ATTACHMENT 4, 14 PAGES
(DOUBLE SIDED)

Parallel Processing using Flat Concurrent Prolog

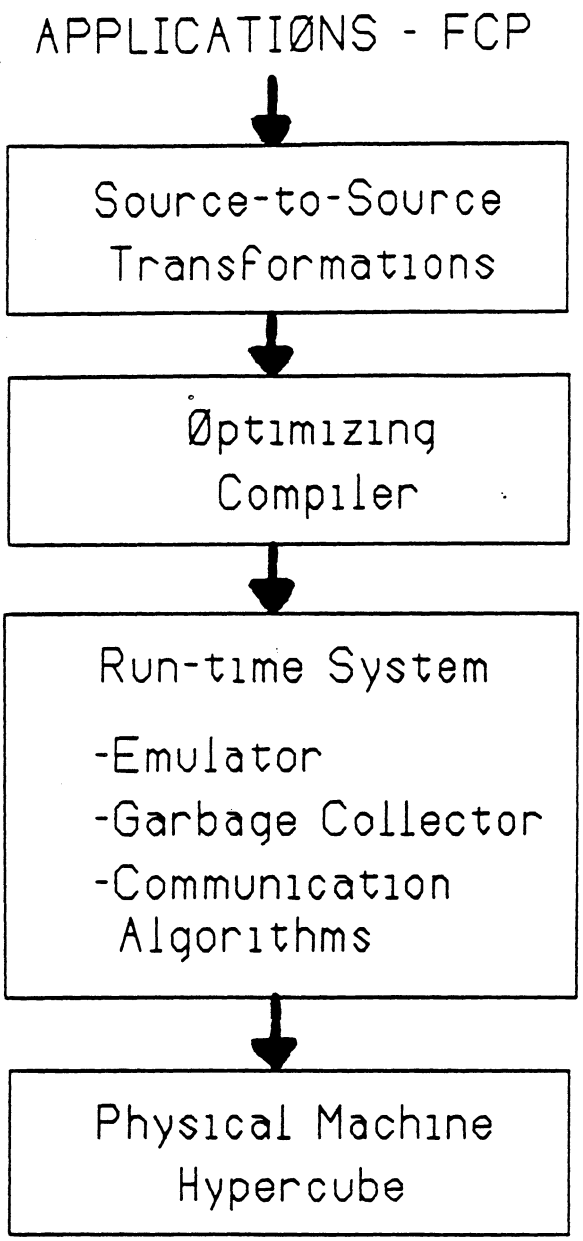
Stephen Taylor

Weizmann Institute of Science
Department of Computer Science
Rehovot, Israel

CONSTRUCT APPLICATIONS BY RECURSIVELY SPECIFYING PROCESS STRUCTURES

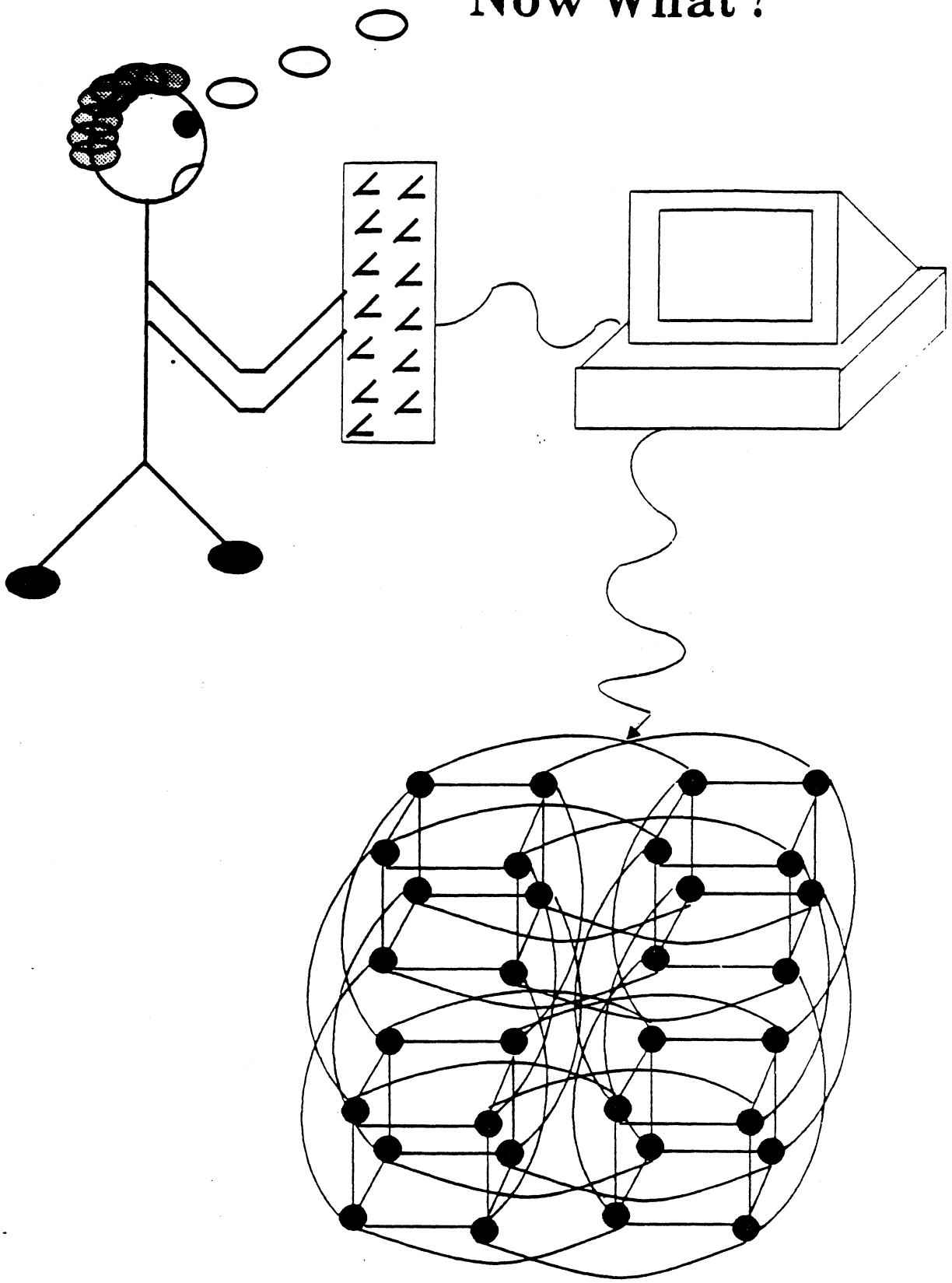
- Looking at various PROBLEMS
- Framework - Flat Concurrent Prolog
- Simple Process Oriented P.L.
- Symbolic / Numeric Applications
  - 2 Directions
    - Investigate Fine Grain Computation
      - Daily
    - "GLUE"
      - Rapidly Put Together Large Grain Applications
      - Deligate Numerics Code to 'C' / FORTRAN
      - Communication + Control at Higher Level of Abstraction

- Complete, Integrated, Working System



- Each section = approach specific problems
- Seminar
  - look at slices
  - highlight problems + solutions
  - keep in mind - part of cohesive whole

Now What?



Reducing Complexity - Approach

- Problems:
  - perform load balancing
  - conveniently map applications/code
- general algorithms, compiler/run-time system
  - not flexible, complicate implementation, semantics?
- simple tools, convenient abstractions
- isolate generic techniques, parameterize, libraries
  - most of time libraries suffice
  - specialize/optimize for particular application
  - implemented in language, no semantic support

FCPic

- non-trivial FCP application - 600 lines
- Polygon based graphic modeling system
- I/P - high level spec. of drawing

$$square(X),\ bigger(X, 10, X1),\ move(X1, 10, x, X2)$$

- O/P - x/y coordinates of each line in drawing
- primitive polygons, transformations, objects

- Dynamic Process Mapping
- Dynamic Load Balancing
- Dynamic Code Mapping

Dynamic Process Mapping

- Map application to a convenient virtual machine
  - abstraction

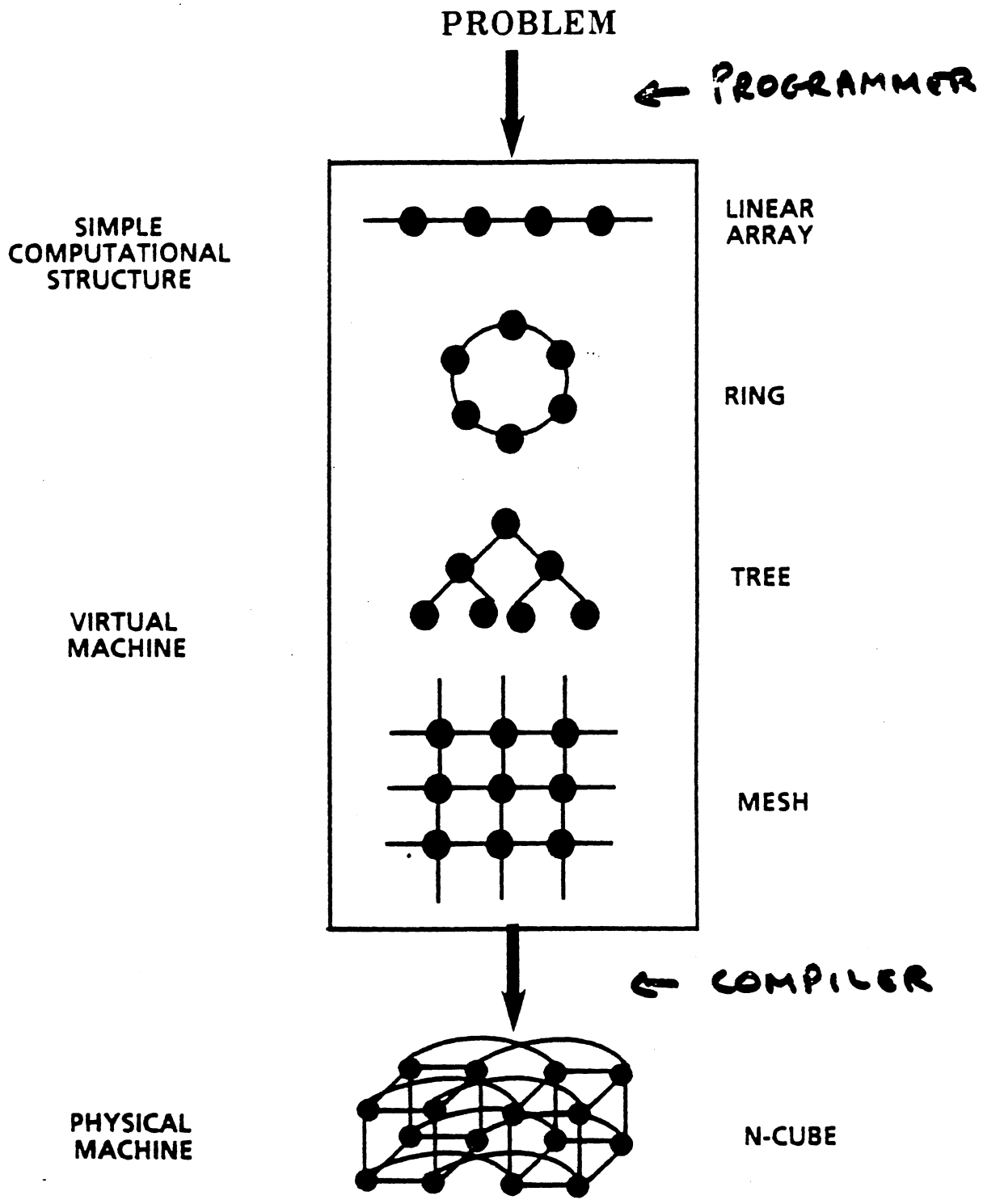
- Compiler maps virtual machine to physical machine
  - libraries

- maintain proximity
  - adjacent nodes in VM adjacent (or near) in PM
  - not so important (wormhole routing)

- Specify VM using meta-interpreter
- Implement via pre-processor

- Many applications on single VM concurrently
- Many VM's concurrently
- Nest VM's in hierarchy (where topology permits)

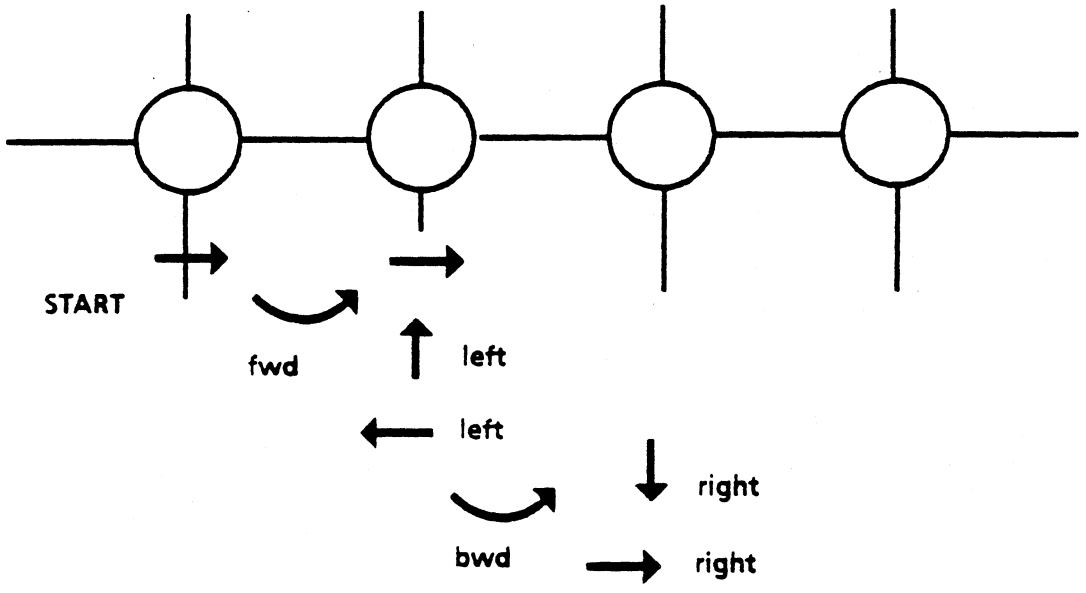
Basic Approach



Turtle Programs

- tool for mapping processes to VM
- similar to turtle programs (Papet)
- processes have position + heading
- specify where processes executes
- Mesh
  - P@T - execute process P at processor T
  - T can be:
    - fwd, bwd - cause process spawning
    - left, right - change heading 90 degrees

- process(...)@(fwd,left,left,bwd,right,right)



- goal is executed 2 processors away in current direction

Streams

- Variables are single assignment
- How to achieve multiple communications with single variable?
  - streams represented by incomplete lists.
  - bind shared variable to cons cell
  - contains new shared variable
  - used in subsequent communications

e.g. $f(X)$, $g(X?)$

$$X = [a \mid X1]$$

$$X1 = [b \mid X2]$$

$$X2 = [b \mid X3]$$

:

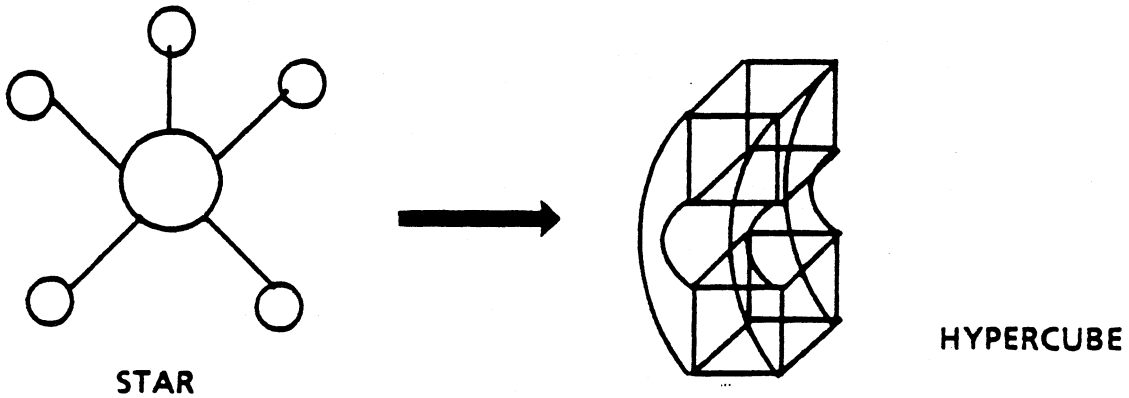
:

etc.

Dynamic Process Mapping

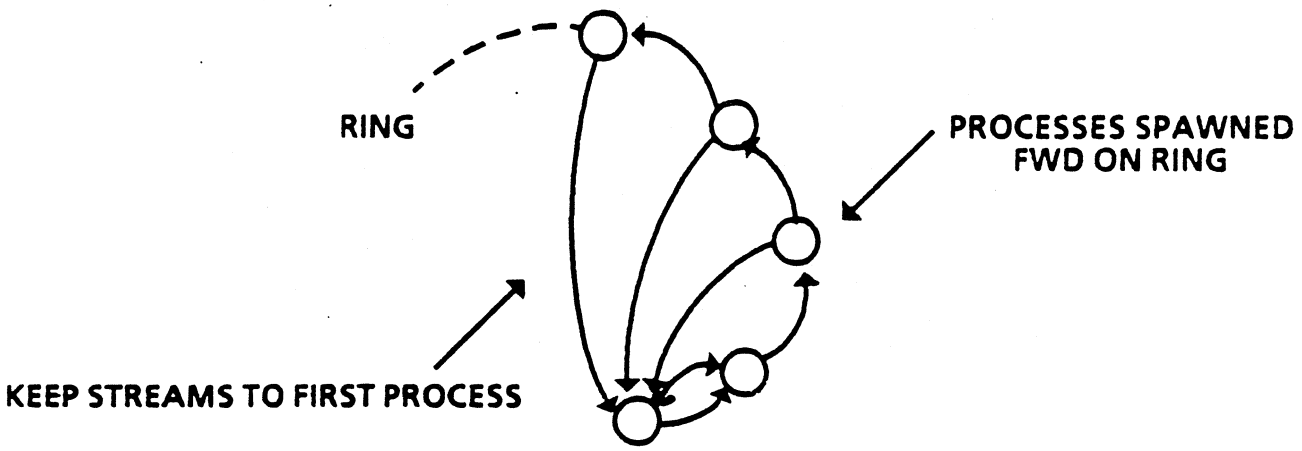
- Use turtle programs

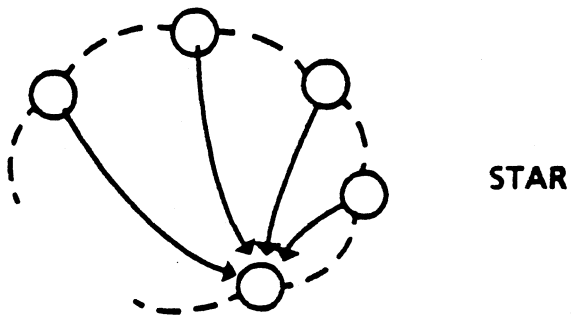
PROCESS STRUCTURE



NOT CONVENIENT

- What structure - mesh?, ring?...
- Try ring.
- Recursive process spawning.





- Outline:

$start \leftarrow$
  $center,$
  $spawn\_star@fwd.$

$spawn\_star \leftarrow$
  $tip,$
  $spawn\_star@fwd.$
$spawn\_star.$

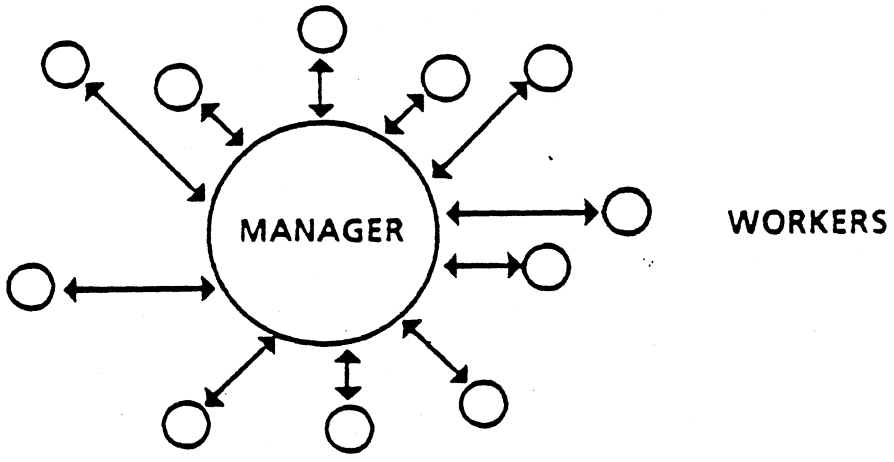
- Code:

$start \leftarrow$
    $center(Streams?),$
    $spawn\_star(15, Streams)@fwd.$

$spawn\_star(N, [S \mid Ss]) \leftarrow$
  $N > 0,\ N1 := N - 1 \mid$
    $tip(S?),$
    $spawn\_star(N1, Ss)@fwd.$
$spawn\_star(0, []).$

Dynamic Load Balancing

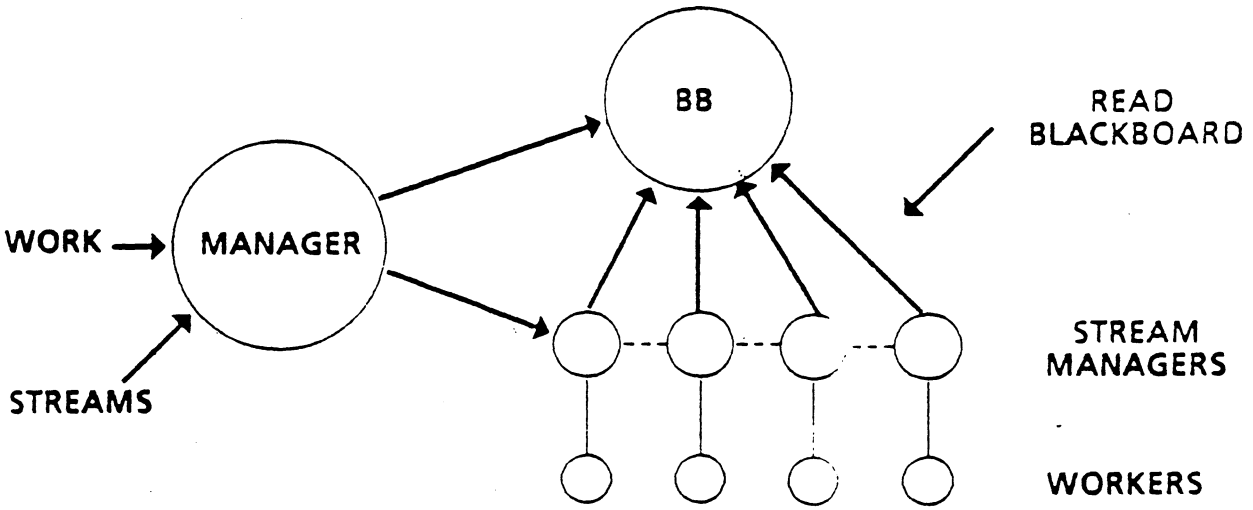
- Example - Generic Technique
  - Manager/Worker (Brandenburg)



- Manager (center)
  - Partitioning/load balancing
  - Gives subproblems to workers
  - Receives replies when worker needs more
- Worker (tip)
  - solve subproblem
  - request more work
- Want Overlapping
  - sends next subproblem while worker executing
- No. subproblems = 10 x No. Processors
  - ramp up/ramp down

Manager Action

- Create blackboard of work
- Spawn stream managers
  - One per worker
  - Responsible for obtaining work from blackboard
  - Coordinates synchronisation of worker.




```
manager:-
  blackboard,
  spawn-managers.

spawn-managers:-
  stream-manager,
  spawn-managers.
spawn-managers.
```

```
manager (Work,Streams):-
  blackboard (Work?, BB),
  spawn-managers(15,BB?,Streams?)

spawn-managers(N,BB,[S|Ss]):-
  N>0,N1:=N-1 |
      stream-manager(N,BB,S),
      spawn-managers(N1,BB,Ss?).
spawn-managers(-,-,[]).
```

Stream Manager

- Complex Synchronization Task
  - remove objects from blackboard
  - mutual exclusion, subproblem per stream manager
  - coordinate sending subproblems to workers
    - overlapping
- Mutual Exclusion
  - all have unique identifier
  - all concurrently try remove subproblem
  - blackboard has following form:

$[\{subproblem1, Var1\},\ \{subproblem2, Var2\}...]$

  - Succeed - bind variable to identifier, send worker
  - Fail - if already bound, try to take next
  - atomic unification - provides a test and set

$stream\_man(N, [\{S, N\} \mid BB], [\{S, Ok\} \mid Ws], ok) \leftarrow$
$\quad stream\_man(N, BB?, Os, OK?).$
$stream\_man(N, [\{\_, M\} \mid BB], Os, OK) \leftarrow$
$\quad N \neq M \ \mid \ stream\_man(N, BB?, Os, OK?).$
$stream\_man(\_, [], [], \_).$

  - remove and send, keep looking, terminate

Performance

- 2 approaches, 2 problems
  - systolic algorithms: MM, merge sort
  - dynamic management: FCPic, Prolog
- Systolic Programming

	Merge Sort Ordered Input (secs)	Merge Sort Reversed Input (secs)	Matrix Multiply (secs)
Uniprocessor	155.21	191.83	372.29
d-4 Hypercube	61.35	56.38	59.92
Speedup	2.53	3.40	6.22

- Dynamic Management

	FCPic - Small Problem (mins)	FCPic - Large Problem (mins)	Prolog (secs)
Uniprocessor	51.78	112.2	560
d-4 Hypercube	5.54	9.35	58
Speedup	9.4	12	9.7

  - small problem extrapolate
  - max = 15

Dynamic Code Mapping

- Code = First Class Object
  - manipulated as string
  - passed around/executed

- Definition of VM
  - processor with disk - "host"
  - uniform: $get\_code(Module)@host$
  - retrieve code and execute

- Already Shown
  - can dynamically map processes to any processor
  - can keep stream to any processor

- While Spawning
  - keep stream to processor with disk - "host"
  - dynamically allocate code by sending messages
  - begin to see mapping annotations = messages
  - generated at compile time - pre-processing

- Granularity and Load Balancing
  - Granularity = Reductions/Communication (msgs)
  - Computation Balance = SDr/Mr
  - Communication Balance = SDc/Mc
- No temporal information, serves to highlight

	Granularity (Reds/Msg)	Comp. Bal.	Comm. Bal.
FCPic	218.2	0.11	0.25
Merge Sort	3.46	0.27	0.65
Matrix Multiply	4.4	0.0	0.28

- FCPic = 50 x larger grain/well balanced
  - MM - spread columns, pipeline rows, inner product
  - increase granularity, pack more columns

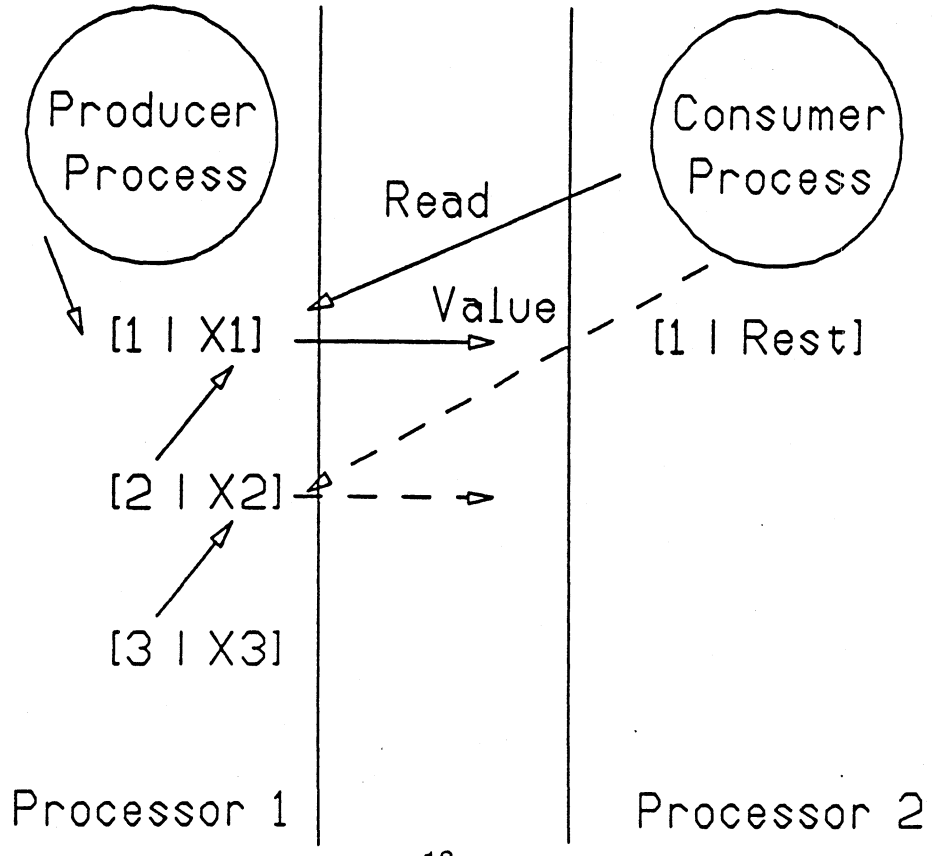
	Speedup	Granularity (Reds/Msg)	Comp. Bal.	Comm. Bal.
Original	6.22	3.46	0.27	0.65
4 Column	10.98	4.4	0.0	0.28

  - granularity x 3, remains well balanced
  - speedup increases to 10.98
  - facilities

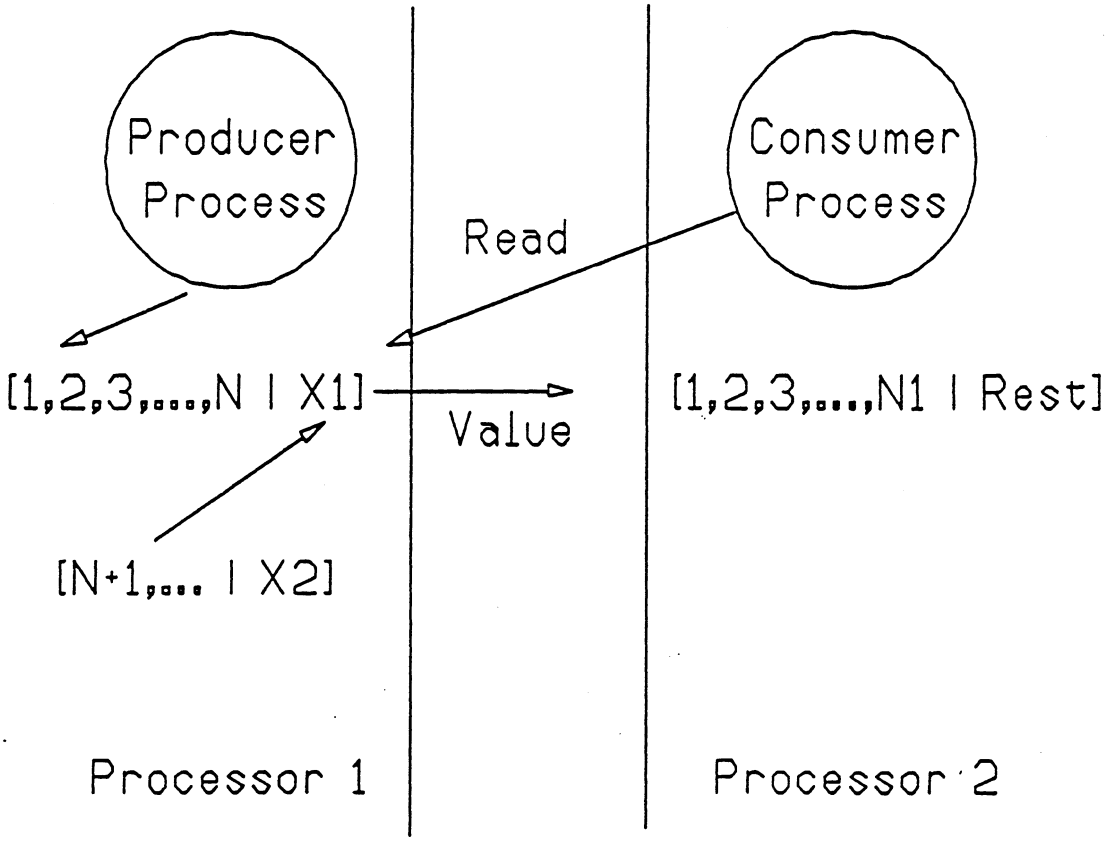
Communication Stereotypes

- Small number programming techniques
  - Stream Producer/Consumer
  - Incomplete Messages
  - Bounded Buffers
  - Blackboards
  - Short Circuit
- Abstract into stereotypical program fragments
  - understand, quantify
  - compiler optimizations
    - aim to improve communication speed

- Stream Producer/Consumer



- naive implementation, not efficient,optimise
- recursive (iterative) processes



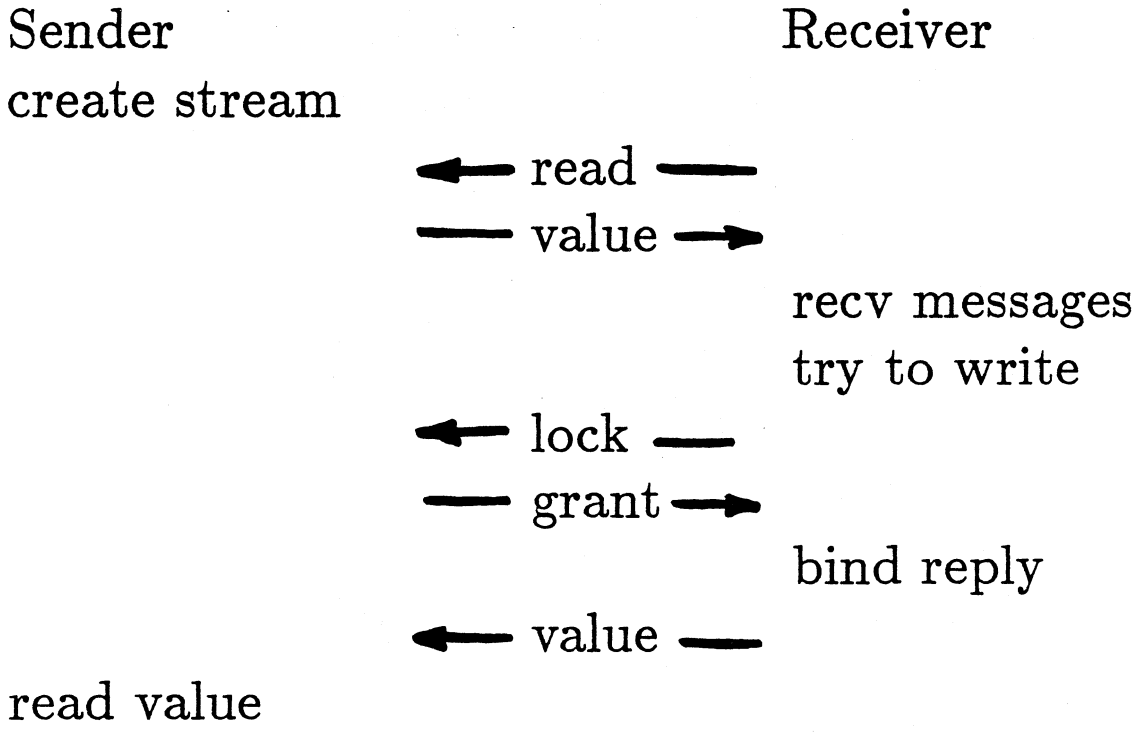
No. of times process iterates = Timeslice (TS)
No. of levels of structure copied = Copy-depth (CD)

	Time (secs)
Uniprocessor	10.1
2 Processors	9.61
Speedup	5%

straightforward, simple protocol, more complex

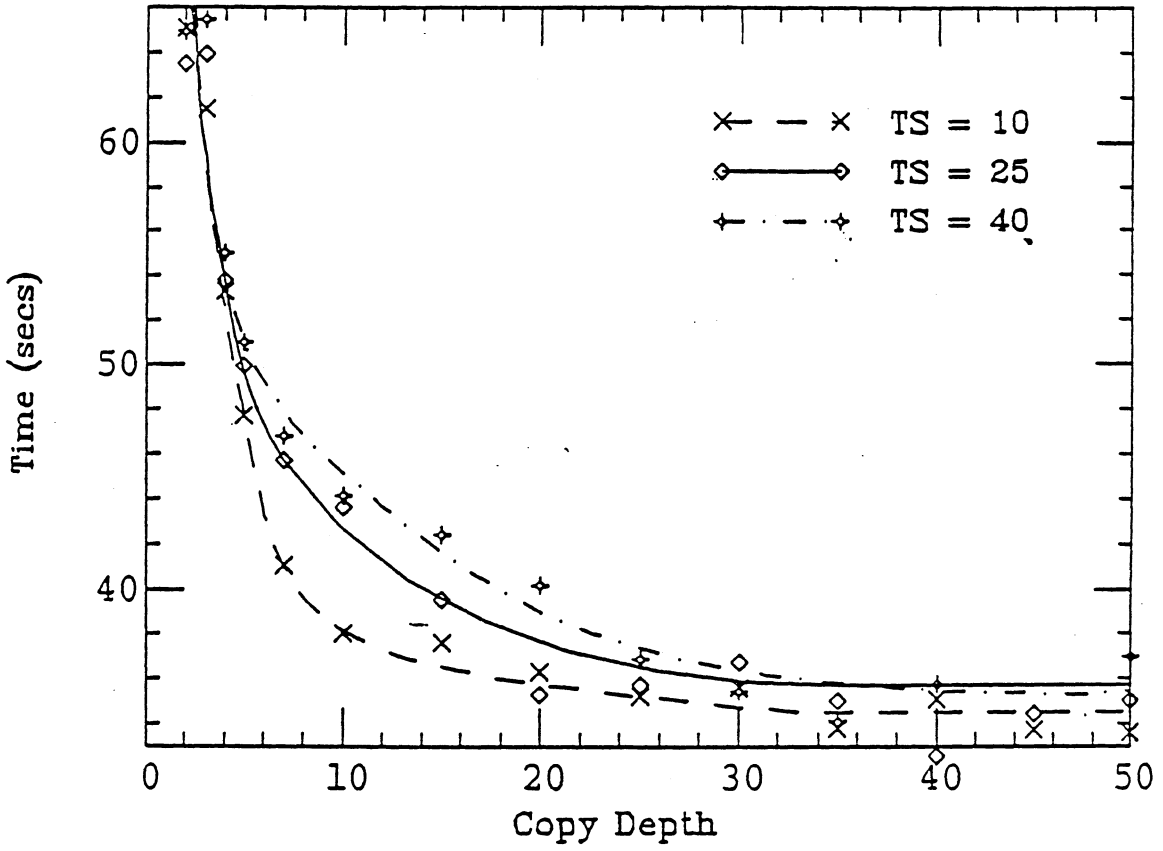
Incomplete Messages

- message contains reply slot (variable)
- combine with mergers
  - receiver reply without sender identify
- Sender
  - $[msg(X1),\ msg(X2),\ msg(X3), \ldots]$
  - $X1? = reply$
- Receiver
  - reads the stream of messages
  - binds reply variables e.g. $X1 = reply$
- Protocol



	Time (secs)
Uniprocessor	33.6
2 Processors	58.9
Increase	1.75

- degradation - locking, extra messages



- 2 parts - separate at TS
  - $CD < TS$
    - performance increases with CD
    - producer runs ahead
  - $CD > TS$
    - consumer faster, forced to wait for producer

- more generally
  - characterize mathematically
  - take TS=CD
  - based on relative speeds of
    producer (R_p) and Consumer(R_c)

$R_p \ll R_c$: $T \approx (N/TS)(P+B)$

$R_p \gg R_c$: $T \approx (N/TS)(C+R+B)$

N - No stream elements
P - time for producer execute TS reductions
C - time for consumer execute TS reductions
B - buffering time at sender
R - time to send request at sender

- no buffering time at consumer - rel. refs

- improve speed
  hardware - reduce B
  uniprocessor speed - reduce P and C
  less messages - reduce R

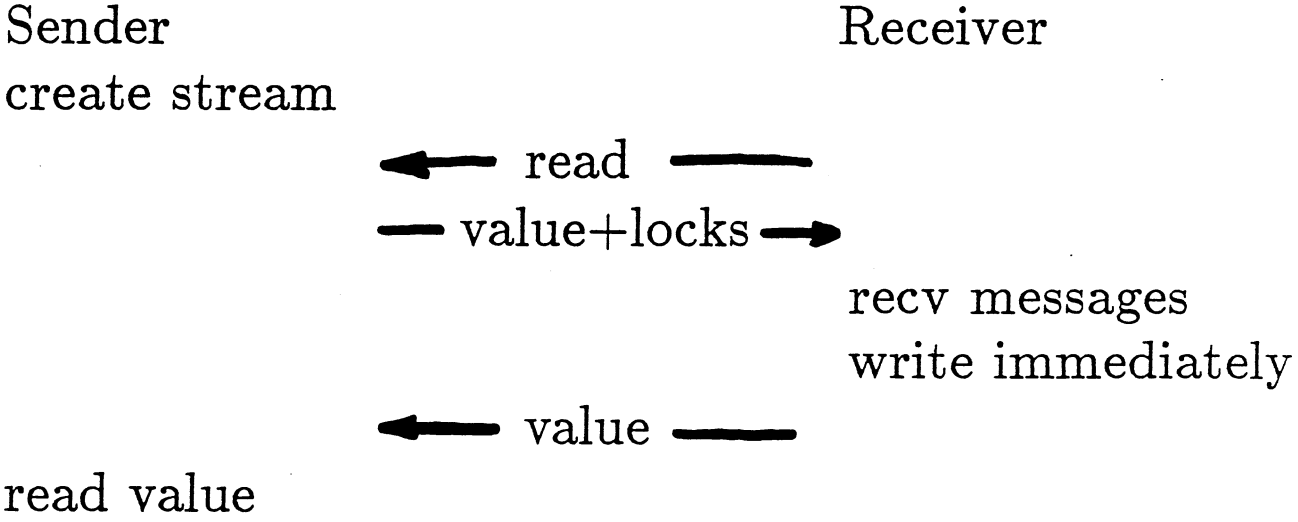
Optimizations

- Global Analysis - detect consumers (C)

$c(..., [msg|Xs], ...) \leftarrow ..., c1(..., X1?, ...), ...$
$c(..., [msg|Xs], ...).$
$c(..., [], ...).$

  - stream operations to known arg. posns.
  - iterative calls in known body posn.
  - generate code uses info.

- Pragmas
  - e.g. variable is only read locally?
  - incomplete message



	Time (secs)	Msgs	Suspensions
Original Uniprocessor	33.6	-	-
Original 2 Processors	58.9	16201	10600
Optimized 2 Processors	35.59	6601	5800
Improvement	40%	59%	45%

- Finally - What About B??
  - detect producers at compile time
  - conventional compiler techniques - open loops
  - possible to allocate messages on fly

Algorithm - Guiding Principles

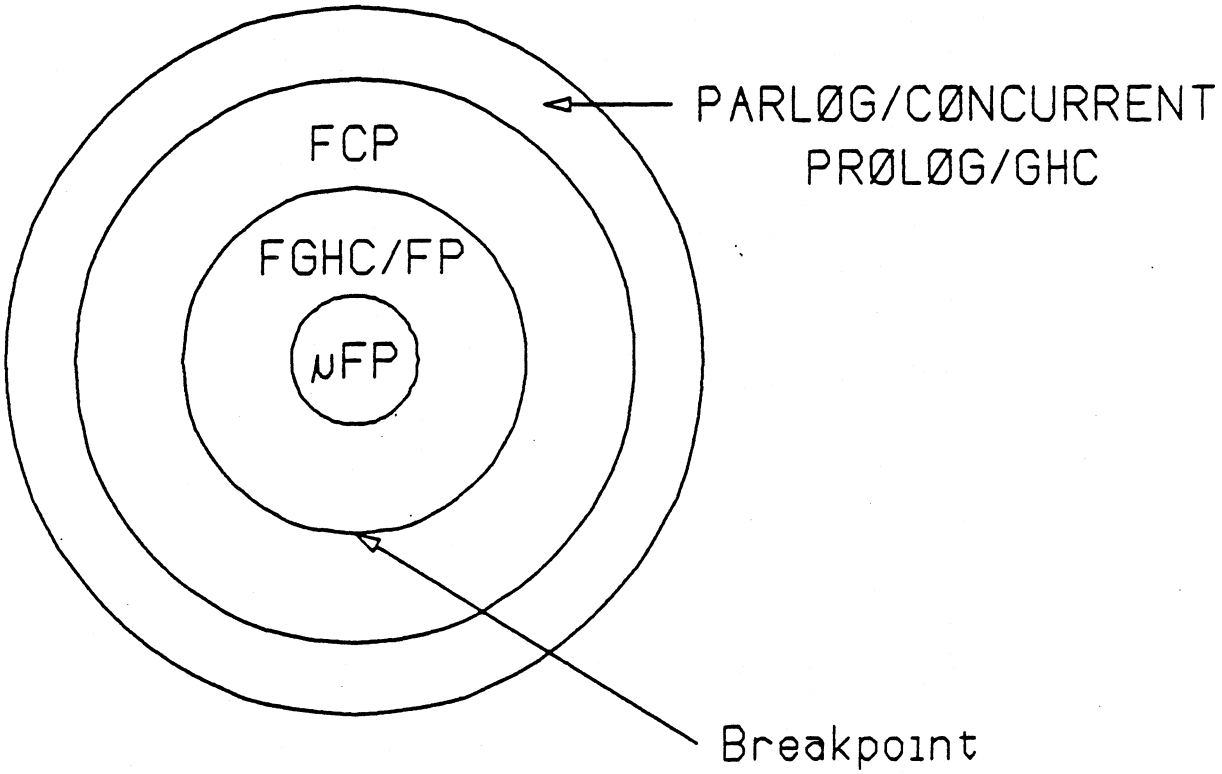
- MIMD, message passing, topology unimportant
- Simulate global address model of language
  - remote references + communication
  - uniprocessor - access data structures via references
  - multiprocessor - access remote data via comm.
- Atomicity - localising writing
- 3 comm. behaviours: read, single/multiple write

Application	% Reading
Merge Sort	99.99
Matrix Multiplication	99.9
FCPic	61.0
Virtual Machines	87.2
Symbolic Differentiation	96.6
Naive Reverse	98.8

- Most writing occurs locally - no comm.
- Algorithm distinguishes reading/writing
  - reading efficient, no locking, simple protocol
- Favour single writers
  - deadlock cannot occur, process write immediately
- Multiple writers - atomic transactions
  - simple deadlock prevention scheme, rarely invoked
  - two phase locking (databases)
  - break symmetry, processor number - decide (OS)

Future Research

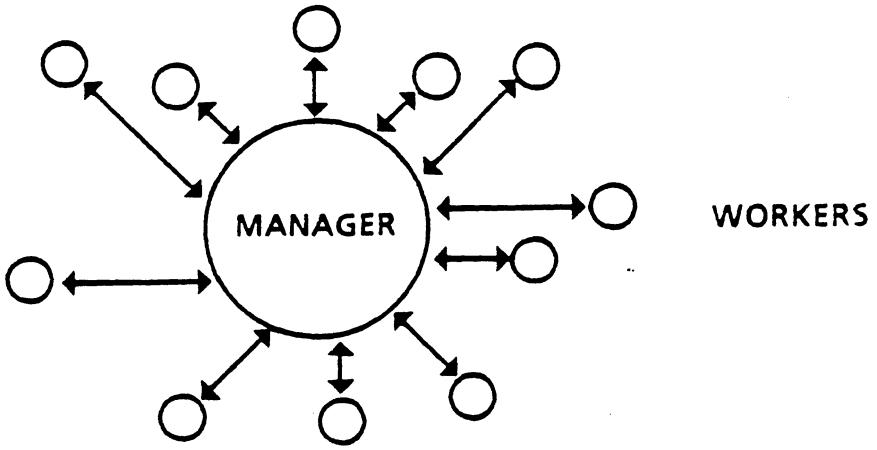
- Problems due to framework
  - 3 Semantics: $f(X, X?) = f(a, a)$
  - Implementation Complexity
    - Atomic transactions



- Focus on most simple framework
  - push as far as possible
  - develop hardware
  - examine tradeoff's between hardware/compiler
- Non-trivial applications
  - load balancing on irregular problems
  - more processors + perf. measurements

Dynamic Load Balancing

- Example - Generic Technique
  - Manager/Worker (Brandenburg)



- Manager (center)
  - Partitioning/load balancing
  - Gives subproblems to workers
  - Receives replies when worker needs more
- Worker (tip)
  - solve subproblem
  - request more work
- Want Overlapping
  - sends next subproblem while worker executing
- No. subproblems = 10 x No. Processors
  - ramp up/ramp down

Summary

- Introduce FCP as Framework to investigate problems

- Outlined language implementation problems
  - main: combining solutions in simple/efficient algorithm
  - guiding principles behind algorithm

- Example program
  - problems: mapping/load balancing
  - demonstrate programming techniques and concepts
  - simple tools, convenient abstractions
  - performance

- Communication Stereotypes
  - problem: how to improve communication speed
  - abstract common programming techniques into stereotypes

Why FCP?

- Parallel Semantics
  - encourage programmer to think
  - problems easier to decompose
  - freedom to compiler writer

- Simple + Uniform
  - execution based on reduction/unification
  - simple => efficient run-time operations
  - uniform => transformations - efficiency
    - partial evaluation/abstract interpretation

- Supports Meta-Programming
  - specify non-trivial system functions without complicating semantics
  - Process/Code Mapping, Debuggers, E.S. Shells etc.

- Practical Language - non-trivial problems
  - more than 30,000 working lines code
  - bootstrapping compiler
  - programming environment
  - sections of operating system
  - circuit simulation

Implementation Problems

- Atomic Transactions - atomic unification

  - Deadlock Prevention
    - multiple processes write shared variables

    e.g. $f(X,Y),\ \ g(Y,X)$

    - typically use locking mechanism
    - f locks X, prevents others, tries to lock Y
    - g locks Y, tries to lock X => deadlock

  - Starvation

    e.g. $f(X),\ \ g(X)$

    $$f(X) \leftarrow X = Y \mid f(Y).$$

    - f locks X, never writes
    - g may not get access

- Incomplete structures - streams
  - processes in different processors may compete to write tail, read sections
  - what communication?, GC?

- Data-Flow Synchronization
  - uniprocessor easy

    e.g. $X? = a$

  - X not bound, unify suspended using structure associated with X
  - when X bound by another process
  - unify woken up via structure
  - processes may reside in different processors

- Secondary Problems
  - Circular References/Terms
  - Garbage Collection - incomplete structures
  - Global Termination Detection

- Start: strange framework, End: standard problems
  - most difficult problem
  - combine solutions in simple/efficient algorithm

Sequencing and Synchronization

- Commit Operator
  - body only executed when guard solved

  Process: $f(6)$

  Program: $f(X) \leftarrow X > 5 \mid g(X).$

- Data-Flow - Read-only variable
  - variables marked read-only e.g. $X?$
  - if process attempts to bind X, suspends until bound by another process

  Processes: $f(X)$, $g(X?)$
  Program:

  $f(a).$
  $g(a).$

  pick g - suspends, pick f(X), X/a, wake g(a), a=a
  success X=a.

  pick f - X/a, pick g(a), a=a, success X=a

  X used for communication

Brief Introduction to FCP

- Single-Assignment,Data-Flow,Symbolic applications
- Computation = communicating processes - streams

$$is\_fat(fred, 200, Fat), \quad is\_fat(bert, 150, Fat1)$$

- Program = set of rules $H \leftarrow G \mid B.$

$$is\_fat(Name, Weight, fat) \leftarrow Weight > 150 \mid true.$$

- Computation Cycle - pick process, reduce it.
- Reduce => Process which *matches* head re-written to processes described by body if guard true.

$$Name/fred, \quad Weight/200, \quad Fat/fat.$$

- Result - bindings for variables
- Parallel Semantics
  - non-deterministic process/clause selection
  - processes any order and/or in parallel
- Unification - two way pattern matching
- Recursively defined data structures

$$fred(person, age(25), brothers([bert, george]))$$

Overview

- Why FCP?

- Brief Introduction

- Problems
  - Implementation
  - Algorithm Design - Programming Approach
  - Compilation

- Programming Approach
  - Non-trivial example - FCPic
    - Dynamic Process and Code Mapping
    - Dynamic Load Balancing
  - Performance

- Compilation - Communication Stereotypes
  - Abstractions
    - understand, quantify, optimize
  - Performance

Interests

- Long Term:

  - Relation between Hardware and Compiler
  - Tradeoffs at boundary

- In the beginning...

  - Construct applications by recursively specifying process structures.

  - What machinery necessary?
  - What practical problems arise?
  - Efficient solutions?

  - Practical problems => concrete P.L.

  - Framework - minimal

  - Flat Concurrent Prolog (FCP)