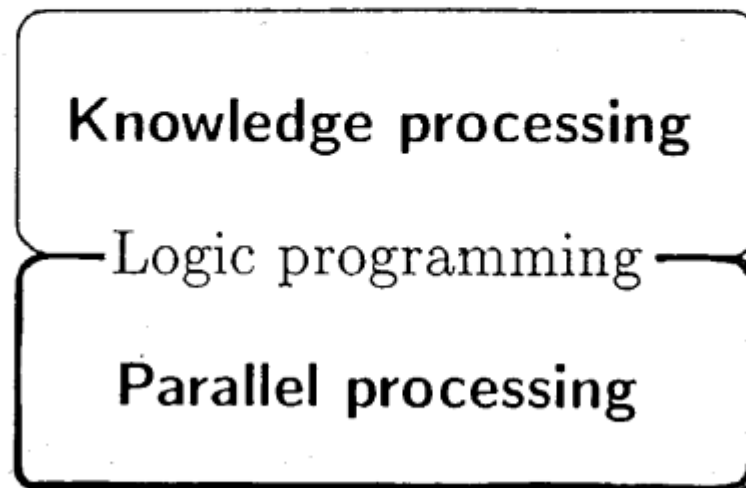
<input type="checkbox"/> Outline of the Project	----- 4
<input type="checkbox"/> Multi-PSI/V2	----- 11
<input type="checkbox"/> PIM/p	----- 17
<input type="checkbox"/> KL1 Language	----- 22
<input type="checkbox"/> PIMOS	----- 32

☐ **Outline of the FGCS Project**

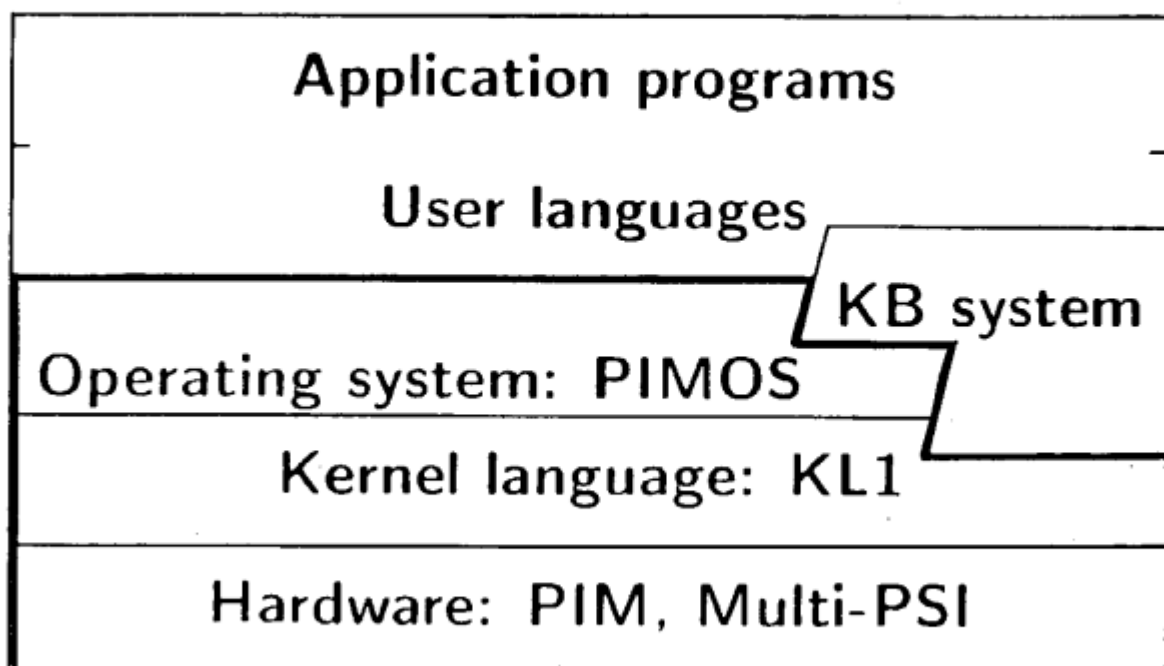
The FGCS Project of Japan

- R & D of technology bases for
 “knowledge information processing systems” in 1990s
- Project spans **1982 to 1992.**

General Framework of R & D



Parallel Inference System



General Research Plan

- R & D of parallel software and parallel hardware must proceed concurrently.
- Stepwise bootstrapping

Parallel software \iff Parallel hardware
Chicken and egg

Egg	-Multi-PSI
Chicken	-PIMOS/V1
Egg	-PIM/p
Chicken	-PIMOS/V2
Egg	-Final PIM
Chicken	-Final PIMOS

- Cultivation of parallel computing

Sequential systems

1984	PSI-I, KL0	37K LIPS(KL0)
	—ESP, SIMPOS	
1986	PSI-II	330K LIPS(KL0) (KL0 append)

Parallel systems

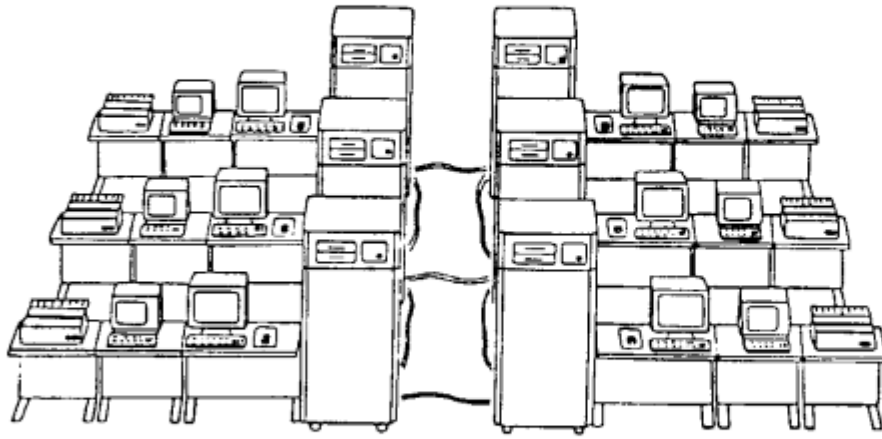
1985	—GHC	
1986	Multi-PSI/V1, FGHC	1K LIPS×6PE(FGHC)
	—Small sample programs	
1988	Multi-PSI/V2, KL1	150K LIPS×64PE(KL1) (KL1 append)
	—PIMOS/V1, <u>Demonstration programs</u>	
Now → 1990	PIM/p	600K LIPS×128PE(KL1)
	—PIMOS/V2, Application programs	
1992	Final PIM system	
	—Final PIMOS, Large application programs	

(LIPS : Logical Inferences per Second
KL0, KL1, : Kernel languages
GHC, FGHC
SIMPOS, PIMOS : Operating Systems
8

Target Domain of Parallel Processing

- **Hardware: Large scale (scalable), and loosely coupled MIMD machines**
 - Powerful processor and large memory pair
 - not SIMD
 - local memory – non global
- **Logic language based**
 - less logic-based than Prolog
- **Domain of application:**
 - Knowledge processing
 - General purpose parallel processing
 - non uniform data/processing
- **Maximum n times speed up using n processors**

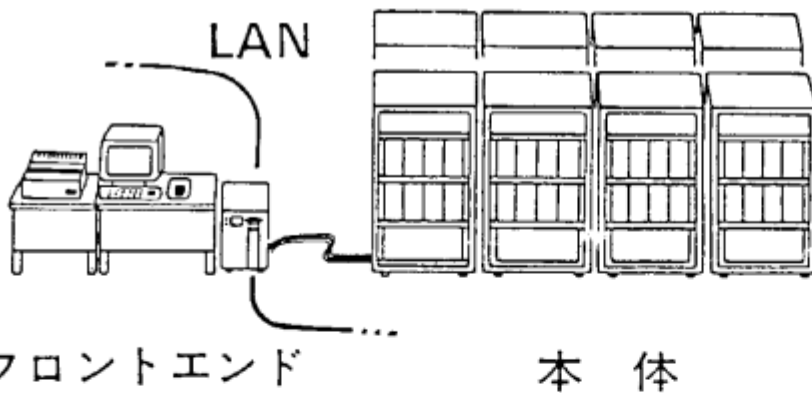
Multi-PSI/v1



6 PE_s
~3KLIPS



Multi-PSI/v2



64 PE_s
2~5MLIPS

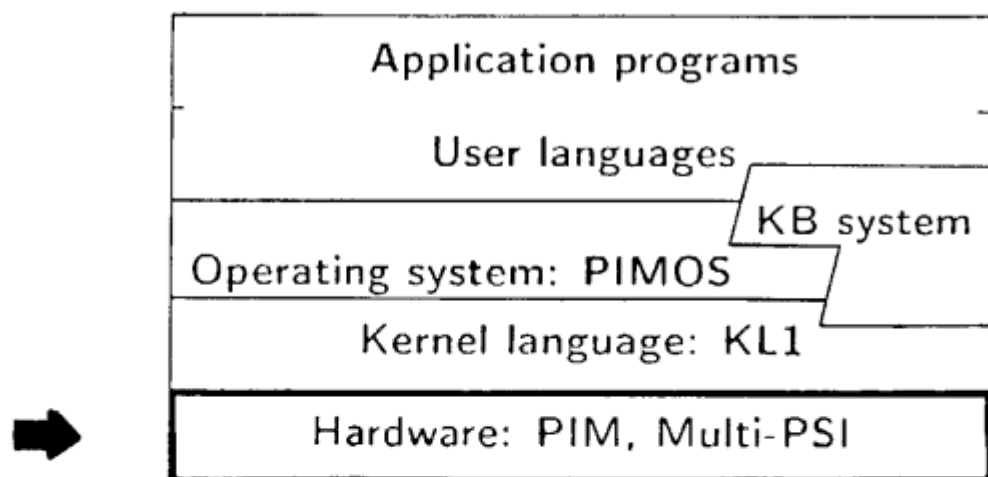


PIM/p

1/2 size, 128 PE_s,
10~20 MLIPS

□ Multi-PSI/v2

- Early prototype of PIM
- R&D tool for parallel software



Multi-PSI/v2

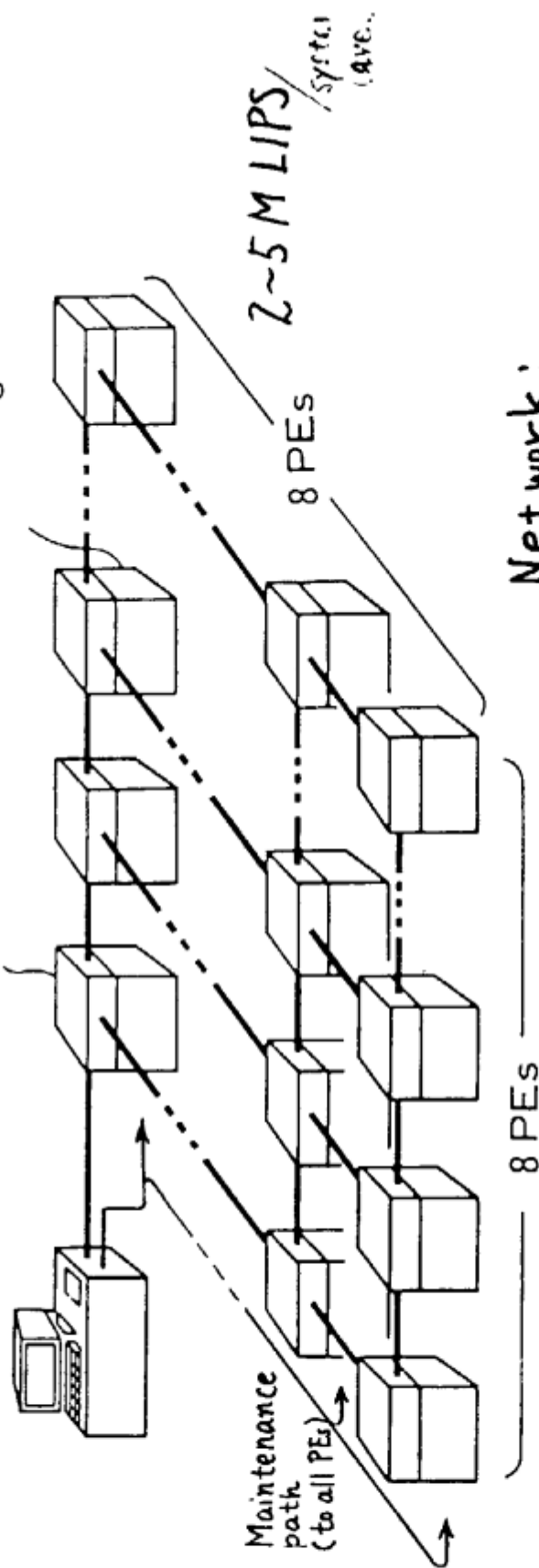
フロントエンド・プロセサ
Front-end processor
(PSI-II)

ネットワーク制御回路

Network controller

要素プロセサ

Processing element (PE)



Network :

64 PEs max (PSI-II CPU each)

2-dimensional mesh

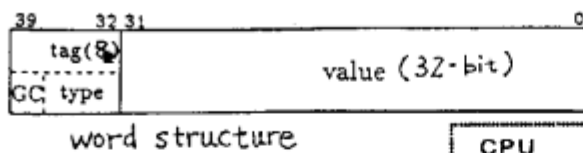
Machine language : KL1-b (written in microprogram)

message exchange

Memory : 16 Mw/PE

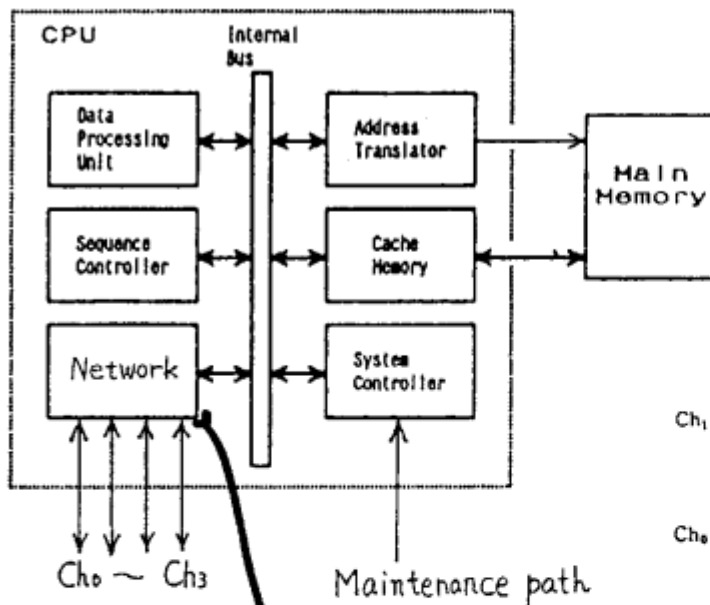
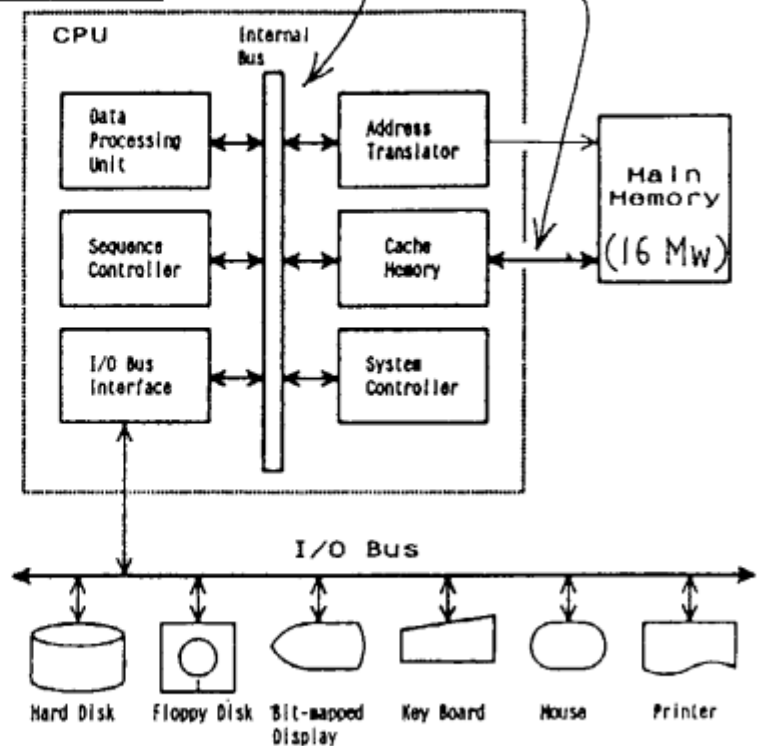
routing function

5MB/s \times 2 directions



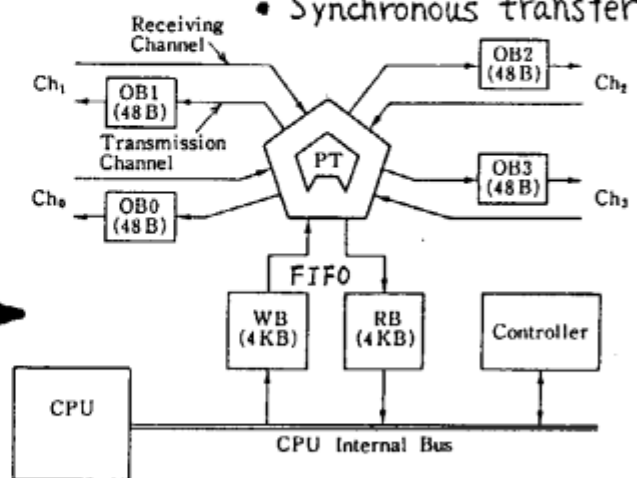
- Microprogram control
- 200nS cycle time
- 4 Kw cache memory
- 53-bit x 16 Kw WCS

PSI-II ➡

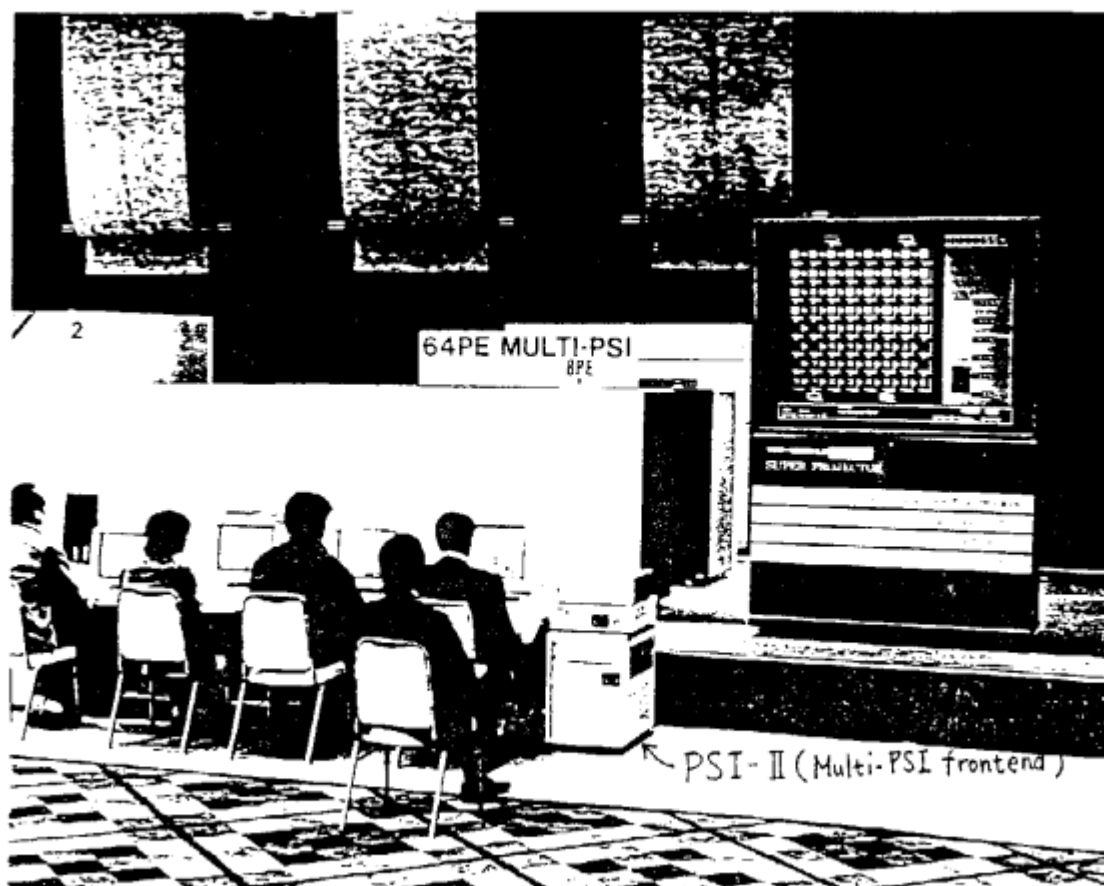


← PE of Multi-PSI/v2
(= PSI-II CPU)

- "Cut through" routing
- Synchronous transfer



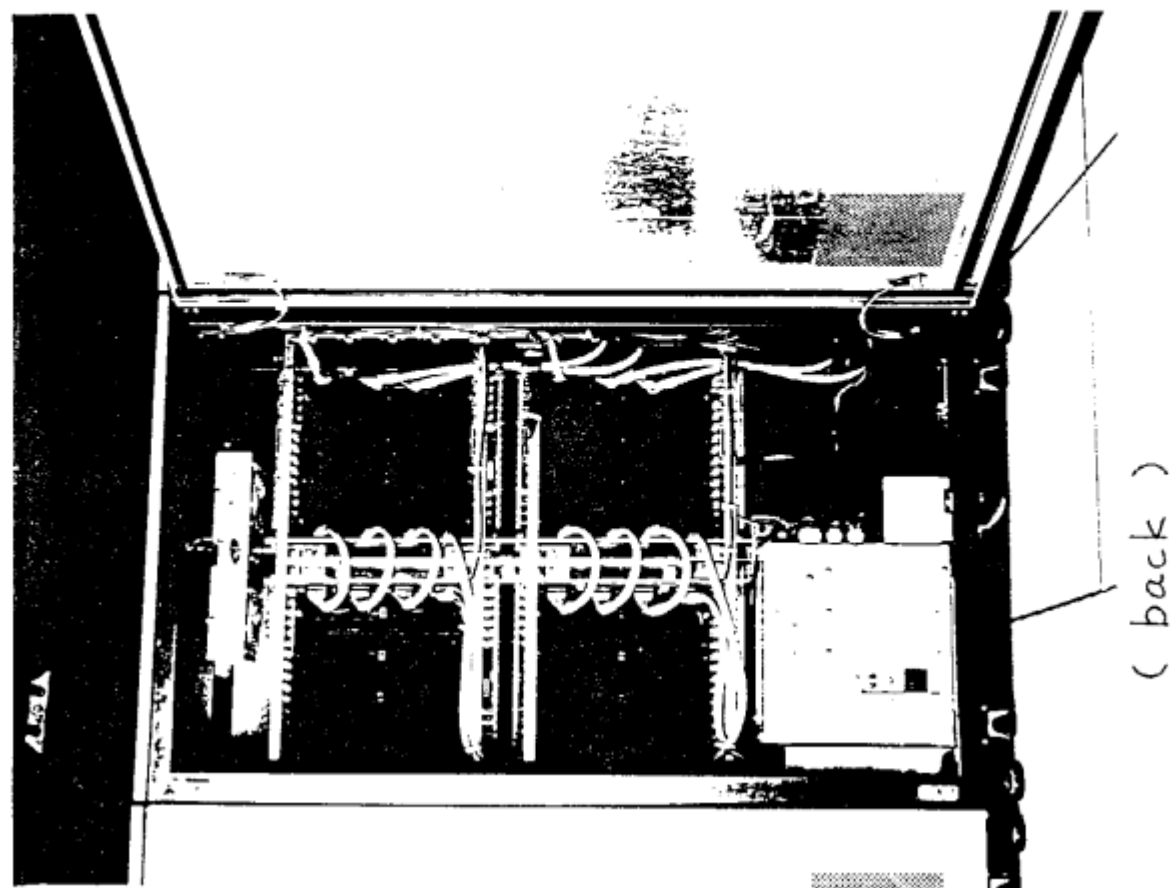
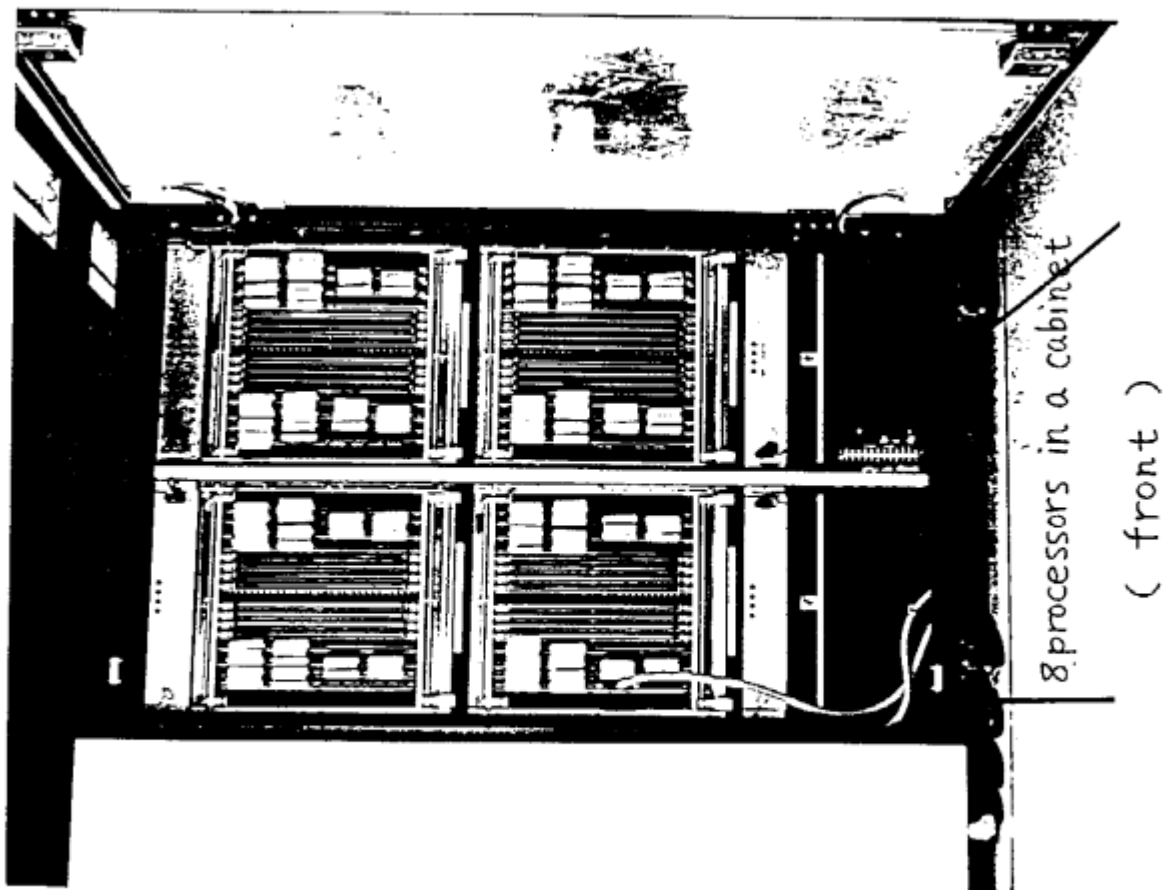
Network of Multi-PSI/v2 (one node)



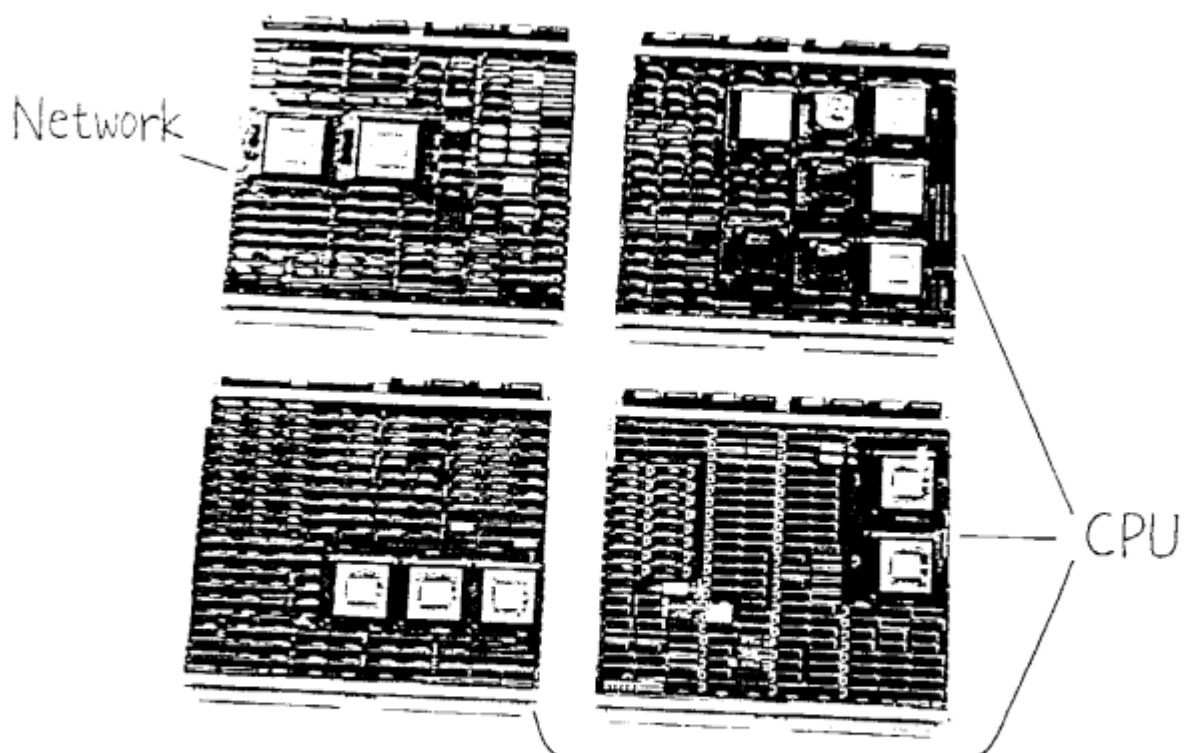
Multi-PSI/v2 at a demo site (FGCS'88)



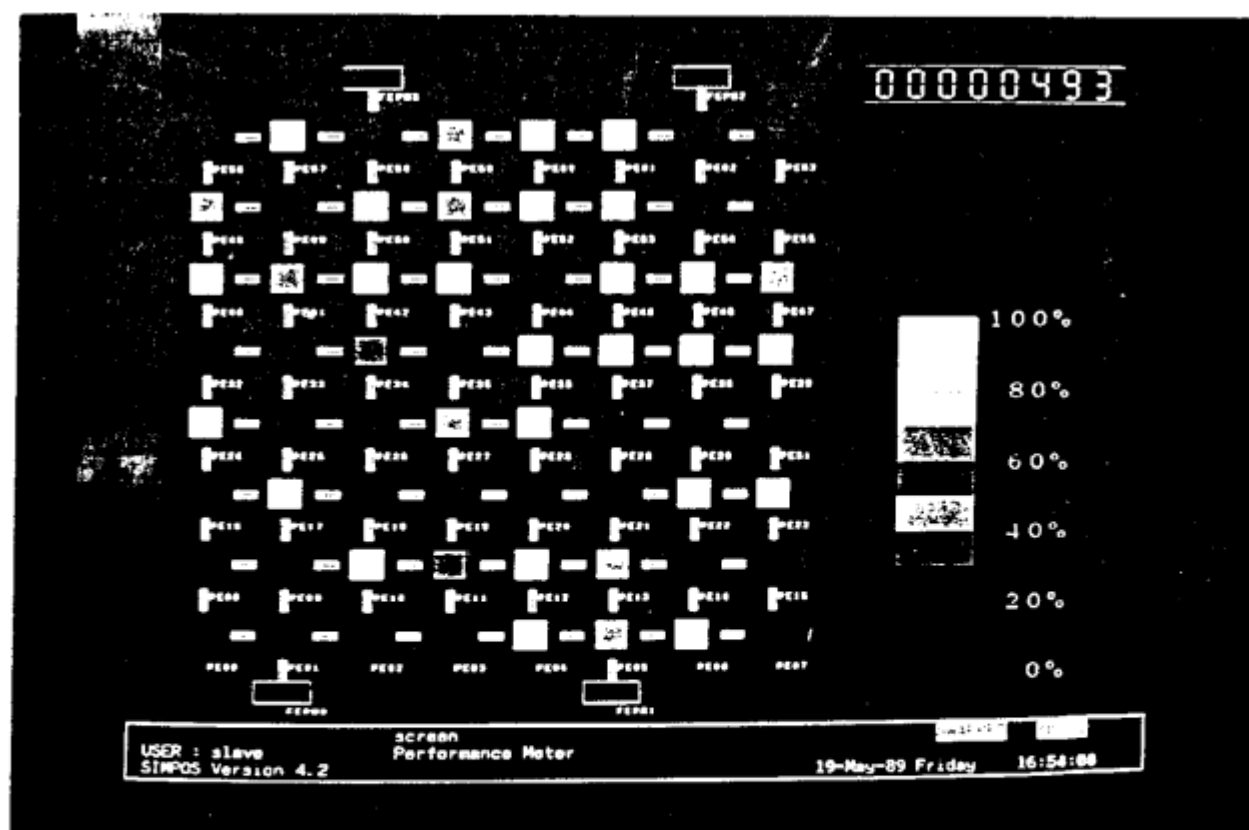
Multi-PSI/v2 system



A cabinet of Multi-PSI/v2



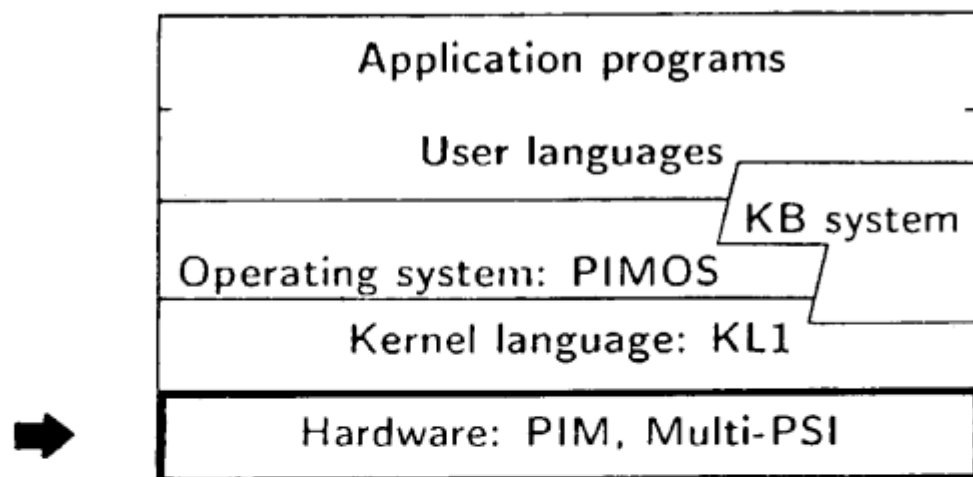
Printed circuit boards of Multi-PSI/v2 CPU



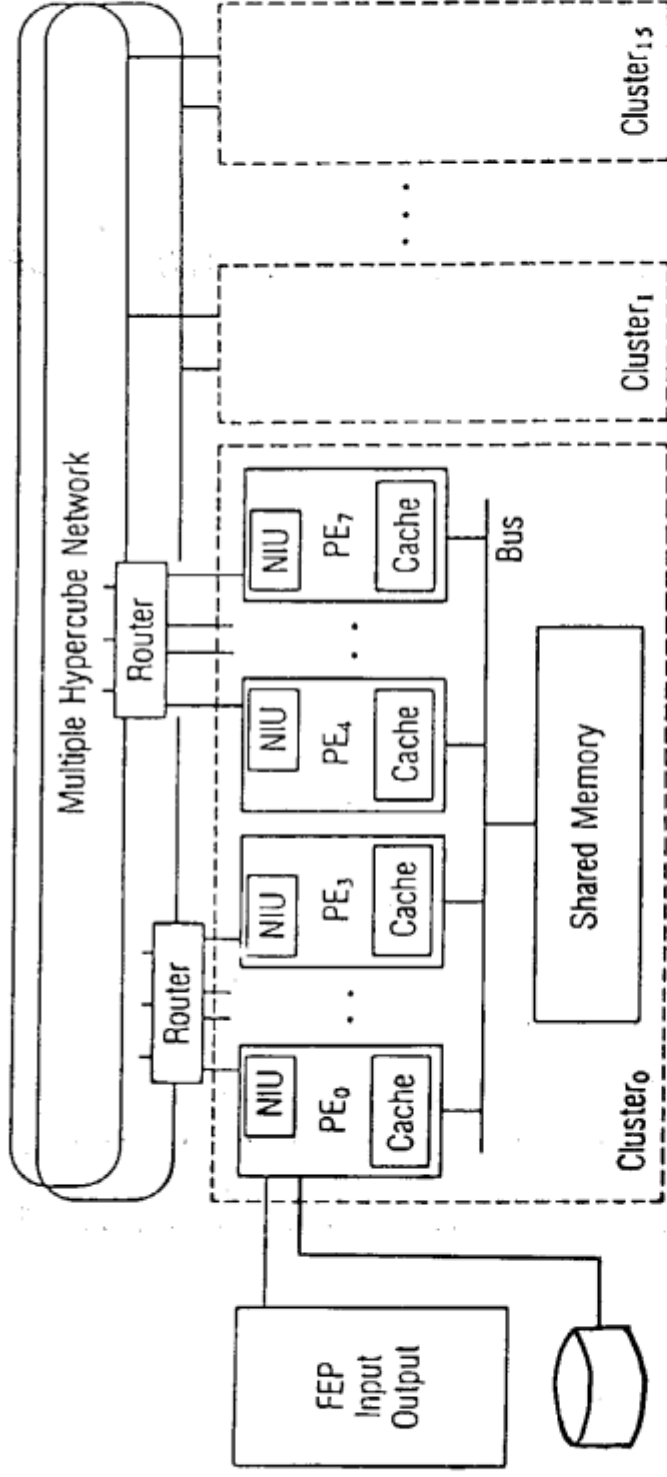
Display of "Performance Meter"

□ PIM/p

— Parallel Inference Machine / model p



PIM/p



Layered structure

8 PEs / cluster

16 clusters / system

(128 PEs total)

Machine language: RISC+KL1-b
macro call

Memory: 32 Mw / cluster

Network: double hyper-cube

message exchange

routing function

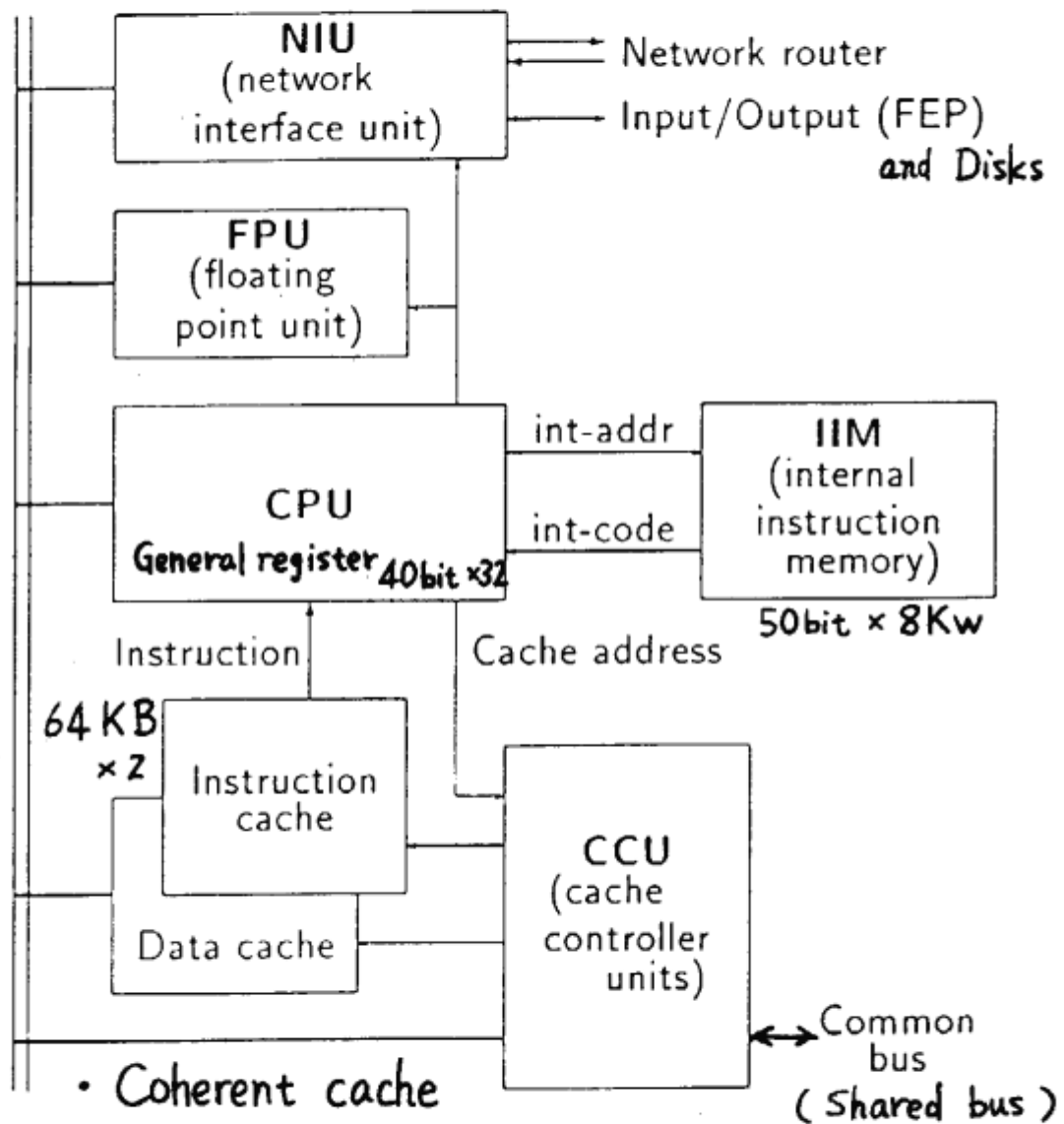
20MB/s \times 2 directions \times 2

VLSI processors

4-stage pipeline

Processor Element Configuration

64-bit data path



Processing Element Design Features

1. Efficient Implementation of KL1-B
(Abstract Instruction Set for KL1)
 - ⇒ Instruction set amalgamating both RISC and CISC features
 - ⇒ Efficient data type checks by tag architecture
 - ⇒ Conditional macro-call instructions for high-level functions.
 2. Efficient implementation of MRB garbage collection
 - ⇒ Special instructions for MRB and dereferencing
- Hopefully 1 instruction / cycle (50nsec)
by 4 stage pipeline
 - *Append* performance : about 600 Krps
(Including GC overhead)

Processor Connection Design Features

1. Hierarchical Structure

- ⇒ Cluster: Eight processing elements connected with shared memory and bus.
- ⇒ Fast network to connect 16 (\leq) clusters.

2. Efficient communication within a cluster.

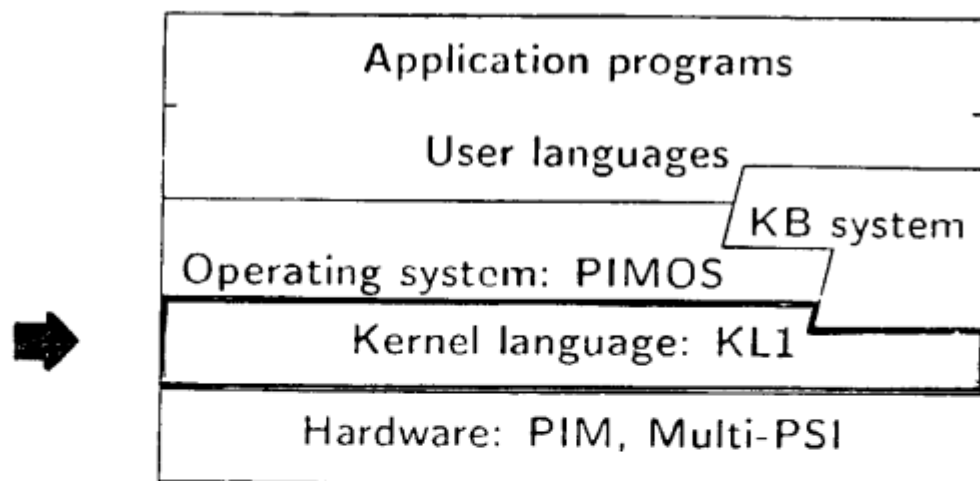
- Coherent cache design using the characteristics of KL1

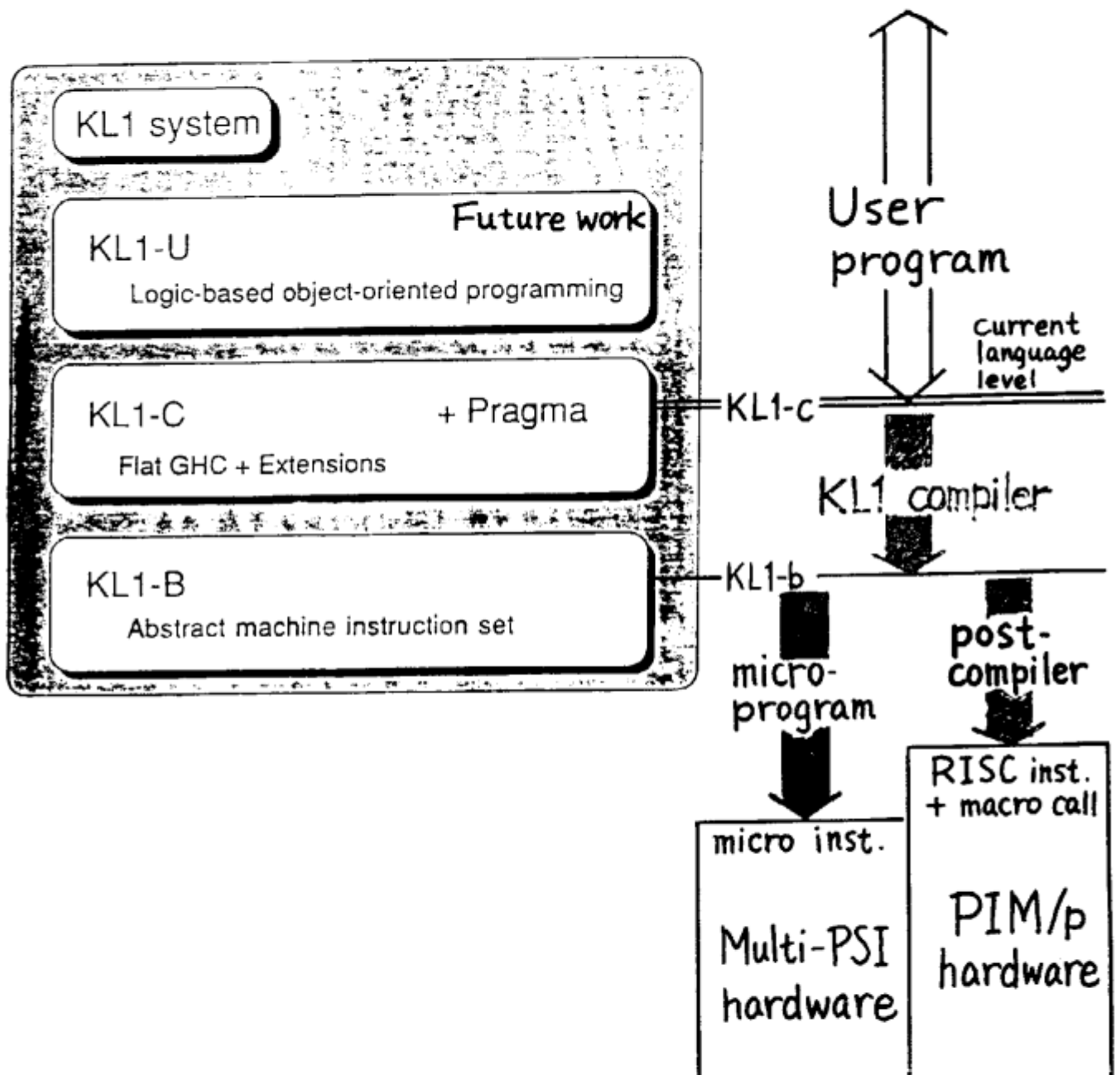
3. High performance inter-cluster network

- ⇒ Network port in each processing element
- ⇒ Hyper-cube network (Doubled)
 - Cluster throughput : 40 M Bytes/sec
- ⇒ I/O device can be connected with most processing elements.

□ KL1 Language

- Kernel language for Multi-PSI and PIM
- Concurrent logic language





Kernel Language: KL1- c

Flat GHC + Meta-control features
+ Features for efficiency

- **Sho-en features:**

- Starting/stopping/aborting execution
- Exception handling
- Resource consumption control

- **Higher order extensions**

- **“Updatable” arrays**

- **Optimized merger**

- **Pragma:**

- Priority control *goal @ priority (X)*
- Load allocation *goal @ processor (Y)*

A Small Program in KL1

`qsort(Xs,Ys):-true|qsort(Xs,Ys,[]).`

`qsort([], Ys0,Ys1) :- true | Ys0 = Ys1.`

`qsort([X|Xs],Ys0,Ys3) :- true | part(Xs,X,S,L),
 qsort(S,Ys0,[X|Ys2]),
 qsort(L,Ys2,Ys3)@processor(n).`

pragma for load distribution

`part([X|Xs],A,S,L0) :- A < X |
 L0=[X|L1], part(Xs,A,S,L1).`

`part([X|Xs],A,S0,L) :- A >= X |
 S0=[X|S1], part(Xs,A,S1,L).`

`part([], X,S, L) :- true` | `S = [], L = [].`

guard
 |
 pattern match and
 condition test
 (synchronization)

↑
 commitment
 operator
 |
 selecting
 a clause

body
 |
 goal expansion
 (parallel execution)

A Small Program in KLI

Meta
Level

Report

control

..., control(Rep,Ctl), execute(qsort(X,Y),Rep,Ctl),.

Task (Shoen)

meta-call

qsort(Xs,Ys):-true|qsort(Xs,Ys,[]).

Object Level

qsort([], Ys0,Ys1) :- true | Ys0 = Ys1.

qsort([X|Xs],Ys0,Ys3) :- true | part(Xs,X,S,L),

qsort(S,Ys0,[X|Ys2]),

. qsort(L,Ys2,Ys3)@processor(n).

pragma for load distributio

part([X|Xs],A,S,L0) :- A < X |

L0=[X|L1], part(Xs,A,S,L1).

part([X|Xs],A,S0,L) :- A >= X |

S0=[X|S1], part(Xs,A,S1,L).

part([], X,S, L) :- true | S = [], L = [].

KL1 - c

- **Born parallel**
 - Fine grain execution model
 - Medium or large grain on real execution
- **Implicit data-flow synchronization**
- **Independency of load mapping specification and logical program structure**
 - *Pragma*
- **General purpose concurrent programming language**

KL1 implementation issues (for PIM/p and Multi-PSI/V2)

1. Intra-cluster/processor:

KL1-b Abstract machine instruction set

Tagged word 8+32 bits

MRB Incremental GC

Copying GC

2. Inter-cluster/processor:

Message passing

Independent address space

WEC Incremental GC

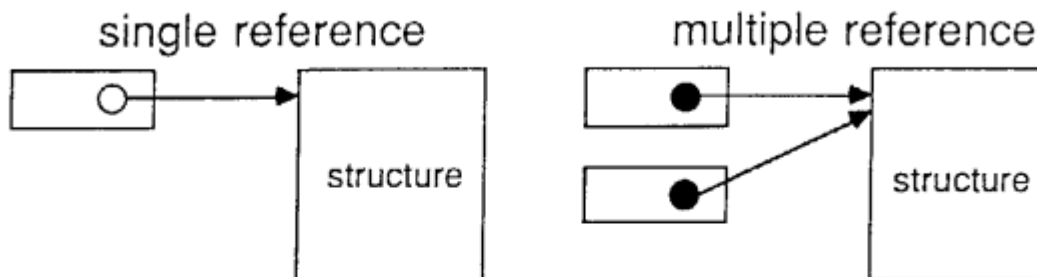
Distributed goal management

Distributed unification

MRB Multiple Reference Bit

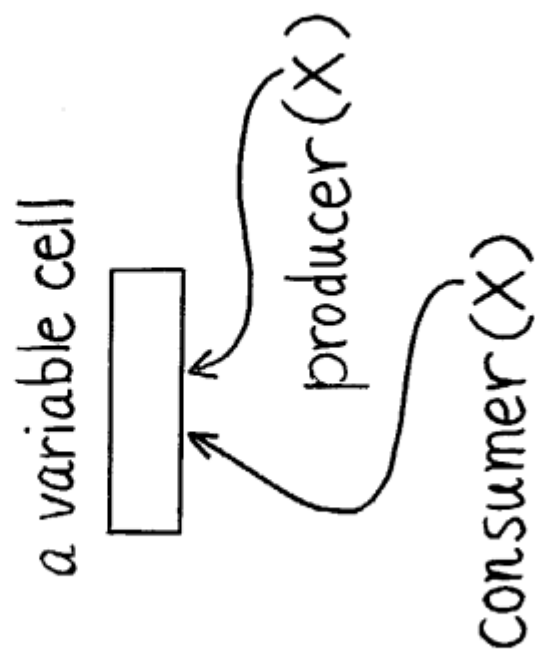
1 bit flag in each pointer of structure or variables to represent multiple reference information.

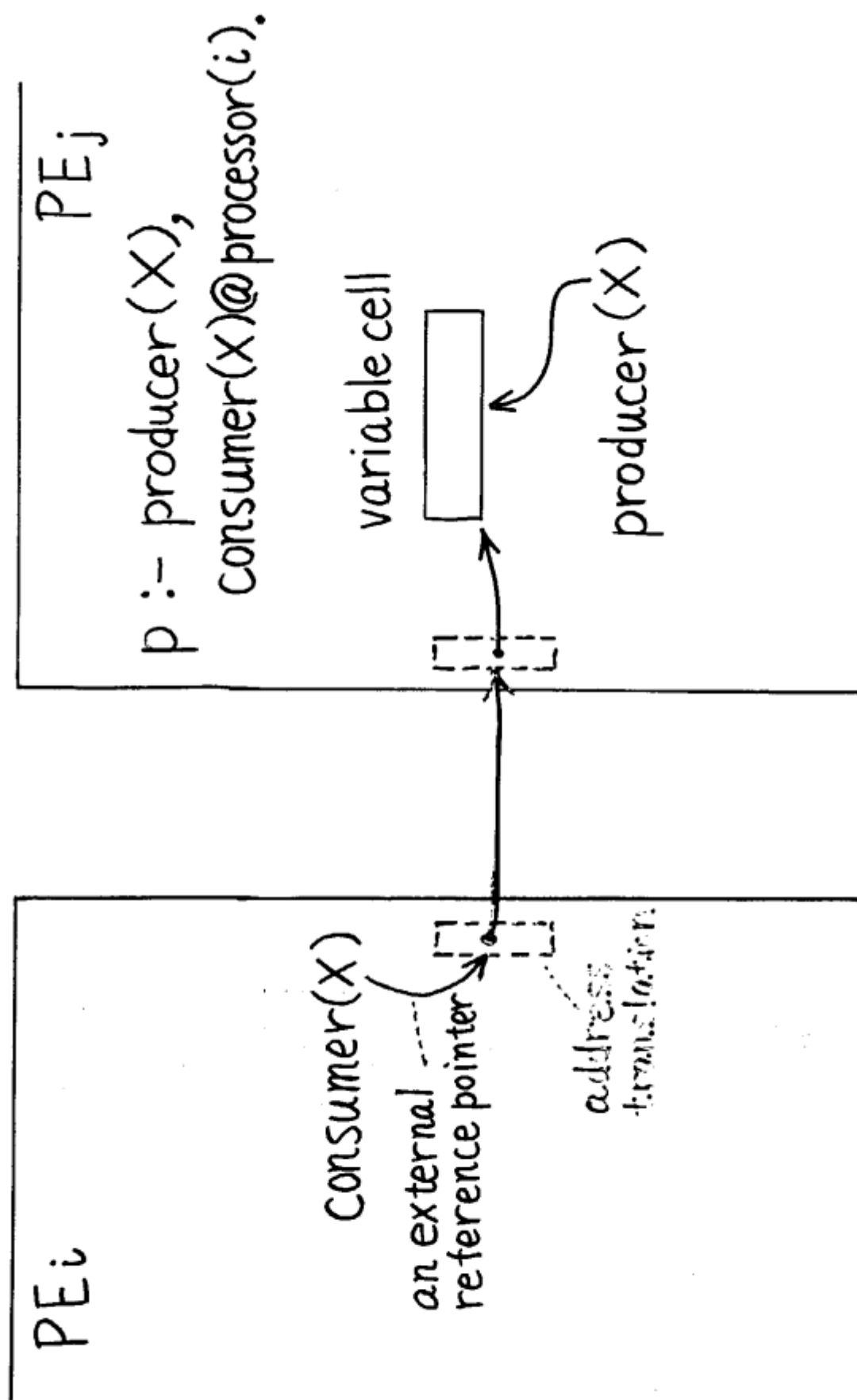
on-the-fly garbage collection
destructive updation of array elements
managemet of out-going and in-coming pointers
in loosely-coupled multi-processors
(Multi-PSI, PIM inter-cluster)



PEj

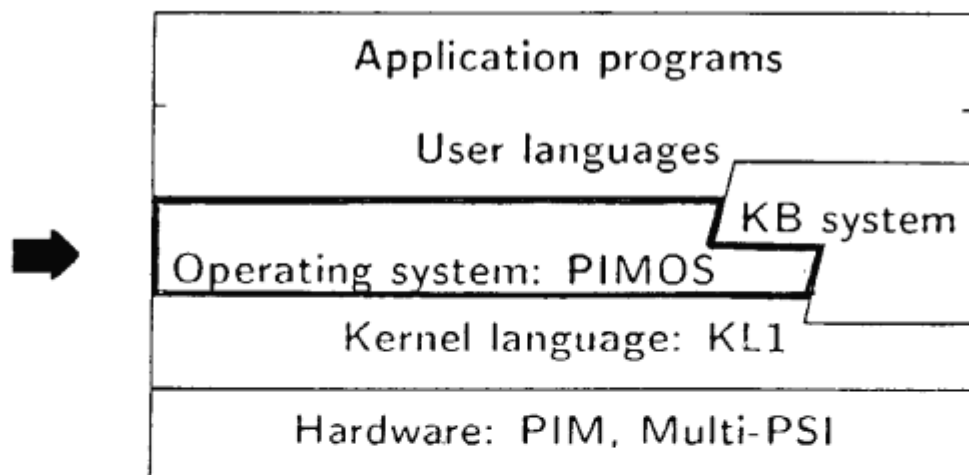
p :- producer(X),
consumer(X).





□ PIMOS (PIM Operating System)

- Parallel operating system for Multi-PSI and PIM



PIM operating system: PIMOS

Design Goals

1. A practical OS for large scale S/W experiments.
2. A stand-alone self-contained OS written in KL1.
3. A single OS showing a parallel machine as one system.
4. Independent from architectural details.

The biggest KL1 program — 100 K lines in KL1 source code

Operating System: Problems of conventional schemes

Optimized for procedural and sequential processing

- Basic notions suited to proc. and seq. processing
Large-grain processes as the unit of management
 - Processes have large inertiaControlling dependency by execution order
 - Limitation on parallelism

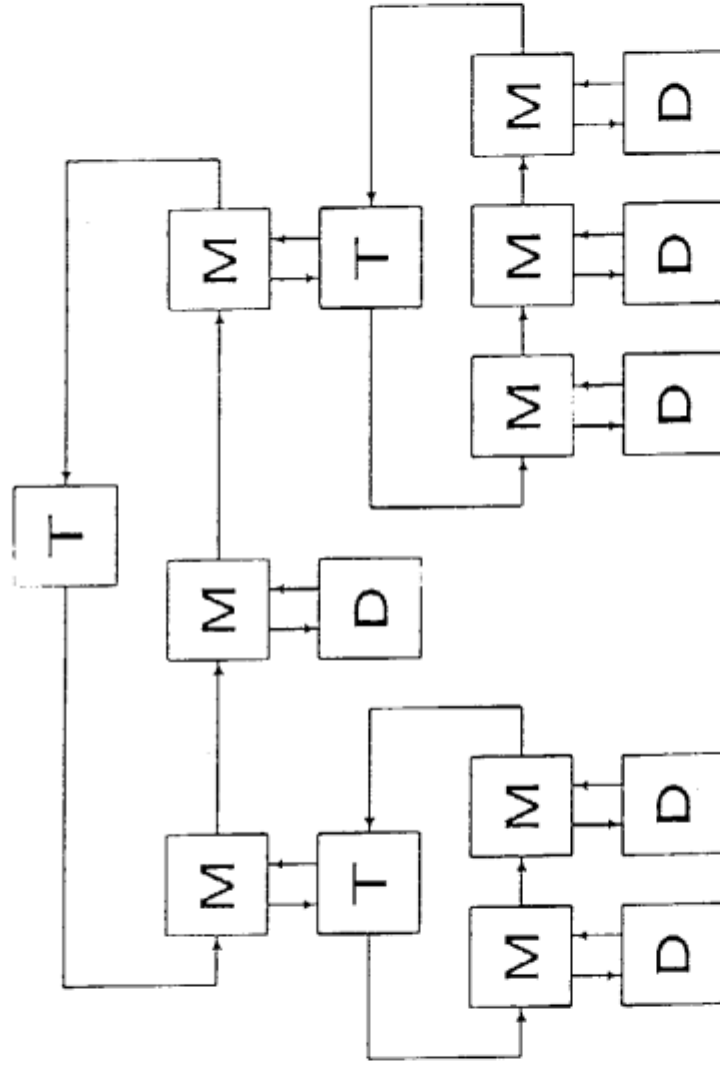
- Operating system itself optimized for seq. processing
Centralized management
 - More communication → OS can be the bottleneck

Operating System: Redesigned for Parallel Processing

PIMOS employs a scheme optimized for parallel execution

- Basic notions suited to parallel execution
 - Group of fine-grained processes as the unit of management
 - Shoen, Task
 - 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 Management of Resource



T: Task management process

M: Resource monitoring process

D: device handler process

Operating System: Software Productivity

- **Synchronization errors:** Source of irreproducible anomalies
 - Much less frequent thanks to the pure concurrent language
- **Dead locks:**
 - Automatic deadlock detection mechanism
(thanks to data-flow nature of the language)
 - Detection (in part) by static analysis (under development)
- **Difficulty in tracing parallel processes:**
 - Selective tracing mechanism based on control flow
 - Selective tracing mechanism based on data flow (planned)
- **Difficulty in performance debugging:**
 - Visualization of program behavior (under development)

Development support

- **PDSS:** PIMOS development support system
 - KL1 language processor on UNIX machines
 - Pseudo-parallel execution
- **Pseudo Multi-PSI**
 - Multi-PSI simulator on PSI-II
 - Compatible execution environment with Multi-PSI/V2
- **Multi-PSI CSP**
 - Multi-PSI console processor system
 - Debugging facilities for Multi-PSI/V2
- **PSL/VPIM**
 - KL1 development environment for PIM

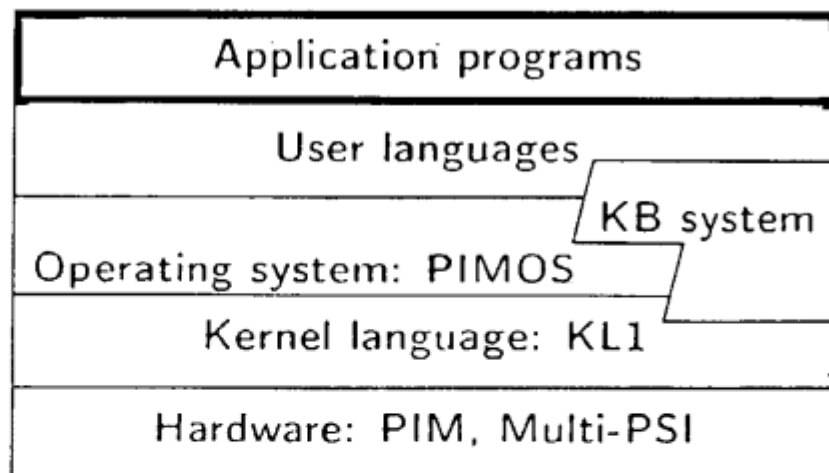
Part 2.

Part 2.

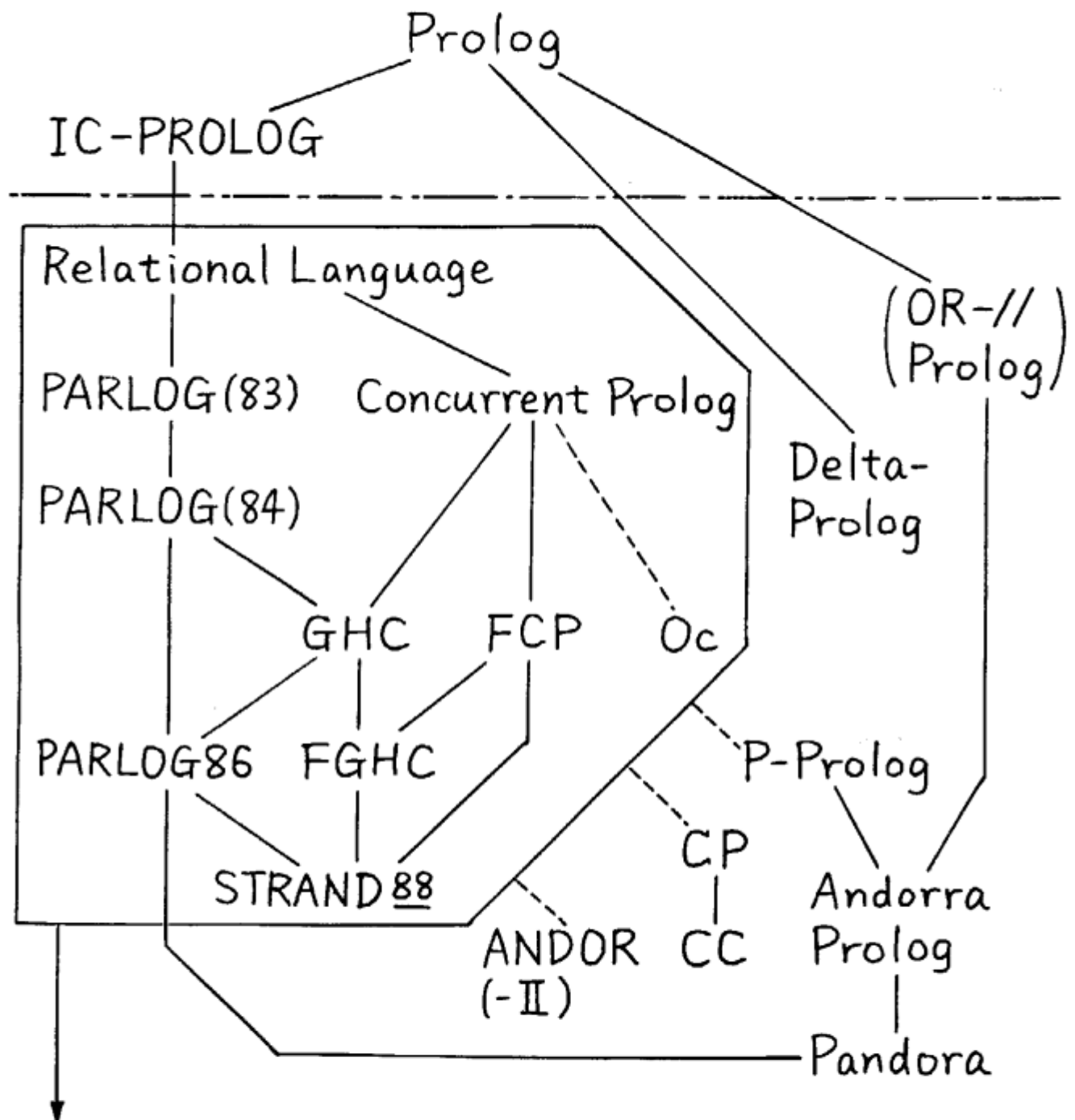
KL1 Programming, Examples and Performance Measurements

- ☐ GHC and KL1 Programming -----41
- ☐ Concurrent Programming and Parallel Processing –Research Themes and an Approach– -----58
- ☐ Concurrent Programming and Parallel Processing –Examples and Performance Measurements– -----63
- ☐ Conclusion and Future Plan -----88

□ GHC and KL1 Programming



Concurrent Logic Languages and GHC



- use guarded clauses
- feature don't-care nondeterminism
- capable of describing concurrency

Overview of GHC

- Syntactically,

$$\begin{array}{ccccc} \boxed{\text{GHC} = \text{Horn clauses} + \text{Guards}} \\ \downarrow & & \downarrow & & \downarrow \\ \text{Algorithm} = & \text{Logic} & + & \text{Control} & \\ & & & \text{[Kowalski]} & \end{array}$$

- Semantically,

$$\begin{array}{l} \boxed{\text{GHC} = \text{Logic Programming} \\ \quad + \text{Partial order on bindings} \\ \quad + \text{Don't-care nondeterminism}} \end{array}$$

cf. Prolog = Logic Programming
+ Total order on
goals and clauses
+ ...

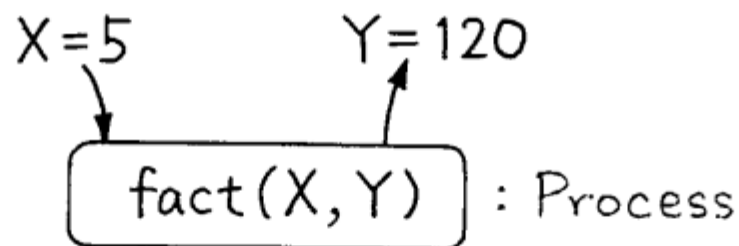
GHC: Syntax and Informal Semantics

Logic Programming and GHC

- The purpose of computation — two aspects:
 - (a) whether $:-G$ can be refuted
 - (b) bindings (substitutions) generated in the course of refutation

Theorem proving: $(a) \sim (b)$

Programming language: $(a) < (b)$ or $(a) \ll (b)$
 $\langle \text{Prolog} \rangle \quad \langle \text{GHC} \rangle$



- GHC introduces partial order on bindings by restricting dataflow caused by unification.
cf. the basic idea of dataflow computation

The Factorial Program

① $\text{fact}(0, Y) :- \text{true} \mid Y=1.$

② $\text{fact}(X, Y) :- X > 0 \mid$
 $X' := X - 1, \text{fact}(X', Y'), Y := Y' * X.$

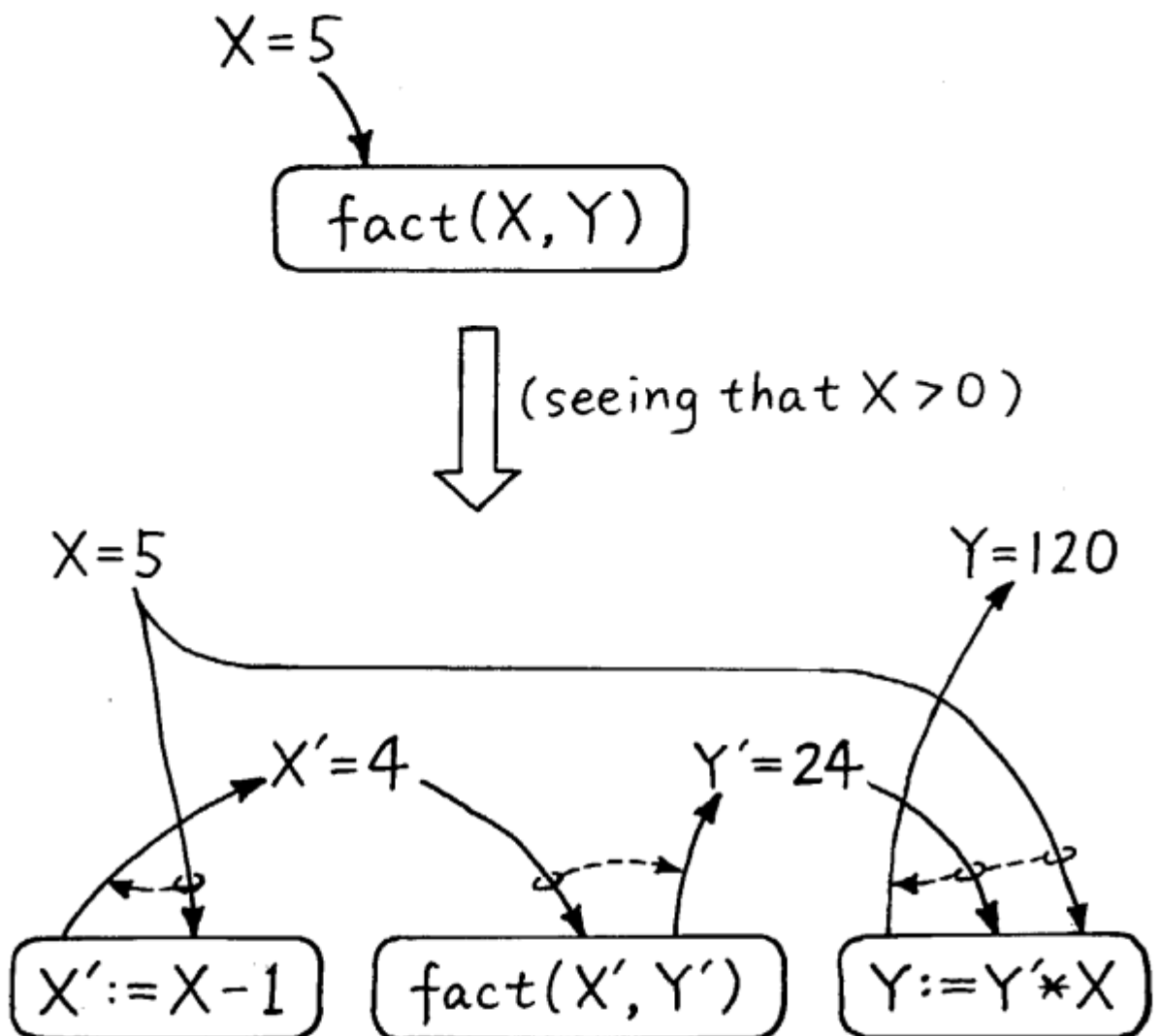
reads:

① } If there is a goal of the form
② } 'fact(,)' and the first argument
is known to be $\begin{cases} 0 \\ \text{positive} \end{cases}$, then ...

★ Goal \doteq Process = Concurrent object

★ Binding = Information (constraint)

Controlling Bindings Replaces Sequencing



- Since each goal has its own direction, we need not serialize goals themselves.

GHC as a Process Description Language

- System of processes \longleftrightarrow Conjunctive body goals (dynamically varying)
- Process \longleftrightarrow Body goal
- Process state \longleftrightarrow (retained by the arguments of subgoals)
- Computation & Communication \longleftrightarrow Observation (guard) and generation (body) of bindings (by unification)
- Synchronization \longleftrightarrow Suspension of unification invoked in guards
- Rewrite rule \longleftrightarrow Program clause of processes

$$\text{filter}(P, \underline{[X|Xs1]}, Ys0) :- \left. \begin{array}{l} X \bmod P \neq 0 \\ Ys0 = [X|Ys1], \\ \text{filter}(P, Xs1, Ys1). \end{array} \right\} \begin{array}{l} \text{reduction} \\ \text{condition} \\ \text{reduced} \\ \text{processes} \end{array}$$

Generating Prime Numbers

Top Level:

```
go(Max) :- true |
    primes(Max,Ps),
    outconv(Ps,Os),
    outstream(Os).
```

Writing the Integer Stream:

```
outconv([X|Xs1], Os0) :- true |
    Os0=[write(X),nl|Os1],
    outconv(Xs1,Os1).
outconv([], Os0) :- true |
    Os0=[].
```

Executing the Program

```
| ?- ghc go(300).
```

2

3

5

...

281

283

293

562 msec.

yes

Prime Sequence Generator

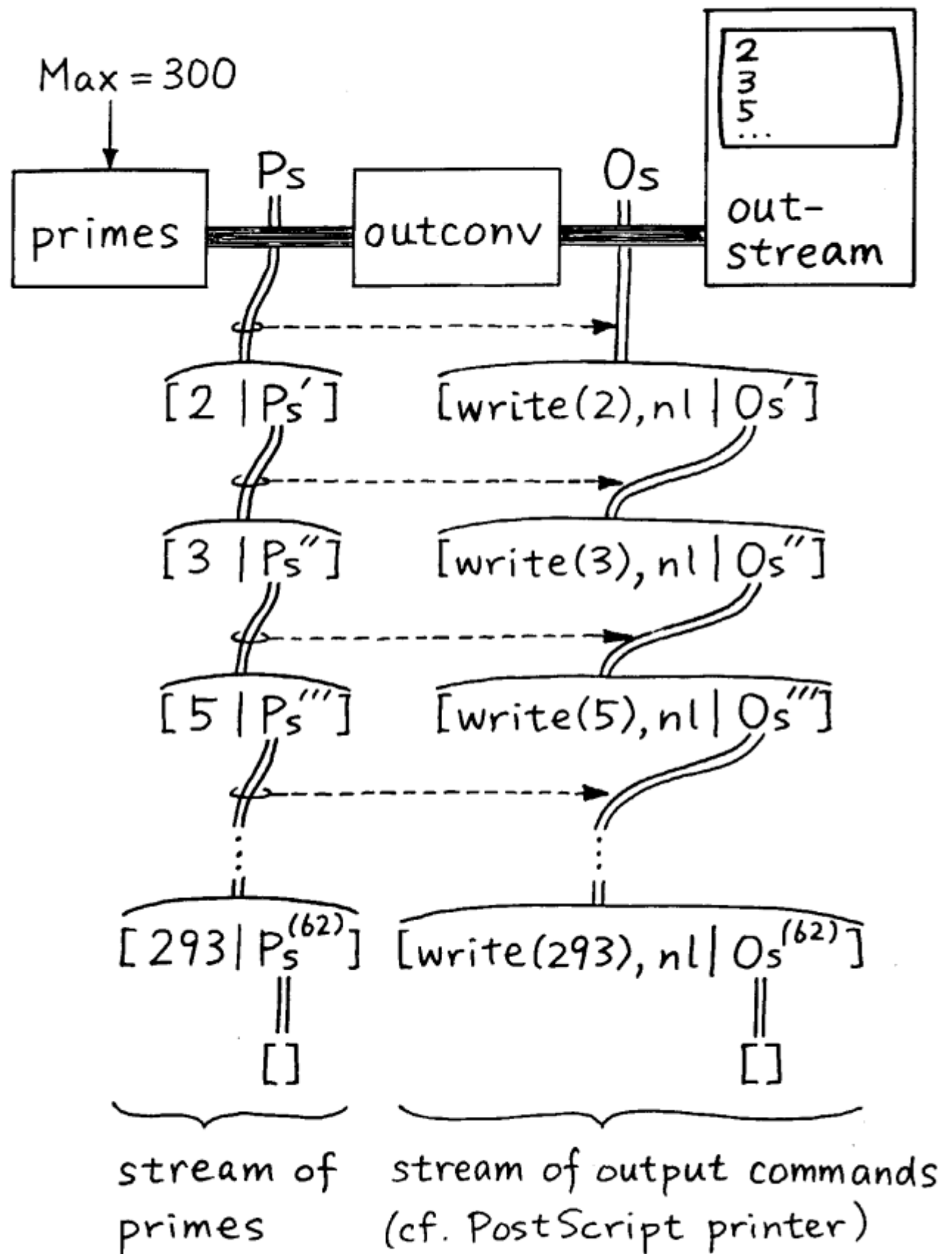
```
primes(Max,Ps) :- true |
    gen(2,Max,Ns),
    sift(Ns,Ps).

gen(NO,Max,Ns0) :- NO=<Max |
    Ns0=[NO|Ns1], N1:=NO+1,
    gen(N1,Max,Ns1).
gen(NO,Max,Ns0) :- NO >Max |
    Ns0=[].

sift([P|Xs1],Zs0) :- true |
    Zs0=[P|Zs1], filter(P,Xs1,Ys),
    sift(Ys,Zs1).
sift([], Zs0) :- true |
    Zs0=[].

filter(P,[X|Xs1],Ys0) :- X mod P=\=0 |
    Ys0=[X|Ys1],
    filter(P,Xs1,Ys1).
filter(P,[X|Xs1],Ys0) :- X mod P==0 |
    filter(P,Xs1,Ys0).
filter(P,[], Ys0) :- true |
    Ys0=[].
```

Displaying Prime Numbers



Generating Fibonacci Numbers

[1, 1, 2, 3, 5, 8, 13, 21, ...]

Eagerly:

```
go :- true |  
    fib(Ns), outconv(Ns,Os), outstream(Os).
```

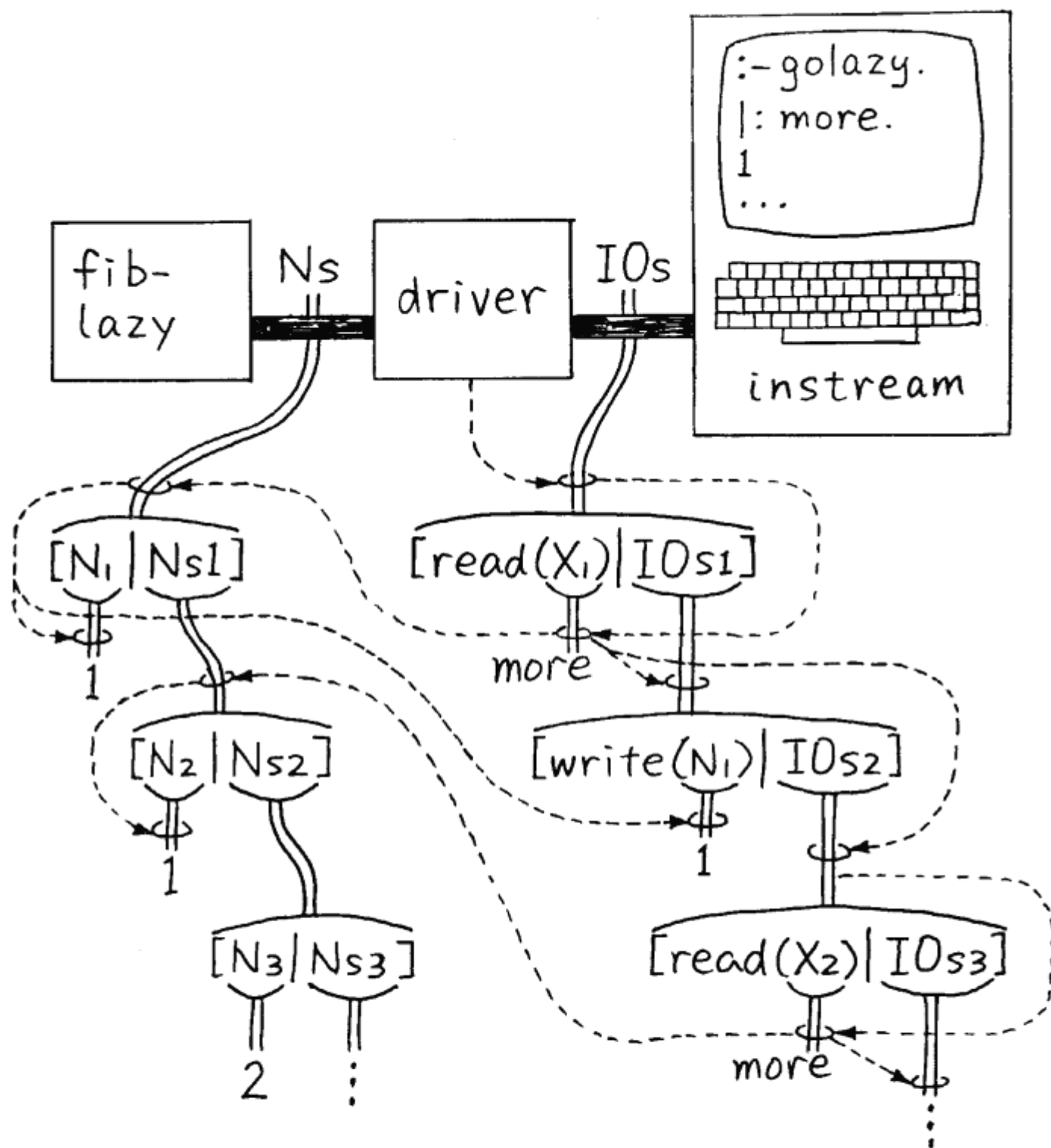
```
fib(Ns)          :- true |  
    fib(1,0,Ns).  
fib(N1,N2,Ns0) :- true |  
    N3:=N1+N2, Ns0=[N3|Ns1], fib(N2,N3,Ns1).
```

Lazily:

```
golazy :- true |  
    fiblazy(Ns), driver(Ns,I0s), instream(I0s).
```

```
fiblazy(Ns)          :- true |  
    fiblazy(1,0,Ns).  
fiblazy(N1,N2,[N3|Ns1]) :- true |  
    N3:=N1+N2, fiblazy(N2,N3,Ns1).  
fiblazy(_,_,[])      :- true | true.
```

```
driver(Ns ,I0s0)      :- true |  
    I0s0=[read(X)|I0s1],  
    driver(Ns,I0s1,X).  
driver(Ns0,I0s0,more) :- true |  
    Ns0=[N|Ns1],  
    I0s0=[write(N),nl|I0s1],  
    driver(Ns1,I0s1).  
driver(Ns, I0s ,done) :- true | Ns=[], I0s=[].
```



History-Sensitive Objects—Stack

```
stack(Xs) :- true | stk(Xs, []).
```

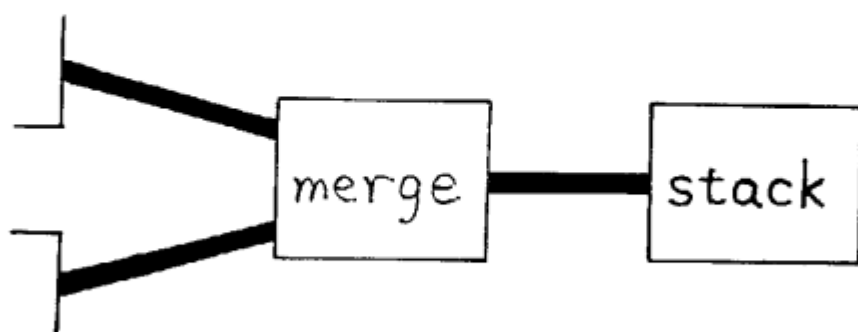
```
stk([push(T)|Xs1], Ls      ) :- true | stk(S, [T|Ls]).  
stk([pop(T) |Xs1], [L|Ls1]) :- true | T=L, stk(Xs1, Ls1).  
stk([pop(T) |Xs1], []      ) :- true | T=error, stk(Xs1, []).  
stk([],           Ls      ) :- true | true.
```

?- ~~stack~~stk(Xs), Xs=[push(5), push(4), pop(X), pop(Y), pop(Z)].

→ X=4, Y=5, Z=error • two-way communication
using incomplete messages

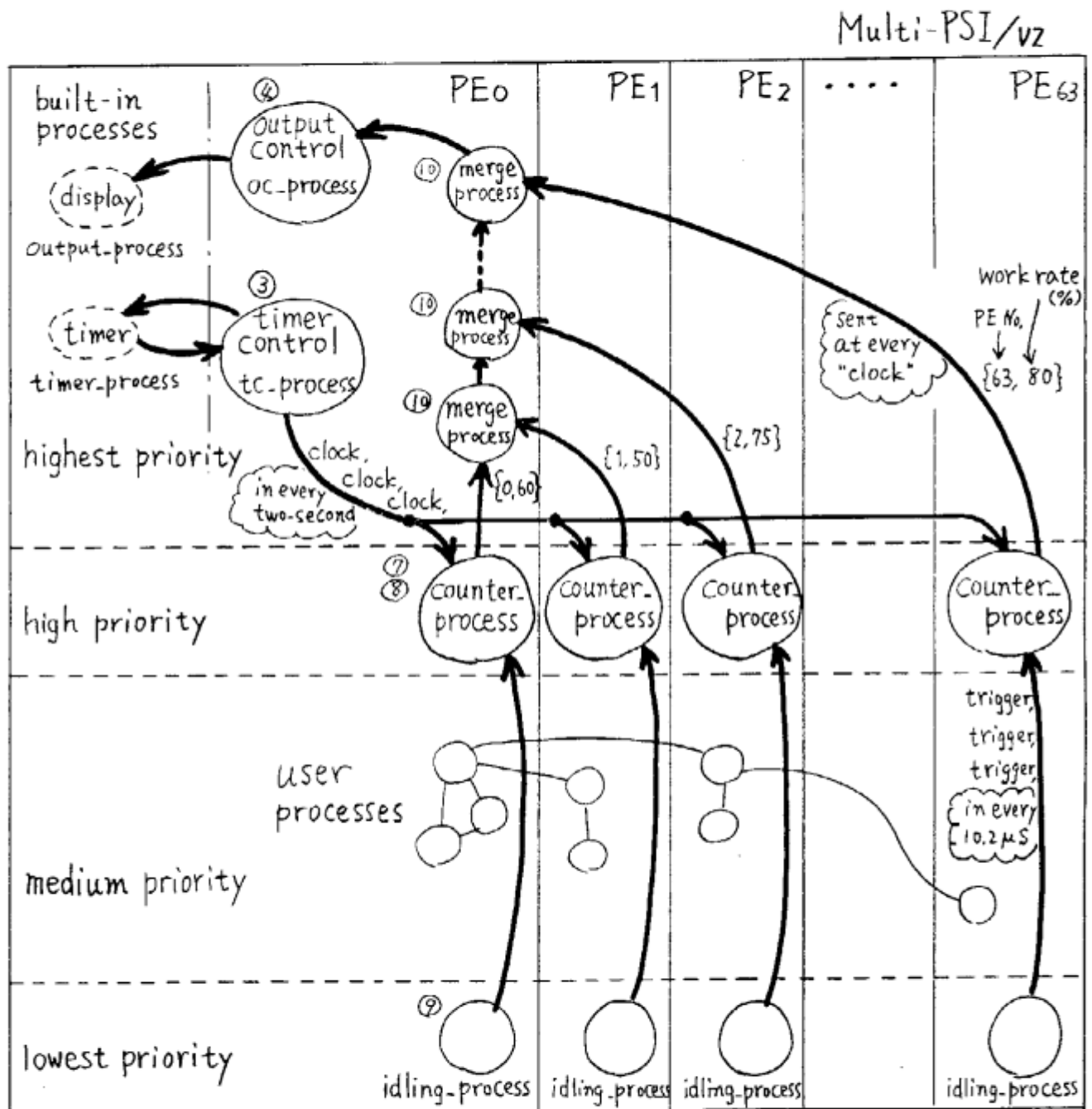
Merging Streams • many-to-one communication

```
merge([A|Xs1], Ys,      Zs) :- true |  
                                Zs=[A|Zs1], merge(Xs1, Ys, Zs1).  
merge(Xs,      [A|Ys1], Zs) :- true |  
                                Zs=[A|Zs1], merge(Xs, Ys1, Zs1).  
merge([],      Ys,      Zs) :- true | Zs=Ys.  
merge(Xs,      [],      Zs) :- true | Zs=Xs.
```



A KL1 program example : "Performans Meter"

- To measure each processor work rate and display in real time (in every two seconds)



Numbers encircled in the figure correspond to numbers in the source list.

Process structure of the "Performans Meter"

" PERFORMANCE METER " -- PROCESS DESCRIPTIONS --

idling_process : Continues to send "trigger" messages to the counter_process (in every 10.2 micro seconds) as long as it can run. Runs only when no runnable user processes. A lowest priority process.

counter_process : Suspended until a message arrives.
Counts up the counter value keeping by itself when "trigger" message arrives, then suspended again.

Calculates sum of idling_process run time (ms) when "clock" message arrives, with dividing the counter value by 98. User process run time is calculated by subtracting the idling_process run time from "clock" interval (ms). PE work rate can be attained as below.

$$\text{Effective work rate(\%)} = \frac{\{ (\text{clock interval(ms)} - \text{counter value} / 98) / \text{clock interval(ms)} \} \times 100}$$

Reports "{PE_number, Work_rate}" to the oc_process through the merge processes, clears the counter, then suspended again.

merge process : Merges two input streams to an output stream.

oc_process (output control process) : Constructs a 64-element vector with receiving cosecutive sixty four "{PE_number, Work_rate}" pairs.

	{60,	50,	75,	,	80}
work rate of	PE0	PE1	PE2			PE63

Sends the vector to the output_process when completed.
Suspended when no data comes.

tc_process (timer control process) :
Sends "{on-after, Interval=2000(ms), Timing}" to the timer_process, then waits (suspended) until "Timing" will be instanciated to "just_now". Sends "clock" to all the counter_processes when "Timing" becomes "just_now". Then, sends the same message to the timer_process again (starting the timer again), and waits.

The timer_process is started with receiving the "{on-after, Interval=2000(ms), Timing}" message. After "Interval" time, the timer_process instantiates "Timing" with a value, "just_now".

```

!!!!!!!!!!!! source program of the performance meter !!!!!!!!!!!!!
!!!!!!!!!!!! written by X.Taki, 1988.10.19 !!!!!!!!!!!!!

:= performance_meter(64,2000,Timer_stream,Output_stream);
timer_process(Timer_stream),      % assumed as a built-in function
output_process(Output_stream).    % assumed as a built-in function

```

①

```

② performance_meter(PE,Interval,Timer_stream,Output_stream) :- true |
    timer_control(Interval,Timer_stream,Timing_stream) @priority(*,4094);
    output_control(Report_stream,Output_stream,PE) @priority(*,4094);
    process_distribution(Timing_stream,Report_stream,
        PE,Interval) @priority(*,4095).

```

②

```

③ timer_control(Interval,Timer_stream,Timing_stream) :- true |
    Timer_stream = [(on_after,Interval,Timing) | Cdr],
    tc_process(Timing,Timer_stream,Cdr,Interval);
    tc_process(Just_now,Timer_stream,Timer_stream,Interval) :- true |
    Timing_stream = [(clock|Tag,Cdr)],
    Timer_stream = [(on_after,Interval,Timing) | Ts_cdr],
    tc_process(Timing,Tag_cdr,Ts_cdr,Interval).

```

③

```

④ output_control(Report_stream,Output_stream,PE) :- true |
    new_vector(Vector,PE),
    oc_process(Report_stream,Output_stream,PE,PE,Vector),
    oc_process(Report_stream,Output_stream,0,PE,Vector) :- true |
    Output_stream = [(display,Vector) | Os_cdr],
    new_vector(New_vector,PE),
    oc_process(Report_stream,Os_cdr,PE,PE,New_vector),
    oc_process([(Pe_no,Work_rate) | Rs_cdr],Output_stream,Count,PE,Vector) :-
    Count =\= 0 |
    set_vector_element(Vector,Pe_no,_,Work_rate,New_vector),
    New_count := Count - 1,
    oc_process(Rs_cdr,Output_stream,New_count,PE,New_vector).

```

④

```

⑤ process_distribution(_Report_stream,0,_) :- true |
    Report_stream = [];
    process_distribution(Timing_stream,Report_stream,Count,Interval) :- true |
    New_count := Count - 1,
    merge(S1,S2,Report_stream),
    measure(Timing_stream,S1,New_count,Interval) @processor(New_count),
    process_distribution(Timing_stream,S2,New_count,Interval).

```

⑤

```

⑥ measure(Timing_stream,Report_stream,Pe,Interval) :- true |
    counter_process(Timing_stream,Trigger_stream,0,
        Report_stream,Pe,Interval) @priority(*,4093);
    idling_process(Trigger_stream) @priority(*,0).

```

⑥

```

⑦ counter_process((clock|Ts_cdr),Trigger_stream,Count,
    Report_stream,Pe,Interval) :- true |
    Exec_time_by_ms := Count / 98,
    work_rate(Exec_time_by_ms,Interval,Work_rate),
    Report_stream = [(Pe,Work_rate) | Rs_cdr],
    counter_process(Ts_cdr,Trigger_stream,0,Rs_cdr,Pe,Interval);
    counter_process(Timing_stream,(Trigger|Trys_cdr),Count,
        Report_stream,Pe,Interval) :- true |
    New_count := Count + 1,
    counter_process(Timing_stream,Trys_cdr,New_count,
        Report_stream,Pe,Interval).

```

⑦

```

⑧ work_rate(Exec_time,Interval,Work_rate) :- Exec_time >= Interval |
    Work_rate = 0;
    work_rate(Exec_time,Interval,Work_rate) :- Exec_time < Interval |
    Work_rate := ((Interval - Exec_time)*100)/Interval.

```

⑧

```

⑨ idling_process(Trigger_stream) :- true |
    Trigger_stream = (Trigger|Ts_cdr), idling_process(Ts_cdr).

```

⑨

```

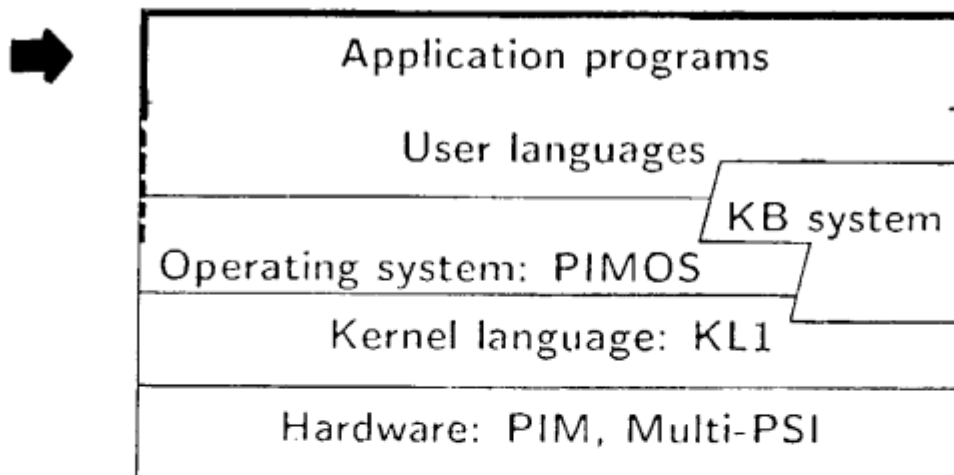
⑩ merge([X|XT],Y,Z) :- true | Z=[X|ZT],merge(XT,Y,ZT);
    merge(X,[Y|YT],Z) :- true | Z=[Y|ZT],merge(X,YT,ZT);
    merge([],Y,Z) :- true | Z=Y;
    merge(X,[],Z) :- true | Z=X.

```

⑩

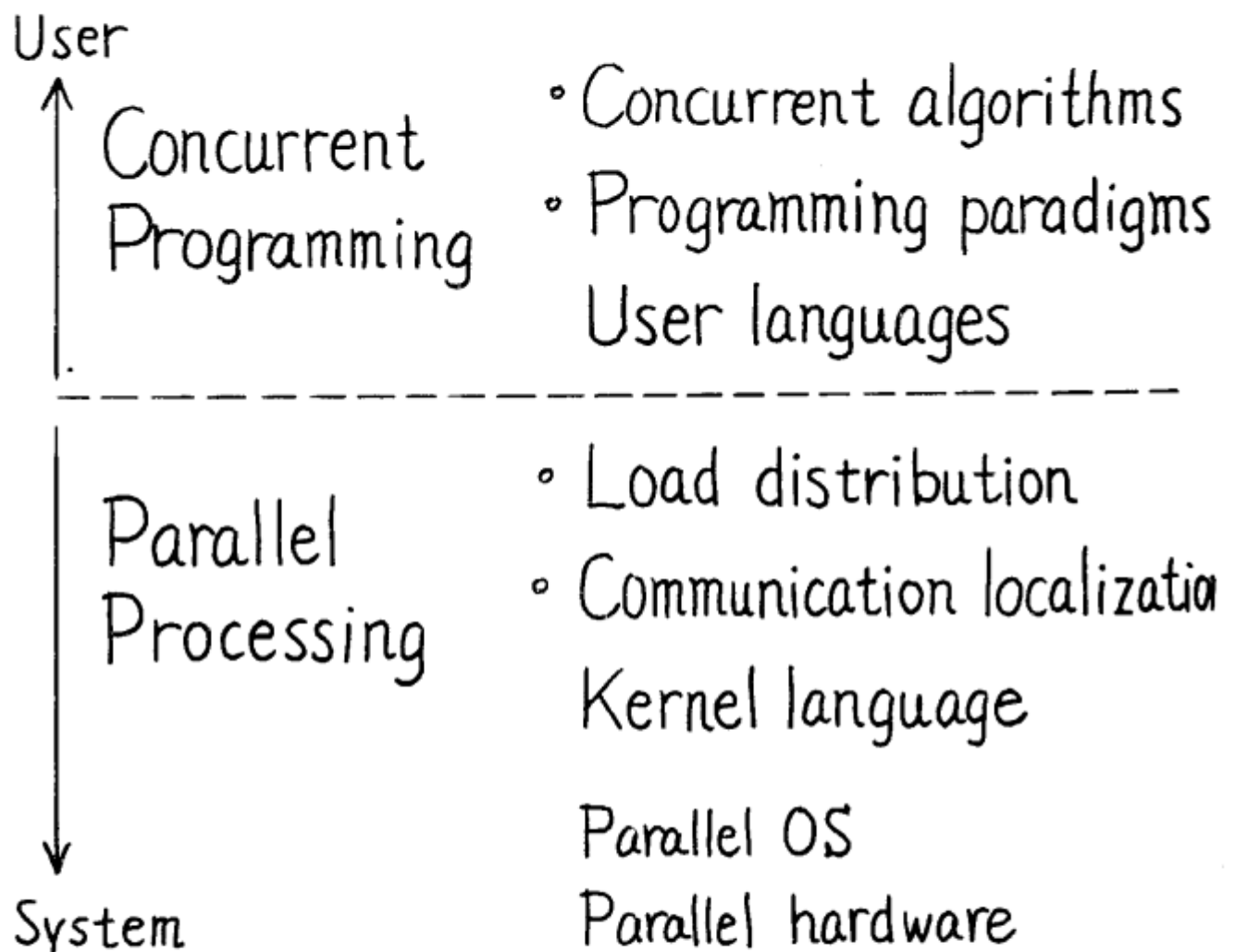
□ Concurrent Programing and Parallel Processing

— Research Themes and an Approach —



Cultivate the Parallel Computing

- Research Themes -



Everything new !

Parallel computing on large scale MIMD machines = New culture of computing

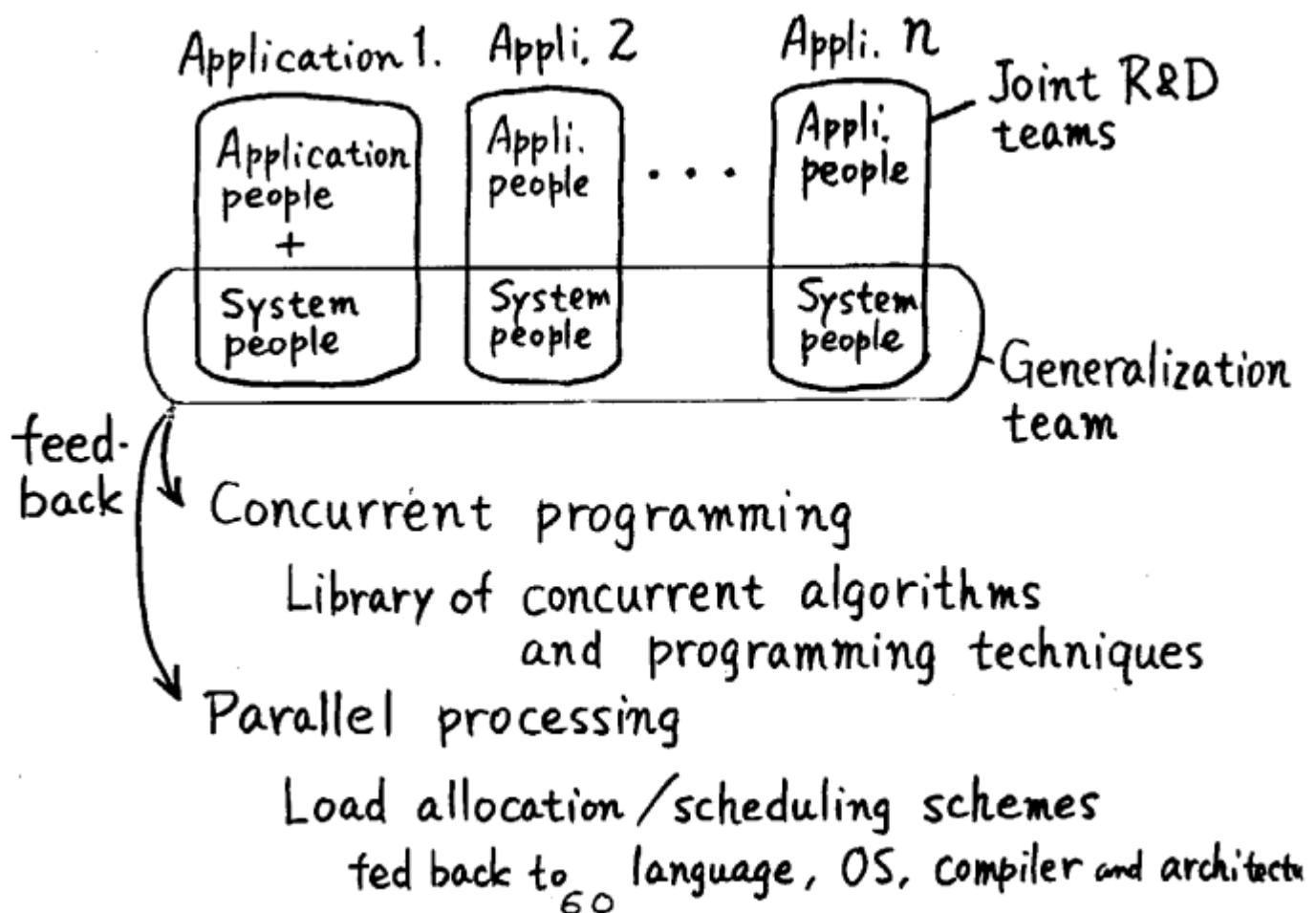
— Approach —

- Preparation of R & D tools

- Joint R & D teams

Application people
+
System builders

- Application specific \rightarrow General



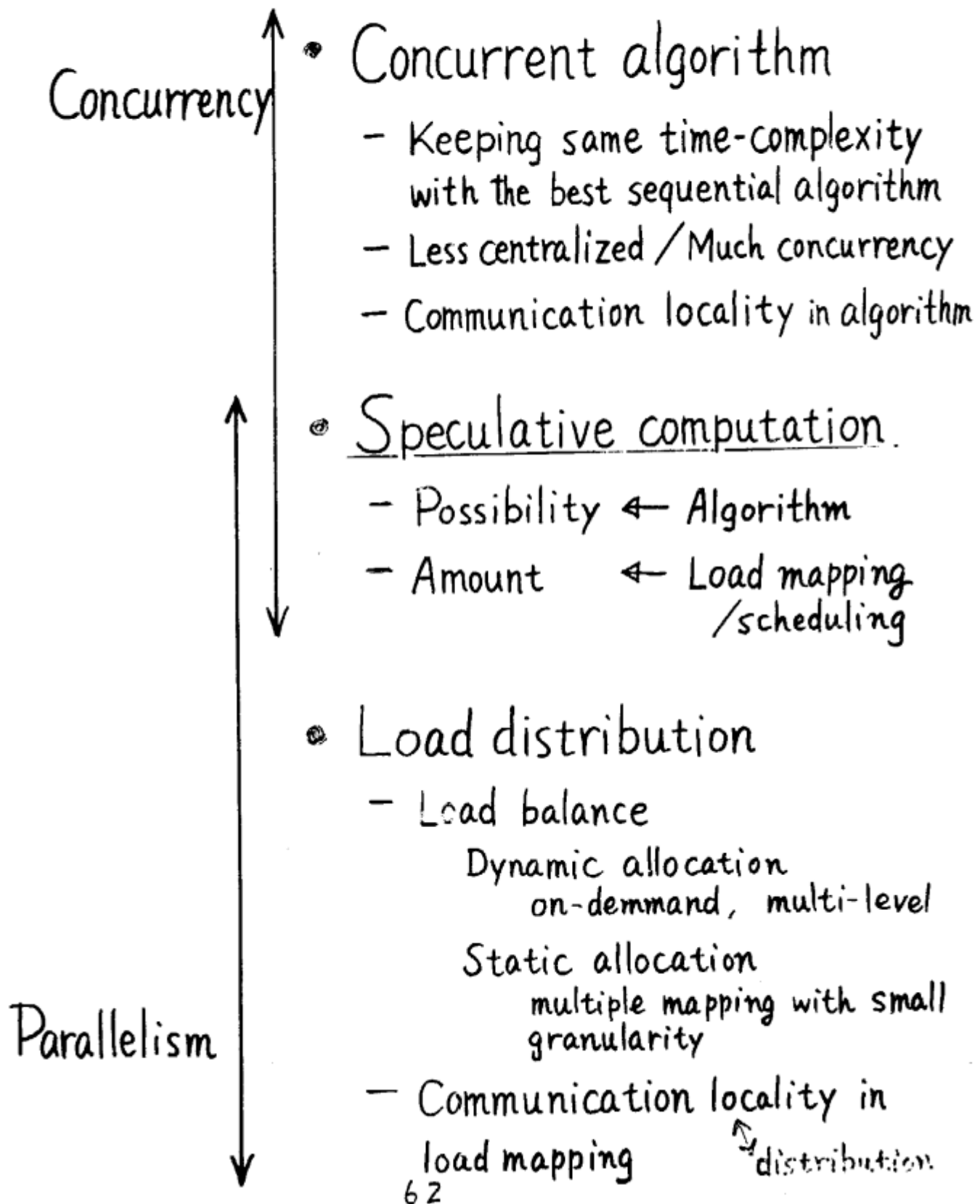
- Examples -

- { - Different type of problems
- { - Different type of program structures and dynamic characteristics
- { - Development of concurrent algorithms
- { - Experimentation of load distribution schemes

The four experimental parallel application programs

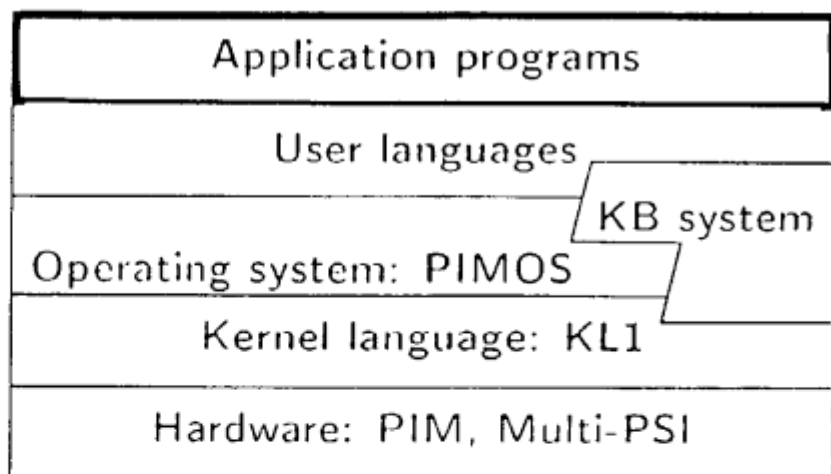
1. PAX : a natural language parser
2. Tsume-go : a board game
3. Packing piece puzzle (Pentomino)
4. Shortest path finding problem

— Designing Concurrency and Parallelism —



□ Concurrent Programing and Parallel Processing

— Examples and Performance —



cf. Appendix. A

1. PAX

- Bottom-up all solution search
- Global communication among processes
- = Load allocation focused on communication localit

2. Tsume-go

- Game tree search
- = Parallel alpha-beta pruning algorithm
- = Dynamic / static load allocation

3. Packing piece puzzle

- "OR-parallel" all solution search
- Independent search among descendant branches
- = Dynamic / static load allocation

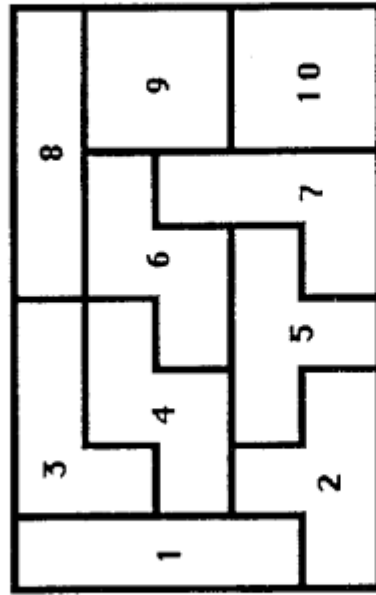
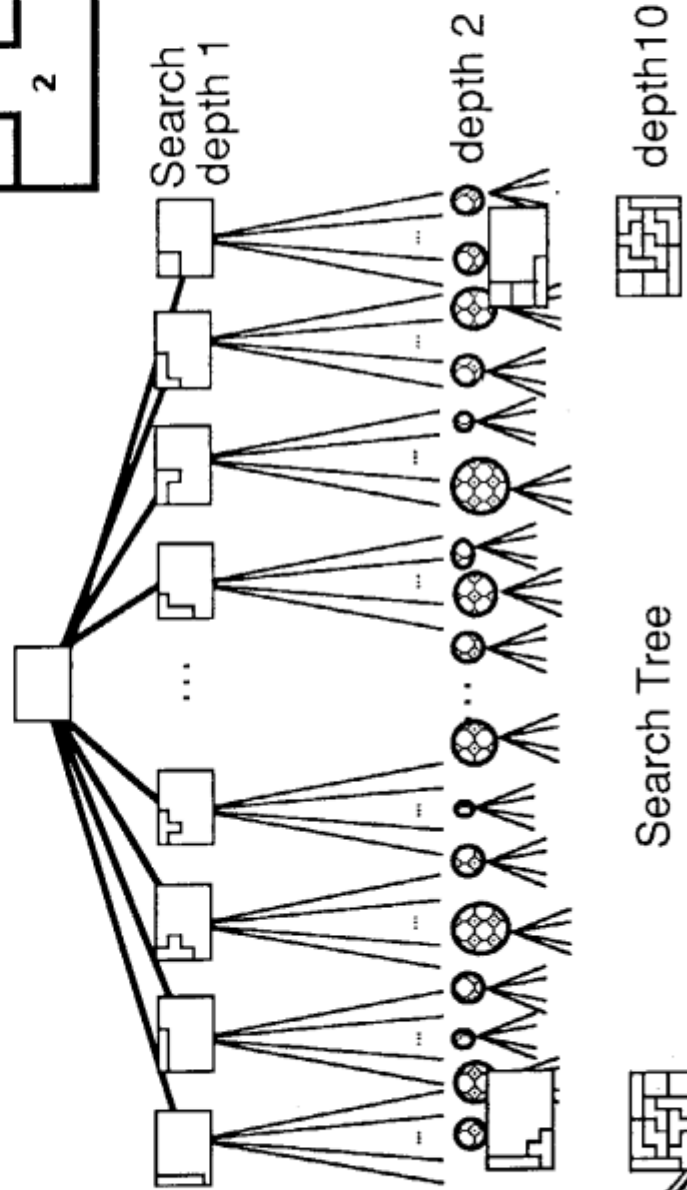
4. Shortest path finding problem

- Best solution search
- = Process oriented algorithm
- = Static allocation with multiple mapping

OR-Parallel Exhaustive Search Problem

Packing Piece Puzzle

Find all possible ways to pack 10 pieces into a box

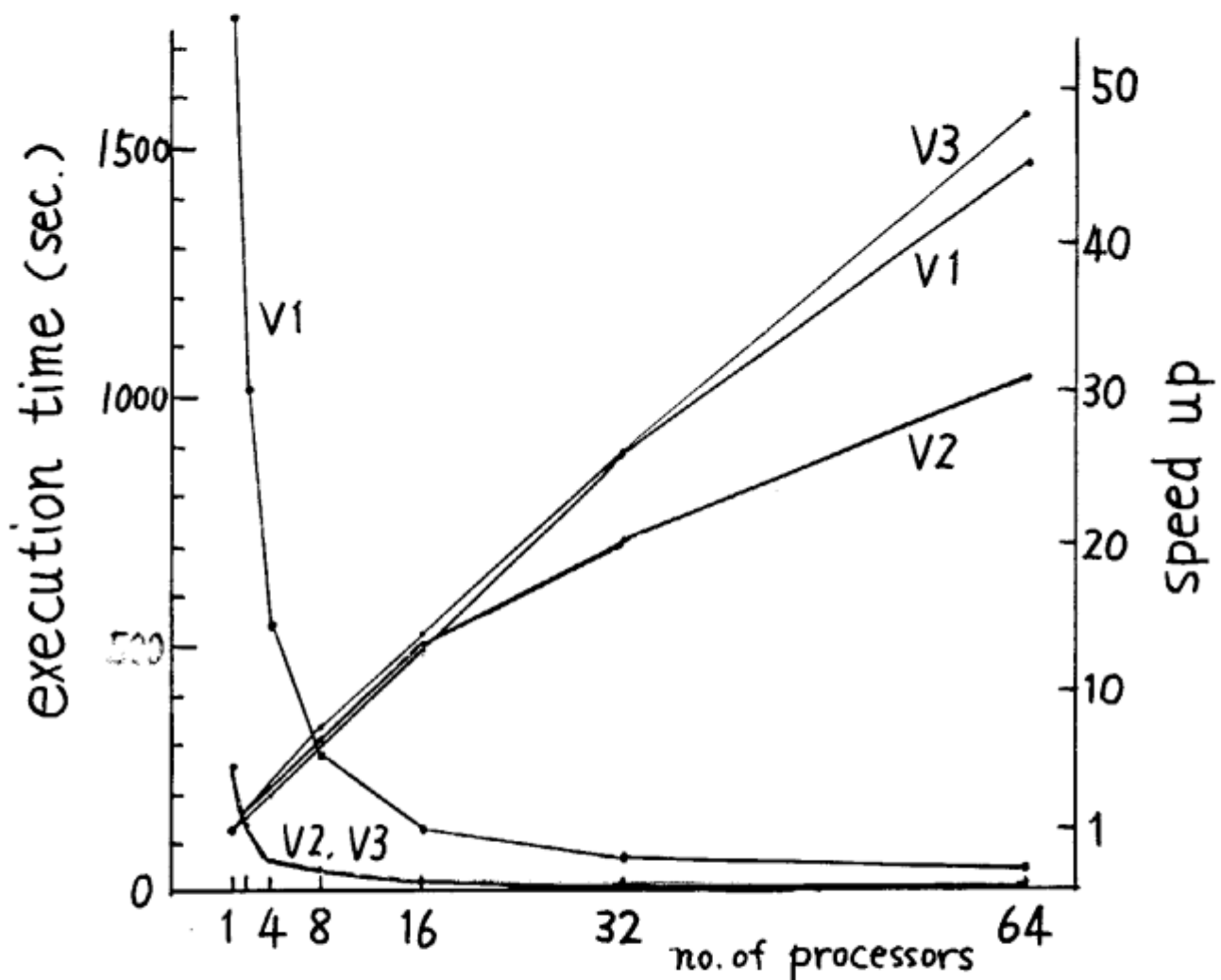


No. of solution:
3,106

Total work:
8 Million reduction

Execution time:
260 sec.on 1PE

— Performance —

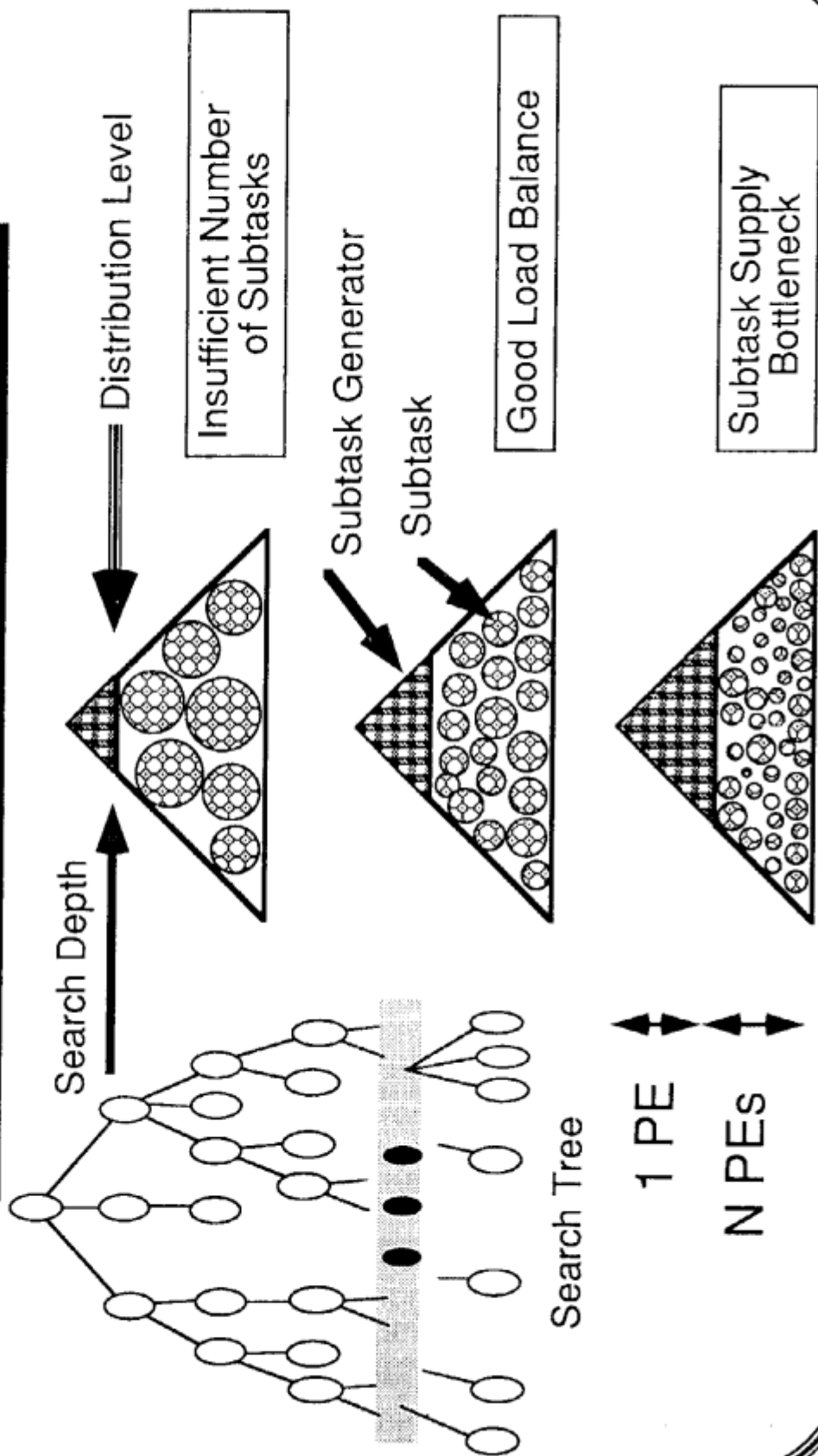


Packing piece puzzle (10 pieces, 5x8 box)

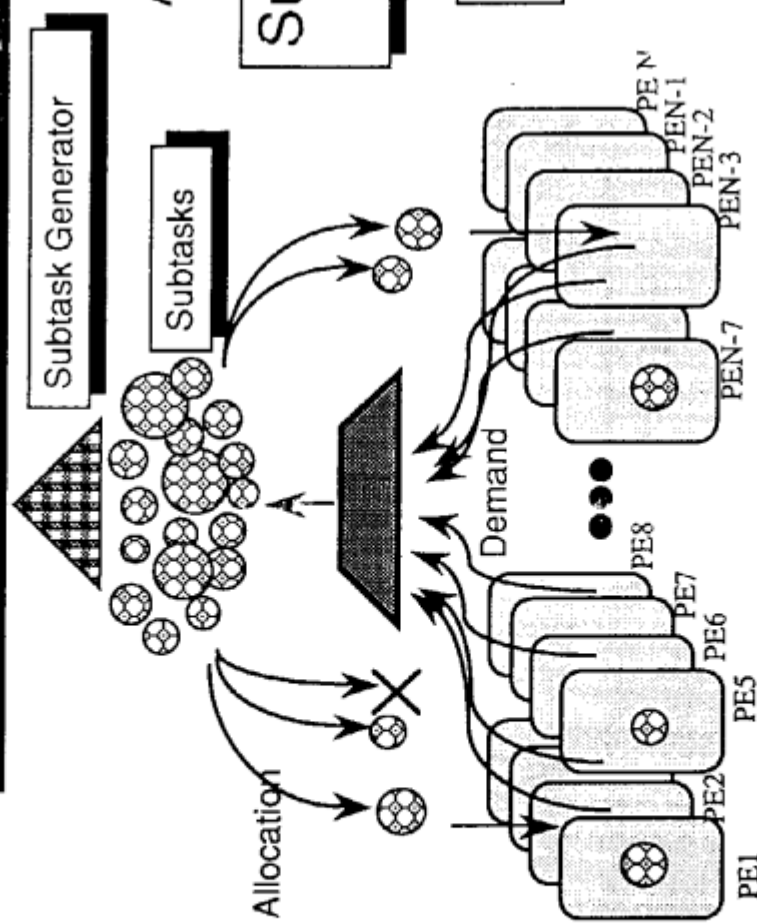
- V1 : FGCS (first) version, static allocation
- V2 : Improved version, dynamic allocation
- V3 : Most recent version, dynamic allocation (2-level)

* Best execution time = 5.3 sec

Tuning the Granularity of Subtasks



Problem of Simple On-demand Dynamic Load Balancing



As the number of PEs increases,

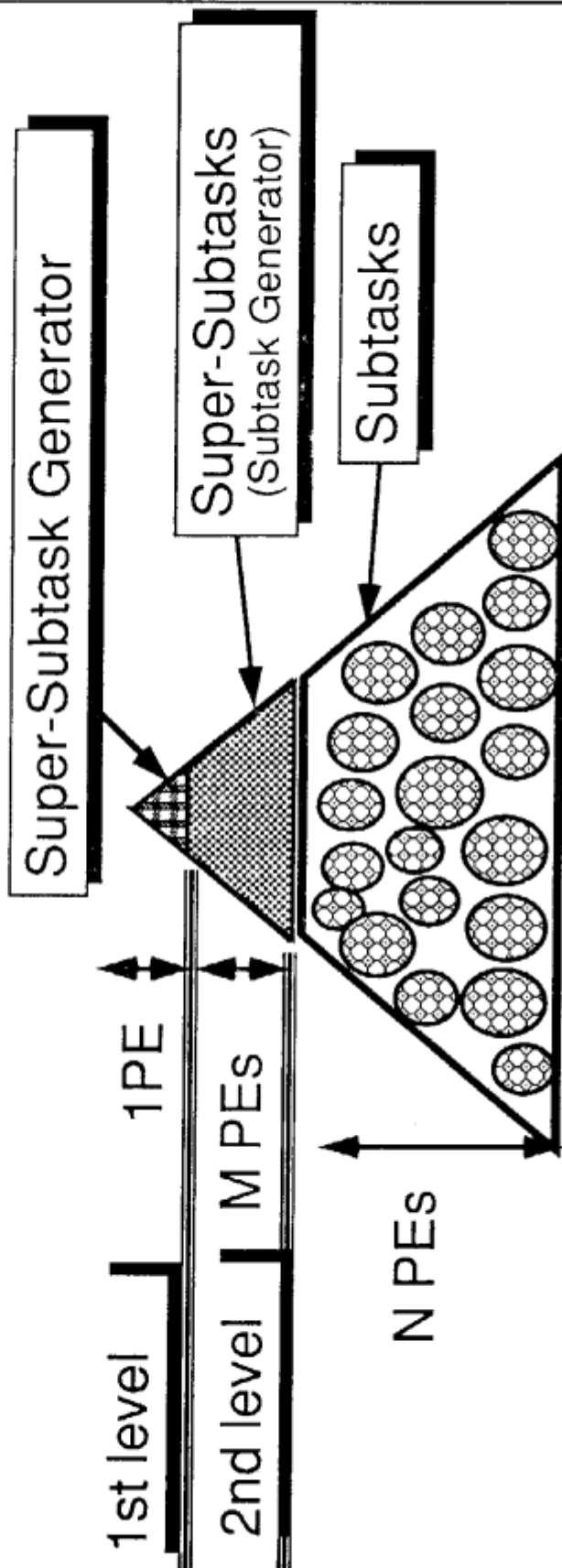
Subtask Supply Becomes Bottleneck

Not scalable

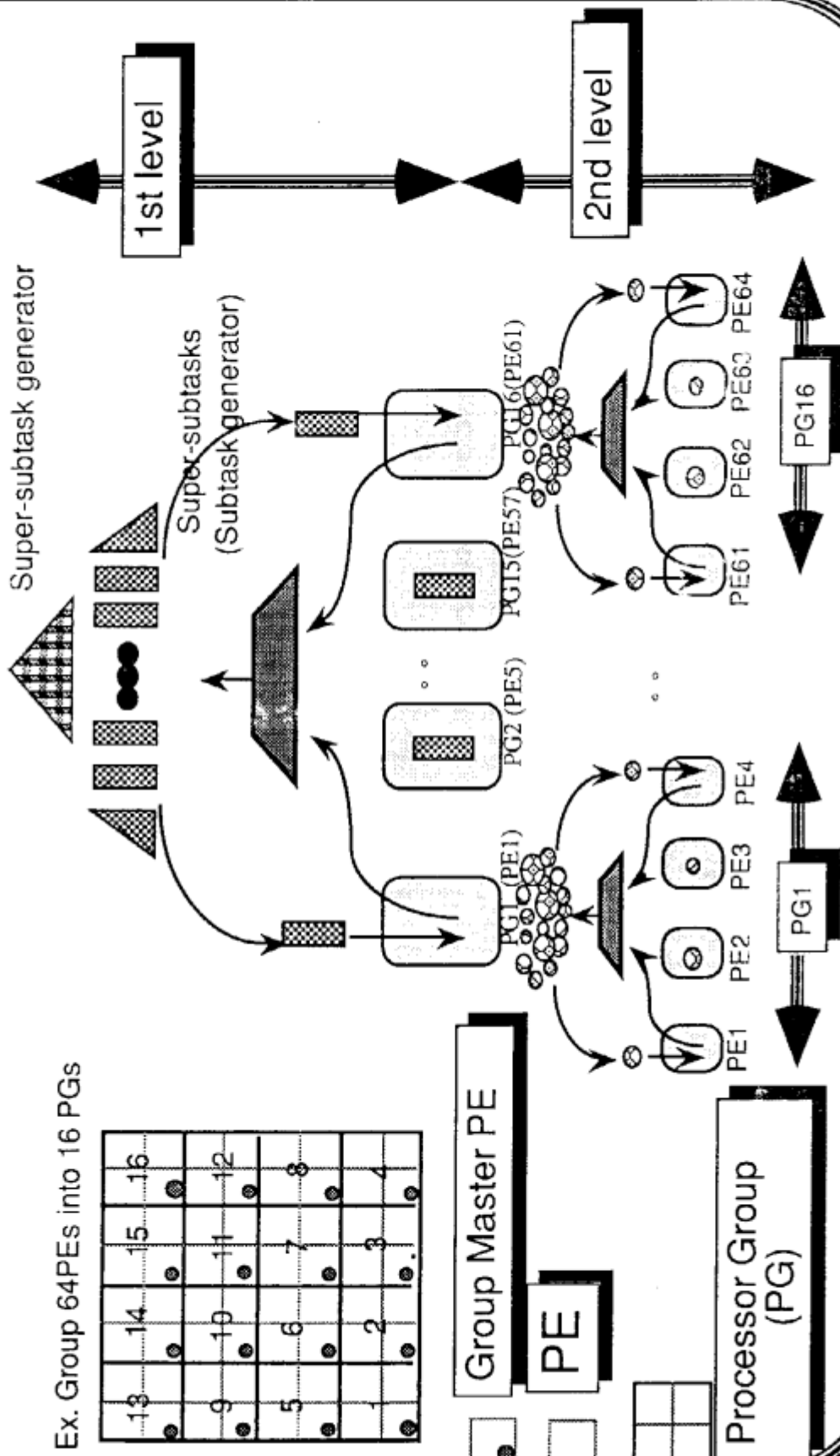


Multi Level Dynamic Load Balancing

Multi Level Load Balancing

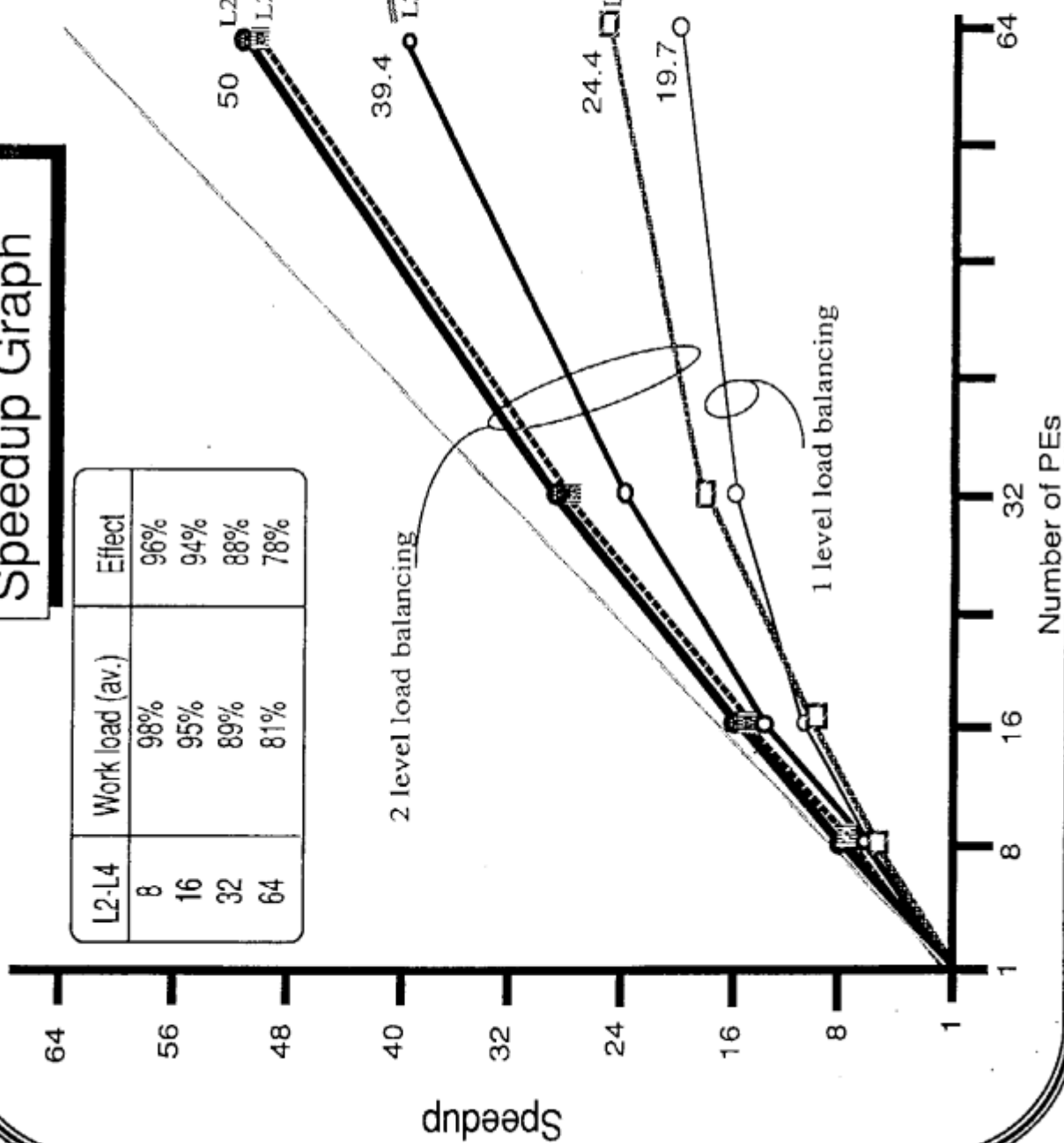


Multi Level Load Balancing (2 level load balancing)



Speedup Graph

L2-L4	Work load (av.)	Effect
8	98%	96%
16	95%	94%
32	89%	88%
64	81%	78%



Discussion

Requirements for good load balancing

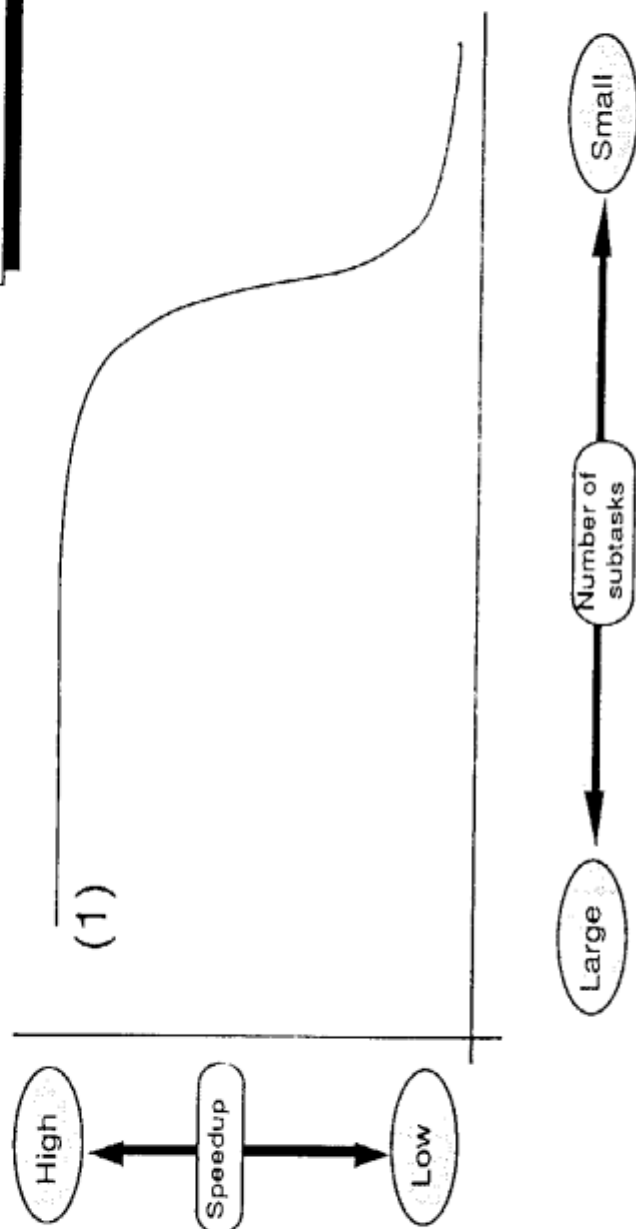
(1) Large number of subtasks

To balance load

(2) Large granularity

To prevent subtask supply bottleneck

Load imbalance

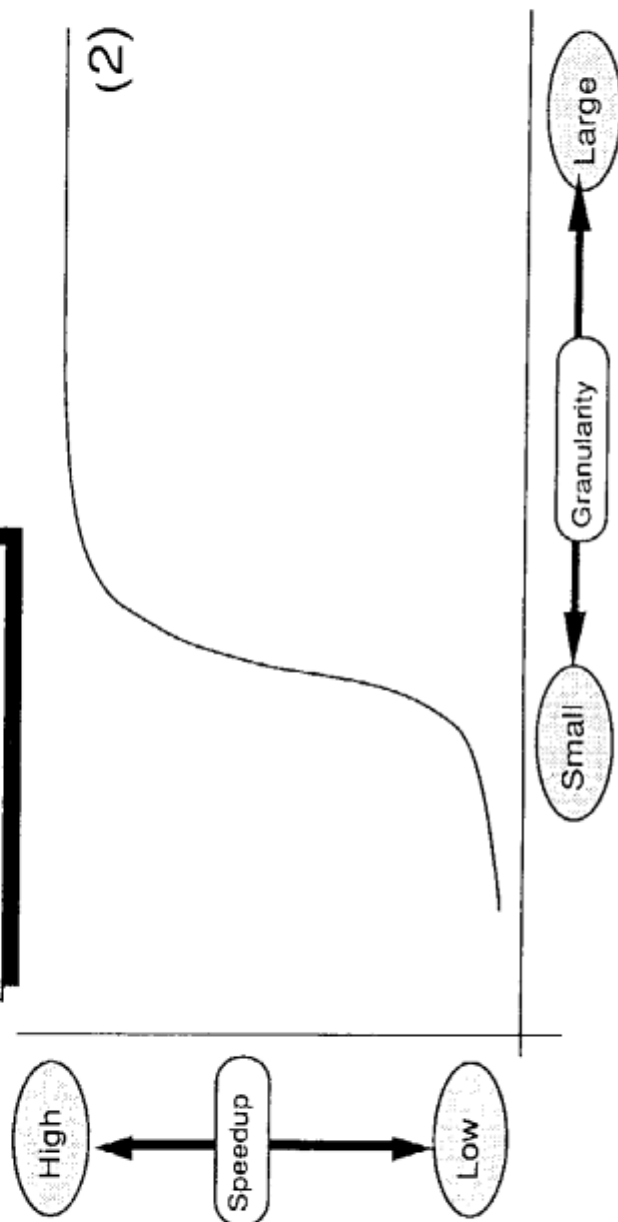


Discussion

Requirements for good load balancing

- (1) Large number of subtasks To balance load
- (2) Large granularity To prevent subtask supply bottleneck

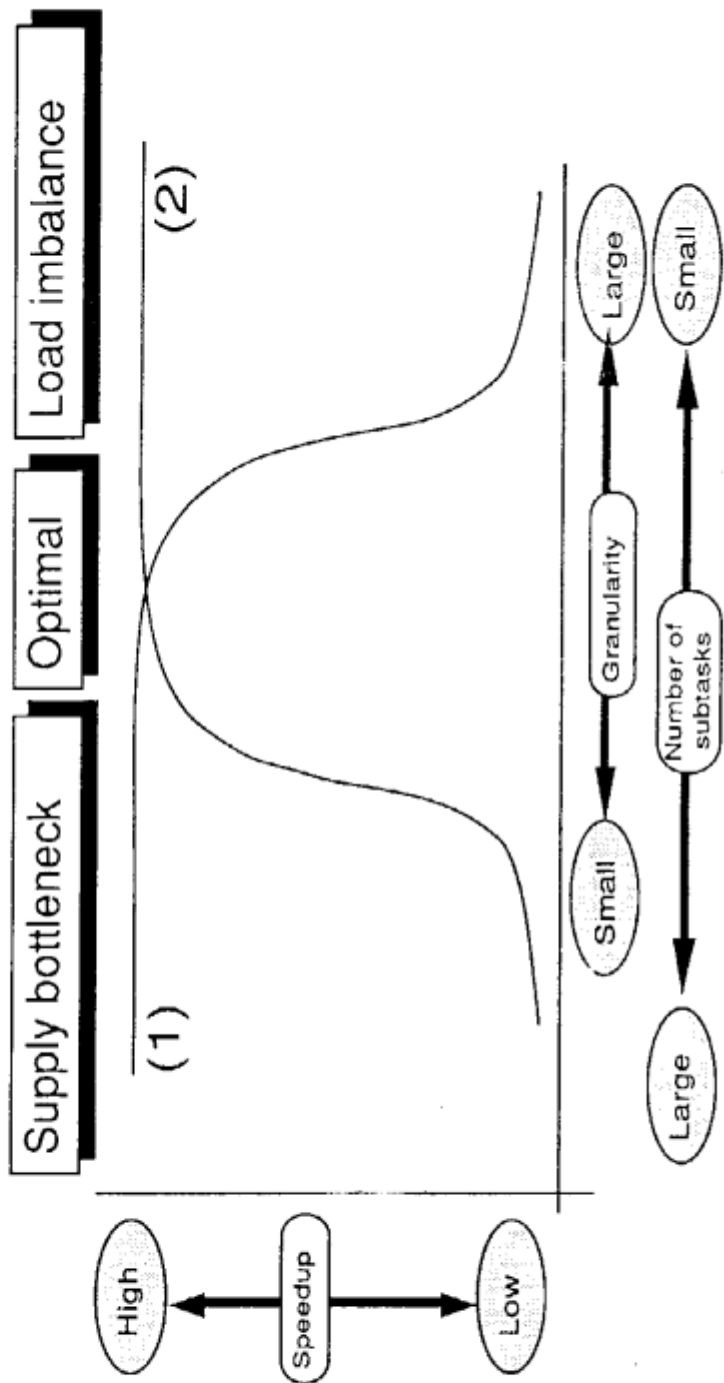
Supply bottleneck



Discussion

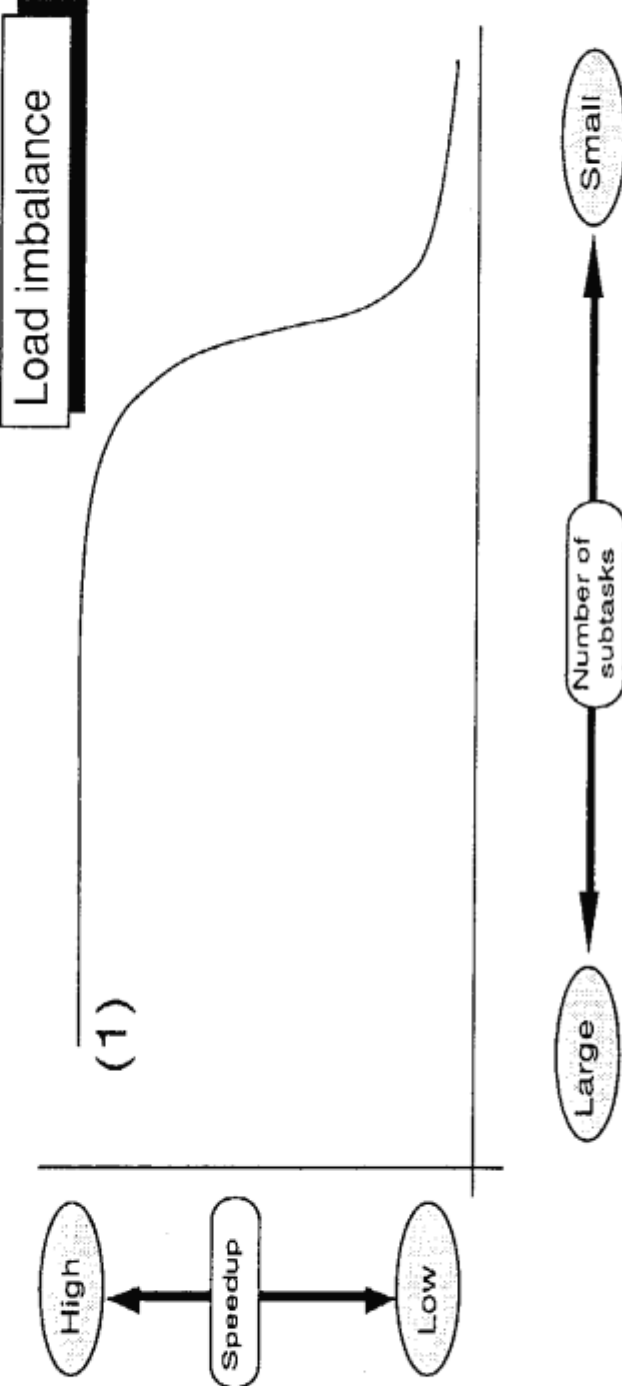
Requirements for good load balancing

- (1) Large number of subtasks To balance load
- (2) Large granularity To prevent subtask supply bottleneck



Requirement on the Number of Subtasks (Not to cause load imbalance)

$$\text{No. of subtasks} \geq \text{No. of PEs} \times \frac{\text{Expected work load}}{1 - \text{Expected work load}} \times \frac{\text{Max Gran.}}{\text{Min Gran.}}$$

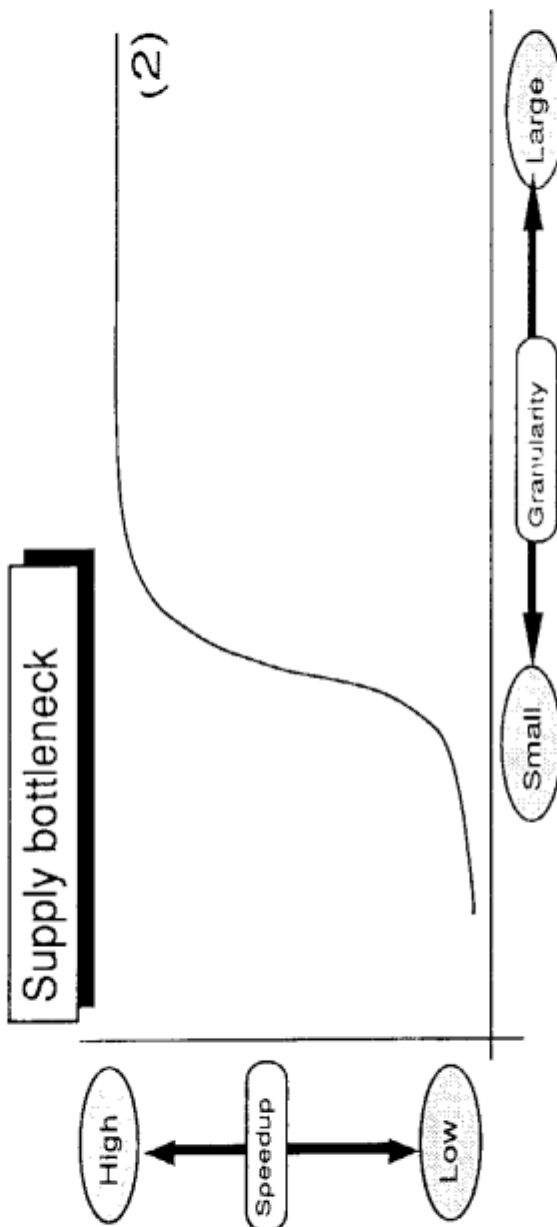


Example

$$\text{No. of subtasks} \geq 64 \times \frac{0.8}{1-0.8} \times \frac{180}{17} = \underline{2,710}$$

Requirement on the Subtask Granularity (Not to become supply bottleneck)

$$\text{Average Granularity} \geq \left(\frac{\text{Generation Cost} + \text{Distribution Cost}}{\text{No. of processor} - 1} \right) \times \text{Reduction}$$



Example

$$\text{Average Granularity} \geq \frac{(40 + 35) \times (64 - 1)}{\text{Reduction}} = \frac{4,725}{\text{Reduction}}$$

Effect of Multi Level Load Balancing

$$\text{Average Granularity} > \left(\frac{\text{Generation Cost} + \text{Distribution Cost}}{\text{No. of processor} - 1} \right) \times \text{No. of processor} - 1$$

4,725 Reduction

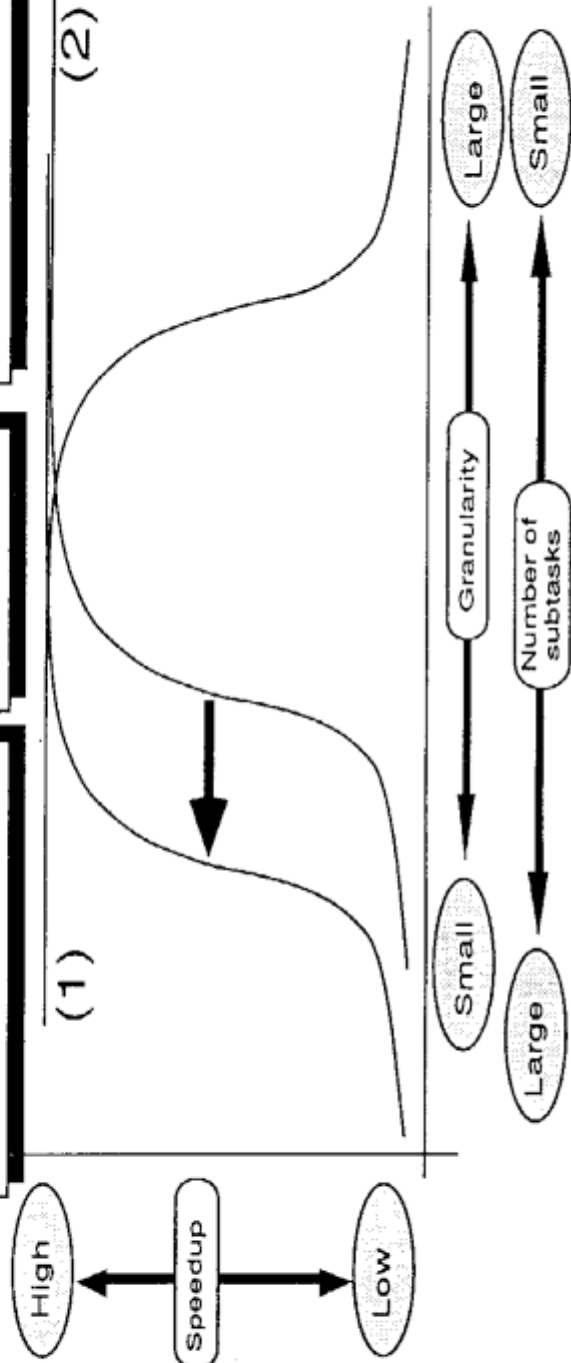
$$\text{Average Granularity} > \left(\frac{\text{Generation Cost} + \text{Distribution Cost}}{\text{No. of processor} - 1} \right) \times \text{No. of processor} - 1$$

225 Reduction

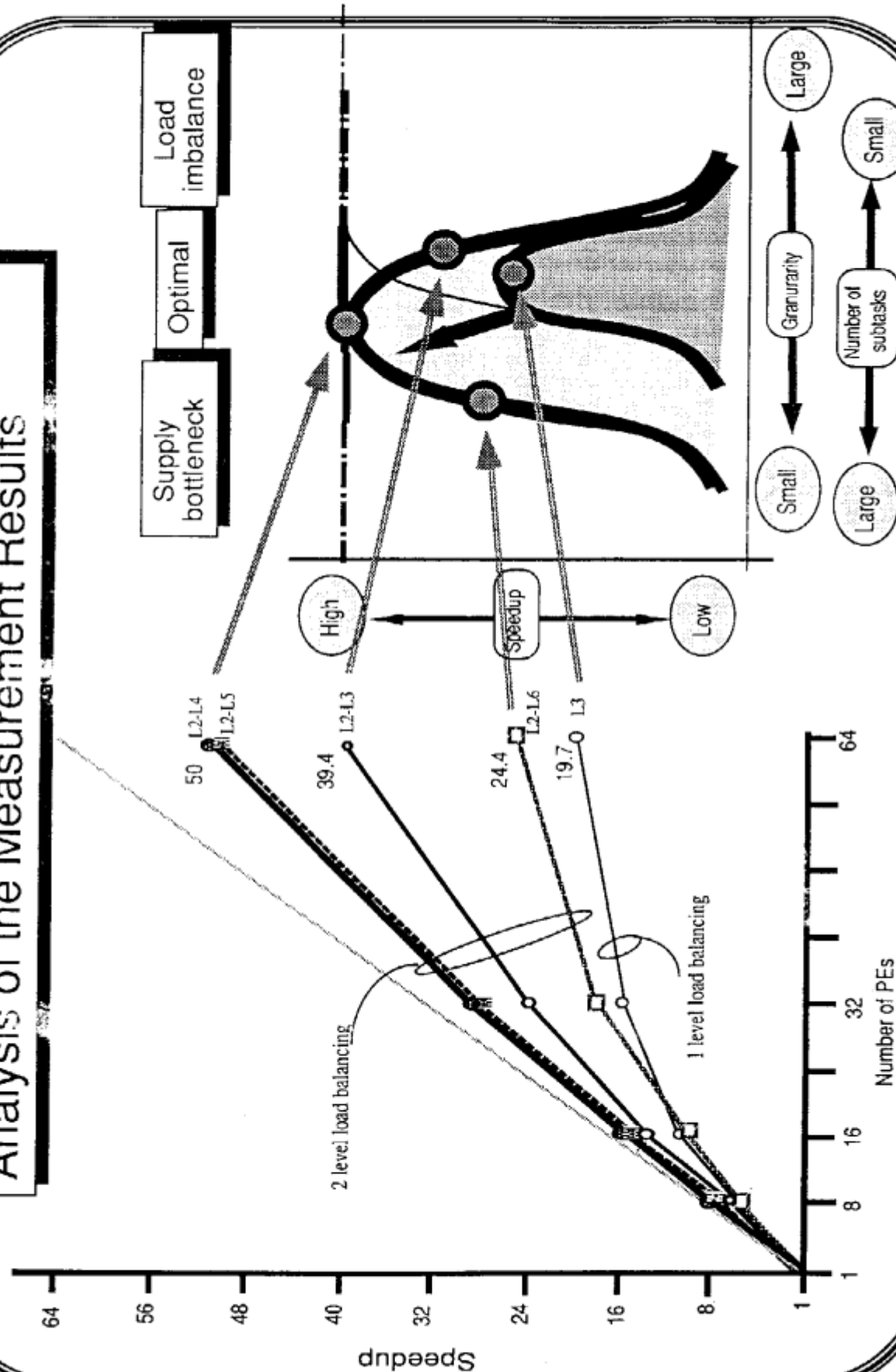
Supply bottleneck

Optimal

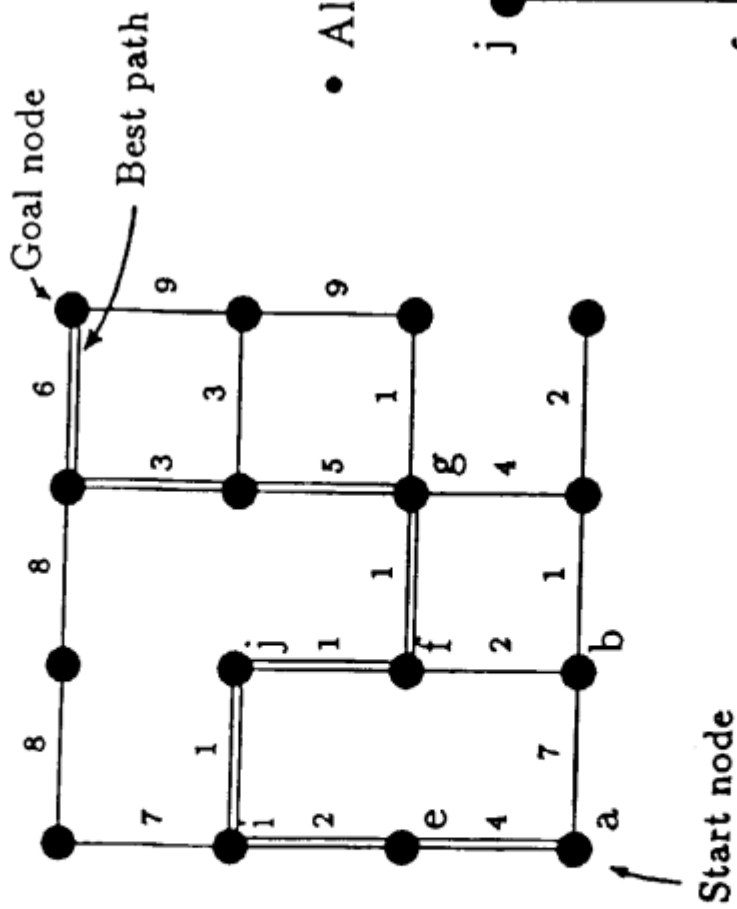
Load imbalance



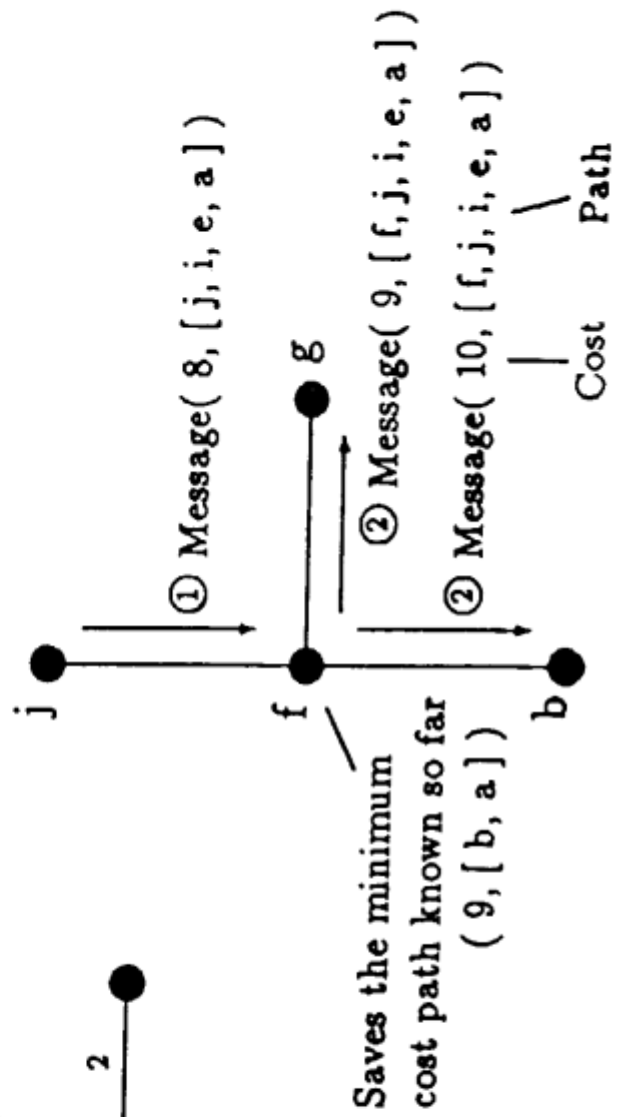
Analysis of the Measurement Results



- Shortest path finding problem
(• Best path problem)



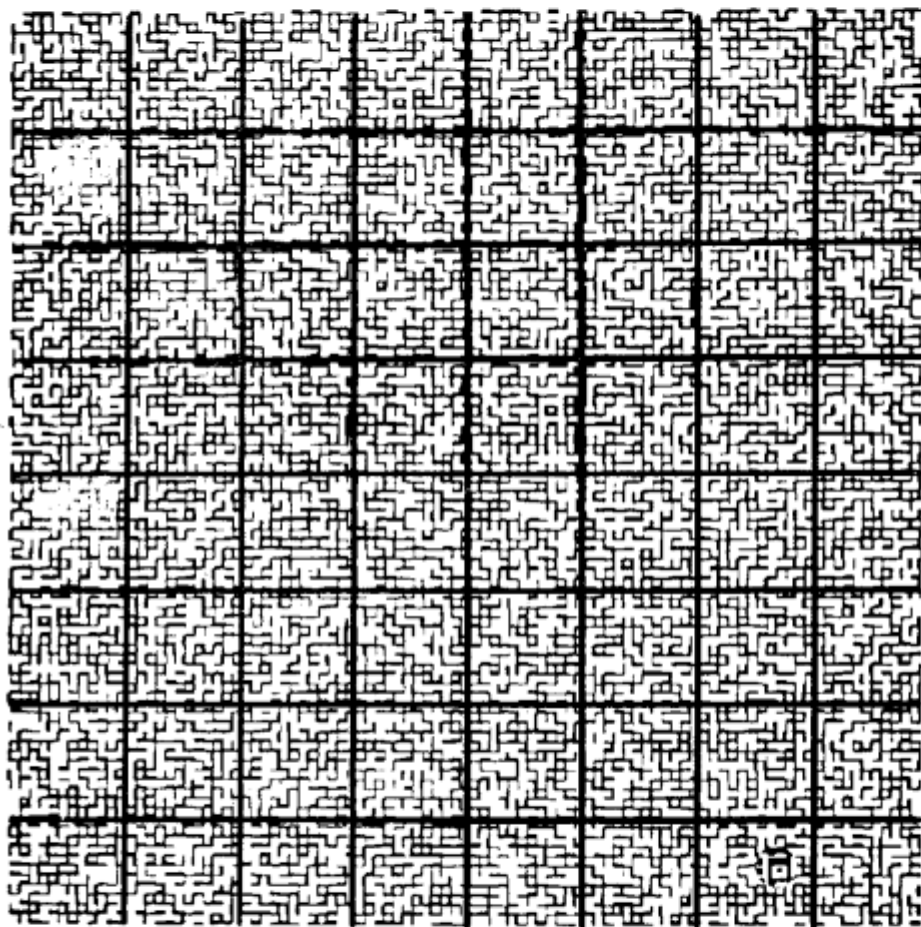
- Algorithm



BESTPATH
PROBLEM
SOLVER
10000
NODES



Start node : 582
Goal node : to all

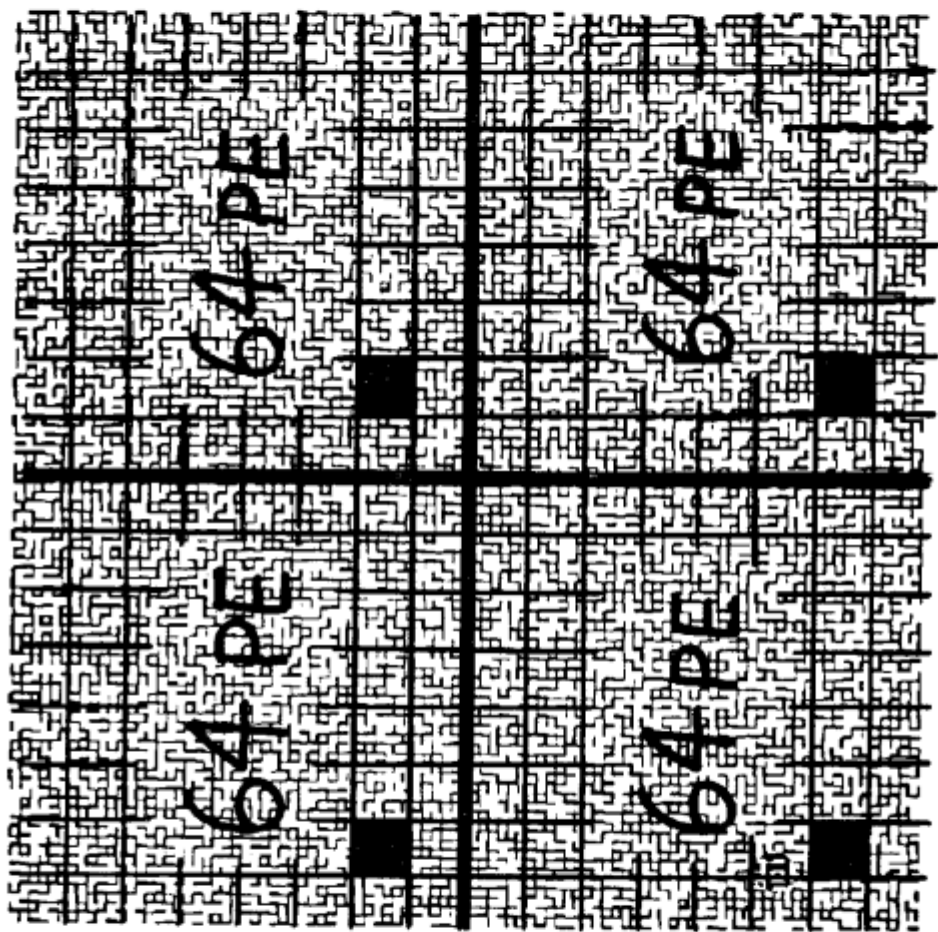


simple mapping

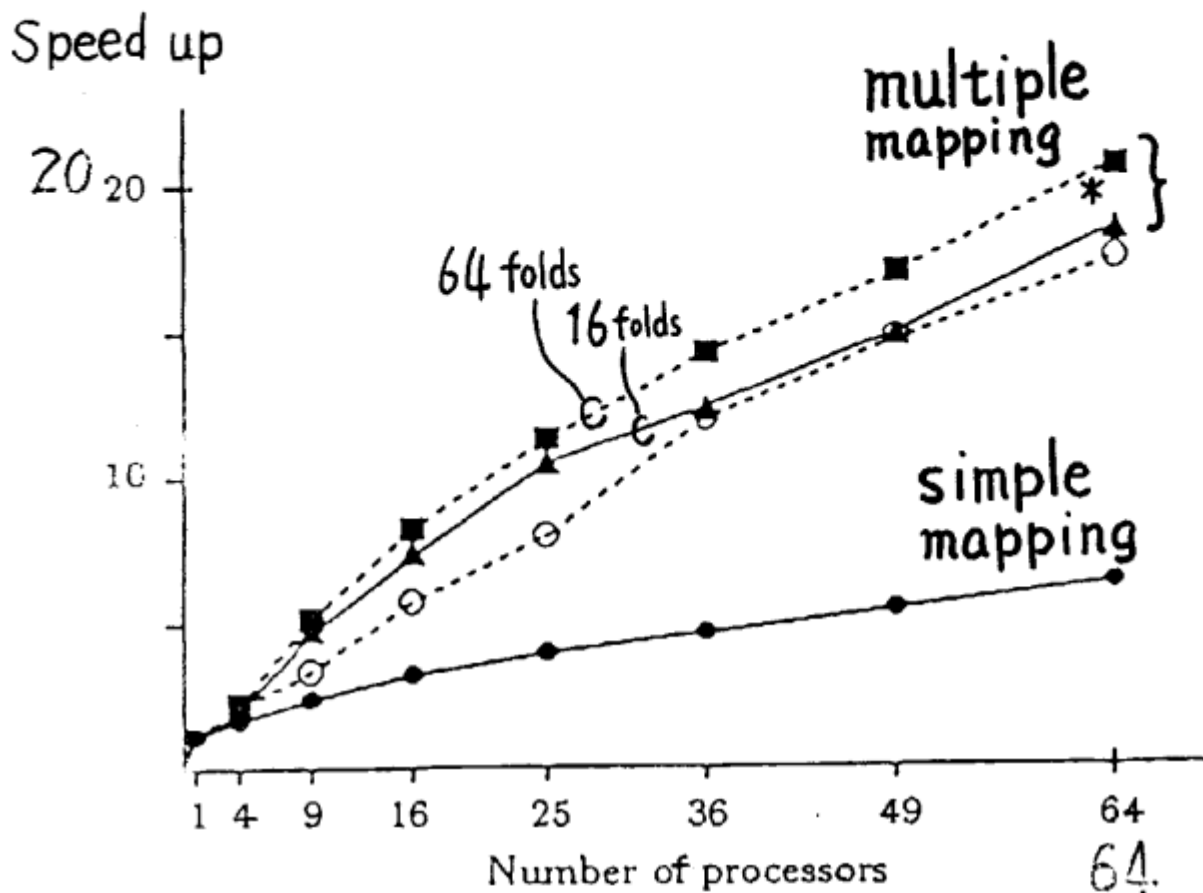
BESTPATH
PROBLEM
SOLVER
10000
NODES

- EXECUTE
- CHANGE PE
PARAMETER
- KEYBOARD
INPUT
- NETWORK
PARAMETER
- CLOSE
- EXIT

Start node : 582
Goal node : to all



multiple mapping (4-fold)



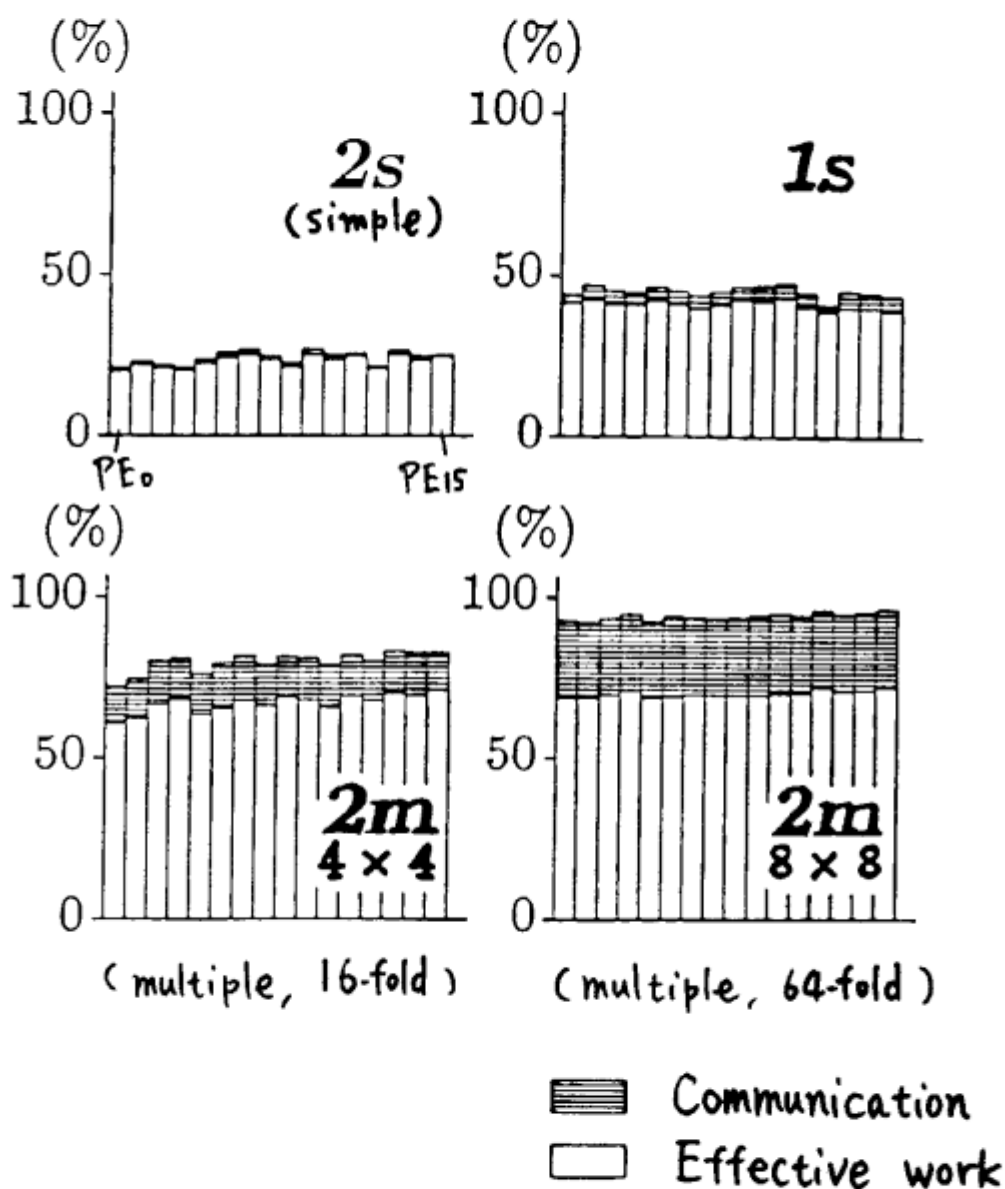
Shortest path finding (40,000 nodes)

- Simple mapping :
 \sqrt{n} speed up with n processors
- Multiple mapping: 2~3 times faster
- 70 times speed improvement from the first version (improvement of algorithm)

* Best execution time = 3 sec
 82

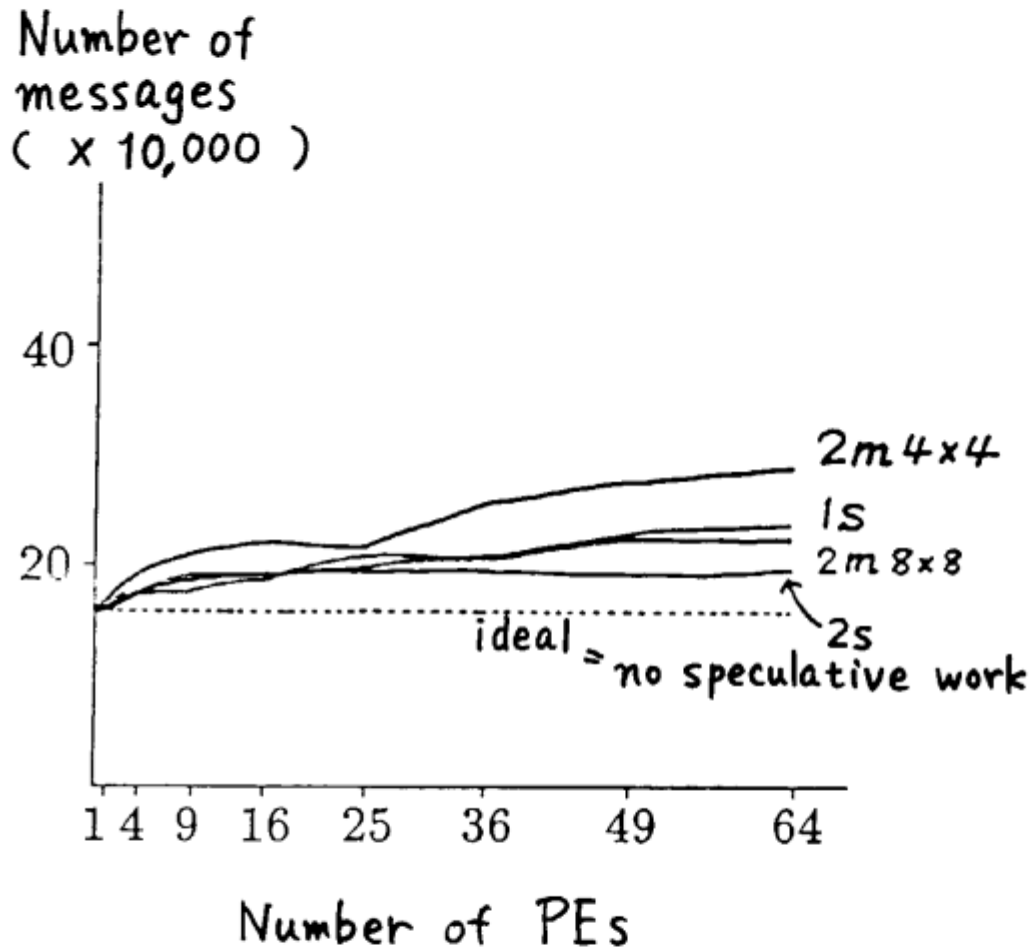
shortest path finding

Processor work rate (in case of 16 PEs)



shortest path finding

Speculative computations



Performance figures for Tsume-go

Problem 1.

Sequential	1696 moves 15 sec	
	Scheduling a	Scheduling b
Parallel (16 PEs)	1696 moves 2.9 sec <u>5.2 times</u> speed up	7901 moves 6.0 sec <u>2.5 times</u> speed up

Problem 2.

Sequential	21257 moves 210 sec	
	Scheduling a	Scheduling b
Parallel (16 PEs)	29269 moves 70 sec <u>3.0 times</u> speedup	48979 moves 31 sec <u>8.8 times</u> speedup

Problem 1 : The best move is the leftmost branch.

Problem 2 : The best move is a far right branch.

Speed up in PAX

Load allocation method	1 PE	16 PEs
method 1	1	0.45
method 2	1	2.4
method 3	1	3.0 *

A sentence with 106 words, 460 solutions

* Best execution time = 15 sec

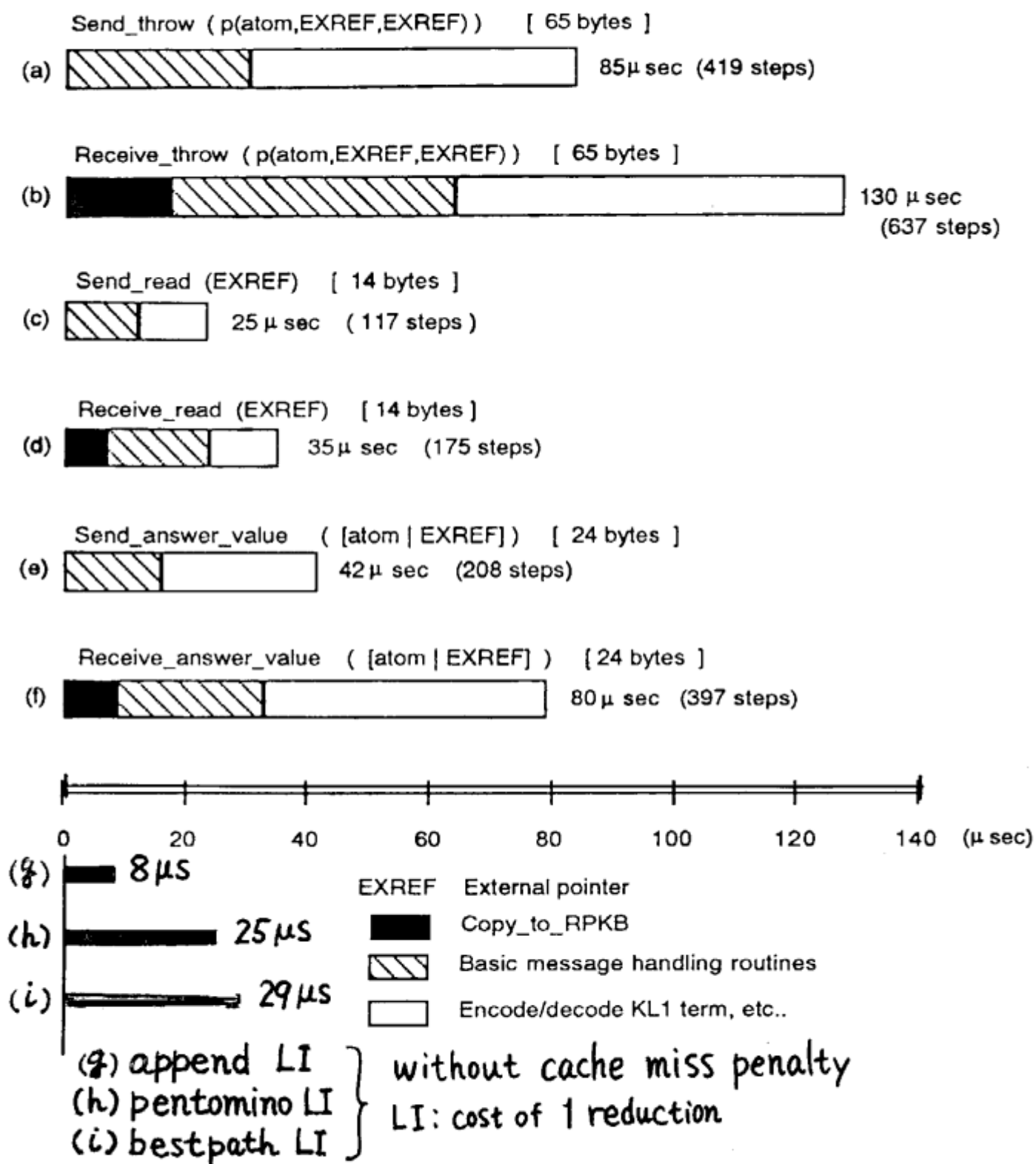
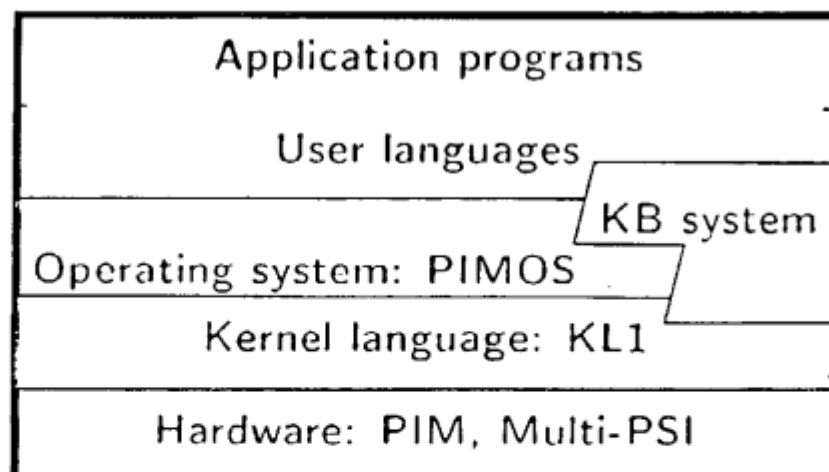


Figure 3: Message Handling Cost

34 KLIPS is used for bestpath instead of 23K

□ Conclusion and Future Plan



— Development of sample programs —

Development of first versions before FGCS '88

program	programmers	size (source lines)
PAX	3	8 K
Tsume-go	2	2.5 K
Packing piece puzzle	2	2.5 K
Best path problem	1	1.5 K

Development period 8/M ~ 11/E, 1988

— Experiences —

- Almost no synchronization/communication bugs
- KL1 is usable.
 - Suitable for process oriented programs
 - Suitable for experimentation on load distribution
- Priority control (with a lot of levels) is valuable.

Summary

R & D of parallel inference systems

- 1985 —GHC
- 1986 **Multi-PSI/V1, FGHC** 1K LIPS×6PE(FGHC)
—Small sample programs
- 1988 **Multi-PSI/V2, KL1** 150K LIPS×64PE(KL1)
2~5 M LIPS
—PIMOS/V1, Demonstration programs
- 1990 **PIM/p** 600K LIPS×128PE(KL1)
10~20 M LIPS
—PIMOS/V2, Application programs
- 1992 **Final PIM system** 1000 PE >100 M LIPS
—Final PIMOS, Large application programs

Continuous research on

- **Concurrent programming and Parallel processing** for large scale, loosely coupled MIMD machines

Future Plan

- Development of PIM/p system
(completed in 1990)
- Also final PIM system — 5G machine
- Improvement of PIMOS
- Continuous research on
Concurrent programming and
Load distribution/Scheduling schemes
- More practical and larger application programs

LSI CAD

Natural language processing

"Go" game solver (full system)

Theorem prover

Expert systems

Genetic information processing, etc.

Appendix A.

Parallel Software Development System and Application Programs

Title	Parallel Software Development System
Purpose	Research and development of parallel operating systems, parallel algorithm design and load distribution methods on an implementation of a concurrent logic programming language on a parallel processor
Outline & Features	<ul style="list-style-type: none"> • The Multi-PSI connecting up to 64 CPUs of the sequential inference machine (PSI) • A high performance distributed language implementation for a concurrent logic programming language (KL1) • A parallel operating system (PIMOS) for research and development of parallel software • Used to extend and optimize the KL1 language processor and the PIMOS for parallel inference machines (PIMs)
System Configu- ration	<p style="text-align: center;">Parallel software development system</p> <div style="text-align: center; border: 1px solid black; padding: 10px; margin: 10px auto; width: fit-content;"> <div style="border: 1px solid black; border-radius: 15px; padding: 5px; margin-bottom: 5px; width: fit-content;">R&D of parallel software in KL1</div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px; width: fit-content;">Parallel operating system PIMOS</div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 5px; width: fit-content;">Distributed language processor for logic programming language KL1</div> <div style="border: 1px solid black; padding: 5px; width: fit-content;">Parallel processor Multi-PSI</div> </div>

Title	Hardware for Parallel Software Development - Multi-PSI
Purpose	<p>The Multi-PSI hardware offers high processing power and useful functions for (1) research and development of parallel software and distributed processing mechanisms, and (2) designing PIM architecture.</p>
Outline & Features	<p>The CPU and memory of the compact version of the sequential inference machine, PSI, are used as the processing elements (PEs) of the Multi-PSI. PEs are connected to each other to form a two-dimensional mesh network by a specially designed message switching and automatic routing network controller. The system can be configured with up to 64 PEs in units of eight PEs.</p> <p>The compact version of the PSI, the PSI-II, is used as the front-end processor (FEP). Up to four front-end processors can be connected to the network to perform I/O functions for the Multi-PSI system.</p> <p>Tag architecture : 8-bit tag + 32-bit data PE control : Horizontal microprogram control Cycle time : 200 nsec (whole system synchronized) Main memory : 80 MB (16 MW)/PE max. Network channel : 5 MB/s Devices : 8K, 20K-gate CMOS gate array LSI, and others</p>
System Configu- ration	<p>The diagram illustrates the system configuration. On the left, a 'フロントエンド・プロセッサ' (Front-end processor) labeled '(PSI-II)' is shown. It is connected to a 'ネットワーク制御回路' (Network controller). The network controller is connected to a 2D mesh of '要素プロセッサ' (Processing element) blocks. The mesh is organized into two rows of four blocks each, with a bracket indicating '8 PEs' for each row. Dashed lines represent the network connections between the blocks.</p>

Processing element (PE)

The wide (53-bit) micro instruction architecture of PEs allows flexible experiments of KL1 language implementation. The tagged architecture enables efficient execution of logic programming languages. It is especially suitable for high level abstract machine instructions because tag manipulation and multi-way branching on tag values can be performed in parallel with an ALU operation.

Network controller

A PE communicates with four adjacent nodes through bidirectional channels, each nine bits wide. Messages received are stored in the Read buffer for the PE in the node, or forwarded to another node through the channel chosen by looking up the Path table with the destination address of the message.

A buffer of 48 bytes for each output channel is used for busy waiting synchronization with the adjacent node.

When a complete message is stored in the Read buffer, the PE of the node is notified of it (by a NWINT signal), and the PE will process the message as early as convenient for its internal processing.

Message sending from the PE is initiated automatically when a complete message is stored in the Write buffer of the network controller.

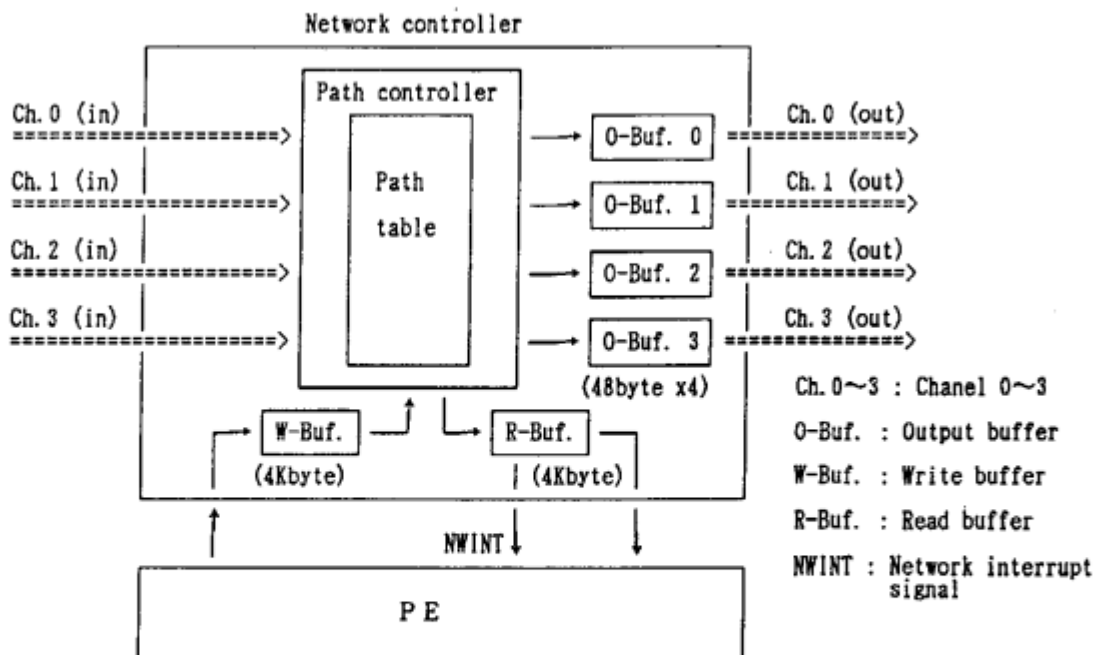


Figure. Network controller and Processing element (PE)

Hardware implementation

A PE is implemented on three printed circuit boards (PCBs) with nine 8K-gate CMOS gate array LSIs. The network controller is implemented on one PCB using two 20K-gate CMOS gate arrays.

One PE can have up to 80M bytes of main memory on four PCBs, each of which contains 20M bytes, using 1M bit dynamic RAMs.

One cabinet of the Multi-PSI contains eight nodes, and one system can have up to 64 nodes by connecting eight such cabinets. The entire system is synchronized by a single clock distributed to all cabinets.

Up to four front-end processors (FEPs) can be connected to the network for I/O. One of them, the master FEP, also performs console processor (CSP) functions, such as system start-up and diagnosis, through a specially devised maintenance path.

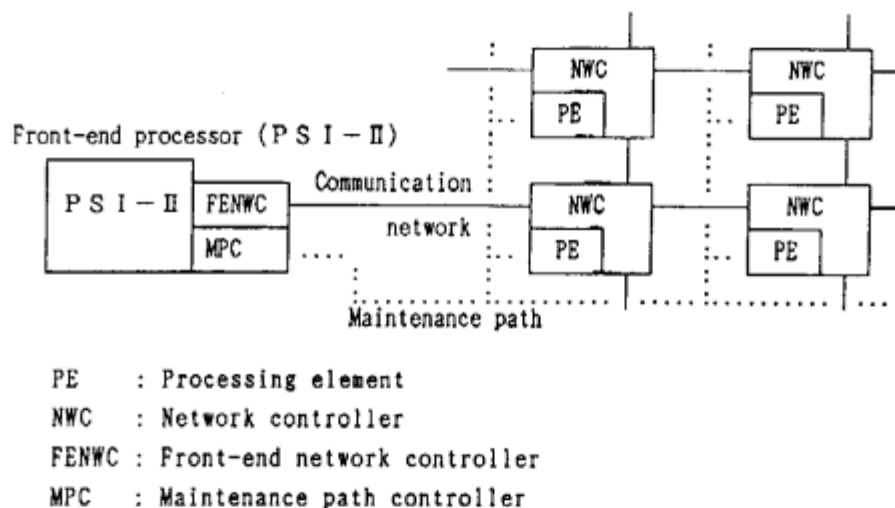
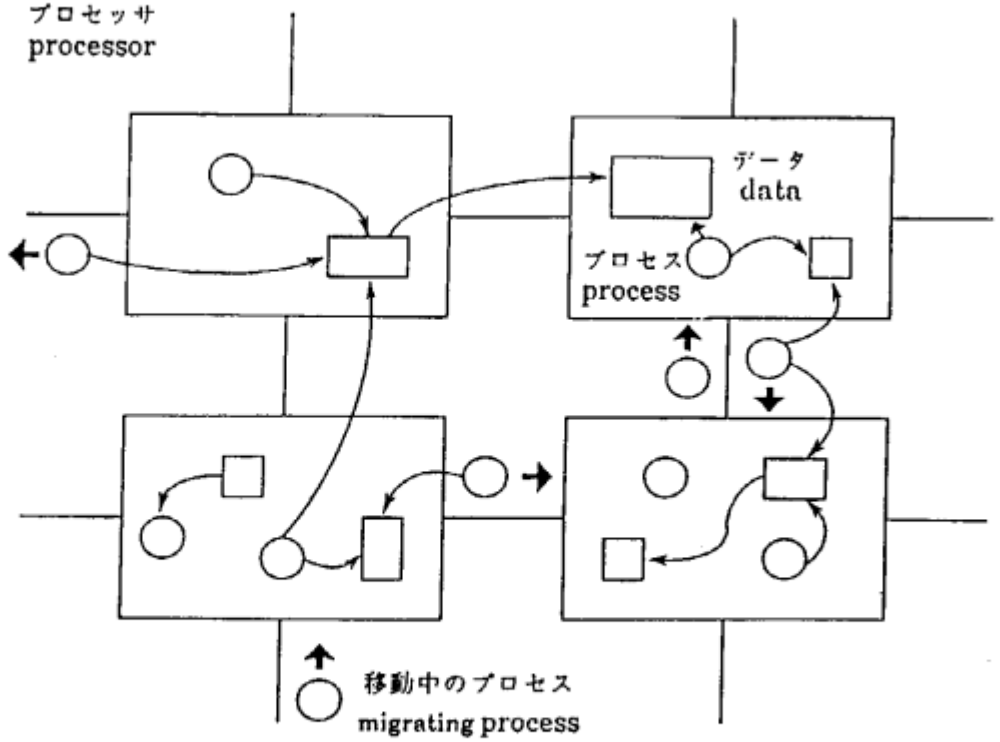


Figure. Multi-PSI and its front-end processor

Title	Distributed KL1 Implementation
Purpose	<p>The distributed KL1 implementation manages KL1 processes and data distributed over the network-connected processors of the Multi-PSI and executes KL1 programs efficiently.</p>
Outline & Features	<p>KL1 programs are compiled by an optimizing compiler into abstract machine instructions, which are executed by the microcode (150 KLIPS (Kilo Logical Inferences Per Second) per processor). The distributed KL1 implementation is designed to reduce the amount of inter-processor communication by utilizing the single-assignment property of KL1 and by various other techniques. Innovative intra- and inter-processor garbage collection schemes are implemented.</p>
System Configuration	 <p>The diagram illustrates the system configuration with four processors arranged in a 2x2 grid. Each processor is represented by a large rectangle containing various internal components and connections.</p> <ul style="list-style-type: none"> Top-left processor: Labeled "プロセッサ processor". It contains a circle at the top, a rectangle in the middle, and another circle at the bottom left. Arrows show data flow between these components and with the top-right processor. Top-right processor: Contains a rectangle labeled "データ data" at the top, a circle labeled "プロセス process" in the middle, and a square at the bottom right. It is connected to the top-left processor and the bottom-right processor. Bottom-left processor: Contains a square at the top left, a circle in the middle left, and a rectangle in the middle right. It is connected to the top-left processor and the bottom-right processor. Bottom-right processor: Contains a circle at the top left, a square in the middle left, and a rectangle in the middle right. It is connected to the top-right processor and the bottom-left processor. <p>At the bottom center, there is a legend: a circle with an upward arrow pointing to it, labeled "移動中のプロセス migrating process".</p>

1 KL1

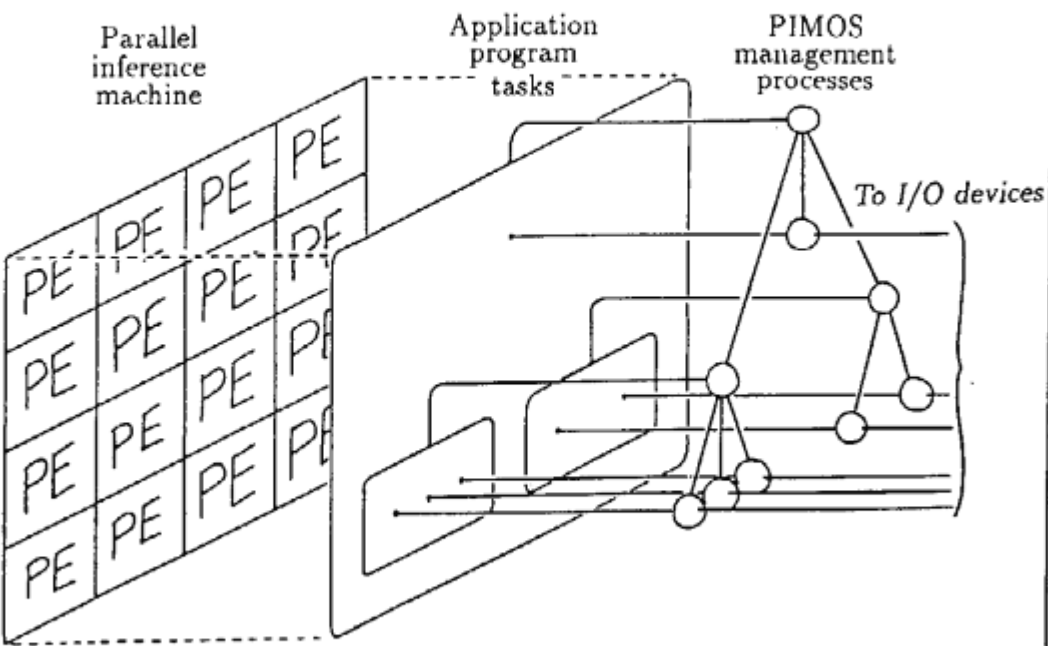
KL1 is a stream AND-parallel logic programming language. In stream AND-parallelism, concurrently executed processes that share data constitute the process structure in the form of cause-consequence chains. Unlike OR-parallelism and independent AND-parallelism, in which concurrently running processes do not communicate with each other, programs originally written for sequential machines cannot be readily executed in a stream AND-parallel manner, but complex cooperative problem solving can be best modeled in stream AND-parallelism.

2 Multi-PSI Architecture

There are two categories in parallel computer architectures: shared and non-shared memory architectures. In the shared memory architecture, processors communicate with each other by writing and reading shared memory; in the non-shared memory architecture; they communicate by sending and receiving messages over the communication channels. The shared memory architecture has the advantage of relatively low communication overhead, but the maximum number of processors is severely limited because of the memory access bottleneck. The advantage of non-shared memory architecture is its scalability. It was chosen for the Multi-PSI, since the machine is to serve as an experimental machine for the Parallel Inference Machine (PIM) project which aims at building a parallel machine with up to 1,000 processors. Programs on a non-shared memory machine, however, need to be designed with the problem of high communication overhead in mind.

3 Distributed KL1 Implementation

The task of KL1 implementation is to execute KL1 programs efficiently on the Multi-PSI. Algorithms have been developed to keep the amount of inter-processor communication as low as possible. Since the rate of memory consumption in KL1 programs is high, new techniques are employed in garbage collection (GC), reclamation of memory area that is no longer used. One is the Multiple Reference Bit (MRB) technique that uses one-bit pointer tags to recognize reclaimable data in a local processor, and another is the Weighted Export Counting (WEC) technique suited for inter-processor incremental GC. The implementation provides metaprogramming capabilities to support the operating system. They include the "shoen" facility — core of resource and task management, priority execution, and user-programmable load distribution mechanism.

Title	Parallel Inference Machine Operating System: PIMOS
Purpose	<p>The PIMOS aims at providing operating system facilities through which application programs can easily and fully utilize the processing power of the parallel inference machines.</p>
Outline & Features	<p>Described in KL1: The PIMOS is completely described in the concurrent logic programming language KL1. Making use of the <i>meta</i>-programming features of the KL1 language, the design of the PIMOS is independent of various hardware parameters, such as the number of available processors in the system.</p> <p>Single OS on multiple processors: The PIMOS is not an aggregate of independent operating systems on each processor, but one single integrated operating system. However, not only the application programs but also various parts of the PIMOS are executed on multiple processors in parallel. Computation and communication bottlenecks due to information centralization are avoided by distributing management information close to the application program tasks.</p> <p>Providing basic system functions: The PIMOS provides basic functions required in operating systems, such as management of execution, resource allocation and input/output devices. Additional services are planned.</p>
System Configuration	<p>All the demonstration programs shown on the Multi-PSI systems are operated under the supervision of the PIMOS.</p>  <p>The diagram illustrates the system configuration. On the left, a 4x4 grid of boxes represents the 'Parallel inference machine' (PE) units. Lines from these units connect to a central area labeled 'Application program tasks' and 'PIMOS management processes'. The 'PIMOS management processes' are shown as a hierarchical tree structure. Lines from this management structure lead to the right, labeled 'To I/O devices'.</p>

The functions of the PIMOS are demonstrated by operating the application programs in their demonstrations.

Stream communication and filters

All communication between the PIMOS and application programs is made through *streams*, which is one of the most advantageous features of the AND-parallel logic programming languages. *Filters* inserted to such streams handle abnormal termination of application programs and protect the operating system from failures in application programs.

Distributed processing by the resource tree

Each application program task or input/output device used in such tasks has a PIMOS management process associated with it. The *processes* in this case are quite light-weight ones provided by the microcode of the KL1 language processor, which correspond to *objects* in object-oriented programming languages.

In the PIMOS, tasks can be created inside a task, which, as a whole, forms a tree structure. Thus, the corresponding PIMOS management processes also form a tree structure. This tree is called the *resource tree*. All the PIMOS management information is distributed to the management processes, which are the nodes of this resource tree.

The management process is allocated on the processor where the corresponding application program task is running (when the task uses multiple processors, the processor on which the request was made to create a new task or to open an I/O device). This distribution of management processes also distributes the management overhead to multiple processors. Also, by placing management information near the application program tasks, communication congestion to a single processor, as expected when all the information is centralized in a single table, is avoided and the total amount of inter-processor communication is minimized.

Shell

Execution of an application programs supervised by the PIMOS can be initiated by invoking it from the command interpreter (*shell*). Based on the functions equipped by the PIMOS, the shell provides the following features for the user.

- Starting, suspending, restarting and aborting the execution of jobs
- Controlling foreground and background jobs
- Defining the standard input/output of jobs
- Inter-task communication via *pipes*
- Controlling resource allocation to jobs
- Handling exceptions in application programs

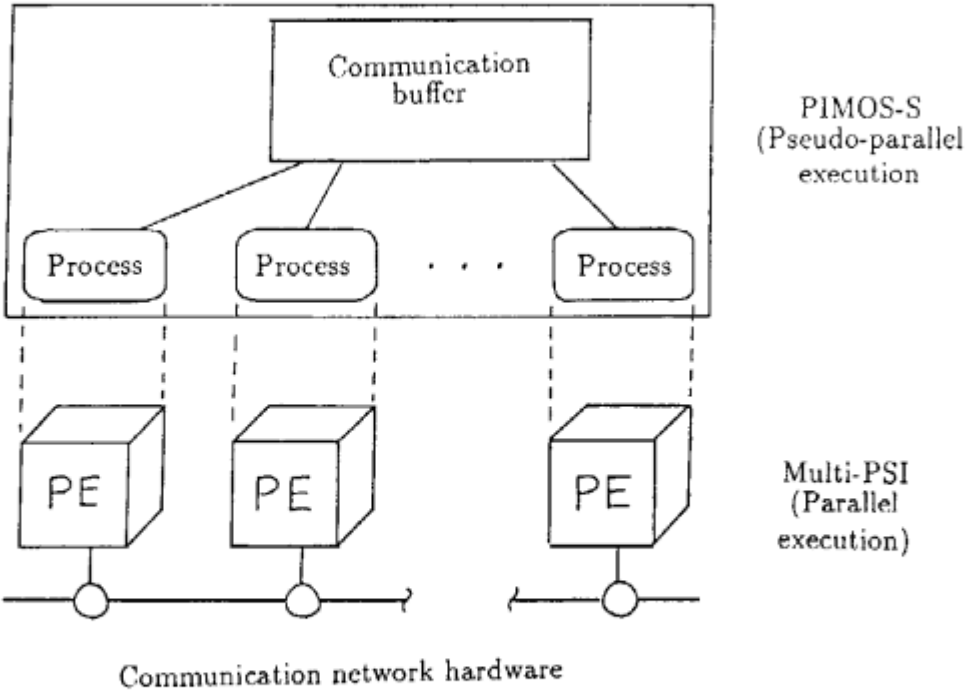
Utility programs invoked from the shell provide monitoring of the status of task execution and allocation of resources such as input/output devices.

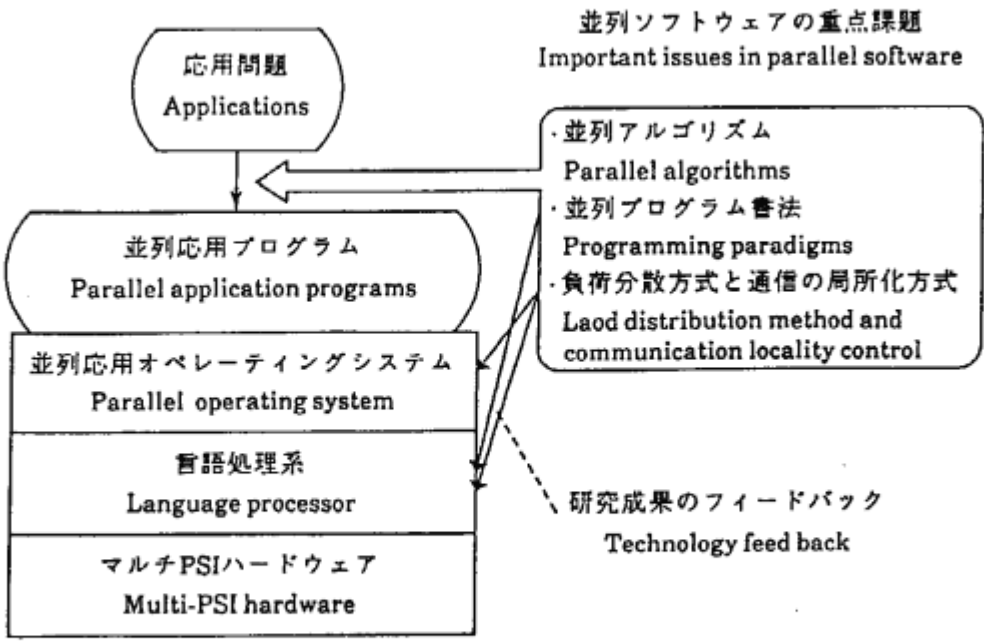
Input and output devices

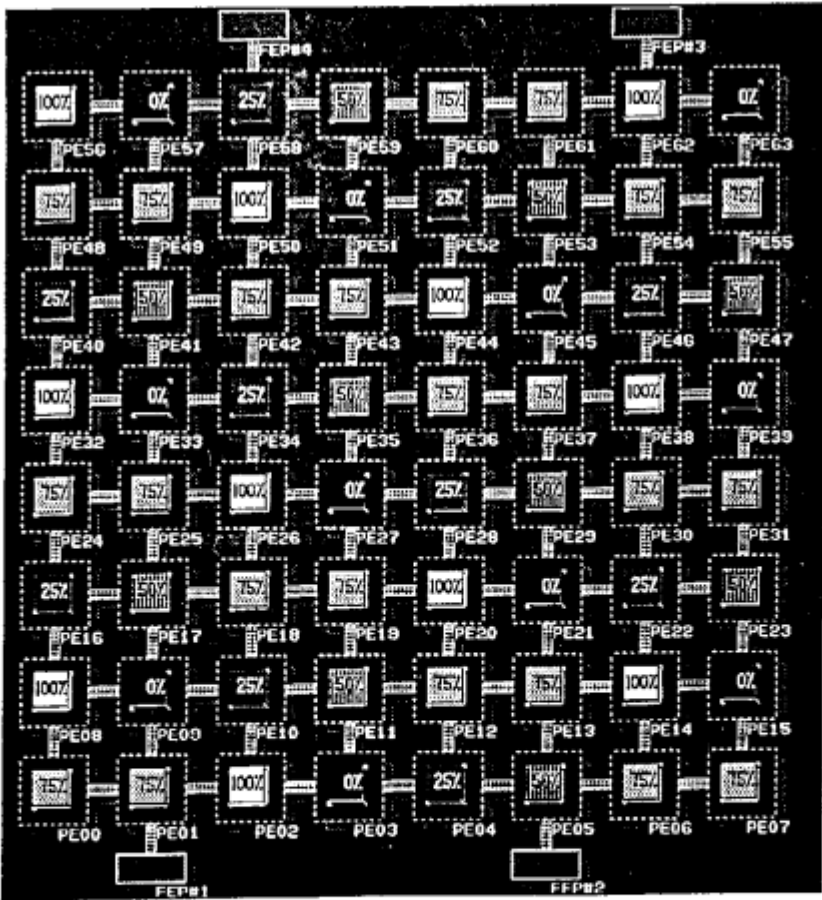
The PIMOS currently provides functions to access high-level I/O features of the operating system (SIMPOS) on the front-end processor (PSI-II), such as files and display windows, from KL1 programs. Various display facilities used in demonstration programs are operating on the front-end processor, controlled by KL1 programs on the Multi-PSI through the standard interface provided by the PIMOS.

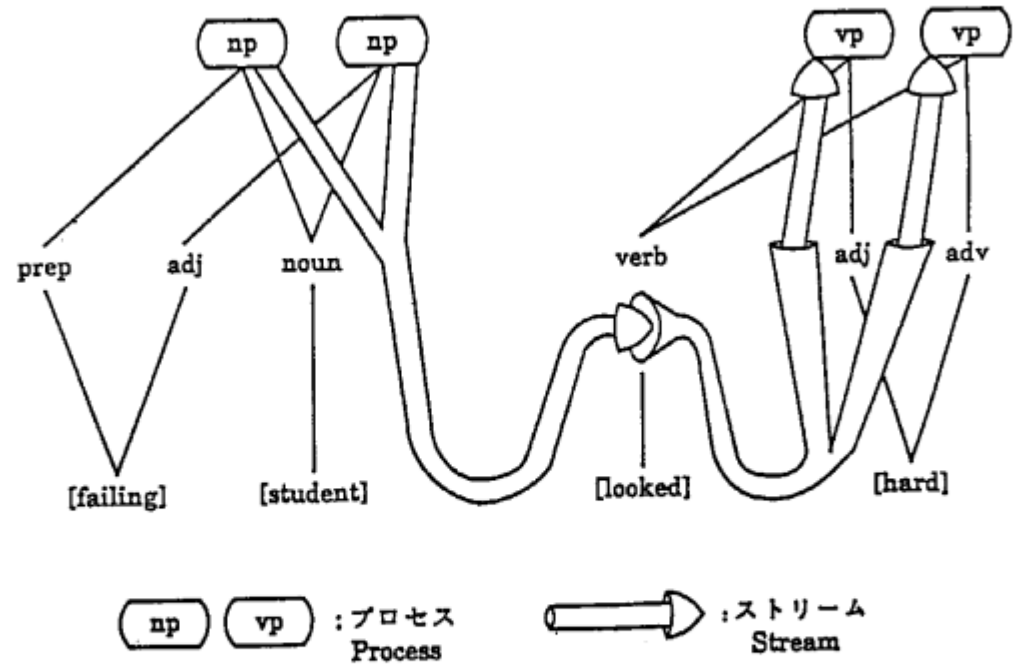
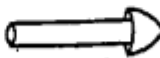
Further details of the design of the PIMOS are given in the following paper, presented at the session ICOT-SS1 (3:30 p.m., Wednesday, Nov. 30th).

"Overview of the Parallel Inference Machine Operating System"

Title	Parallel Software Development Environment on PSI-II: PIMOS-S
Purpose	The PIMOS-S provides a parallel software development environment qualitatively equivalent to the Multi-PSI system on much less costly PSI-II workstations.
Outline & Features	<p>Compatibility: The language and operating system features provided are fully compatible with the Multi-PSI system. Parallel software developed on PSI-II workstations using PIMOS-S can be executed on the Multi-PSI without any change.</p> <p>Pseudo-parallelism: Processing elements of the Multi-PSI running in parallel are simulated by pseudo-parallel processes.</p> <p>High performance: The same microcode as used in the Multi-PSI system is utilized. Thus, the provided performance is equivalent to its one processing element.</p> <p>Decent debugging environment: A debugging environment qualitatively equivalent to the Multi-PSI system is provided. It is even better in some aspects; e.g., non-determinacy due to parallel execution can be eliminated by applying pseudo-random scheduling.</p>
System Configuration	 <p>The diagram illustrates the system configuration for both PIMOS-S and Multi-PSI. At the top, a box labeled 'Communication buffer' is connected by lines to three rounded rectangular boxes labeled 'Process'. These 'Process' boxes are grouped under the label 'PIMOS-S (Pseudo-parallel execution)'. Below each 'Process' box, a dashed vertical line connects to a 3D cube labeled 'PE' (Processing Element). These 'PE' boxes are grouped under the label 'Multi-PSI (Parallel execution)'. At the bottom, a horizontal line with three circular nodes represents the 'Communication network hardware'. Each 'PE' box is connected to one of these nodes. The overall structure shows that PIMOS-S simulates parallel execution using multiple processes connected to a single communication buffer, while Multi-PSI uses multiple physical processing elements connected to a shared communication network hardware.</p>

Title	Parallel Software Research
Purpose	To develop novel software technology essential for making large scale network-connected parallel machines work efficiently for various application fields.
Outline & Features	<p>Almost every software technology in programming and execution of application programs must be reconstructed for parallel processing to achieve a satisfactory execution efficiency of large scale parallel machines. Especially those listed below are essential.</p> <ul style="list-style-type: none"> • To develop algorithms with much parallelism without increasing the amount of required computation • To accumulate programming styles or paradigms which give guidelines to parallel programming for various types of large scale problems • To develop load distribution methods for balanced work load and high communication locality <p>The research has just started by implementing parallel programs for several types of applications (e.g. the programs demonstrated). The results of the research will be fed back to improve the functions of the operating system and the language processor.</p>
System Configu- ration	 <p>The diagram illustrates the system configuration and feedback loop for parallel software research. It features a vertical stack of four components: '並列応用プログラム' (Parallel application programs), '並列応用オペレーティングシステム' (Parallel operating system), '言語処理系' (Language processor), and 'マルチPSIハードウェア' (Multi-PSI hardware). Above the stack is an oval labeled '応用問題' (Applications). Arrows show a flow from 'Applications' to 'Parallel application programs', and from 'Parallel application programs' to 'Parallel operating system', 'Language processor', and 'Multi-PSI hardware'. A separate box on the right, titled '並列ソフトウェアの重点課題' (Important issues in parallel software), lists four key areas: '並列アルゴリズム' (Parallel algorithms), '並列プログラム書法' (Programming paradigms), '負荷分散方式と通信の局所化方式' (Load distribution method and communication locality control), and '研究成果のフィードバック' (Technology feed back). Arrows indicate that these issues are addressed within the 'Parallel application programs' and 'Parallel operating system' layers, with a dashed arrow pointing from the '研究成果のフィードバック' box back to the 'Language processor' and 'Multi-PSI hardware' layers.</p>

Title	Processor Work Rate Measurement Program for The Multi-PSI — Performance Meter —
Purpose	To evaluate load distribution of user programs through real time visual display of processor work loads.
Outline & Features	Work rate of each processor of the multi-PSI is displayed graphically at real time. Display interval is two seconds (can be changed). The measurement program is written in KLI.
System Configu- ration	<p>The program is constructed of measuring processes allocated to all processors and a management process which gathers the measurement results. The results are packed and sent to a display device on the front-end processor via the operating system, PIMOS.</p>  <p>work rate</p> <p>processor number</p> <p>Display example of the Performance meter</p>

Title	Parallel Application Program (1): Natural Language Parser
Purpose	The natural language processing system originally used in DUALS is implemented in parallel using <i>layered stream method</i> . A load distribution method is studied and evaluated for it in order to realize a very fast natural language parser.
Outline & Features	<p>Outline The PAX analyzes natural language sentences, makes parse trees, and displays them. Parsing is performed by bottom-up, parsing messages between processes which correspond to each node of parse tree.</p> <p>Parsing program The parsing program is generated from definite clause grammar by a translator, which adds load distribution code automatically.</p> <p>Problem characteristics All solutions are searched in the parsing algorithm. However, the result is not always unique because of the ambiguity of grammar, particularly in natural language.</p> <p>Programming paradigm The <i>layered stream method</i>, a broadly applicable paradigm for all solution search problems, is used.</p> <p>Load distribution A load allocation which minimizes inter-PE communication is examined.</p>
System Configuration	 <p>The diagram illustrates the parse tree and layered stream communication for the sentence "[failing] [student] [looked] [hard]".</p> <ul style="list-style-type: none"> Parse Tree Structure: <ul style="list-style-type: none"> The root node is a large vertical oval. It branches into two main sections: a left section for the noun phrase and a right section for the verb phrase. Left Section (Noun Phrase): <ul style="list-style-type: none"> Roots into two np (noun phrase) nodes. The first np node branches into prep (preposition) and adj (adjective), which point to the words "[failing]" and "[student]" respectively. The second np node branches into noun, which points to the word "[student]". Right Section (Verb Phrase): <ul style="list-style-type: none"> Roots into a verb node and another large vertical oval. The verb node points to the word "[looked]". The second large oval branches into two vp (verb phrase) nodes. The first vp node branches into adj (adjective) and adv (adverb), which point to the words "[hard]" and "[hard]" respectively. The second vp node branches into adv (adverb), which points to the word "[hard]". Legend: <ul style="list-style-type: none"> np vp : プロセス Process  : ストリーム Stream Caption: <p>解析木と階層化されたストリーム通信 Parse tree and layered stream communication</p>

Configuration

The PAX is a natural language processing system. It analyzes natural language sentences, makes parse trees, and displays them in the window. This system is divided into three parts, the input part of sentences, the analysis part (parser), and the output part of results (parse trees).

The parser is the main part of this system and only this part runs on the Multi-PSI in parallel. The parsing program is generated from definite clause grammar by a translator.

Both the input program and the output program are written in ESP and run on the front end processor PSI-II machine.

Problem characteristics

For parsing, a bottom-up breadth-first algorithm called left-corner parsing is used. Generally, all solutions are searched in the parsing algorithm. However, the result is not always unique because of the ambiguity of grammar, particularly in natural language.

In general, we use the following way to write all solution search programs in languages that do not support the backtracking mechanism, such as KL1. First, we gather all solutions as a set, then the set is sieved gradually to select the adequate ones. For the PAX, the *layered stream method*, a broadly applicable paradigm for all solution search problems, is used.

Algorithm

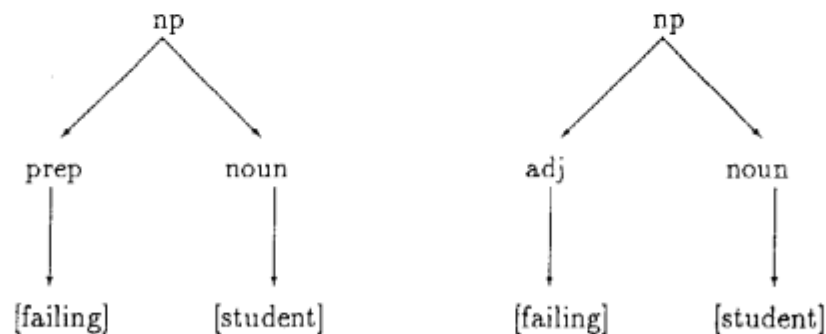
The PAX analyzes the input sentence by using the bottom-up parsing method with generating processes which correspond to each node of the parse tree, based on each word. The analysis progresses by messages passing between two adjacent processes through the stream. Each message contains partial results (partial parse trees).

We explain it by using an example shown in the figure of first page. This example analyzes the sentence "failing student looked hard" with the following grammar.

s	--> np, vp.	adj	--> [failing].
np	--> adj, noun.	adj	--> [hard].
np	--> prep, noun.	prep	--> [failing].
vp	--> verb, adj.	adv	--> [hard].
vp	--> verb, adv.	verb	--> [looked].
		noun	--> [student].

First, the input words generate processes with streams which connect two adjacent words.

This picture is a snapshot of parsing. The noun phrase (*np*) and verb phrase (*vp*) processes have already been made, and each *np* process (there are two) sends partial parse trees that it made to *vp* processes through the stream. The partial parse trees are as follows:



There are two *np* and *vp* processes, because there are two candidates for *failing*: adjective (*adj*) and preposition (*prep*). *hard* also has two candidates, adjective (*adj*) and adverb (*adv*).

Communication between an *np* process and a *vp* process uses the layered stream that was connected when processes were generated from the words.

1. Each *np* process sends its partial parse trees through the stream between *student* and *looked*.
2. Both the *adj* process and *adv* process are adjacent to the *verb* process and are connected with the stream between *looked* and *hard*. When they communicate with each other, the *verb* process puts the stream connecting *student* and *looked* into the message.
3. On receiving messages from the *verb* process, the *adj* process and *adv* process generate the *vp* process above them. The stream between *student* and *looked* are got from the message and given to the *vp* processes.
4. Each *vp* process receives the stream between *student* and *looked*, and can receive the message from the *np* process through it.

In this way, the *np* process communicates with the *vp* process. Each *vp* process receives two partial trees from the *np* processes, and constructs its trees and generates the last node *sentence*. (This completes parsing.) It means that four parse trees are made.

Load Balancing

Two kinds of parallel execution are applicable to the PAX:

1. Making adjacent nodes can be done in parallel. In the above example, the processes to make an *np* process and a *vp* process can be executed in parallel.
2. When there are two or more node candidates, each process corresponding to node can be executed in parallel. In the above example, there are two candidates for *failing* (*prep* and *adj*), and they can be executed in parallel.

If all the processes mentioned above are distributed to different processors, too much inter-processor communication occurs. Because the messages from some candidates are merged into one stream, which is sent some candidates again. That is *N* to *one* to *M* communication occurs.

In order to minimize inter-processor communication, we use following load balancing method. Processes that receive a message from the same stream are executed on the same processor.

First, processes that correspond to each word are allocated to different processors. A process corresponding to a word puts its processor number into the head of a stream which connects the process and a process of next word. The process of next word reads this stream and moves to the processor that is designated by the processor number. In this way, inter-processor communication decreases to *N* to *one*.

Demonstration

To demonstrate the PAX, we analyze some sentences as follows. The PAX analyzes the sentences according to the grammar that appears in "Oxford Advanced Learner's Dictionary of Current English".

- Display the result of analysis (parse trees).
- Show the load balancing during analysis by using the performance meter.
- Show the change of processing time and load balancing by analyzing sentences of different length.
- Analyze two or more sentences in parallel.
- Show the speed up ratio by parallel execution. Processing time for a sentence is measured twice, changing the number of processors used.
- Compare the processing time with that of a sequential processing system.

Title	Parallel Application Program (2): Tsumego Solver
Purpose	A parallel algorithm, programming paradigm and load distribution method for the game tree search problem are studied and evaluated implementing a tsumego solver.
Content & Feature	<p>The solution (life, death or "ko") is calculated for a given tsumego problem and the first move is made.</p> <p>Characteristics of the problem: Game tree search (to leaf nodes)</p> <p>Algorithm: The alpha-beta pruning method is modified for parallel execution. The search tree is expanded in parallel.</p> <p>Priority is attached to each sub-tree controlling the execution order.</p> <p>Load distribution: Large grain processes are allocated at random in one method and allocated to idle processors in the other.</p>
System Configu- ration	<p>詰め碁問題 Tsumego problem</p> <p>詰め碁のゲーム木 Game tree in the problem</p> <p>ノードプロセス Node process</p> <p>並列化αβ枝刈り法 Parallel alpha-beta pruning</p> <p>あらかじめ決められた深さ Specified depth</p> <p>大粒度プロセス Large grain processes</p> <p>逐次αβ枝刈りを実行するプロセス群 Each process executes sequential alpha-beta pruning.</p> <p>部分問題 Sub-problem</p> <p>PE_n PE_m</p> <p>High ← Priority → Low</p>

1 Overview

A parallel algorithm, scheduling and load distribution method for the game tree search problem are studied. The alpha-beta pruning method is modified for parallel execution. Priority is attached to all the node processes to realize an efficient pruning for the search tree. Processes are allocated dynamically in order to balance work loads of all processors.

2 Tsumego problem and solver

Tsumego problem is to determine life, death or tie of the surrounded stones given the Go board state (placement of white and black stones) and whose turn is next. The program shows the first move according to the result.

The program is essentially a game tree search. In the search, the alpha-beta pruning method is used with some modification for parallel execution.

Exhaustive search down to leaf nodes is made in this solver.

3 Alpha-beta pruning for parallel execution

In the conventional alpha-beta search, the game tree search is done in depth first manner. Since the result of a subtree search is used for pruning searches of other part of the game tree, the alpha-beta search has a sequential bottleneck. Therefore, the alpha-beta pruning method should be modified for parallel execution.

If the game tree search were done in purely parallel breadth first manner, there would be no pruning of search space. A priority control is used to realize an efficient pruning for the search tree.

4 Algorithm

The game tree is expanded in one master processor to a certain depth using the parallel alpha-beta pruning method. Below the depth, the conventional sequential alpha-beta search is executed. Each sequential alpha-beta search is performed by a large grain process, which is the unit of distributing computational load to processors. The parallel alpha-beta search is as follows.

1. Make a move for the given Go board situation. Pick up stones to be captured if any; Trace of adjacent same colored stones and if they are completely surrounded by enemy stones, they are captured.
2. Make judgment whether the game tree is expanded to a certain depth. When the game tree is expanded to a certain depth, sequential alpha-beta search is executed.
3. When not expanded, the search is continued spawning children node processes.

Each node process tests conditions for branch pruning or termination, exchanging information through streams. The information for renewal of alpha and beta values flows downwards, values of terminating nodes upwards, and termination commands caused by pruning downwards.

5 Scheduling using priority

In the parallel alpha-beta search, following three kinds of scheduling are tried.

- (a) The moves of the first player are sequentially searched.
- (b) The searches of the first player's moves are given priorities so that left-hand tree searches always prior than right-hand ones. The moves of the second player are given the same priorities.
- (c) Each search of the first player's moves is given individual priority according the individual game tree. The moves of the second player are given the same priorities.

Scheduling (a) is closer to the sequential alpha-beta search than (b) and (c), and has less parallelism.

It is expected that for a problem instance where the pruning effect in the sequential alpha-beta search is large, scheduling (a) will do well, while (b) and (c) will have good speedup for a problem instance in which the pruning effect of sequential search is small.

6 Load distribution

On-demand dynamic load distribution is used in order to balance work loads of all processors.

When a processor becomes idle, it sends a message requesting a new process to the master processor in which the game tree is expanded in parallel. On receiving the message, the master processor distributes a process as the response to the message.

7 Outline of the demonstration

1. Sequential alpha-beta search is tried to the problem where the best move is searched first.
2. Parallel alpha-beta search using 16 PEs is tried to the same problem above. Both scheduling (a) and (b) are used.
3. Parallel alpha-beta search using 16 PEs is tried to the problem where the best move is searched after many moves.
4. Using scheduling (c), parallel alpha-beta search is tried to the same problem above.

Title	Parallel Application Program (3): Packing Piece Puzzle
Purpose	Parallel algorithms, programming paradigms and load distribution methods for two different problems are studied and evaluated by implementing solvers for packing piece puzzles in two different ways.
Outline & Features	<p>The following two different solvers are demonstrated.</p> <p>1 Exhaustive search</p> <p>Program structure: An OR-Parallel exhaustive search by forking processes at each alternative choice, forming a tree structure.</p> <p>Load distribution: Semi-static load allocation suitable for OR-parallel problems.</p> <p>2 Applying neural network simulation†</p> <p>Program structure: Simulation of a neural network by exchanging messages between processes corresponding to neurons.</p> <p>Load distribution: Experiments on load allocation methods suitable for communication oriented problems.</p> <p>†Based on the Gaussian Machines neuron model (in cooperation with Anzai Laboratory, Keio University).</p>
System Configuration	<p>Packing piece puzzle [Applying neural network Simulation]</p> <p>○ and ● are neurons which stand for possible allocations for each piece. Inhibitory links of neuron A. All other neurons also have inhibitory links. ● represents fired neurons having converged in a solution.</p>

1 Overview

In the demonstration, packing piece puzzle of 10 pieces(Fig.1) is solved with different number of processing elements (PEs), and speedup by parallel execution and effectiveness of load balancing are shown.

The demonstration is carried out as follows.

- Program is executed on 1 processor, and execution time is displayed.
- Program is executed on 16 processors, and execution time is displayed.
- Load balancing can be observed by the performance meter.
- Near-linear speedup is obtained.

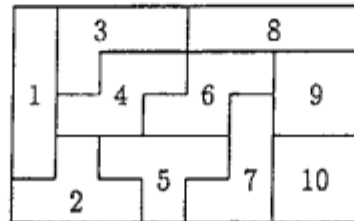


Figure 1: Packing Piece Puzzle

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, 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 pack one piece. 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.

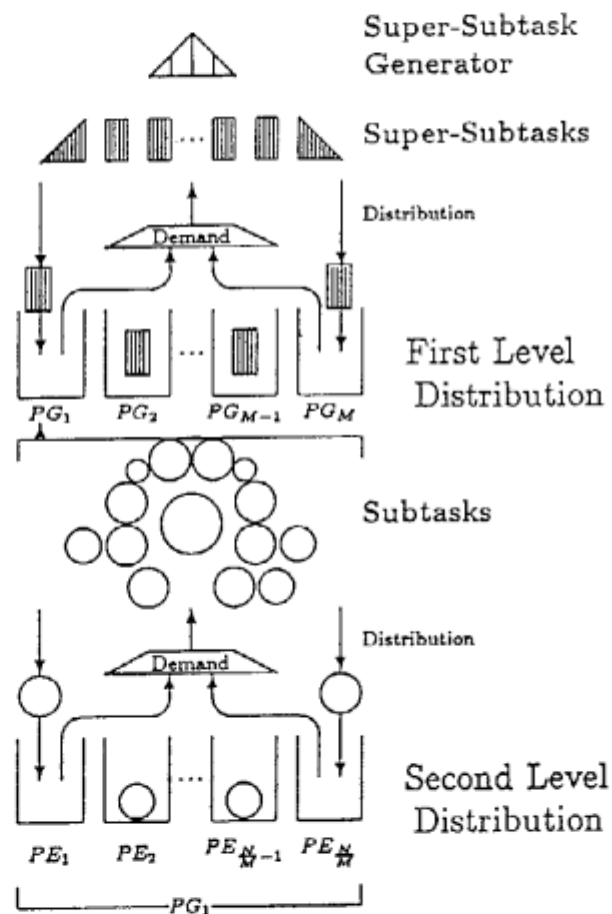


Figure 2: Structure of Multi-Level Load Balancing

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 is getting 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.

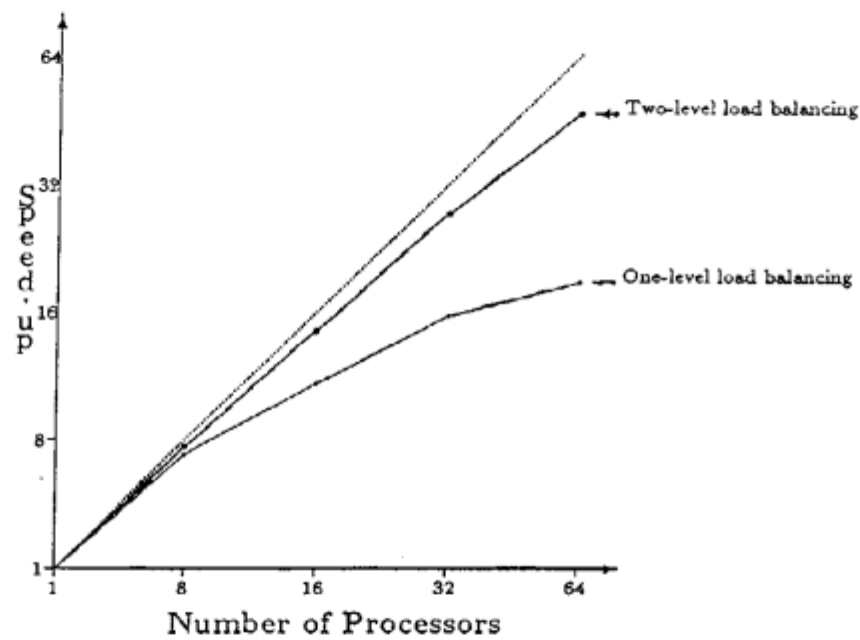


Figure 3: Speedups

5 Conclusion and Future Works

This scheme is efficient not only for OR-parallel search problems, but also applicable to some types of search problems such as alpha-beta pruning problems, which does not involve frequent inter-processor communication. Applying the multi-level load balancing scheme to such programs is our future works.

Title	<h1>Parallel Application Program (4): Shortest Path Problem Solver</h1>
Purpose	<p>A parallel algorithm, programming paradigm and load distribution method for the best solution search problem are studied and evaluated by implementing a shortest path problem solver.</p>
Outline & Features	<p>The single-source shortest path problem is to search for the minimum cost paths between a given start node and all other nodes of a network in which each network branch has a non-negative cost. Large networks with tens of thousands of nodes are generated using random numbers as the test data.</p> <p>Type of the problem : Best solution search.</p> <p>Algorithm : Processes corresponding to each network node exchange messages with each other. Each message contains path and cost from the start node. Priority is attached to each message so that a message with lower cost is sent earlier than a message with higher cost. Each node remembers the shortest path notified by the messages arrived so far and its cost.</p> <p>Programming technique : A message is represented by a process so that a message has a priority.</p> <p>Load distribution : Making more processors work for the part of the network where communication is dense.</p>
System Configu- ration	<div data-bbox="438 1265 997 1724"> </div> <div data-bbox="726 1444 1340 1915"> <p>• Algorithm</p> <p>Message1 will be sent earlier than message2.</p> </div>

Outline

The single-source shortest path problem is to find the minimum cost paths between a given start node and all other nodes of a network in which each network branch has a non-negative cost. In the demonstration, the network consists of about ten thousand nodes and is generated using random numbers.

In the demonstrated program, processes are generated for each network node and computation is performed by exchanging messages between them. The order of required computation with this algorithm is smaller than that with the algorithm in which processes are forked for each candidate path. Using priority control, efficient pruning for the search branches is done. As a result of that, the program works in the same order of computational complexity as well-known Dijkstra's algorithm.

Algorithm

A message contains the path from the start node to the receiver node and its cost. The computation is initiated by sending a message with an empty path and zero cost to the start node. All the nodes remember the minimum cost to reach the node notified by the messages received so far. Initially, the cost remembered by all the nodes is infinite (Figure 1).

When a message arrives at a node and the cost notified by the message is smaller than the minimum cost remembered in the node, the new cost is saved and messages with better paths and costs are sent further to the adjacent nodes (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 message means a creation of a message process, while receiving message means an execution of a message process. Each message process has a priority decided by the cost. Thus, a message with a lower cost is received earlier than a message with a higher cost.

When all the messages on the network are discarded, each node has the shortest path from the start node and its cost.

Load Balancing

The heaviest part of the processing is communication, and the communication is initiated at the start node and propagates gradually to the whole network in waves.

The program tries to balance the load based on the following two ideas.

- Divide the network into sub-networks and distribute processes for sub-networks to distinct processors.
- Make more processors work for the part of the network where communication is dense.

Mapping Strategies

The following three mapping strategies are tried. In each mapping, $p = q^2$ processors are employed.

Two-Dimensional Simple Mapping

Divide the network into $q \times q$ sub-networks and map each sub-network onto the corresponding processor.

Two-Dimensional Multiple Mapping

Divide the network into k super-sub-networks, each of which is again divided into p sub-networks just as in the two-dimensional simple mapping. Each processor is responsible for k sub-networks, each one from each super-sub-network.

One-Dimensional Simple Mapping

Divide the network simply as p narrow rectangular strips and map them onto the processors.

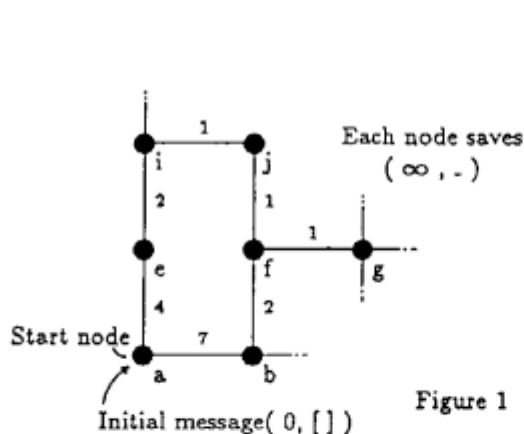


Figure 1

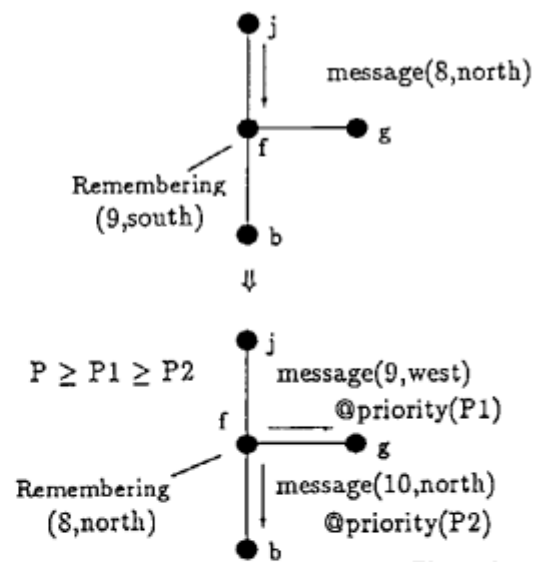
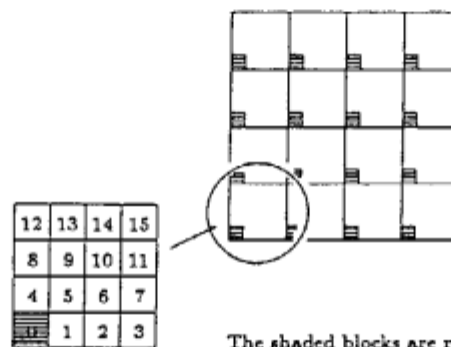


Figure 2

12	13	14	15
8	9	10	11
4	5	6	7
1	2	3	

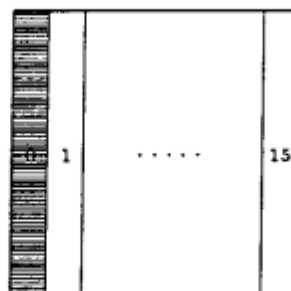
The shaded block is mapped onto processor 0.

Figure 3: The decomposition of a graph for the two-dimensional simple mapping



The shaded blocks are mapped onto processor 0.

Figure 4: The decomposition of a graph for the two-dimensional multiple mapping



The shaded block is mapped onto processor 0.

Figure 5: The decomposition of a graph for the one-dimensional simple mapping

Appendix B.

General Information of Parallel Inference Systems Research in FGCS Project –The FGCS Computing Architecture–

THE FGCS COMPUTING ARCHITECTURE

Kazuo Taki

Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan

ABSTRACT

In the fifth generation computer systems (FGCS) project of Japan, *logic programming* and *parallel processing* are adopted as the principles of both software and hardware system development. Their amalgamation, *parallel inference systems*, is being investigated at ICOT.

Both hardware and software development have been carried out based on the kernel language (KL1) as the software-hardware interface. KL1 is a concurrent logic programming language extended from flat GHC. The hardware development target is a parallel inference machine (PIM) with about 10^3 element processors. A smaller scale prototype machine, PIM/p, with about 10^2 processors is under development. The processors are dedicated for efficient execution of KL1 programs on a distributed implementation of the KL1 language processor.

One major software project is the development of a common operating system for all the parallel inference machines, named PIMOS (parallel inference machine operating system), which prototype has been working. Several program development environments have also been prepared. The Multi-PSI, another inference machine prototype, is one such powerful environment, which connects up to 64 personal sequential inference machines (PSI-II) relatively loosely. Further improvements of the PIMOS, and research on various concurrent algorithms and load distribution schemes are in progress on the Multi-PSI.

1 INTRODUCTION

The Japanese fifth generation computer systems (FGCS) project aims at building a prototype of a high performance knowledge information processing system (KIPS). The project spans ten years, from April 1982 to 1992. One of the principal functions of KIPS is its highly parallel inference feature. The target of the hardware system is a highly parallel inference machine (PIM) with about 10^3 processing elements and with an inference speed of more than 10^8 LIPS (logical inferences per second).

All R&D around the PIM (Uchida et al. 1988) has

been based on a concurrent logic programming language called KL1 (Chikayama et al. 1988), which is an extension of Flat GHC (Ueda 1986). Machine hardware and the language processor have been carefully designed for efficient parallel execution of KL1. The parallel inference machine operating system (PIMOS) (Chikayama et al. 1988) and application programs (mainly knowledge processing systems) are and will be also written in KL1 or in its extensions. Various software technologies for highly parallel processing, such as highly concurrent algorithms, programming paradigms for practical concurrent programs, and experiments on load balancing schemes, will be studied and developed based on the KL1 language.

Everything, from language to the hardware system, operating system, concurrent algorithms, and programming style is completely new in our R&D approach. They are designed for highly parallel processing on large-scale network-connected MIMD-type multiprocessor systems. The program development environment is very important to cultivate these new computing technologies. Bootstrapping of the R&D, from a simple and small prototype to a large and complex target, is also important. The Multi-PSI was developed as an R&D tool for concurrent software technologies, and also as an early prototype of the parallel inference machines (Taki 1986, Taki 1987).

This paper reports the outline of our FGCS computing architecture by looking at the hardware system, language, operating system and program development, and also reports their progress.

2 LANGUAGES

KL1 is the common kernel language for parallel inference systems in the FGCS project, based on Flat GHC (Ueda 1986). GHC is a concurrent logic programming language similar to Concurrent Prolog (Shapiro 1983) and Parlog (Clark and Gregory 1984).

The advantage of using a concurrent logic programming language is in its implicit concurrency and synchronization feature. Without being explicitly specified in the program, concurrency of the program is exploited and data-flow synchronization is made automatically at

and under the language implementation level. The implicit data-flow synchronization mechanism has a great advantage in eliminating synchronization errors. The language is powerful enough to describe everything as long as it can be modeled as communicating processes. Any processes with small grain size or short lifetime can be implemented without large overhead.

Flat GHC is a subset of GHC. Only unification and calls to certain built-in predicates are allowed in the guard part of a clause. This makes efficient implementation easier without losing the essential descriptive power of the language.

The KL1 language has several extensions from the original Flat GHC (Chikayama et al. 1988). One of the most essential extensions is the notion of *sho-en*. *Sho-en*, or *manor* in English, is similar to the meta-call mechanism seen in other concurrent logic programming languages. Like the meta-call, the *sho-en* mechanism can be used to protect the outside of the *sho-en* from failure inside the *sho-en*. In addition, limits on computational resources (e.g., execution time and memory) consumed in a *sho-en* can be controlled and monitored from outside. This feature is essential in writing an operating system in KL1.

The KL1 language also supports the functions of priority control and load allocation. Execution priority can be specified for a *sho-en* or goal. A goal can be allocated to a certain processing node specified by a node number. The annotation, which specifies the priority or node number for a goal (e.g., *goal@priority(X)* or *goal@node(Y)*), is called the pragma. The pragma only affects execution efficiency of a program but is independent of the program semantics. This feature is useful for tuning the execution efficiency and the load balance of a program. Current experimental implementation on the Multi-PSI supports 4096 priority levels which will be used in user programs.

3 PIM PILOT MACHINE: PIM/p

3.1 Target Performance

PIM/p is the first pilot machine for our target PIM system. Several other models are also being developed. The performance of a PIM/p processing element is 200K to 500K LIPS. A PIM/p system will contain 128 processing elements and will achieve 10M to 20M LIPS of effective performance.

3.2 Overall Structure

PIM/p has the hierarchical structure shown in Figure 1. Eight processing elements (PEs) form a cluster with shared memory and bus. The PIM/p consists of 16 clusters connected by the inter-cluster network. Processing elements in a cluster share the same address space, whereas address spaces for each cluster are separated.

A cluster forms a substructure with low communication cost and response time which is utilized in the load allocation. The hierarchical structure allows an easier and better implementation of the dynamic memory management than the structure sharing all the memory. It is also more effective to reduce the physical memory size per a processing element than the size of the completely non-shared memory structure.

3.3 Processing Elements

The PIM/p processing element is designed for the efficient execution of KL1-B (Kimura and Chikayama 1987), which is the common abstract instruction set for the KL1 used in our inference machines. KL1 programs are compiled to KL1-B instructions and then translated to target machine instructions. PIM/p has a RISC-like instruction set which is executed in a four-stage pipeline (Goto et al. 1988). To reduce the static code size, PIM/p supports the conditional macro-call feature (Shinogi et al. 1988), which is a sort of subroutine call to the internal instruction memory (IIM) with a specialized argument passing mechanism. It is used to implement complicated KL1-B functions by common modules in the IIM with low invocation overhead. The tag architecture has been adopted for the PIM/p processor with an 8-bit tag and 32-bit data. Each data word is aligned with the 64-bit memory boundary in the current implementation.

The role of KL1-B is similar to that of WAM (Warren 1983). The major differences are the synchronization feature and functions for incremental garbage collection, called MRB (Chikayama and Kimura 1987).

A PIM/p processing element is implemented on a single board with several custom CMOS LSIs and about 20 static RAMs, as shown in Figure 2. The pipeline cycle is expected to be 50 nanoseconds. There are two cache memories, instruction cache and data cache with 64K bytes each. They are write-back caches with the cache coherency protocol (Matsumoto et al. 1987). They also support the word locking mechanism and software cache functions optimized for KL1 execution (Goto et al. 1988). The common bus cycle is the same with the processor pipeline cycle. The bus data width is 64 bits. The network interface unit (NIU) and the floating point unit (FPU) are the co-processors of the CPU. The peak performance for the append program will be over 600K LIPS including MRB garbage collection.

3.4 Inter-cluster Network

A message exchanging network with hyper-cube topology has been introduced to connect PIM/p clusters, placing each cluster on a hyper-cube node. Inter-cluster communication is invoked by a unification or a goal forking across the cluster boundary in KL1 program execution. Two sets of hyper-cube networks are used to

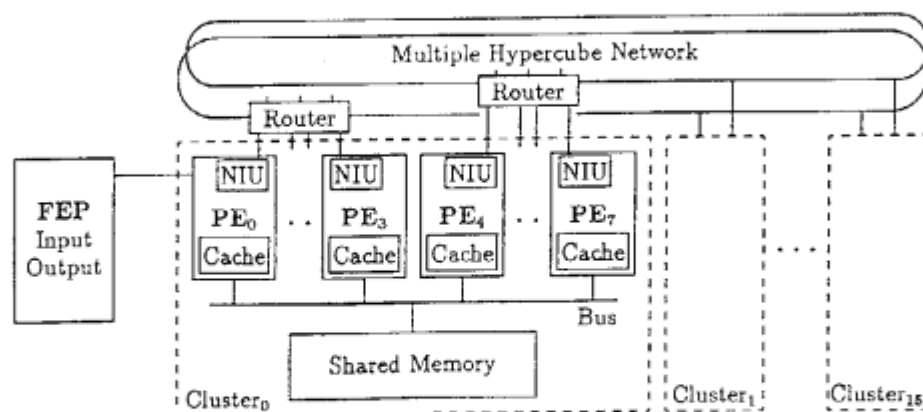


Figure 1: The Pilot Machine: PIM/p

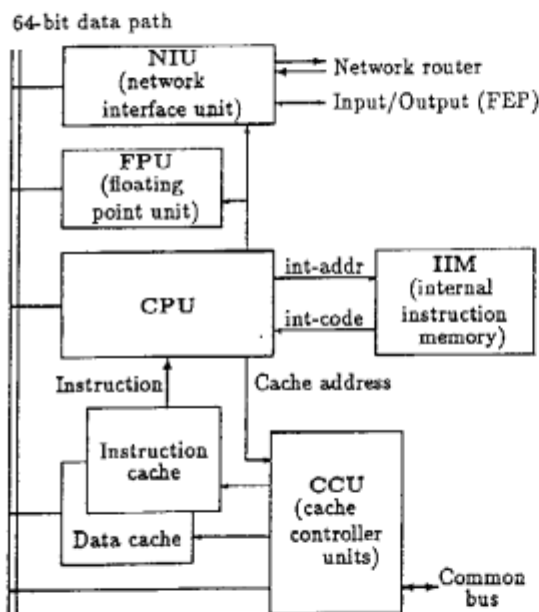


Figure 2: PIM/p Processing Element Configuration

increase the network bandwidth as shown in Figure 1. Every processing element has a connection to one network. A message is routed to the destination cluster automatically, as preset. Each communication path has a throughput of 20M bytes/second in both directions.

3.5 KL1-B Implementation Issues

KL1-B is the abstract instruction set which defines the basic behavior of the KL1 language processor. Each KL1-B instruction is normally expanded into a RISC instruction sequence. However, several complicated functions such as resource management in the *sho-en*, inter-cluster processing mechanisms, and garbage collection, are not open-coded as are other more primitive features. They are implemented as common modules in the IIM,

and invoked when certain conditions occur.

There were many difficulties in the KL1-B implementation, especially in the inter-cluster processing mechanisms, such as the distributed resource management for the *sho-en* (Ichiyoshi et al. 1987), the efficient goal termination detecting mechanism among clusters (Rokusawa et al. 1988), export and import mechanisms of reference pointers between clusters, and the distributed unification mechanism (Ichiyoshi et al. 1988). These problems were basically solved in the KL1-B implementation on the Multi-PSI system.

3.6 Garbage Collection

Efficient garbage collection is essential for the KL1 implementation because it had to be implemented in heap-based style rather than stack-based and consumes much memory at run time. Intra-cluster and inter-cluster garbage collection will be implemented separately for ease of implementation and efficiency.

Both incremental and stop-and-collect GC will be implemented for the intra-cluster GC. An incremental GC, MRB GC (Chikayama and Kimura 1987), is being experimented on the Multi-PSI. It is a simple subset of reference counting, only distinguishing between single and multiple references by one bit. Structures with single reference often appear in KL1 programs. MRB GC is effective in such cases, making the working set size smaller and the interval of stop-and-collect GC longer.

Inter-cluster reference pointers are implemented with export and import tables which are a sort of address translation table. The role of inter-cluster GC is to maintain these table entries and to remove garbage inter-cluster pointers. The weighted export counting (WEC) scheme (Ichiyoshi et al. 1988) has been introduced to realize the incremental GC for these pointers. WEC is an application of weighted reference counting (Watson and Watson 1987, Bevan 1987).

3.7 Goal Scheduling

A goal is a unit of parallel execution and scheduling. A goal is replaced or expanded into the body goals of a clause in the invoked predicate when the clause is committed. The leftmost body goal is executed immediately while others are pushed into the ready-goal stack. That is, the depth-first scheduling is adopted for body goals and also for the ready-goal stack. Since each goal is associated with its execution priority, ready-goal stacks are managed corresponding to each priority level. When all the guard unifications are suspended, the goal is suspended, hooking itself to the variables that caused the suspensions (Ichiyoshi et al. 1987). The goal is resumed when one of those variables is instantiated. That is, the *non-busy waiting* method has been adopted.

How to keep the processing load well-balanced is a key issue in making the best use of parallel processing resources. An automatic load balancing is adopted in a cluster. Each processing element has a goal stack for the highest-priority ready goals to avoid conflicts of access to the common ready-goal stack. The highest-priority goals are distributed to keep the processor loads in good balance. We found *on-demand* distribution to be an effective way of realizing a good balance within a cluster while reducing the amount of wasteful communication among processors (Sato and Goto 1988). In this scheme, an idle processor sends a request for load allocation to a busy processor.

Load distribution among clusters should be done carefully because the communication cost is more expensive than within a cluster. Several distribution schemes have been tested on the Multi-PSI in which the load distribution algorithms are buried in the KL1 programs specifying goal allocation by the pragma (`goal@node(X)`). Several standard schemes will be supported by the operating system in the future.

4 OPERATING SYSTEM

PIMOS is the common operating system for our inference machines. It has been developed on the Multi-PSI. The primary functions of the PIMOS are I/O resource management, execution control of user tasks, and management of programs (Chikayama et al. 1988). The programming system has not been implemented in the current version. PIMOS has the following characteristics.

Logic-based: PIMOS is described entirely in KL1, without using extra-logical features at all. Even I/O devices connected to the inference machine have *logical* interfaces. Each I/O device looks like a perpetual process from the user program, communicating through a stream interface.

Integrated: Although PIMOS is an operating system for parallel machines, it is an integrated operating system working as one unit, rather than consisting of many operating systems distributed on processing elements.

Born parallel: Various functions of the PIMOS, which exploit the power of parallel inference machines, have been implemented to work in parallel to prevent the PIMOS becoming a bottleneck in highly parallel systems.

Practical: Although PIMOS has many experimental features, its purpose is to provide a practical programming environment for parallel algorithm development, keeping robustness.

5 PROGRAM DEVELOPMENT

Three systems providing the program development environment have been developed. The primary system is called the PIMOS development support system (PDSS), written in C, and an operating system called Micro-PIMOS. All the KL1 features except for real parallel execution are provided with several debugging facilities. The system was mainly used for the development of the PIMOS and in the early stage of the application program development.

The second system, Multi-PSI, and the third system, Pseudo Multi-PSI, have been developed in parallel. The Multi-PSI (Taki 1986, Taki 1987) is a collection of PSI-II processors (Nakashima and Nakajima 1987) connected by a two-dimensional mesh network (Takeda et al. 1988). The full system contains 64 processing elements. KL1-B has been implemented in microprogram. The processing element speed is approximately 150K LIPS for the append program with MRB GC. The KL1 compiler and debugging support system were implemented on the front-end processor (FEP), PSI-II. The FEP also supports I/O functions controlled by the PIMOS. The Multi-PSI is used with the PIMOS as a main tool for the development and evaluation of various concurrent algorithms and load balancing schemes. Three full systems have been working since 1988.

The Pseudo Multi-PSI is a simulator of the Multi-PSI implemented on a PSI-II machine. The behavior of the Multi-PSI with any number of processing elements can be simulated. KL1 programs are executed by the same microprogram as that of the Multi-PSI, overlaid on the PSI-II micro code area. The KL1 execution speed for simulating one processing element is equivalent to that of the Multi-PSI, which is unusual for simulation systems. The system performs the same as the real Multi-PSI except for the round robin scheduling of the pseudo processors and smaller memory size. The Pseudo Multi-PSI is mainly used for general KL1 program debugging, both for intra-PE errors and inter-PE errors.

Four KL1 sample programs were developed for the demonstration at the FGCS'88 conference. They are the natural language parser, PAX; a board game, tsumego; the packing piece puzzle (Furuichi et al. 1990); and the shortest path finding problem. There is a total of 14.5K source program lines in KL1, and the development period is around three months, with eight programmers. Several concurrent algorithms and load balancing schemes are being experimented in the program development.

6 CONCLUDING REMARKS

In the past, research on parallel computer hardware has been relatively independent from parallel software research. Basically, the hardware or system implementation research was for implementing more efficient environments for executing *already existing* software.

The principle of the parallel inference systems development in ICOT is rather different in this point; software and hardware research should be combined more closely. Software or even algorithms optimized for sequential machines may not be optimal for parallel machines. Thus, software should change when the hardware changes.

However, there is a chicken-and-egg problem: without parallel hardware, practical parallel software cannot be developed; without parallel software, it is hard to know what kind of parallel hardware is appropriate. ICOT's approach to solve this problem is stepwise bootstrapping. The first step was to settle on a software-hardware interface, namely, the KL1 language, and implement it (as the multi-PSI system). The next step is to develop various software systems on it (including PIMOS). By running the resultant software, many unknown parameters of the behavior of parallel software will be revealed. The next generation (the PIM/p system) will be based on these experiences, and software will be developed on this machine.

Development of the PIM/p hardware system will be completed in 1990, and then the improved KL1 language processor will be implemented on it, inheriting various research results from the software development on the Multi-PSI system.

Our challenge to develop the parallel processing technologies for large-scale MIMD multiprocessors from both software and hardware sides must be the creation of a *new computing culture*.

Acknowledgement

This paper is based on various R&D activities carried out by many researchers at ICOT and cooperating manufacturers.

REFERENCES

- (Bevan 1987) D.I. Bevan. Distributed Garbage Collection using Reference Counting. In *Proceedings of Parallel Architectures and Languages Europe*, pages 176-187, June 1987.
- (Chikayama and Kimura 1987) T. Chikayama and Y. Kimura. Multiple Reference Management in Flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 276-293, 1987.
- (Chikayama et al. 1988) T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.
- (Clark and Gregory 1984) K. Clark and S. Gregory. Notes on Systems Programming in PARLOG. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 299-306, Tokyo, 1984.
- (Furuichi et al. 1990) 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. on Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, Seattle, March 1990.
- (Goto et al. 1988) A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the Parallel Inference Machine Architecture (PIM). In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.
- (Ichiyoshi et al. 1987) N. Ichiyoshi, T. Miyazaki, and K. Taki. A distributed implementation of flat GHC on the Multi-PSI. In *Proceedings of the Fourth International Conference on Logic Programming*, 1987.
- (Ichiyoshi et al. 1988) N. Ichiyoshi, K. Rokusawa, K. Nakajima, and Y. Inamura. A New External Reference Management and Distributed Unification for KL1. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.
- (Kimura and Chikayama 1987) Y. Kimura and T. Chikayama. An Abstract KL1 Machine and its Instruction Set. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 468-477, 1987.

- (Matsumoto et al. 1987) A. Matsumoto et al. Locally Parallel Cache Designed Based on KL1 Memory Access Characteristics. TR 327, ICOT, 1987.
- (Nakashima and Nakajima 1987) H. Nakashima and K. Nakajima. Hardware Architecture of the Sequential Inference Machine: PSI-II. In *Proceedings of 1987 Symposium on Logic Programming*, pages 104-113, San Francisco, 1987.
- (Rokusawa et al. 1988) K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An Efficient Termination Detection and Abortion Algorithm for Distributed Processing Systems. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 1 Architecture, pages 18-22, August 1988.
- (Sato and Goto 1988) M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *Proceedings of IFIP Working Conference on Parallel Processing*, Pisa, Italy, April 1988.
- (Shapiro 1983) E.Y. Shapiro. A subset of Concurrent Prolog and Its Interpreter. TR 003, ICOT, 1983.
- (Shinogi et al. 1988) T. Shinogi, K. Kumon, A. Hattori, A. Goto, Y. Kimura, and T. Chikayama. Macro-call Instruction for the Efficient KL1 Implementation on PIM. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.
- (Takeda et al. 1988) Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama, and K. Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.
- (Taki 1986) K. Taki. The parallel software research and development tool : Multi-PSI system. In *Programming of Future Generation Computers*, edited by K.Fuchi and M.Nivat, North-Holland, Amsterdam, 1988, pages 411-426. Also in TR 237, ICOT, 1986.
- (Taki 1987) K. Taki. Measurements and evaluation for the Multi-PSI/V1 system. In *Programming of Future Generation Computers II*, edited by K.Fuchi and L.Kott, North-Holland, Amsterdam, 1988, pages 365-391. Also in TR 370, ICOT, 1987.
- (Ueda 1986) K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. TR 208, ICOT, 1986. (Also in *Programming of Future Generation Computers*, North-Holland, Amsterdam, 1987.).
- (Uchida et al. 1988) S. Uchida, K. Taki, K. Nakajima, A. Goto, and T. Chikayama. Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project. In *Proc. of the International Conference On Fifth Generation Computing Systems 1988*, Tokyo, November 1988.
- (Warren 1983) D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, Artificial Intelligence Center, SRI, 1983.
- (Watson and Watson 1987) P. Watson and I. Watson. An Efficient Garbage Collection Scheme for Parallel Computer Architecture. In *Proceedings of Parallel Architectures and Languages Europe*, pages 432-443, June 1987.