Parallel Logic Programming Languages

Ian Foster & Steve Gregory
Imperial College
London
Contents

- Introduction and history
- PARLOG: the language
- Other parallel logic programming languages
- Applications
- Case study: PPS
- PARLOG implementations
- Some recent developments
- Conclusions
Parallel Logic Languages

Horn clause logic:

\[ H \leftarrow B_1, B_2, \ldots, B_n. \]

Declarative interpretation:

\( H \) is true if all \( B_i \) are true.

Procedural interpretation:

to prove \( H \), prove all \( B_i \)

Process interpretation:

\( H \) reduces to a network of concurrent processes \( B_i \).

Parallel logic languages =

Horn clause logic + AND-parallelism communication synchronization committed-choice non-determinism

They are interesting because they are:

logic programming languages parallel languages expressive
Parallel Logic Programming
A Partial History

1977
// functional programming
Kahn + MacQueen

1978
PL as language for // programming.
Van Emden + Lucena
Deterministic

1973
Prolog

IC-PROLOG 1977/80
Clark + McCabe + Gregory
parallel evaluation, "guards",
read-only variable, streams, run-time mode checks, backtracking

1974/75
Guarded commands
Dijkstra

1976 Lazy LISP
Friedman + Wise,
Henderson + Morris

1979/80
Non-deterministic LISP
Friedman + Wise

1980-
or-parallel
PROLOGS

PARLOG
1982 ->
Clark + Gregory
RLPP + set constructor
interface: or-parallel,
and-parallel

RELATIONAL LANGUAGE FOR parallel programming
Clark + Gregory 1980/81
parallel evaluation, guards, streams,
compile-time mode checks, committed choice non-determinism

Need for variables in messages

Concurrent Prolog
Shapiro 1983
RLPP + read-only variable - compile-time mode checks

GHC/KL1
ICOT 1985

FCP
Mierowsky et al 1985

CSP 1978 Hoare
AND-PARALLELISM

Relation call = Process

Conjunction = Process network

e.g. C1, C2, process(s)

Clause with one call in body = Change of process state

e.g. process(State) ← process(New-state).
**AND-PARALLELISM**

Clause with >1 calls in body =
Creation of new processes

\[
\text{mode process(?).}
\]
\[
e.g. \quad \text{process(State) } \leftarrow \\
\quad \text{process(State1), process(State2)}
\]

Unit clause =
Process termination

\[
e.g. \quad \text{process(State)}.
\]
AND-PARALLELISM

mode on-list(? , ?).
on-list(Item, [Item | List]).
on-list(Item, [U | List]) ← Item \= U:
on-list(Item, List).

e.g. on-list(1, [5, 2, 1, 3])
AND-PARALLELISM

mode off-tree(?1, ?2).
off-tree(Item, t(T1, T2)) ←
off-tree(Item, T1), off-tree(Item, T2).
off-tree(Item, L) ←
integer(L), Item \not\leq L : true.

e.g. off-tree(1, t(t(5, 2), t(1, 3)))
COMMUNICATION

Shared variable = Communication channel or memory location

e.g. process1(V), process2(V)

Binding shared variable = sending message

N.B. Single-assignment
mode sum_tree(?,[\to]).
sum_tree(t(T1,T2), V) \leftarrow
  sum_tree(T1, V1), sum_tree(T2, V2),
  plus(V1, V2, V).

sum_tree(L, V) \leftarrow integer(L);
  V= L.

e.g. sum_tree(t(t(5,2),t(1,3)), V)
Stream communication

Sequence of partial bindings to shared variable = Stream of messages

\[ V = f(X, Y) \quad X = t_1 \quad Y = g(Z) \quad Z = t_2 \]
mode flat_tree(? , ^).
flat_tree(t(T1 , T2) , S) \leftarrow 
  flat_tree(T1 , S1) , flat_tree(T2 , S2) ,
  append(S1 , S2 , S).
flat_tree(L , [L]) \leftarrow integer(L) : true.

mode append(?, ?, ^).
append([U|X] , Y , [U|Z]) \leftarrow append(X , Y , Z).
append([], Y , Y).

\textit{e.g.} \texttt{flat\_tree(t(t(5,2), t(1,3)) , S)}
SYNCHRONIZATION

Delay reduction (call \( \rightarrow \) clause body) until:

1. Input arguments available, and
2. Guard succeeds

mode process(?), process1(?).
process(pattern(State)) \( \leftarrow \)
process1(State).

process1(State) \( \leftarrow \) guard(State):
process2(State).

Wait for \( S = \text{pattern}(\text{State}) \)
Wait for \( \text{guard}(\text{State}) \)
SYNCHRONIZATION

Before reduction

No access to output arguments of call.
Only input (read-only) access to input arguments of call.

1. One-way unification (matching)

Unification (call - clause head) cannot bind call variables:
suspends on attempt to do so.

Call \( r(t_1, ..., t_K) \)
\[ \Downarrow \]
\[ \Downarrow \]
Clause head \( r(t_1', ..., t_K') \)

2. Safe guards

Guard cannot bind call variables.

After reduction

Call arguments are unified with head arguments, in output positions.
May bind call variables.
Synchronization

Kernel PARLOG - make matching explicit

mode p(?,:?,?,:).
p(t1,t2,t3,t4) ← guard: body.

can be written:

\[ p(X1,X2,X3,X4) \leftarrow t1 \leq X1, t3 \leq X3, \text{guard:} \]
\[ X2 = t2, X4 = t4, \text{body.} \]

Matching primitive

\[ t1 \leq t2 \]

Unify \( t1, t2 \) but suspend on attempt to bind variables in \( t2 \).
SYNCHRONIZATION

1. mode append(?, ?, ?).

append([U|X], Y, [U|Z]) ← append(X, Y, Z).
append([], Y, Y).

2. append(T, Y, S) ← [U|X] <= T :
   S = [U|Z], append(X, Y, Z).
append(T, Y, S) ← [] <= T :
   S = Y.
COMMITTED CHOICE NON-DETERMINISM

Clauses in procedure = alternative ways to reduce process

Committed choice of "candidate" clause
("candidate": successful input matching and guard)

mode process(\(?)\).
e.g. process(pattern(S1)) \iff\ guard1(S1) : process1(S1).
process(pattern2(S2)) \iff\ guard2(S2) : process2(S2).
process(pattern3(S3)) \iff\ guard3(S3) : process3(S3).

Wait for \(S = \text{pattern}(S1)\) and \(\text{guard}(S1)\)

Parallel search
Read-only access to arguments during search
- "safe" guards.

Output only after "commitment".
- committed to output.

May be many candidate clauses.
- time dependency.

Parallel search for clause.
- "committed" or-parallelism.
Committed choice

Non-determinism

Committed or-parallelism

```
mode on_tree(? , ?).

on_tree( Item , t(T1, T2)) ← on_tree(Item, T1): true.

on_tree(Item , t(T1, T2)) ← on_tree(Item, T2): true.

on_tree(Item, Item).
```
Sequence of partial bindings to shared variable can be made by different processes.
THE LOGICAL VARIABLE

Back communication

Sequence of partial bindings to shared variable, some by this process, some by others.

e.g. \( p: \ V = [t(U1)|V1] \)

\( V1 = [t(U2)|V2] \)

c: \( U1 = t1 \)

\( U2 = t2 \)

\[ p(V), c(V) \]
THE LOGICAL VARIABLE

mode pri-queue(?), pri-queue(?;?),
pri-queue(M) ← pri-queue(M, []).

pri-queue([add(Item) | M], Q) ←
  insert(Item, Q, New-q),
pri-queue(M, New-q).
pri-queue([remove(Item) | M], [H | Q]) ←
  Item = H,
pri-queue(M, Q).

\[\text{pri-queue} \leftarrow \left[\text{add(Fred), add(john), remove(X)}, \ldots\right]\]
THE LOGICAL VARIABLE

e.g. priority spooler

user1 → merge → pri-queue

user2 → merge → pri-queue

printer → merge → pri-queue

printer → merge → pri-queue
THE LOGICAL VARIABLE

Eager "read list":

mode eager-read(\(\uparrow\)).

\texttt{eager-read([[]]) \leftarrow \text{end-of-file} : \text{true};}
\texttt{eager-read([U|X]) \leftarrow}
\texttt{read(U) \&}
\texttt{eager-read(X).}

Lazy "read list":

mode lazy-read(?).
\texttt{lazy-read([U|X]) \leftarrow \text{end-of-file}:
U = \text{end-of-file};}
\texttt{lazy-read([U|X]) \leftarrow
read(U) \&}
\texttt{lazy-read(X).}
METALEVEL PROGRAMMING

Problem:

To allow a process ("metaprogram") to examine and control evaluation of another ("object program").

Solution 1:

Transform the object program to signal its status and respond to control messages.

Solution 2:

Encapsulate object program in special metacall to achieve same effect.
The PARLOG metacall

call (Goal?, Status↑, Control?)
EXAMPLE: SR LATCH

PARLOG specification of NAND gate

```
mode nand(?, ?, ↑).
nand([1|X], [1|Y], [0|Z]) ← nand(X, Y, Z).
nand([0|X], [1|Y], [1|Z]) ← nand(X, Y, Z).
nand([1|X], [0|Y], [1|Z]) ← nand(X, Y, Z).
```

SR latch using NAND gates

```
mode sr-latch(?, ?, ↑, ↑).
sr-latch(S, R, Q, Q_0) ←
```

EXAMPLE: PARSER

Grammar

Expr → Term Rest_expr

Rest_expr → Add_op Expr
Rest_expr → empty

Term → Number
Term → '(' Expr ')

PARLOG

mode expr(?,'_'), rest_expr(?,'_'), term(?,'_').

expr(Tokens_h, Tokens_t) ←
   term(Tokens_h, Tokens), rest_expr(Tokens, Tokens_t).

rest_expr([Op | Tokens_h], Tokens_t) ← add_op(Op):
   expr(Tokens_h, Tokens_t) ;
rest_expr(Tokens, Tokens).

term([N | Tokens], Tokens) ← number(N): true.
term([ '(' | Tokens_h], Tokens_t) ←
   expr(Tokens_h, [')' | Tokens_t]).
EXAMPLE: PARSER

Process interpretation

```
[ ]   rest_expr  "+3"  term  "(1+2)+3"
```

expr
**EXAMPLE: LABEL TREE**

Problem:

Mark nodes of a labelled binary tree that occur on parallel branches.

```
  3
 /   \
1     5
  \\   \
  5     \
  |     |
  4     9
```

```
  3
 /   \
1     5
   |   |
par(5) 3
   |     |
par(4) 2      par(4)
   |          |
   |          |         9
```

EXAMPLE: LABEL TREE

mode label_tree(I_tree?, 0_tree↑, Par-this?, In-this↑).

label_tree(t(X, U, Y), t(X1, U1, Y1), Par-this, [U|In-this]) ←
replace(U, U1, Par-this),
label_tree(X, X1, Par_X, In-X),
label_tree(Y, Y1, Par_Y, In-Y),
append(Par-this, In-Y, Par-X),
append(Par-this, In-X, Par-Y),
append(In-X, In-Y, In-this).
label_tree(empty, empty, Par-this, []).

mode replace(Node?, New_node↑, Par_nodes?).

replace(Node, par(Node), Par_nodes) ←
on-list(Node, Par_nodes) : true ;
replace(Node, Node, Par_nodes).
EXAMPLE: LABEL TREE

\[ t(X, U, Y) \]

\[ t(X_1, U_1) \]

\[ t(X_1, U_1) \]
EXAMPLE: COMMUNICATION PROTOCOLS

mode dev(State?, In?, Out↑).

dev(s0, In, [syn|out]) ← dev(s1, In, Out).
   % Establish

dev(t0, [syn|In], [syn_ack|Out]) ← dev(t1, In, Out).
   % Acknowledge

dev(t0, [syn|In], [syn_ack|Out]) ← dev(t2, In, Out).
   % Decline

dev(s1, [syn_ack|In], [ack|Out]) ← dev(s2, In, Out).
   % Confirm

dev(s1, [syn_ack|In], Out) ← dev(s2, In, Out).
   % Abandon

dev(t1, [ack|In], Out) ← dev(t2, In, Out).
   % Connect
PARLOG specification of simple connection establishment protocol

**EXAMPLE: COMMUNICATION PROTOCOLS**

receive SYN-ACK
send ACK

receive SYN
send SYN-ACK

receive ACK

dev(s0, TS, ST),  dev(t0, ST, TS)
EXAMPLE: PARLOG FOR SPECIFICATION

CSP specification of (illogical?) variable

\[
\text{var}_X = (\text{update}\, ?\, Y \rightarrow \text{var}_Y \mid \text{read}! X \rightarrow \text{var}_X)
\]

Advantage: calculus for reasoning about behaviour

PARLOG specification

\[
\begin{align*}
\text{mode } \text{var}(?, ?). \\
\text{var}(X, [\text{update}(Y) \mid M]) & \leftarrow \text{var}(Y, M). \\
\text{var}(X, [\text{read}(R) \mid M]) & \leftarrow R=X, \text{var}(X, M). \\
\text{var}(X, []) & .
\end{align*}
\]

Advantage: some properties clear from logical reading
EXAMPLE: SYNCHRONIZED COMMUNICATION

mode sync_send(?,↑,?), sync_receive(?,?,↑), succeeded(?).

sync_send(Term, [(Term,Ack)|Ch], Ch) ⇐
succeeded(Ack).

succeeded(succeeded).

sync_receive(R_term, [(Term,Ack)|Ch], Ch) ⇐
R_term = Term &
Ack = succeeded.
EXAMPLE: SYNCHRONIZED COMMUNICATION

process1(E), ..., process2(E)

sync_send(3, E, E1)

sync_send(t(a, y), E1, E2)

sync_receive(t(z, b), E1, E2)

X = 3

t(a, y) = t(z, b)

sync_receive(X, E, E1)
Page 38 missing: more of "Example: Synchronized Communication"?
Concurrent Prolog Differences

Synchronization: read-only variable

- Read-only annotation ('?') on consumer occurrences of variables.

- Suspend on attempt to bind a variable via a read-only occurrence.

- No other restrictions on variable bindings made by unification or guards; call variables may be bound before commitment

=> multiple environments

e.g. append([U|X],Y,[U|Z]) :- append(X?,Y?,Z).
    append([],Y,Y).

"mode" depends on annotations in call.

- Difficulties implementing multiple environments

=> serious implementation work thus far restricted to Flat Concurrent Prolog (FCP)

- Flat: only calls to system primitives in guards
GHC Differences

Syntax

Assumed mode (?,?,?, ?).

Output unification done explicitly.

e.g. PARLOG

mode append(?,[?]^).
append([UIX],Y,[UIZ]) <- append(X,Y,Z).
append([],Y,Y).

GHC

append([UIX],Y,Z1) :- Z1 = [UIZ],
append(X,Y,Z).
append([],Y,Z1) :- Z1 = Y.

Synchronization

1. One-way unification (matching):

   Call/head unification cannot bind call variables

   (like PARLOG).

2. Guard suspends on attempt to bind call variables

   (PARLOG: guard must be "safe").

N.B. Flat GHC = Flat PARLOG.
Applications of Parallel Logic Languages

Why use parallel logic languages?

• Parallel logic languages =
  
  Horn clause logic + concurrent, committed-choice proof procedure

• Horn clause logic: (declarative content)
  
  parallel logic programs can be read as executable specifications

• Proof procedure: (parallel execution)
  
  declarative programs can be executed in AND-parallel

• Proof procedure: (process interpretation)
  
  can be exploited to provide useful operational behaviour
Useful Operational Behaviour

The proof procedure of parallel logic languages permits them to implement many useful 'real-world' behaviours

- Concurrent, communicating entities

  ..., keyboard(In), user(In, Out), screen(Out), ...

- Side-effects in the external world

  \[ \text{screen}(\text{display}(X) / \text{In}) \leftarrow \text{output}(X) \land \text{screen}(\text{In}). \]

- Time dependent treatment of events:

  \[
  \text{keyboard}([\text{Ch} / \text{Chars}], \text{Interrupts}, \text{Out}) \leftarrow \\
  \quad \text{handle_char}(\text{Ch}, \text{Out}, \text{NewOut}), \\
  \quad \text{keyboard}(\text{Chars}, \text{Interrupts}, \text{NewOut}).
  \]

  \[
  \quad \text{keyboard}(\text{Chars}, [\text{Int} / \text{Interrupts}] \text{Out}) \leftarrow \\
  \quad \text{handle_interrrupt}(\text{Int}, \text{Out}, \text{NewOut}), \\
  \quad \text{keyboard}(\text{Chars}, \text{Interrupts}, \text{NewOut}).
  \]
Applications of Parallel Logic Languages
(Some Examples)

(a) **Behavioural:** describe, implement concurrent systems

(emphasis on operational behaviour)

operating systems / programming environments
telephone exchange control
simulation

(b) **Algorithmic:** describe, implement parallel algorithms

(emphasis on declarative content)

parallel parsers
parallel router
compilers
image processing
qualitative reasoning

(c) **Language Implementation:**

concurrent implementations of other language formalisms

Vulcan
POLKA
LOTOS

Concurrent object-oriented languages
Formal description language for concurrent systems
A Large Application: PARLOG Programming System (PPS)

- An operating system designed to support logic programming on parallel machines

- A prototype implemented on SUN workstations is a major PARLOG application

- Uses some UNIX facilities but implements computation control, secondary storage management etc in PARLOG

- Implementation exploits extended PARLOG metacall:
  - an exception handling mechanism
  - modularity
  - computation priorities

- Implementation demonstrates:
  - Use of PARLOG to implement a complex system
  - PARLOG programs as executable specifications
  - Use of PARLOG control metacall
  - PARLOG programming techniques: back communication, synchronization variables
PPS Facilities

PPS provides:

- modularity: programs are divided into databases
- persistence: file system is invisible
  databases persist between PPS invocations
- declarative environment:
  user view: system = \{databases\}
  user interaction: execute queries wrt databases
  queries calculate relations over system states
- metaprogramming:
  programs can:
  - reason about other programs
  - generate new versions of other programs
  programs define relations over system states.
- multiprogramming:
  task control
  concurrency control
- user-definable inference mechanisms:
  inheritance
  query-the-user
An Extended PARLOG Metacall

- Primitive for describing initiation, monitoring and control of a computation
- Efficient implementation techniques developed

\[
\text{call}(\text{Program}, \text{Priority}, \text{Goal}, \text{Status}, \text{Control})
\]

- stop
- suspend
- continue
- priority(_)
- succeeded
- failed(_)
- stopped
- exception(_)
- exception(_,_,_)

\textbf{Program}: names a module

\textbf{Priority}: programmer control of underlying scheduling mechanism

\textbf{Status} and \textbf{Control}: interfaces to monitoring and control functions of underlying machine
### Exception Handling

1. The *exception(T,G,NG)* message indicates an unsolvable goal *G*; *T* indicates why the goal was unsolvable (undefined, div-by-zero, ...)

2. The unsolvable goal is replaced with a continuation variable *NG*

3. The monitoring program can instantiate *NG* to a new goal (in the example, false)

**Applications:**

- implement alternative exception handlers (including inheritance, query-the-user, ...)
- enhance expressive power of language by implementing system calls
A Simple Exception Handler

- Reports termination
- Aborts program if any run-time errors reported
- Closed world: fails all goals unsolvable in program

... call(Db,1,G,S,C), monitor(S,C,Output), ...

\[
\text{mode initiate(Db?,G?,Output↑).}
\]

\[
\text{initiate(Db,G,O) ← call(Db,1,G,S,C), monitor(S,C,O).}
\]

\[
\text{mode monitor(Status?,Control↑,Output↑).}
\]

\[
\text{monitor(failed(_),C,[failed]).}
\]

\[
\text{monitor(succeeded,C,[succeeded]).}
\]

\[
\text{monitor([exception(T)]/_J,stop,[exception(T)]).}
\]

\[
\text{monitor([exception(T,G,NG) / S],C,[exception(T,G) / O]) ← NG = false, monitor(S,C,O).}
\]
Concurrent Control in PPS

- PPS is a multiprogramming system: several queries can execute concurrently

- PPS supports metarelations that permit a program to:
  - access terms representing system state \((\text{definition, ...})\)
  - generate new states \((\text{new definition, ...})\)
  - specify a new state, to apply on success

State is the set of all databases defined in PPS

- A PPS program may thus access a database in three ways:
  - to execute its code
  - to read its source
  - to assert new versions of predicates

- Control mechanisms are required to:
  - avoid contention due to concurrent access/update
  - maintain declarative semantics: a PPS program is a relation over states

- Implementation of these mechanisms illustrates use of PARLOG for systems programming
Metarelations

\textit{current}(\textit{State} \uparrow)
\textit{definition}(\textit{State}?,\textit{Db}?,\textit{Relation}?,\textit{Definition} \uparrow)
\textit{new\_definition}(\textit{State}?,\textit{Db}?,\textit{Definition}?,\textit{NewState} \uparrow)
\textit{next}(\textit{State}?)

\textit{etc} ...

1. Program transformation

Apply a transformation to all predicates in \textit{Db}.

\textit{transform\_db}(\textit{Db}) \leftarrow
\textit{current}(\textit{S}),
\textit{dict}(\textit{S},\textit{Db},\textit{Dict}),
\textit{transform\_db}(\textit{S},\textit{Db},\textit{Dict},\textit{S'}),
\textit{next}(\textit{S'}).

\textit{mode transform\_db}(\textit{State}?,\textit{Db}?,\textit{Dict}?,\textit{NewState} \uparrow).
\textit{transform\_db}(\textit{S},\textit{Db},[\textit{R}/\textit{Dict}],\textit{S''}) \leftarrow
\textit{definition}(\textit{S},\textit{Db},\textit{R},\textit{Defn}),
\textit{transform}(\textit{Defn},\textit{Defn'}),
\textit{new\_definition}(\textit{S},\textit{Db},\textit{Defn'},\textit{S'}),
\textit{transform\_db}(\textit{S'},\textit{Db},\textit{Dict},\textit{S''}).
\textit{transform\_db}(\textit{S},\textit{Db},[],\textit{S}).

2. Alternative Worlds

Simulate execution of \textit{Q} in \textit{Db} and in \{\textit{Db} + \textit{Fact}\}

\textit{try}(\textit{Db},\textit{Q},\textit{Fact}) \leftarrow
\textit{current}(\textit{S}),
\textit{add\_fact}(\textit{S},\textit{Db},\textit{Fact},\textit{S'}),
\textit{demo}(\textit{S},\textit{Q}),
\textit{demo}(\textit{S'},\textit{Q})
For each query, PPS must:

- maintain alternative states
- commit modifications on successful termination of query
- abort commitment if conflicting accesses

Queries and databases are represented as goals in a parallel conjunction: that is, as processes.

Queries communicate with databases by query (q) and metaquery (mq) messages:

\[
\text{message}(\text{To}, q(Q, \text{Done}), \text{Result}) \\
\text{message}(\text{To}, mq(Q, \text{Done}), \text{Result})
\]

Incomplete messages are used to return results of requests

\[\text{Result} = true, \text{Result} = error(Q)\]

A Done variable associated with a query is included in all messages. This is bound when query terminates. This is an example of a synchronization variable.

Updates (new states) are cached in the query process and applied to databases on success using a two-stage commit procedure.
Concurrency Control: Implementation

Assume three databases, db1, db2, db3.
Two concurrent queries, db1: q1 and db3: q2

- Virtual copies record modifications to databases
- PPS applies updates if query succeeds
- Updates are not applied if concurrent queries active in any database modified by query
PPS Structure

\[
init(Qs) \leftarrow \text{queries}(Qs, DbRs), \text{databases}(DbRs).
\]

\[\text{dbl: } q_1, \text{db2: } q_2\]

mode queries(Queries?, DbRequests↑).

queries([ (Db : Q) / Queries ], DbRs) \leftarrow
\text{query}(Db, Q, QRs),
\text{merge}(QRs, Rs, DbRs),
\text{queries}(Queries, Rs).

- \text{queries} process waits for query messages
- Spawns a \text{query} process to control query evaluation
- Merges \text{query}'s database request stream into general stream
The Query Process

- Generates a query request message to the specified database
- Starts computation using the metacall
- Spawns a monitor process to control evaluation

\[
\text{mode query}(Db?, \text{Query}? , \text{DbRequests} \uparrow) .
\]

\[
\text{query}(Db,Q,[\text{message}(Db,q(Q,\text{Done}),R) / Rs]) \leftarrow \\
\text{monitor}(\text{Done},[], Db,Rs,S), \\
\text{call}(Db,1,R,S,C).
\]
The Monitor Process

- The monitor process controls a query evaluation
- It responds to query status messages by accessing States and, if necessary, generating messages to databases

\[
\text{mode monitor}(\text{Done?}, \text{States?}, \text{Db?}, \text{Requests} \uparrow, \text{Status?}).
\]

Failure:

\[
\text{monitor}(D, \text{States}, Db, [ ], \text{failed}() \leftarrow D = \text{failed}.
\]

Success:

\[
\text{monitor}(D, \text{States}, Db, \text{Requests}, \text{succeeded}) \leftarrow \text{commit( States, Requests)} \land D = \text{done}.
\]

Metarelations: (definition and new definition)

\[
\text{monitor}(D, \text{States}, Db, Rs, [ \text{exception(meta, G, R) / S}]) \leftarrow \\
\text{meta relation}(D, \text{States}, Rs, G, R, \text{States'}, Rs'), \\
\text{monitor}(D, \text{States'}, Db, Rs', S).
\]

Calls to relations in other databases:

\[
\text{monitor}(D, \text{States}, Db, [\text{message(ODb, q(G,D), R)} / Rs], \\
[\text{exception(undef, ODb#G, N_G) / S}]) \leftarrow \\
\text{NG} = \text{call(ODb, R)}, \\
\text{monitor}(D, \text{States}, Db, Rs, S).
\]
On Success, Two-Stage Commit

(1) Generate a commit message to each database to be updated

\[ \text{commit}(\text{Updates}, \text{Oki}, \text{Go}) \], \text{Oki} \text{ and } \text{Go} \text{ variables}

(2) If all \( \text{Oki} = \text{ok} \), instantiate \( \text{Go} \) to \( \text{go} \)

If any \( \text{Oki} = \text{abort} \), instantiate \( \text{Go} \) to \( \text{abort} \)

\[
\text{mode commit}(\text{States}?, \text{Requests} \uparrow).
\]
\[
\text{commit}(\text{States}, \text{Requests}) \leftarrow
\quad \text{updates\_required}(\text{States}, \text{Updates}) ,
\quad \text{request\_commit}(\text{Updates}, \text{Requests}, \text{Go}, \text{Oks}),
\quad \text{confirm\_commit}(\text{Oks}, \text{Go}).
\]

\[
\text{mode request\_commit}(\text{Updates}?, \text{Rs} \uparrow, \text{Go} \uparrow, \text{Oks} \uparrow).
\]
\[
\text{request\_commit}([\text{db}(\text{Db}, \text{Us}) \mid \text{Updates}],
\quad [\text{message}(\text{Db}, \text{commit}(\text{Us}, \text{Ok}, \text{go})_{R_s}) \mid R_s], \text{Go},
\quad [\text{Ok} \mid \text{Oks}]) \leftarrow
\quad \text{request\_commit}(\text{Updates}, \text{Rs}, \text{Go}, \text{Oks}).
\]
\[
\text{request\_commit}([\ ], [\ ], \text{Go}, [\ ]).
\]

\[
\text{mode confirm\_commit}(?, \uparrow).
\]
\[
\text{confirm\_commit}([\text{ok} \mid \text{Oks}], \text{Go}) \leftarrow
\quad \text{confirm\_commit}(\text{Oks}, \text{Go}).
\]
\[
\text{confirm\_commit}([\text{abort} \mid \text{Oks}], \text{abort}).
\]
\[
\text{confirm\_commit}([\ ], \text{go}).
\]
Database Processes

- Each PPS database is represented by a database process
- Each has a stream connection to a disk process, which it can request to retrieve and store source and object code
- A database accepts queries, metaqueries and commits.
- Queries and metaqueries are processed at any time, and their Done variable recorded
- Commits are only processed if there are no concurrent accesses to database (all Done variables are bound).

\[
\text{mode database}(R_s?, D_b?, Source?, \text{DoneVars}?).
\]

\[
database([\text{message}(D_b,q(Q,D),R) / R_s], D_b, S, DVs) \leftarrow \quad R = Q, \quad \text{database}(R_s, D_b, S, [D / D_b]).
\]

\[
database([\text{message}(D_b,mq(Q,D),R) / R_s], D_b, S, DVs) \leftarrow \quad handle_{mq}(Q, S, R),
\]

\[
database(R_s, D_b, S, [D / DVs]).
\]

\[
database([\text{message}(D_b,commit(Us,Ok,Go),R) / R_s] \quad D_b, S, DVs) \leftarrow \quad try_{commit}(Us,Ok,Go,DVs,DVs',S,S') \; & \; \text{database}(R_s, D_b, S', DVs').
\]

57
Commitment: Databases

- A *commit*(Us,Ok,Go) message indicates a query wishes to commit

- If no concurrent executes or reads to database, signal that commit may proceed (*Ok = ok*)

- If query signals that commit should proceed (*Go = go*) apply updates *Us*

\[
\text{mode try_commits}(Us?,Ok\uparrow,Go?,DVs?,DVs'?,S?,S'\uparrow).\\
\text{try_commits}(Us,ok,Go,DVs,[ ]\!,S,S') \leftarrow \\
\quad \text{check}(DVs), \\
\quad \text{valid_updates}(Us) : \\
\quad \text{try_commits2}(Us,Go,S,S'); \quad \% \text{Note sequential OR} \\
\text{try_commits}(Us,abort,Go,DVs, DVs, S,S).
\]

\[
\text{mode try_commits2}(Updates?,Go?,Source?,Source'\uparrow).\\
\text{try_commits2}(Us,go,S,S') \leftarrow \text{apply_updates}(Us,S,S'). \\
\text{try_commits2}(Us,abort,S,S).
\]

\[
\text{check}([D \mid DVs]) \leftarrow \text{not}(\text{var}(D)): \text{check}(DVs). \\
\text{check}([\ ]).)
PPS: Summary

- The PPS is a sophisticated programming environment / operating system for PARLOG, written in PARLOG

- Individual relations in its implementation can be read as specifications of programming environment components

- Components execute concurrently

- Directional unification (dataflow) is used to constrain reduction to obtain desired operational behaviour

- Back communication and synchronization variables provide a succinct representation of a parallel algorithm: distributed database update

- The control metacall is used to initiate, monitor and control user computation

- Exception messages provide for communication between user tasks and operating system
PARLOG Implementation

Two approaches: Shallow or-parallel (non-flat) and flat.

- **Non-flat**: user defined predicates in guards ⇒ AND-OR tree.
- **Flat**: only simple tests in guards ⇒ process pool.

Non-flat implementation work: restricted to PARLOG.

Flat implementation work: Flat PARLOG; FCP; FGHC.

(Flat PARLOG + metacall = Full PARLOG)

- **Non-flat PARLOG**: AND-OR tree computational model
  
  Sequential PARLOG Machine (SPM)  Sequential
  PARLOG on Prolog  Sequential
  **Shared Memory Multiprocessors**  Parallel

- **Flat PARLOG**: Process pool computational model
  
  ALICE packet rewrite machine  Parallel
  **Flat PARLOG Machine (FPM)**  Sequential/
  (Distributed)

- **PARLOG control metacall**
  
  Sequential PARLOG Machine
  **Flat PARLOG Machine

  (** = described here)
Shallow Or-Parallelism in PARLOG

*Deep guards* mean that program execution may create an *AND-OR tree.*

PARLOG's *guard safety* mean that multiple environments are *not* required during OR-parallel evaluation

\[
f \leftarrow g_1 : b_1. \quad g_1 \leftarrow a, b. \quad g_2 \leftarrow c. \quad f \leftarrow g_2 : b_2.
\]

AND/OR Tree

Nodes may be regarded as processes

**Non-leaf nodes** await evaluation of offspring

**Leaf nodes** correspond to reducable or suspended processes
Crammond's Process Model

Two types of process:

Goal processes - (AND processes)
responsible for creating child clause processes

Clause processes - (OR processes)
responsible for executing a single clause

Process States:

Runnable, executing  Suspended on variable
Runnable, queued    Suspended on child

Execution:

- A goal process creates a clause process for each clause
- Each clause process spawns goal processes to solve guard goals
- First clause process to succeed with guard commits and spawns goal processes to solve body goals
- Processes communicate by signals

Process Signals: DONE + QUIT

Goal process succeeds  -------->  DONE
Goal process fails      -------->  QUIT

Clause process succeeds -------->  QUIT
Clause process fails        -------->  DONE
Implementation: Sequent Multiprocessor

- Sequent: shared memory multiprocessor with hardware locks
- One Unix process / processor; executes PARLOG processes
- Each processor has its own data areas
- Locks used when writing shared data areas
- Procedures are encoded using Warren-like code
- Processes are represented as data structures

- Processors 'steal' processes from neighbours when necessary
- Optimisations for flat programs:
  do not generate clause processes if a test on a single argument can reduce choice to a single clause
- Significant speedups achieved: 4.6 times on 5 processors
Flat PARLOG Computational Model

- Computation = process pool

\[
p \leftarrow a, b, c, d.
\]

\[
d \leftarrow g1 : b1, u, v.
\]

\[
d \leftarrow g2 : b2.
\]

- Repeatedly select and attempt to reduce processes using clauses defining procedure

- Three phase reduction cycle:

  \[\text{test then bind then spawn.}\]

Input mode arguments + guard calls define \textit{tests}

Output mode arguments + '=' calls define \textit{binds}

\[
\begin{align*}
\text{test} & \quad \text{bind} & \quad \text{spawn} \\
gl & \rightarrow b1 & \rightarrow u, v \\
?d & \rightarrow & \\
g2 & \rightarrow
\end{align*}
\]

- A process try may succeed, fail or suspend.
Compilation of Flat Parlog

Flat Parlog:

mode lookup(Key?, Data^, Dict?, NewDict^).

lookup(K, D, [{K,D} | Ds], [{K,D} / Ds ]).

Standard Form:

lookup(K,D, A3,A4) ←

[{K1,D1} | Ds] ← A3, K == K1 : D = D1, A4 = [{K1,D1} / Ds].

<test>

Compiled Form:

[P | Ds] ← A3
{K1, D1} ← P
K == K1
D = D1
A4 = [W / X]?

var

list

X := Ds

W := {U,V}

U := K1

V := D1

X = Ds

var
tuple/2

U := K1

U = K1

V := D1

V = D1
Compilation of Flat Parlog contd

• Compile unification to basic operations:

\[
\begin{align*}
T & \leftarrow V & \text{Input matching} & \text{test\_integer, test\_list} \\
V & := T & \text{Assignment} & \text{put\_integer} \\
V & = T & \text{Binding} & \text{bind\_integer, bind\_list} \\
V & = V & \text{General unification} & \text{unify} \\
V & == V & \text{Equality} & \text{equals}
\end{align*}
\]

• All instructions single mode (alternative: moded instructions)

• Input matching and output unification compiled
Lookup/4:
\[
\text{load(4)} \\
\text{try_me_else(Lookup4)}
\]
\[
\text{test_list(2,4)} \quad [P \mid D] \leftarrow A_3 \\
\text{test_tuple(4,2,6)} \quad \{K_1,D_1\} \leftarrow P \\
\text{equal(6,0)} \quad K == K_1
\]
\[
\text{unify(1,7)} \\
\text{bind_list(3)} \\
\text{get(8)} \\
\text{put_value(5)} \\
\text{put_tuple(2,8)} \\
\text{put_value(0)} \\
\text{put_value(7)} \\
\text{halt}
\]
Lookup1:
\[
\text{get(2,8,11)} \\
\text{bind_tuple(8,2,Lookup3)} \\
\text{put_value(0)} \\
\text{put_value(7)}
\]
Lookup2:
\[
\text{unify(11,5)} \\
\text{halt}
\]
Lookup3:
\[
\text{get(2,9,10)} \\
\text{unify(9,0)} \\
\text{unify(10,7)} \\
\text{goto(Lookup2)}
\]
Lookup4:

*Note absence of a commit operator*
Flat Parlog Machine

Process pool computational model +

- scheduling structure:
  to avoid selecting suspended processes

- tail recursion:
  to avoid repeated selection of processes

![Diagram of process pool computational model](image-url)
PARLOG Control Metacall

\[ \text{call(Module,Priority,Goal,Status,Control)} \]

- Initiate a computation
- Signal termination of a computation
- Allow control of a computation
- Provide fair and prioritized scheduling of a computation

Implementation via extensions to Flat PARLOG Machine

(a) New computational model

Flat PARLOG + metacall = computation tree

A process pool may contain subpools

- A subpool is a computation or task
- Reduction repeatedly selects \{computation,process\} pair
Extensions to Flat PARLOG Machine, contd:

(b) Allow for termination of a computation

- process failure = success of computation + failed status message
- no processes in a computation = success of computation
- no reducable processes in a computation = computation deadlock

(c) Fair and prioritized computation scheduling

- fairness = bounded depth-first round-robin scheduling
- priorities = directed selection of \{computation, process\} pair

*Benchmarks on uniprocessors (SUN-3) show only 2-3 % performance degradation when FPM is extended to support control metacall.*
Parallel Execution of Flat PARLOG

(Based on Taylor's FCP Implementation)

Assume:

- many processors connected by regular interconnection topology (mesh, hypercube)
- no global memory; each processor has local memory
- processors may communicate by message passing
- Flat PARLOG computation (process pool) distributed over many processors
- each processor executes Flat PARLOG Machine

Three issues:

- distributed unification (discussed here)
- distributed computation control (metacall)
- load balancing, code mapping
Distributed Unification

PARLOG:

mode f(?,^).
f(1,2) <- g.

Standard form:
f(X0,X1) ← X0 ← 1 : X2 = 2, g.

Reduction Cycle:

test: X0 ← 1 ? (read) } may encounter

bind: X1 = 2 (write) } remote references

spawn: g

Remote reference = processor number + address.

Unification + remote refs = extended unification algorithm:

- test phase: generate read messages and suspend process if remote references encountered (e.g. X0 is remote)
- bind phase: generate unify message if remote reference encountered (e.g. X1 is remote)

Messages dereference remote references until values or variables found
Distributed Unification: Reading

1. \( X \leq 1 \) : value is available.

\[
\begin{array}{c}
\text{P1} \\
\quad \text{read}(P1,X,X') \\
\quad \text{value}(X,1)
\end{array}
\]

2. \( X \leq 1 \) : value is not available

\[
\begin{array}{c}
(a) \\
\quad \text{P1} \\
\quad \text{read}(P1,X,X')
\end{array}
\]

\[
\begin{array}{c}
(b) \\
\quad \text{P1} \\
\quad \text{P2}
\end{array}
\]

\[
\begin{array}{c}
(c) \\
\quad \text{P1} \\
\quad \text{P2}
\end{array}
\]

\[X = 3\]
Distributed Unification: Writing

(a) Both local

(b) One local, One remote

(c) Both remote

\[
\text{value}(X) = Y
\]
Recent Developments

1. Move to Flat Languages

Recall: A flat parallel logic language only permits calls to system primitive predicates in clause guards.

Why flat?

(a) Necessity

\[
\begin{align*}
\text{CP} & \rightarrow \text{FCP} \quad \text{appears} \\
\text{GHC} & \rightarrow \text{FGHC} \quad \text{essential for implementation}
\end{align*}
\]

(b) Convenience

\[
\text{PARLOG} \rightarrow \text{Flat PARLOG} \quad \text{simplicity + efficiency}
\]

(Recall: Flat PARLOG + metacall = PARLOG)

Justification: two suppositions:

"Most programs are flat"

"Flat languages can be implemented more efficiently"

Both of which are true ... some of the time

- certain applications (e.g. AI) make extensive use of PARLOG's shallow OR-parallelism
- clever PARLOG implementations (e.g. Crammond's) can approach flat language efficiency
Recent Developments

2. Restoring Completeness

PARLOG and GHC are correct but incomplete logic languages

In certain applications, this lack of completeness is a disadvantage

2.1. P-Prolog

- incorporates exclusive relations

- synchronization mechanism: an exclusive relation cannot reduce until at most one clause is applicable

Example:

\[
\text{append}([X \mid Xs], Ys, [X \mid Zs]) \leftarrow \text{append}(Xs,Ys,Zs). \\
\text{append}([], Ys, Ys).
\]

This cannot reduce until its first OR second and third arguments are instantiated.

- Permits multi-moded relations

- Alternative syntax permits non-exclusive relations

- Not clear that P-Prolog can be efficiently implemented
Recent Developments

2.2 PARLOG & Prolog United

- PARLOG has two set constructor interfaces to pure Horn clauses:

  \[ \text{set}(\text{Solutions} \uparrow, \text{Term}?, \text{Conjunction}?) \]
  \[ \text{incrementally binds } \text{Solutions} \text{ to a list of solutions} \]

  \[ \text{subset}(\text{Solutions}?, \text{Term}?, \text{Conjunction}?) \]
  \[ \text{generates solutions as } \text{Solutions} \text{ is bound to a list of variables} \]

- Recently, Clark and Gregory propose linking PARLOG and Prolog more tightly

- General two-way communication between Prolog and PARLOG computations

- Applications and implementation not yet clear

2.3 Compiling Prolog to Parallel Logic Languages

- Ueda, Tamaki describe techniques for compiling Horn clause programs to GHC

- Can handle exhaustive search programs and generate and test

- Restrictive: modes must be specified at compile-time
Who's Using Parallel Logic Languages?
(Partial list)

Research groups:

PARLOG Group, Imperial College, London.
ICOT Research Centre, Tokyo.
Weizmann Institute, Israel
Vulcan Group, Xerox PARC

Industry in Uk, Sweden, Japan, ...

Implementations

Distribution

PARLOG Group: PARLOG on Unix and Prolog
Weizmann: FCP on Unix
ICOT: FGHHC on Prolog

Experimental

PARLOG Group: Sequent, ALICE, Flat Parlog Machine
Weizmann: Hypercube
ICOT: multi-PSI
others in UK, Sweden, Japan, ...
Directions for Future Development

- Fast parallel implementations
  
  On custom hardware:
  
  ICOT: multi-PSI 100K RPS/PSI-II x 100 = ...

  Conventional machines
  
  native code compilation of FCP: 75K RPS on SUN-3s

- Formal semantics
  
  facilitating:
  
  program transformation
  program analysis and debugging

- Declarative programming environments

- Language extensions and new languages
  
  restoring completeness
  constraints
  typing

- New applications
  
  distributed systems
  AI

- Commercial exploitation
  
  as a specification and implementation language for
  distributed systems

  for parallel symbolic processing
Conclusions

• Parallel logic languages =
  
  concurrent evaluation
  Horn clause logic + dataflow synchronization
  committed-choice non-determinism

• Declarative content makes parallel logic programs:
  easy to understand
  easy to transform and analyze

• Parallel evaluation permits parallel symbolic processing

• Operational characteristics make parallel logic languages a powerful programming formalism

• Efficient parallel implementations are being developed

• Broad range of applications

• Parallel logic languages are a viable language for expressing and implementing parallel algorithms
Selected References

1. Languages


Foster, I.T. and Taylor, S. 1987, "Flat PARLOG: a basis for comparison". Research report DOC 87/5, Imperial College, London. Contrasts Flat PARLOG and FCP, presents benchmarks comparing their efficiency and describes an abstract machine for the implementation of Flat PARLOG.


2. Applications


3. Implementations


