# Parallel Algorithms

## Lecture 3: Parallel Languages

Ronald Veldema

## Languages

- Many languages now have some form of parallel construct
  - Integrated multithreading support
  - Integrated support for parallel arrays
  - Integrated support for task/data mapping
  - Parallel 'for' loops
  - Parallel processes (with some commication support)

## Process Algebra (1)

- Process algebra is a meta language for theorists to describe parallel systems
- Used to prove
  - Determinism
  - Deadlocks
  - Race conditions
  - etc

## Process Algebra (2)

- State transition: $A \xrightarrow{s} B$
- Sequential: AB (run A then B)
- Composition (A) (A is a subprocess)
- Choice A + B (run A or B, not both)
- Parallel A || B (run A and B in parallel)
- Communication y(a,b)
  - Communicate: b sends message to a, a does receive

## Process Algebra (3)

- Question ?
  - A || B == AB | BA ?
  - A(B+C) == AB + AC ?

## Process Algebra (4)

- Deadlock
  - State without outgoing transition
  - $A \rightarrow B$
- Livelock
  - Set of states without outgoing transition
    - $A \rightarrow B$
    - $B \rightarrow A$

# Spin (1)

- Prove that a number of parallel processes can't get into an illegal/unhandled state
    - *http://spinroot.com/spin/whatisspin.html*
  - Use to prove deadlock/livelock free-ness of parallel programs
  - Use to prove that the system is complete: in all states, all messages that can arrive there are processed
  - Use to prove that programmers assertions always hold

# Spin (2)

- Example:
  - Pathfinder mars mission failed because of concurrency control problem
    - Gist of problem:
      - Producer/consumer problem
        - » High priority process acquires/releases mutex in loop
      - Consumer: always runnable
      - Producer: runs only if nothing runnable
      - What happenes if high prio. process starts while low level priority process holds lock ?
        - » Low priority process stops running while holding the lock

# Spin (3)

```
mtype = { free, busy, idle, waiting, running };

show mtype h_state = idle;
show mtype l_state = idle;
show mtype mutex = free;

active proctype high()/* can run at any time */
{
end:  do
      :: h_state = waiting;
       atomic { mutex == free -> mutex = busy };
       h_state = running;
       /* critical section - consume data */
       atomic { h_state = idle; mutex = free }
      od
}
```

```
active proctype low()
      provided (h_state == idle) /* scheduling rule */
{
end:  do
      :: l_state = waiting;
       atomic { mutex == free -> mutex = busy};
       l_state = running;
       /* critical section - produce data */
       atomic { l_state = idle; mutex = free }
      od
}
```

***Spin reports bug: waiting on atomic entry while nothing else is runnable***

# Bug Finding (1)

- 3 * 2 * 3 states
  - Cpu states: (idle,waiting,running) cpus
  - Mutex is free or busy
- Spin tries all states:

| H_state, | L_state, | mutex | |
|---|---|---|---|
| Running, | Running, | free | ← Illegal (2 running at same time) |
| Running, | Running, | busy | ← Illegal (2 running at same time) |
| Running, | waiting, | free | ← Possible ! |
| Running, | waiting, | busy | |
| Running, | idle, | free | |
| Running, | idle, | busy | |
| …. | | | |
| …. | | | |

# Bug Finding (2)

- For each step in process 1, exhaustively try all possible interleavings of process 2

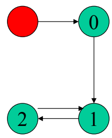
| Process1 | Process2 | OR Process2 | OR Process2 | etc… |
|---|---|---|---|---|
| A1 | | | | |
| | B1 | B1; B2 | B1; B2; B3; B4 | |
| A2 | | | | |
| | B2 | B3; B4 | | |
| A3 | | | | |
| | B3 | | | |
| A4 | | | | |
| | B4 | | | |

# Parallel State Machines (1)

- Normal state machine
  - Number of states with a special 'start state'
  - Outgoing transitions based on some triggering condition
- Parallel state machine
  - Each task has his own state state machine
  - Tasks can send messages to state machines
- Parallel state machines can be rewritten to normal state machines !

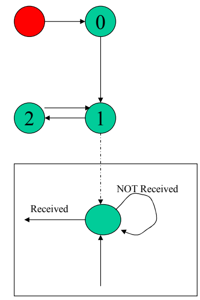
# Parallel State Machines (2)

```
I = 0; // state 0
while (true)
{
        if (I == 0)
                // state 1 -> state 2
                I = 1;
        else
                // state 2 -> state 1
                I = 0;
}
```




# Parallel State Machines (3)

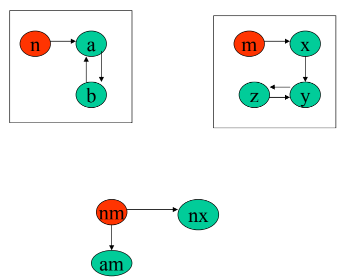
```
I = 0; // state 0
while (true)
{
        if (I == 0)
                // state 1 -> state 2
                I = 1;
                send_msg(…);
        else
                // state 2 -> state 1
                I = 0;
}
```



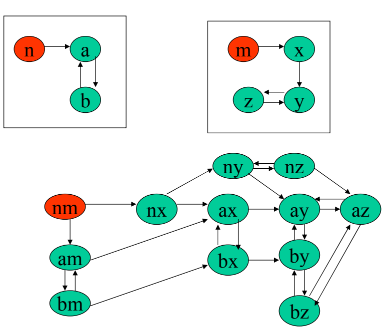

# Parallel State Machines (4)

- Map parallel State machine to normal state machine:
  - 2 machines SA and SB with states SA1 – SA4, SB1 --- SB 3
  - Create new states (SAx, SBy) with x=1…4, y=1…3.
  - If SAx can move to SAp then draw edge from (SAx,*) to (SAp,*) in new state machine
  - If SBx can move to SBp then draw edge from (*, SBx) to (*,SBp) in new state machine

# Parallel State Machines (5)




# Parallel State Machines (6)




# Linda (1)

- Global tuple space that each cpu has access to:
  - ("jim", 34, 3), ("michael", 44, 5)
- Out("jim", 34, 3)
  - Puts this tuple in the tuple space
- In(jim, 34, ?salary)
  - Gets tuple matching (jim, 34) and binds salary to 3.
  - Tuple is removed from tuple space
- Read = same as 'in' except that tuple not removed

## Linda (2)

- Note:
  - Network independence
  - Location independence
  - Automatic synchronization
    - When multiple CPUs try an 'in' only one will succeed
  - Automatic partitioning possible based on for example:
    - Hashing fields of tuple to owning cpu
      - Advanced compiler work…
      - Dynamic load balancing of tuples over cpus
    - Search: not whole data base but only partition

## Parallel Prolog (1)

```
Color(sky, blue).          % database
Color(sea, blue).
Color(grass, green).

State(sky, gas).
State(sea, liquid).
State(grass, solid).

Thing(thing, color, state) :-      % rules
        Color(thing, color),
        State(thing, state).

?- thing(X,blue,liquid)        % query
```

## Parallel Prolog (2)

1) Maintain a stack of predicates still to be matched
2) Push to-be-proven goals on the stack
3) Pop a goal, try and match with known truths
   - If valid, unify and push goal as known truth
4) Continue until stack is empty

## Parallel Prolog (3)

- Example:
  - Push "thing(X, blue, liquid)"
    - Head unification: Color=blue, State=liquid
    - Push "Color(X, blue)" and "State(X, liquid)"
      - For each X where color = blue, test State(X, liquid)
        - » Three alternatives for "color/2" (sea and sky)
      - Bind X to sky, is "state(sky, liquid)" a fact ?
        - » Backtrack and try to bind X with "sea"
      - State(sea, liquid) is a fact and thing(sea, blue, liquid) thus now also a fact

## Parallel Prolog (4)

- Sources of parallelism:
  - Match(X,Y,Z) :- testme1(X,G), testme2(Y,Z)
    - And parallelism
      - Prove 'testme1' and 'testme2' in parallel
    - Or parallelism
      - Prove multiple alternatives in parallel:
        - » Example: testme1(X,blue) and testme1(X,red)

## Set/Array/Script Mini-Languages

- Set a = sort( set b – set c )
  - Sort common set between 'b' and 'c' and place in 'a'
- SPMD programming styles
- Explicit parallelism

# Fortran (HPF)

- SPMD programming
- Explicit parallelism
  - A = B * C ;; A, B, C are arrays
- Implicit parallelism
  - do I = 1, N
    - A(I) = B(I) * C(I)
  - Enddo
  - Note: compiler must do all the work !

# Fortran (HPF) (2)

- Implicit parallelism
  - DO I = 1, N
    - A(I) = A(I) * A(I-1)
  - Enddo
  - Note: compiler must do all the work !

# Fortran (HPF) (3)

- Implicit parallelism
  - !HPF$ INDEPENDENT
  - DO I = 1, N
    - A(I) = A(I) * A(I-1)
  - Enddo
  - Note: programmer must do all the work !

# Fortran (HPF) (4)

- !HPF$ PROCESSORS pr(16)
- REAL X (1024)
- !HPF$ DISTRIBUTE X(block) INTO pr

- Each processor gets 1024/16 elements of X in roundrobin fashion

# Fortran (HPF) (5)

- !HPF$ PROCESSORS pr(16)
- !HPF$ DISTRIBUTE X(CYCLIC) ONTO pr
- Real X(1024)

- Each processor gets every Nth element

# Fortran (HPF) (6)

- !HPF$ PROCESSORS pr(16)
- !HPF$ DISTRIBUTE X(block,cyclic)
- REAL X(1024,1024)

- Each processor gets 1024/16 rows and of each row every Nth element

# Fortran (HPF) (7)

- !HPF$ ALIGN source_array WITH target_array
- Real source_array(1024), target_array(1024)

- Says that each element of source_array should be on the same cpu as target_array

# Fortran (HPF) (8)

- !HPF\$ ALIGN source_array(I) WITH target_array(I * 2)
- !HPF\$ ALIGN source_array(I,J) WITH target_array(J,I)
- !HPF\$ ALIGN source_array(I,*) WITH target_array(J)
- !HPF\$ ALIGN source_array(I) WITH target_array(J,*)

# Fortran (HPF) (9)

- Question
  - What if distribution/align is perfect for one function but not for another ?
    - Remap to different distribution "on the fly" ?
    - Ignore inefficiency ?

# CC++ (1)

- "Concurrent C++"
- C++ with extra syntax
- par { x++; y++ }
- parfor (int I=0;I<10;I++) <statement>

# CC++ (2)

- Global class A { }
- Float *global ptr
- Atomic void func() {}
- CCvoid &operator << (CCvoid &, const TYPE &obj_in)
- CCvoid &operator >> (CCvoid &, TYPE &obj_in)
- proc_t location(node_t(``machine_nameX"));
- G = new (location) Type();

# Java

- Threads
  - new Thread().start();
  - synchronized (ptr) { statements }
    - Translates to
      - "lock(ptr) statements unlock(ptr)"
- Remote Method Invocation (RMI)
  - Remote Procedure Call

# JavaParty (1)

- Extension to the Java language
- Each class can have a “remote” modifier
  - Instances thereof are remotely allocated
  - Methods thereof are remotely invoked

# JavaParty (2)

- Parameter to members of remote classes are passed by copy

```
Remote class A {
  void foo(Data d) {
    PrintReference(d);
  }
}
```

```
A a = new A();
Data d = new Data();
a.foo(d);
a.foo(d);
```

# JavaParty (3)

```
remote class A {
    void foo() {
    }
}

class B {
    A a;

    void foo() {
      DistributedRuntime.setTarget(cpu_num);
      a = new A();
    }
}
```

# Synchronizing Resources (SR) (1)

```
resource foo()
        write("Start A")
        process A
                fa k := 1 to 2 -> write("In A");
                af
        end A

        write("Start B")

        process B
                fa k := 1 to 2 -> write("In B");
                af
        end B
        write("All done")
end foo
```

Start A
Start B
All done
In A
In B
In A
In B

# Synchronizing Resources (SR) (2)

```
resource Cotest()

        procedure me(X: string[10])
                write(X)
        end

        co me("A") // me("B") oc

        write("At the end")
end
```

A
B
At the end

Or

B
A
At the end

# Orca (1)

- Object like model with RPCs
  - Modula like syntax
  - Processes
    - Can dynamically ‘fork’ more processes on potentially different CPUs
  - Objects
    - Process can share objects with forked children
  - Operations
    - indivisible

# Orca (2)

- Arc Consistency Problem
  - N input values
  - Binary constraints between some pairs of values
  - Repeatedly eliminate values from domains until all constrains satisfied

Example: constraint type = '>', constraint vector = 1,0,1,1,0,1
Values = 10,30,103,30,40,20
Values = 30, 103, 40, 20
Values = 103,40,20

# Orca (3): Arc Consistency Problem

```
OBJECT Domain;

        TYPE ValueSet = SET OF Integer;
        Domains: ARRAY[1..N] OF ValueSet;
        # compiler sees this is a write operation
        OPERATION eliminate(v:integer; S: ValueSet);
        BEGIN
                Domains[v] := Domains[v] – S;
        END;
        # compiler sees this is a read operation
        OPERATION values(v:integer): ValueSet; BEGIN
                RETURN Domains[v];
        END;
END;
```

# Orca (4): guards

- Guards are boolean operations
  - If any guard in an operation delivers true then operation may continue
  - If no guard is true then operation blocks
  - As soon as ANY guard becomes true is operation atomically executed.

# Orca (5)

```
OBJECT WorkAdmin;
        TYPE VariableSet = SET OF Integer;
        recheck: VariableSet;
        ActiveProcesses: Integer;
OPERATION Ready(); BEGIN
        ActiveProcesses -:= 1;
END;

OPERATION WaitForWork(S: VariableSet): boolean;
BEGIN
        # wait until: intersection of S and recheck is non empty
        # or intersection is empty and all processes are idle
        GUARD SIZE(S * recheck) > 0 DO
                ActiveProcesses +:=1;
                RETURN true;
        OD;
        GUARD ActiveProcesses = 0 AND SIZE(S*recheck) = 0 DO
                RETURN false;
        OD
END;
END;
```

# Orca (6)

- Orca replicates objects everywhere
  - Send RPC to all machines
- Or put object on one machine and migrate object to machine that uses it most

# Parallel Lisp

- Lisp: functional language
  - '(' expression '(' params ')' ')'
  - ( <pre-expression> ( EXEC <expression>) <sibling-expressions>) <post-expression>
    - Expression evaluates in parallel with sibling expressions
  - Important to lisp: functional transparancy
    - Means that functions have no side-effects

# Parallel Lisp: Matrix Multiply

```
(defun matmul (a b c n m k)
 (declare (type (simple-array (unsigned-byte 32) (*)) a b c)
          (fixnum n m k))
(let ((sum 0)
      (i1 (- m))
      (k2 0))
  (declare (type (unsigned-byte 32) sum) (type fixnum i1 k2))
  (dotimes (i n c)
      (declare (fixnum i))
      (setf   i1 (+ i1 m)) ;; i1=i*m
          (dotimes (j k)
             (declare (fixnum j))
             (setf sum 0)
             (setf k2 (- k))
             (dotimes (l m)
                (declare (fixnum l))
                (setf k2 (+ k2 k)) ;; k2= l*k
                (setf sum (the (unsigned-byte 32) (+ (the (unsigned-byte 32) sum)
                                                    (the (unsigned-byte 32) (* (aref a (+ i1
l))
                                                                               (aref b (+ k2
j))))))))
         (setf (aref c (+ i1 j)) sum)))))
```