

# Languages and Formalisms for Parallel Computing

## *A Survey*

Vivek Haldar

vhaldar@ics.uci.edu

Information and Computer Science

University of California, Irvine

# Overview

- Taxonomy of parallel programming languages
  - *Unit* of parallelism
  - Communication and synchronization mechanisms
- Formalisms/calculi for parallel computing
  - Communicating Sequential Processes [Hoare]
  - Calculus of Communicating Systems [Milner]
  - Chemical Abstract Machine [Berry and Boudol]
  - Formalising Tuple Spaces?

# Languages vs. Libraries

**Library approach:** OS handles parallelism. Applications written in sequential language, invoking OS primitives.

*Superficially* advantageous:

- easily extend existing environment to handle parallelism

But there are problems:

- Difficult to express complex communication.
- Sending complex data structures as part of message — OS does not know about data layout — *language does*

**Why parallel programming languages?**

- higher level and more abstract programming model
- easier to reason about and understand
- automatic checking such as static type checking

# Unit of Parallelism

- Processes
- Statements
- Logical clauses
- Functions
- Objects
- Data parallelism

# Unit of Parallelism: Processes

- **Process:** logical sequential processor, has own state and data
- *Big advantage* — Familiarity. Every major OS has process concept.
- *Orthogonal* to language used — many languages have interfaces to call and manipulate underlying OS primitives for processes.
- Similar to *library approach*
- **Issues:**
  - creation — implicit or explicit?
  - termination

# Unit of Parallelism: Objects

- **Object:** encapsulates both *data* and *behavior*
- Possibilities of parallelism:
  - Object is active without getting message
  - Object continues execution even after returning result
  - Send messages to several objects
  - Sender and receiver run in parallel
- **Criticism**
  - Parallelism an afterthought
  - Boils down to message passing?

# Unit of Parallelism: Statements

**Basic Idea:** group together statements which can be executed in parallel

SEQ

s1 ;

s2 ;

PAR

s3 ;

s4 ;

PAR i=0 FOR n

a[i] = b[i] + c[i]

Used in Occam.

# Unit of Parallelism: Functions

- Functional programs — *no side effects* — implicitly parallel
- Functions without input-output dependency can be executed in parallel

$f( g(x, y), h(y, z) )$

can execute  $g$  and  $h$  in parallel

- **Major Issue:** Weigh communication overhead against gain for parallelism

# Unit of Parallelism: Logic clauses

In logical programming languages such as Prolog

$A :- B, C, D$

$A :- E, F$

- **OR Parallelism** prove (B, C, D) and (E, F) in parallel, until one succeeds or both fail
- **AND Parallelism** prove B, C, D in parallel, until all succeed or any one fails. Same for E, F.
- **Problem:** Shared logical variables — mutual exclusion

$A :- B(X), C(X)$

# Unit of Parallelism: Data Parallelism

- *Data vs. control* parallelism
- **Collection oriented languages:**
  - Basic data type is an *aggregate* of primitive data types. E.g. lists, arrays, sets etc.
  - Primitive functions operate on these aggregate data types
- Parallelism = acting on large collections of data with one operation

# Communication and Synchronization

- Closely related — must wait if sender not ready
- Introduces *non-determinism* — may get data from many senders, and don't know which one ready first
- classification by communication mechanism
  - **logically distributed** — distributed address space, communication by explicit message passing
  - **logically non-distributed** — shared global address space, communication through shared data

# Communication: Message Passing

- who sends it, what is sent, to whom, is receipt guaranteed....?
- Simplest — point to point reliable messages
- Always sent explicitly, but receipt may be *explicit* or *implicit* — explicit is more flexible
- **Naming:** direct or indirect? Symmetric or asymmetric?

# Synchronous or Asynchronous?

- **Synchronous:** sender blocked until receiver accepts message — also does synchronization
- **Asynchronous:** sender sends message and carries on, does not wait for receiver to be ready
  - how will these be *buffered*?
  - in what *order* will they be processed?

# Communication: Rendezvous

- Problem with point-to-point messages: only does one way communication — but often interactions are *two way*
- Two way interaction can be done with message passing — but need a single *higher level mechanism*
- **Rendezvous:** does both communication and synchronization
- Consists of :
  - entry declaration — like procedure declaration
  - entry call – like procedure call
  - accept statement — list of (maybe empty) statements to execute when entry called
- **Processes blocked only while in accept block**

# Remote Procedure Call

- Like procedure call with caller and callee process different
- caller blocked while waiting for results
- Semantics?
  - *Transparent*: try to be just like a regular procedure call — *lots of problems!* — pointers, crashes...
  - *Non-transparent*: restrict semantics

# Distributed Data Structures

- ... are data structures which can be manipulated by simultaneously by many processes.
  - First in **Linda**, using concept of **tuple space** — logically global shared space made of tuples  
Tuple space operations:
    - *out* add a tuple to the TS
    - *in* read and remove tuple from TS
    - *read* read a tuple from TS
- These operations are **atomic** — solves *mutual exclusion* problem
- Tuples do not have addresses — referenced by **contents**

# Mapping Computations to Processors

- similar to *load balancing* on operating systems — difference is that *fairness* is not an issue
- Parallel mapping = load balancing + local scheduling?
- Issue: is mapping under
  - user control?
  - done transparently by language runtime or operating system?  
Disadvantage: problem specific strategies ruled out

# Formalisms for Parallel Computing

- **Why?** Parallelism is unintuitive! Need formal, rigorous, mathematical framework for describing and understanding it.



$$\frac{\lambda - calculus}{sequential\ computation} = \frac{??}{parallel\ computation}$$

# Communicating Sequential Processes

- A number of **sequential** processes running in **parallel** that **communicate** with each other

- Processes have *process labels* to identify them

$P :: x := x + 1$

- Parallel composition:

$P \parallel Q \parallel R$

- **Communication:**

- $X!z$  — to process  $X$ , output value  $z$

- $X?z$  — from process  $X$ , input a value and assign it to  $y$ .

- *Non-determinism* controlled by using *guarded commands* :  $x > y \rightarrow a := 1$

- Denotational semantics given

# Calculus of Communicating Systems

- Mathematically model *communicating* systems (similar to CSP)
- Examples:
  - $a.A$  - do action  $a$ , then behave like  $A$ .
  - $A + B$  - non-deterministic choice between  $A$  and  $B$ .
- Focus on **equivalence** between systems
  - strong and weak bisimulation
  - observational equivalence
- Operations semantics : example (inference rule for choice)

$$\frac{A_j \rightarrow^a A'_j}{\sum_{i \in I} A_i \rightarrow^a A'_j}$$

# Chemical Abstract Machine

- **Problem:** formalisms are based on architectural concepts (like communication) — inflexible
- **Goal:** free parallelism from having to model **control** — have freely moving components which communicate when they come in “contact”
- **Basic ideas:** Mimic chemical solutions
  - Chemical solutions (multisets)
  - Molecules, ions (ready to react)
  - Reaction rules (specify how conversion to/from ions/molecules)
- This model is **intrinsically** parallel

# Semantics for Tuple Spaces

- Attempts made only recently [Ciancarini et al] — exposed many unresolved issues and ill-defined semantics which need to be formalised
- One approach: model using CCS  
Major problem: Tuple spaces are asynchronous, CCS is synchronous
- Model using other formalisms:
  - Petri Nets
  - Chemical abstract machine

# Relations between formalisms?

- How are formalisms related?
- Can some be shown equivalent to others?
- Can some be embedded in others?
- No formal attempts to relate various formalisms — only informal passing comments

# Ignoring Asynchronicity?

- Asynchronicity ignored by both Hoare and Milner because:
  - Synchronous systems are easy to realise physically
  - Buffering can be simulated by a process — programmer responsible
  - Complex to handle buffering issues
- Perhaps chemical abstract machine can be used — but does not model issues like order in which messages are processed

# Conclusion

- Need languages, and related formalisms, to
  - express
  - understand
  - reason about parallelism
- Vast range — units of parallelism, communication mechanisms, formalisms, semantics