# Concurrent Revisions

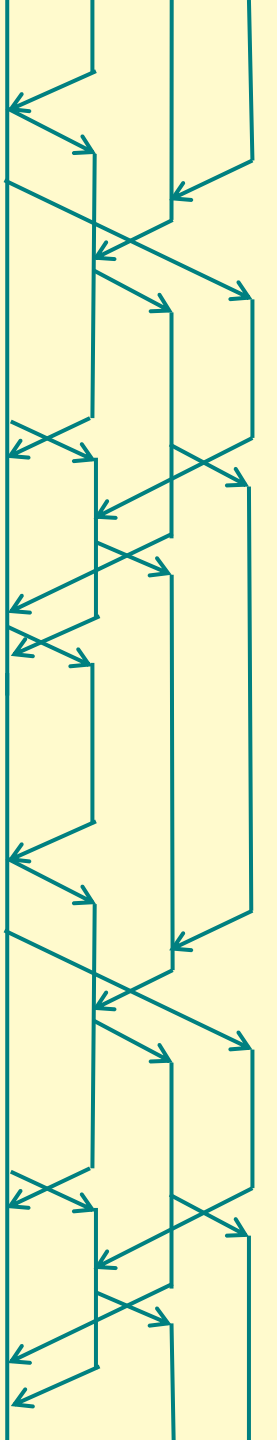
## Lecture 1: From Tasks to Revisions

Sebastian Burckhardt

Microsoft Research

Joint work with Alexandro Baldassin, Daan Leijen

(see also our papers in OOPSLA 2010, ESOP 2011)



# Outline

- Parallel Programming With Tasks
  - Task Dags and Scheduling
  - Conflict Elimination
  - Motivation: Game Example
- Concurrent Revisions
  - Fork/Join and Revision Diagrams
- Semantics
  - Syntax and Semantics
  - Determinacy

# Task Programming Model

- Program expresses task dag (directed acyclic graph)
- Runtime dynamically schedules task dag onto multiprocessor
- Commonly used paradigm
  (CILK, Intel TBB, java.util.concurrent, Microsoft TPL, …)
- Abstracts machine details
  (sync primitives, number of processors)
- Often dynamic
  (graph generation and scheduling are interwoven)

# Expressing parallelism

- Example:

```
for (i = 0; i < n; i++)
{
   work1(i);
   work2(i);
}
```

Can parallelize loop iterations and/or loop body.

```
parallel for
 (i = 0; i < n; i++)
{
   work1(i);
   work2(i);
}
```

```
for
 (i = 0; i < n; i++)
{
   parallel { work1(i), work2(i) }
}
```

What does it mean, exactly?

# Comparison with TPL syntax

```
Parallel.For(0, n, i =>
{
    work1(i);
    work2(i);
})
```

```
parallel for
 (i = 0; i < n; i++)
{
    work1(i);
    work2(i);
}
```

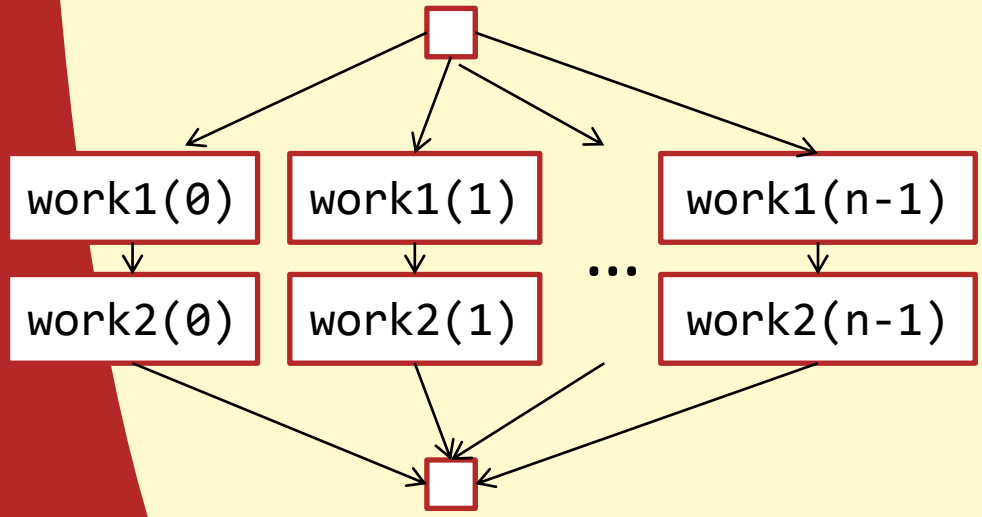
```
for
 (i = 0; i < n; i++)
{
  Parallel.Invoke(
      () => { work1(i); },
      () => { work2(i); }
  });
}
```

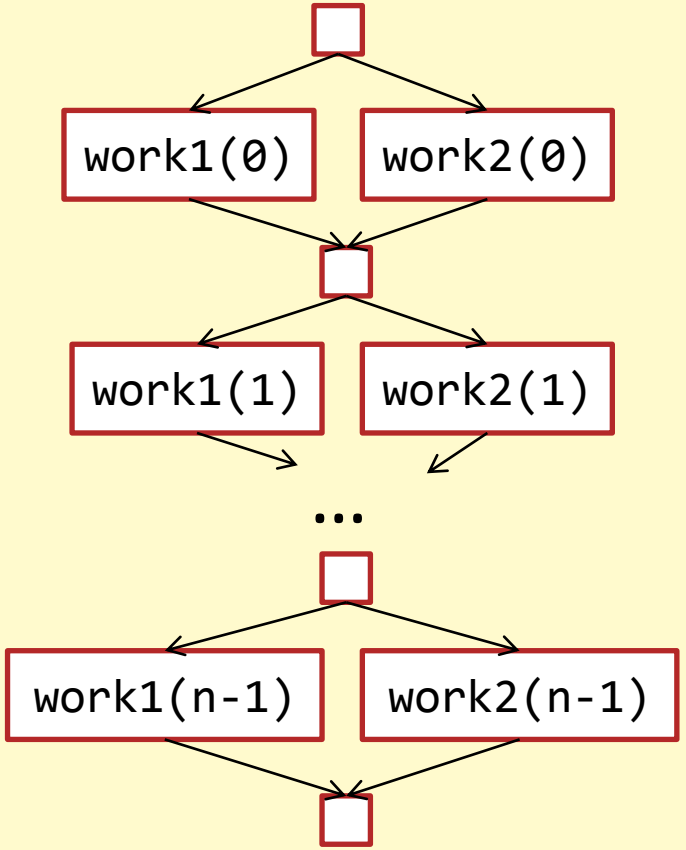
```
for
 (i = 0; i < n; i++)
{
    parallel { work1(i), work2(i) }
}
```

# Task graphs

```
parallel for
 (i = 0; i < n; i++)
{
   work1(i);
   work2(i);
}
```

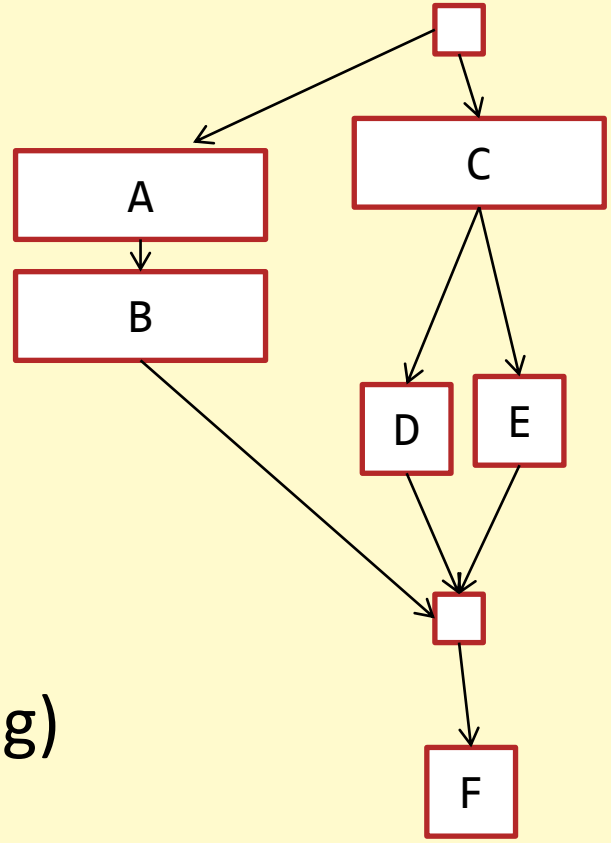
```
for (i = 0; i < n; i++)
{
   parallel { work1(i), work2(i) }
}
```



# What do we mean by task graphs?

- Vertices
  - Represent tasks or synchronization
- Directed Edges
  - Represent scheduling constraints
- No cycles
  - Thus, directed acyclic graph (dag)




*Task graphs (a.k.a **task dags**) are an excellent visualization for understanding parallelism.*

# The meaning of edges

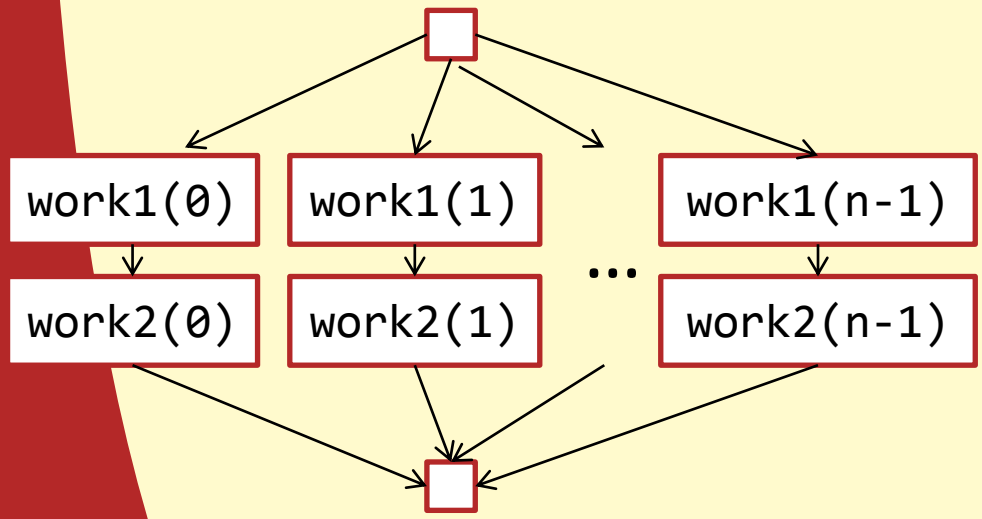
- An edge v -> w means
  “task v must complete before task w starts”

- Is a transitive property!
  (a before b) and (b before c) implies
  (a before c)

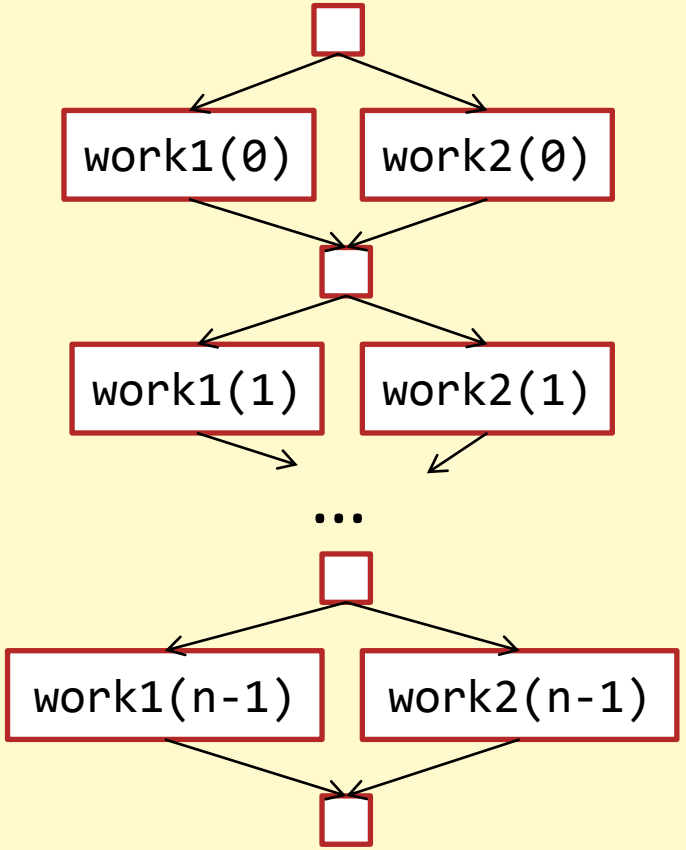
- We omit edges that are implied by transitivity
  (take the “transitive reduction”)

# Examples

```
parallel for
 (i = 0; i < n; i++)
{
   work1(i);
   work2(i);
}
```

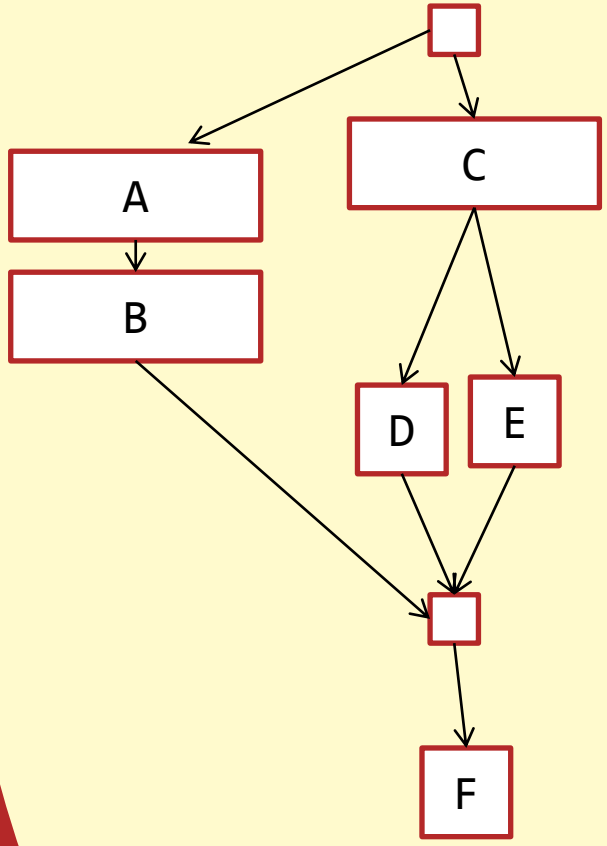



```
for
 (i = 0; i < n; i++)
{
   parallel { work1(i), work2(i) }
}
```



# Executing Task Dags

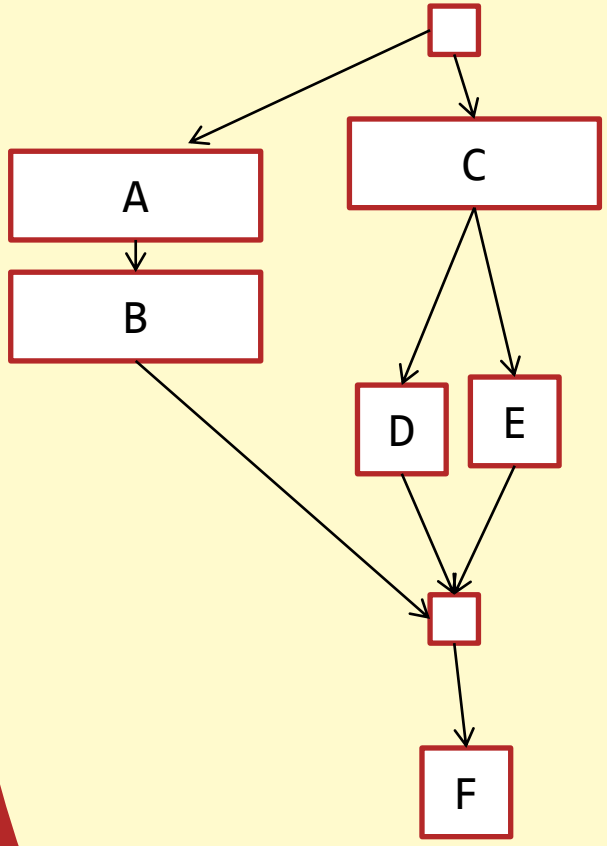
- On a single processor: serial schedules




**Many schedules possible:**
ABCDEF, ABCEDF, ACBDEF, ACBEDF, ACDBEF, ACEBDF, ACDEBF, ACDEBF, ACEDBF, …, CADEB, CAEDB, … CDEABF, CEDABF

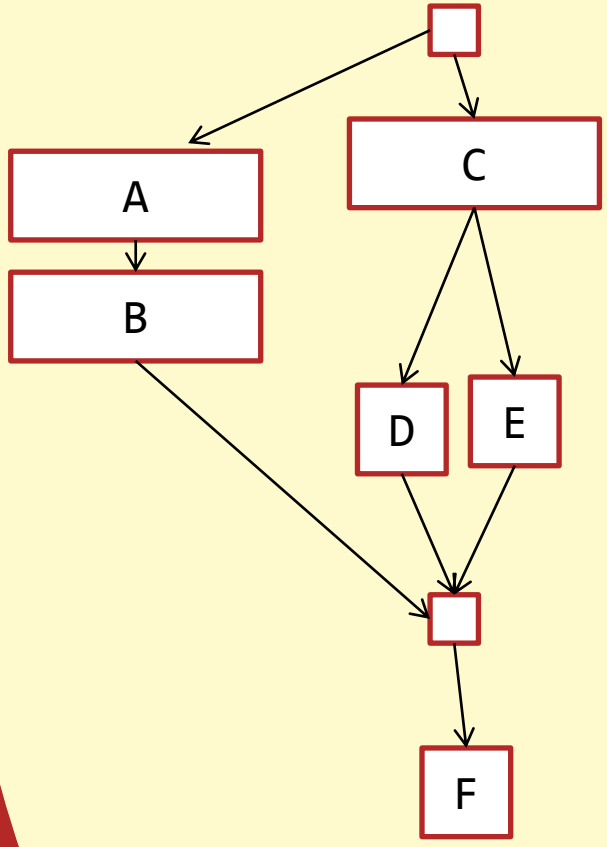
- All schedules have same execution time:
- $\mathbf{T_1}(G) = A + B + C + D + E + F$

# Parallel Schedules




- Suppose we have an idealized parallel processor (infinite number of processors, no synchronization overhead)
- How long does it take to execute this task dag?

# Parallel Schedules




- Each task can start as soon as all previous tasks complete
- Execution time = critical path = time of longest path through the graph
- $\mathbf{T}_{\infty}(G) = \max \{ A + B, C + \max \{D, E\} \} + F$

# Work and Span

- We call the serial execution time $T_1(G)$ the **work** of G.

  $T_1(G) = \sum \{ T(v) \mid v \text{ in } G \}$

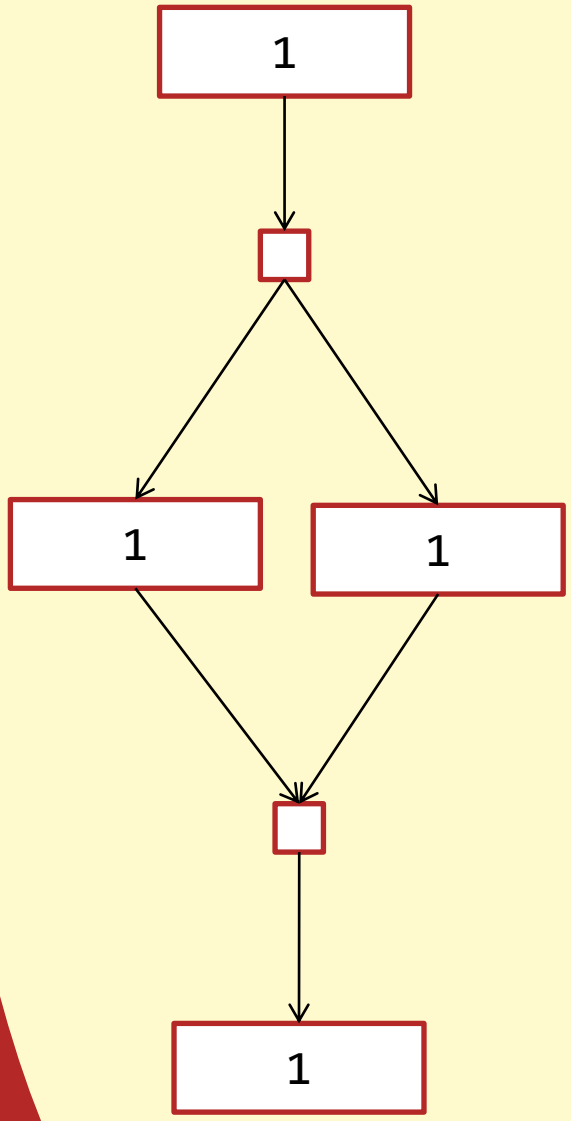
- We call the ideal parallel execution time $T_\infty(G)$ the **span** of G:

  $T_\infty(G) = \max \{ T_1(p) \mid p \text{ maximal path of } G \}$

# Speedup

- Divide work by span:
- What does the ratio $T_\infty(G) / T_1(G)$ mean?
- It's how much we can improve the execution time by parallel execution! We call this par(G), **parallelism** of G.
- It's an upper bound on the possible speedup.

# Example



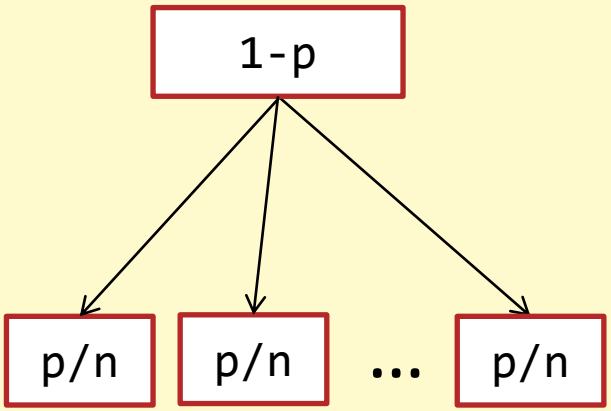

- Work $T_1(G)$ is 4.
- Span $T_\infty(G)$ is 3.
- Parallelism par(G) is 4/3

- Speed improvement from parallelism is never more than 33%

- Clear from picture: Using more than 2 processors has no benefits.

# Special case: Amdahl's law

- Don't remember formula? No problem, can derive it instantly.
  - p : parallel fraction of program
  - 1-p : sequential fraction of program
  - n : number of processors




- Work is 1
- Span is (1-p) + p/n
- Parallelism is work/span

$$\frac{1}{(1-p) + p/n}$$

# Execution time on n processors

- Define $T_n(G)$ to be the best possible execution time on n processors
  (that is, the smallest time we can achieve for any schedule of G on n processors)

- Clearly, we can give bounds

$$T_\infty(G) \leq T_n(G) \leq T_1(G)$$

# FYI: How hard to find best schedule?

## Finding the best schedule is hard

- Computing $T_n(G)$ is NP-complete for n>2
  [Gary, Johnson 1979]

## Finding a reasonably good schedule is easy

- [Brent's Theorem] for any dag G,

$$T_n(G) \leq T_1(G)/n + T_\infty(G)$$

  [Blumofe/Leiserson 1999]:
  Any greedy schedule achieves this bound.

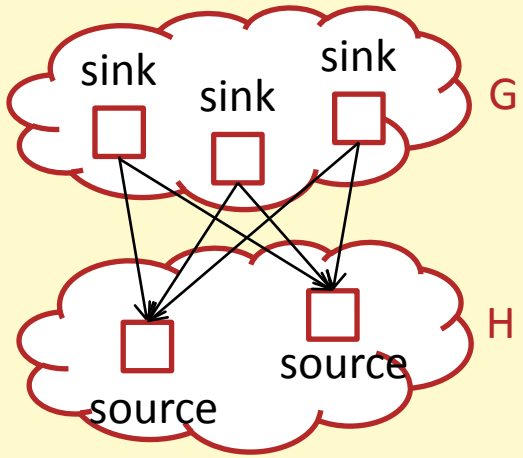
# Series-Parallel Task Dags

- We can build many task dags using just parallel and serial composition.
- These dags are called ‘series-parallel’
- If our programs express parallelism using only the following constructs, the dags are always series-parallel:
  - parallel { }
  - parallel for

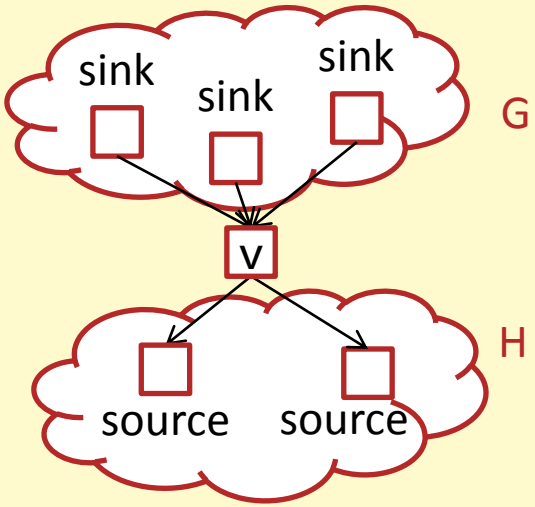
# Serial Composition of Task Dags

Given two task dags G, H, build serial composition (G ; H) by

- connecting all sinks of G to all sources



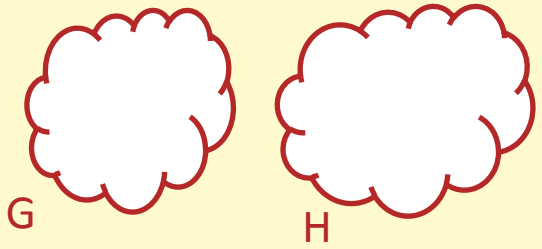

- To use fewer edges, we can construct
  (G; v; H)
  where v is an extra synchronization vertex
  (with T(v) = 0)



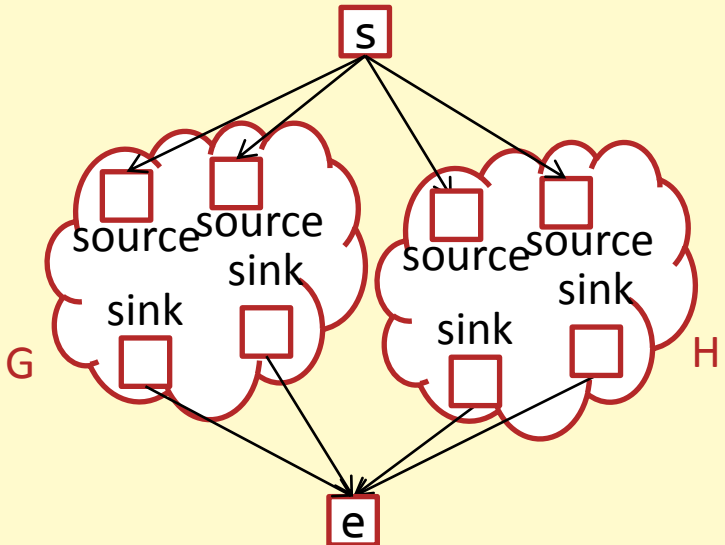
# Parallel Composition of Task Dags

Given two task dags G, H, build parallel composition (G ll H) by

- Building disjoint union of G, H

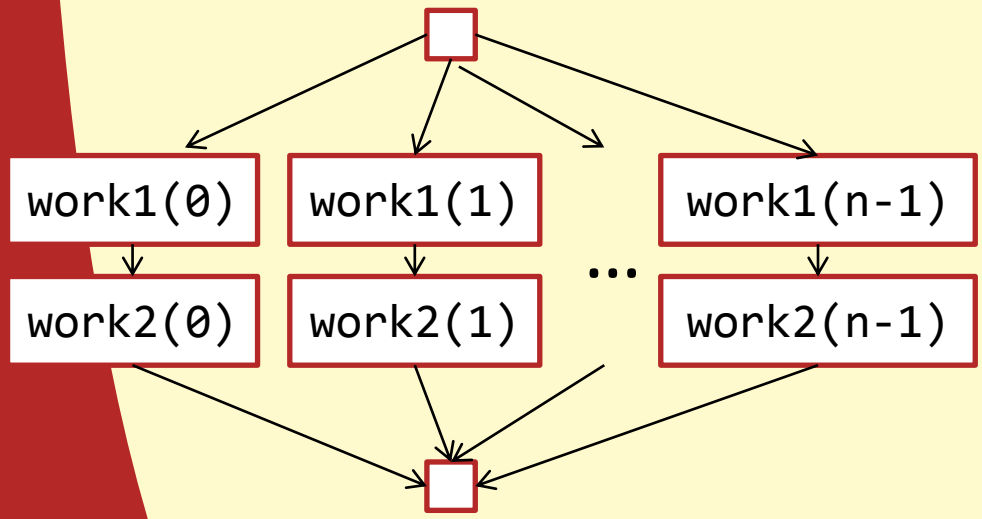



- Or, to keep dag connected, we can construct
  (s ; ( G ll H ) ; e)
  where s,e are extra synchronization vertices
  (with T(s) = T(e) = 0)

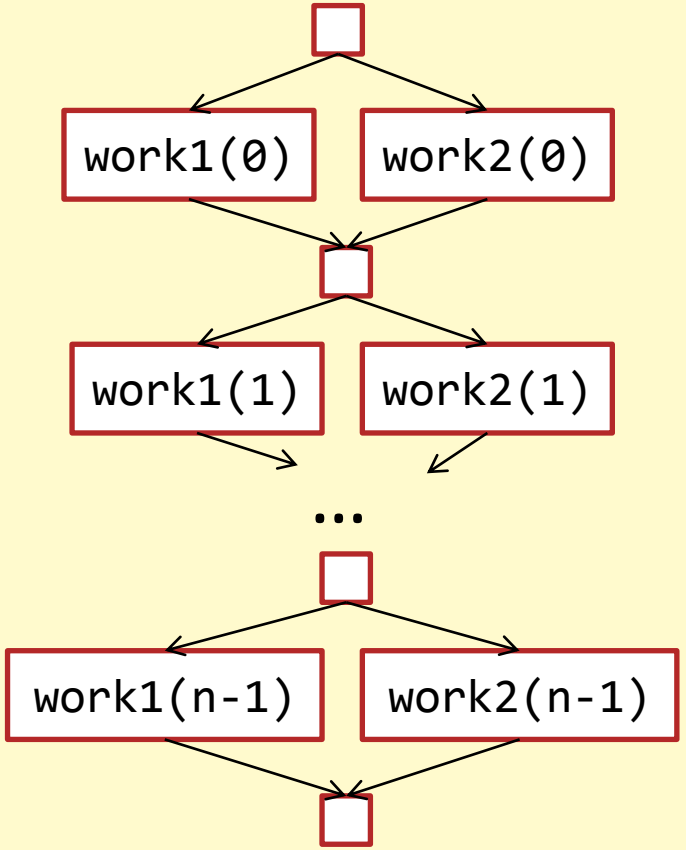


# Examples

```
parallel for
 (i = 0; i < n; i++)
{
   work1(i);
   work2(i);
}
```




```
for
 (i = 0; i < n; i++)
{
   parallel { work1(i), work2(i) }
}
```



# More ways to express parallelism

- Can use fork, join primitives to explicitly create parallel tasks

```
var t;

t = fork {
   A();
}

B();

join t;

C();
```

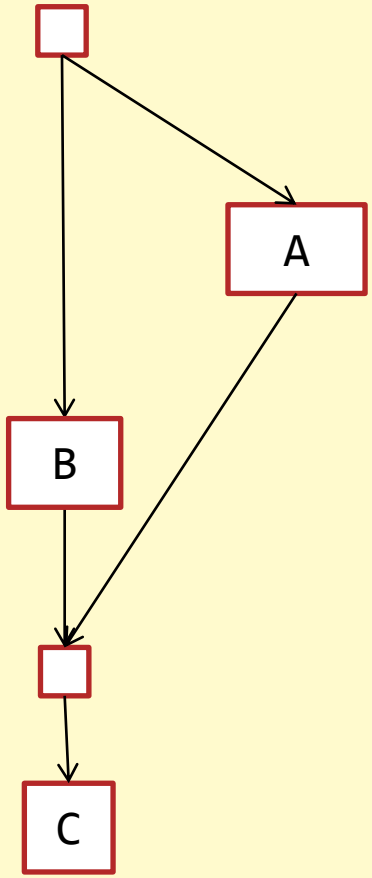
# More ways to express parallelism

- Can use fork, join primitives to explicitly create parallel tasks

```
var t;

t = fork {
    A();
}

B();

join t;

C();
```

**Starts a task that executes the block in curly braces; stores a handle to it in t**

**Waits for t to finish before continuing**



# Comparison to TPL syntax

```
Task t;

t = fork {
    A();
}

B();

join t;
```

```
Task t;

t = Task.Factory.StartNew(() =>
    {
        A();
    });

B();

t.Wait();
```

Our join is more restricted than wait…

Must be called exactly once.

# Can express parallel using fork,join

```
function
myparallelinvoke(f, g)
{
    var t = fork {
        f();
    }
    g();
    join t;
}
```

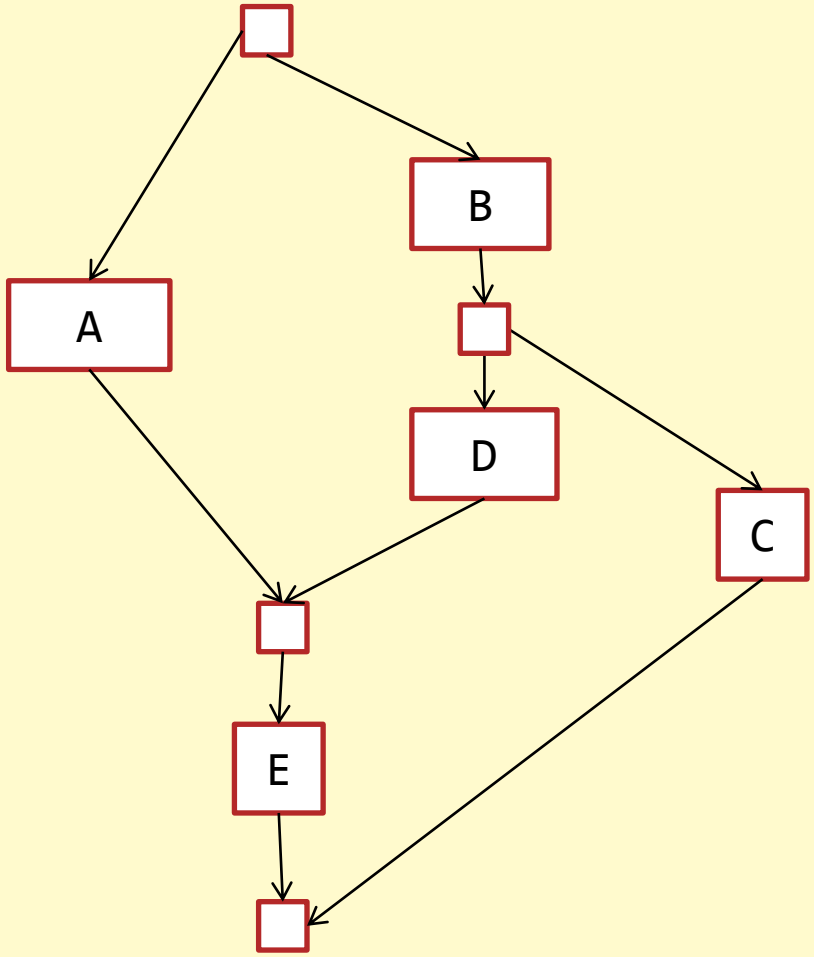
```
function myparallelfor(a, b, f)
{
    var t = new Task[n];

    for (int i = a; i < b; i++)
        t[i] = fork { f(i); }

    for (int i = a; i < b; i++)
        join t[i];
}
```

- Can we do the opposite, i.e. express fork,join using parallel?

# Not all dags are series-parallel

- For example, we can use asymmetric fork and join to create task graphs like this

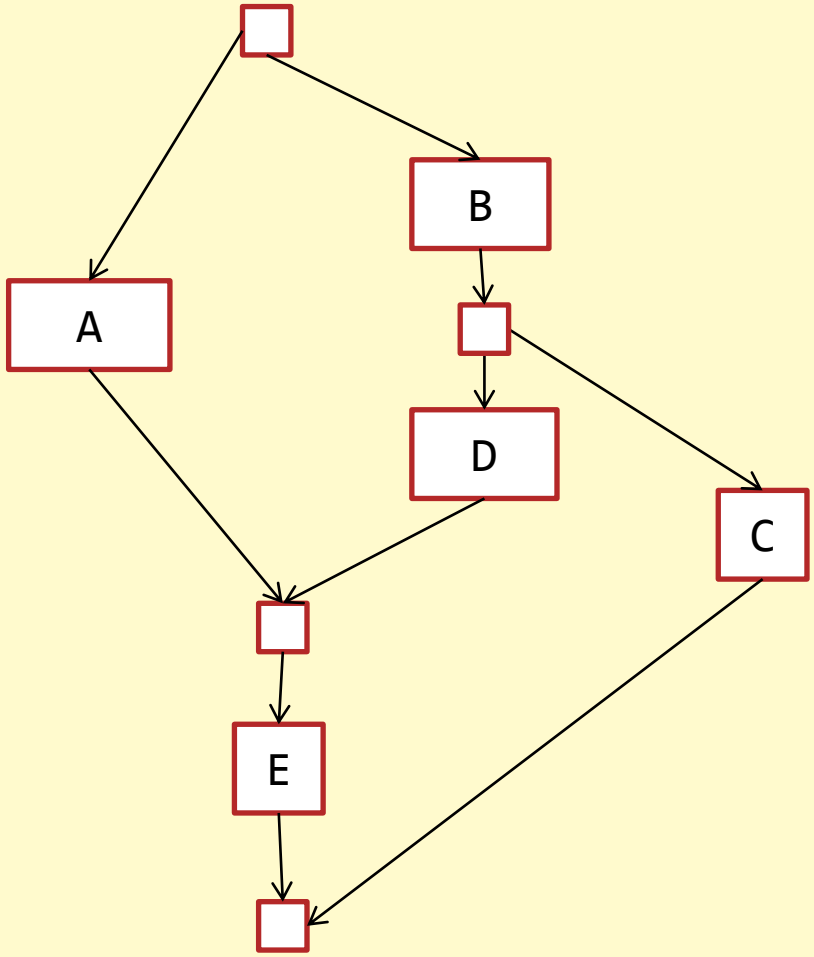
```
var t;

parallel
{
   A(),
   {
     B();
     t = fork {
      C();
     }
     D();
   }
}
E();
join t;
```



# Not all dags are series-parallel

- For example, we can use asymmetric fork and join to create task graphs like this

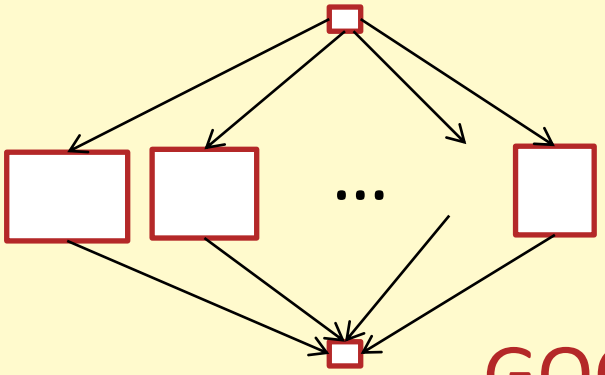
```
var t;

parallel
{
   A(),
   {
     B();
     t = fork {
      C();
     }
     D();
   }
}
E();
join t;
```



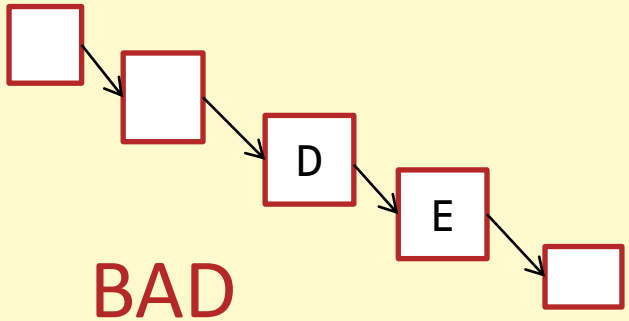
So, what does all this mean for how we should write parallel programs?

# Thus, for fastest execution…

- Make everything as parallel as possible.




GOOD




BAD

- Unfortunately, very few programs are fully parallel (“embarassingly parallel” or “pleasantly parallel”)
- Because tasks usually exhibit dependencies and conflicts!

# Parallelism Blockers

- Conflicts
  - Two tasks A, B access the same data, and one (or both) modify it
  - (If both tasks are only reading, we don't call it a conflict)
  - Two tasks that don't have conflicts can run in parallel
- Dependence (a type of conflict)
  - A conflict as above, where A writes some result and B reads that result

# Terminology: Conflict vs. Data Race

- All data races are conflicts.
- Not all conflicts are data races.

```
parallel
{
  x = 1,
  x = 2
}
```

- Has conflict.
- Has data race.

```
parallel
{
  lock (l) { x = 1 },
  lock (l) { x = 2 }
}
```

- Has conflict.
- does NOT have data race.

# Hazards

- Sequential execution is not equivalent to parallel execution if there are *hazards*.

```
{
    x = work1();
    work2(x);
}
```

**RAW hazard**
(read after write)

a.k.a.
data dependency

```
{
    work1(x);
    x = work2();
}
```

**WAR hazard**
(write after read)

a.k.a.
Anti-dependency

```
{
    x = work1();
    x = work2();
}
```

**WAW hazard**
(write after write)

# Hazards

- Hazards in sequential composition correspond to *conflicts* in parallel composition.
- Conflict = concurrent tasks access shared data, and at least one of them modifies it.

```
{
    x = work1();
    work2(x);
}
```

```
parallel
{ x = work1(),
    y = work2(x) }
```

Read-Write conflict

```
{
    work1(x);
    x = work2();
}
```

```
parallel
{ work1(x),
    x = work2() }
```

Read-Write conflict

```
{
    x = work1();
    x = work2();
}
```

```
parallel
{ x = work1(),
    x = work2() }
```

Write-write conflict

# Why are conflicts bad?

- Nondeterminism
  - The order of the accesses affects the outcome
- Data Consistency
  - Many typical data representations (e.g. linked list) are not safe unless carefully crafted (see M.H.) or protected by locks
- **Performance**
  - Cache coherence: Conflicts cause cache misses (shared memory is not actually shared)

# Why are conflicts bad?

- Nondeterminism
- Data Consistency
- Performance

Essential insight:

**Eliminate frequent conflicts.**
**Make rare conflicts safe.**

# Some conflicts are superfluous.

- WAR and WAW can usually be eliminated.

```
{
    work1(x);
    x = work2();
}
```

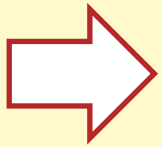
```
{
    x = work1();
    x = work2();
}
```

- Because they are not true dependencies but a matter of reading/writing correct versions
- work2 can proceed without work1's result if
  - It does not interfere with work1's input (RAW)
  - It ensures to overwrite work1's output (WAW)

# Some conflicts are superfluous.

- For example, we can eliminate the following WAR / WAW hazards (privatization).

```
var x;

for (i=0; i<n; i++)
{
    x = work1();
    work2(x);
}
```



```
var x;

var xx[n];
parallel for (i=0; i<n; i++)
{
   var xx[i] = work1();
   work2(xx[i]);
}
x = xx[n-1];
```

- Trick: copying / renaming
  - Register renaming extremely effective in hardware: Tomasulo's algorithm

# The Art of Task-Parallel Programming

- Identify tasks to run in parallel
  - Make tasks large enough
  - Make tasks small enough
  - Make tasks parallel enough
- Carefully consider conflicts
  - Eliminate or reduce conflicts
    - Tricks of the trade
  - Make remaining conflicts safe
    - Concurrency control

# The Art of Task-Parallel Programming

- Identify tasks to run in parallel
  - Make tasks large enough
  - Make tasks small enough
  - Make tasks parallel enough
- Carefully consider conflicts
  - Eliminate or reduce conflicts
    - Tricks of the trade
  - Make remaining conflicts safe
    - Concurrency control

**QUANTITATIVE CRITERIA**
depend on characteristics of workload and machine

**QUALITATIVE CRITERIA**
Do not depend on machine or workload

## Make tasks small enough

- total execution time >= time for longest task

## Make tasks parallel enough

- total execution time >= time along critical path

  (= path from source to sink whose sum of task times is maximal)

# Make tasks large enough

- Danger:
  - Processors are very fast at sequential execution.
  - Processors are slow at scheduling/starting/ending tasks.
  - Unless a task contains a significant amount of work, it is not worth to try parallel execution!
- Usually not a problem when using parallel for and lots of iterations
  - Smart implementation of parallel for chunks iterations automatically and dynamically

# Example: Histogram Computation

```
a = new int[8];
for (i = 0; i < n; i++)
{
   a[f(i)]++;
}
```

- Suppose f is a function with range 0..7
- We want to count how many times each number in 0..7 is produced when calling f on 0…n-1
- We assume n large enough to make parallelism worthwhile (e.g. n = 1000000)

# Simple but incorrect solution

```
a = new int[8];
parallel for (i = 0; i < n; i++)
{
   a[f(i)]++;
}
```

- Does not count correctly because ++ is not atomic

# Correct but slow solution

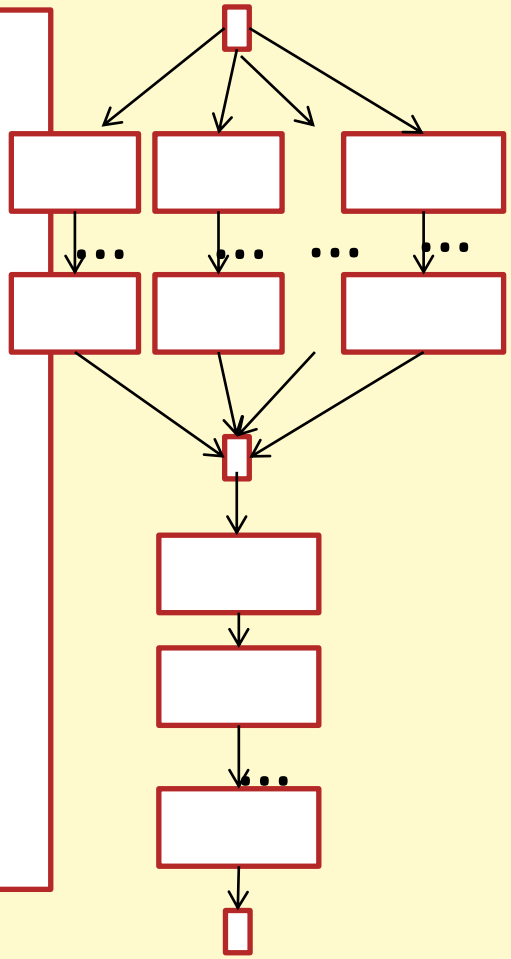
```
a = new int[8];
l = new lock[8];
parallel for
   (i = 0; i < n; i++)
{
   var r = f(i);
   lock( l[r] )
      { a[r]++; }
}
```

```
a = new int[8];
parallel for
   (i = 0; i < n; i++)
{
   var r = f(i);
   atomic-increment(&(a[r]));
}
```

- Atomic increment is typically a bit faster than lock/unlock.
- Both solutions are correct, but overall performance is quite bad (worse than sequential)

# A better solution

```
a = new int[N][8]; // N: number of processors

// partition by processor
parallel for (p = 0; p < N; p++)
   for (i = 0; i < n/N; i++)
   {
       var r = f(p*N + i%N);
       a[p][r]++;
   }
// combine histograms
for (p = 1; p < N; p++)
   for (j = 0; j < 8; j++)
      a[0][j] += a[p][j]

// final result at a[0]
```



- We have eliminated the conflicts.
- Worse in theory (task dag), but much better in practice

# The toolbox

**Eliminate frequent conflicts**

- Architectural Patterns
  - Producer-Consumer
  - Pipeline
  - Worklist
  - ….
- Replication Patterns
  - Immutability
  - Double Buffering
  - Concurrent Revisions

**Make rare conflicts safe**

- pessimistic concurrency control

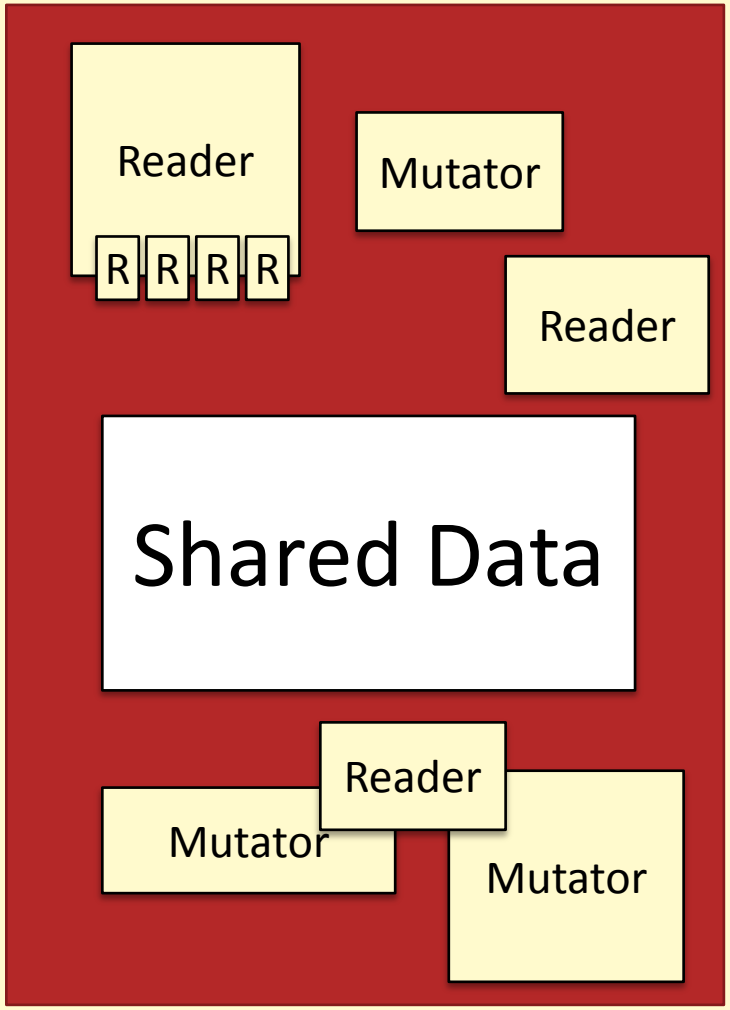
  coarse- or fine-grained locking
- optimistic concurrency control

  speculate on absence of conflicts, roll back if speculation fails
  - Transactional memory

# PART II:
# INTRODUCTION TO CONCURRENT REVISIONS

## Targeted Application Scenario

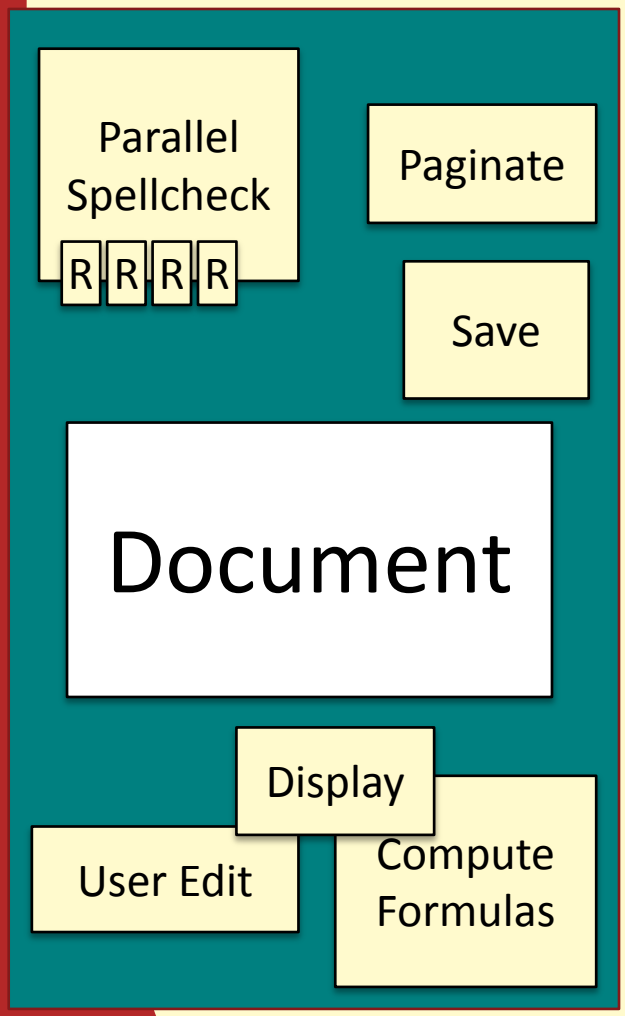
# Shared Data and Parallel Tasks



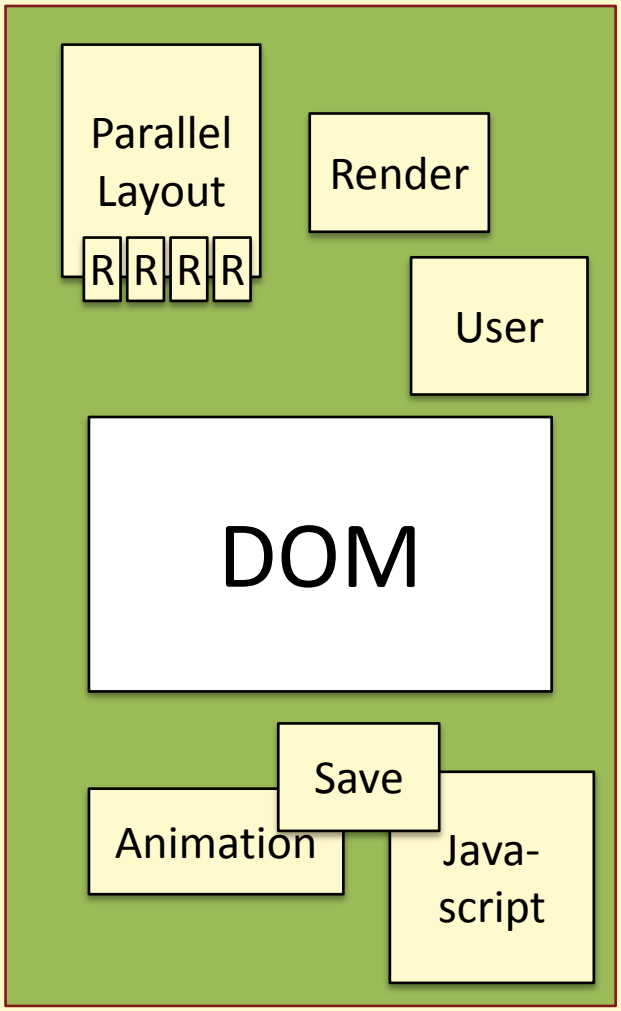

- Shared state is read/mutated by many tasks.
- Both inter- and intra-task parallelism.
- Challenge: Tasks exhibit frequent conflicts.

# 3 Examples of this Pattern:

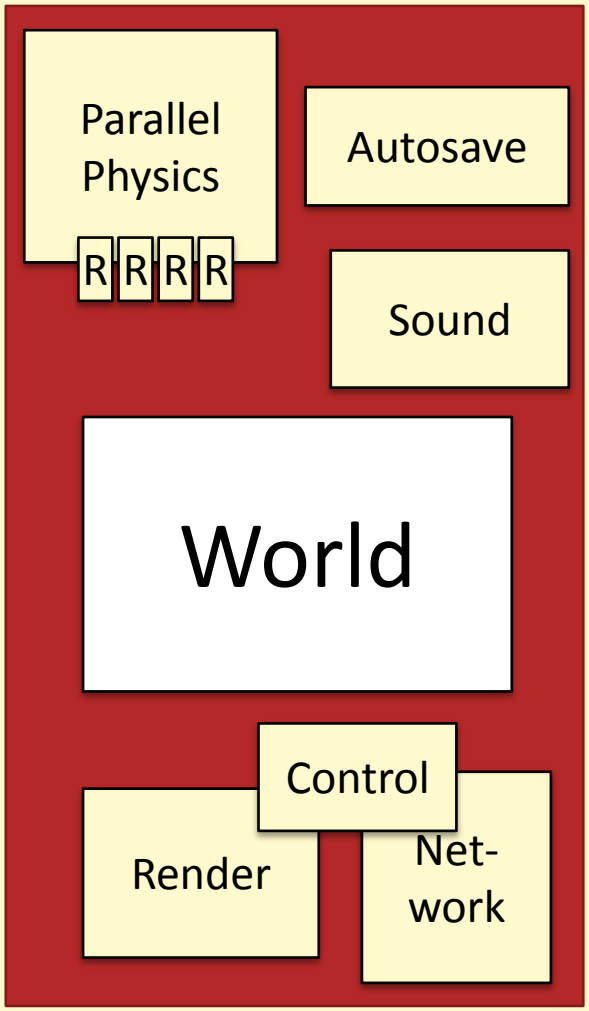
## Office

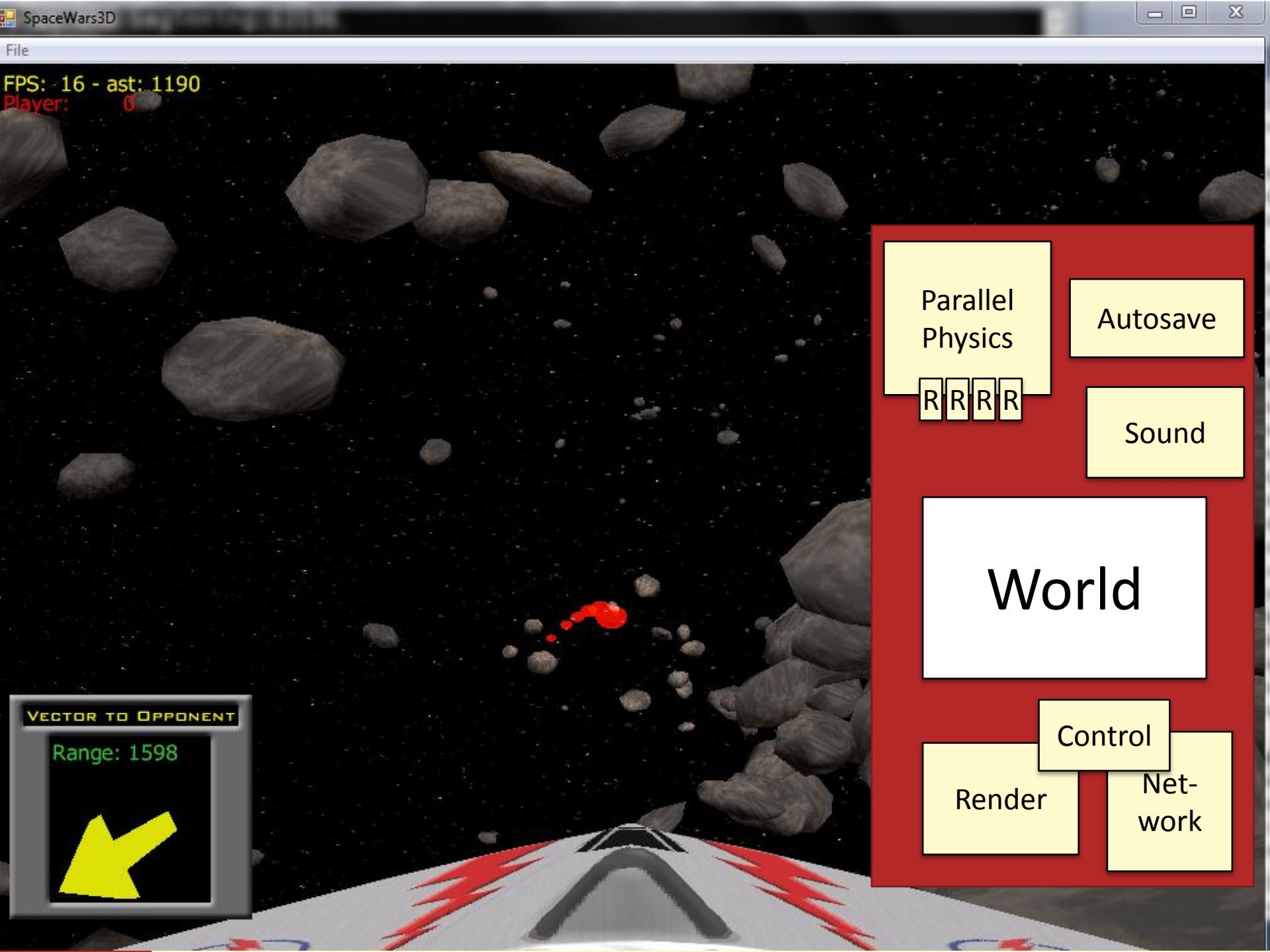



## Browser

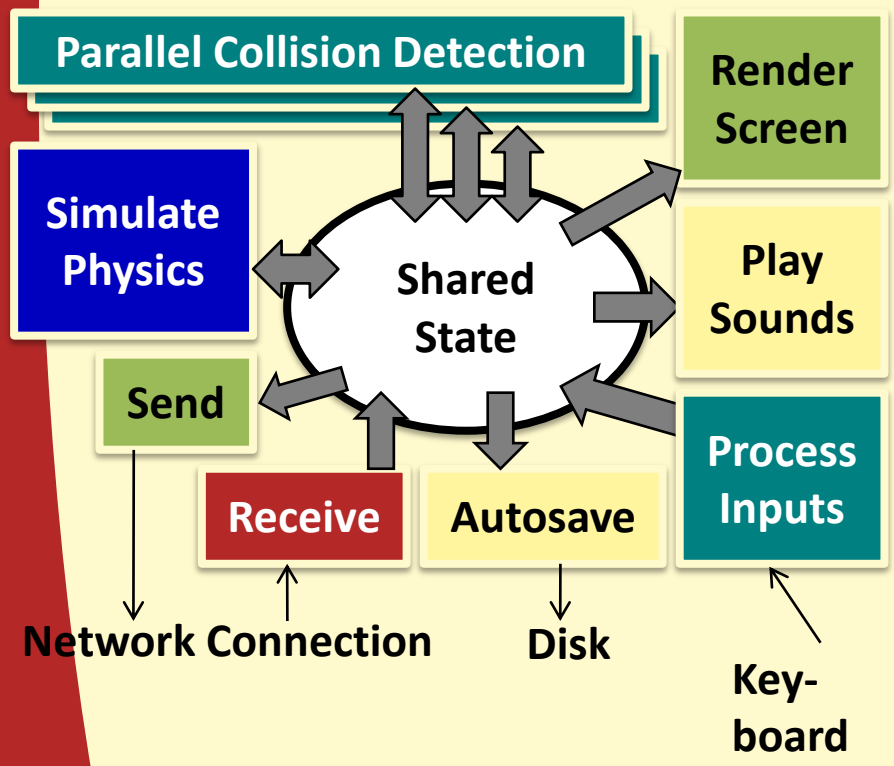



## Game




SpaceWars3D
File
FPS: 16 - ast: 1190
Player: 0
Parallel Physics
R R R R
Autosave
Sound
World
Control
Render
Net-work
Vector to Opponent
Range: 1598

# SpaceWars Game




**Sequential Game Loop:**

```
while (!done)
{
    input.GetInput();
    input.ProcessInput();
    physics.UpdateWorld();
    for (int i = 0; i (physics.numsplits; i++)
        physics.CollisionCheck(i);
    network.SendNetworkUpdates();
    network.HandleQueuedPackets();
    if (frame % 100 = 0)
        SaveGame();
    ProcessGuiEvents();
    screen.RenderFrameToScreen();
    audio.PlaySounds();
    frame++;
}
```

# Can you parallelize this loop?

```
while (!done)
{
    input.GetInput();
    input.ProcessInput();
    physics.UpdateWorld();
    for (int i = 0; i < physics.numsplits; i++)
        physics.CollisionCheck(i);
    network.SendNetworkUpdates();
    network.HandleQueuedPackets();
    if (frame % 100 == 0)
        SaveGame();
    ProcessGuiEvents();
    screen.RenderFrameToScreen();
    audio.PlaySounds();
    frame++;
}
```

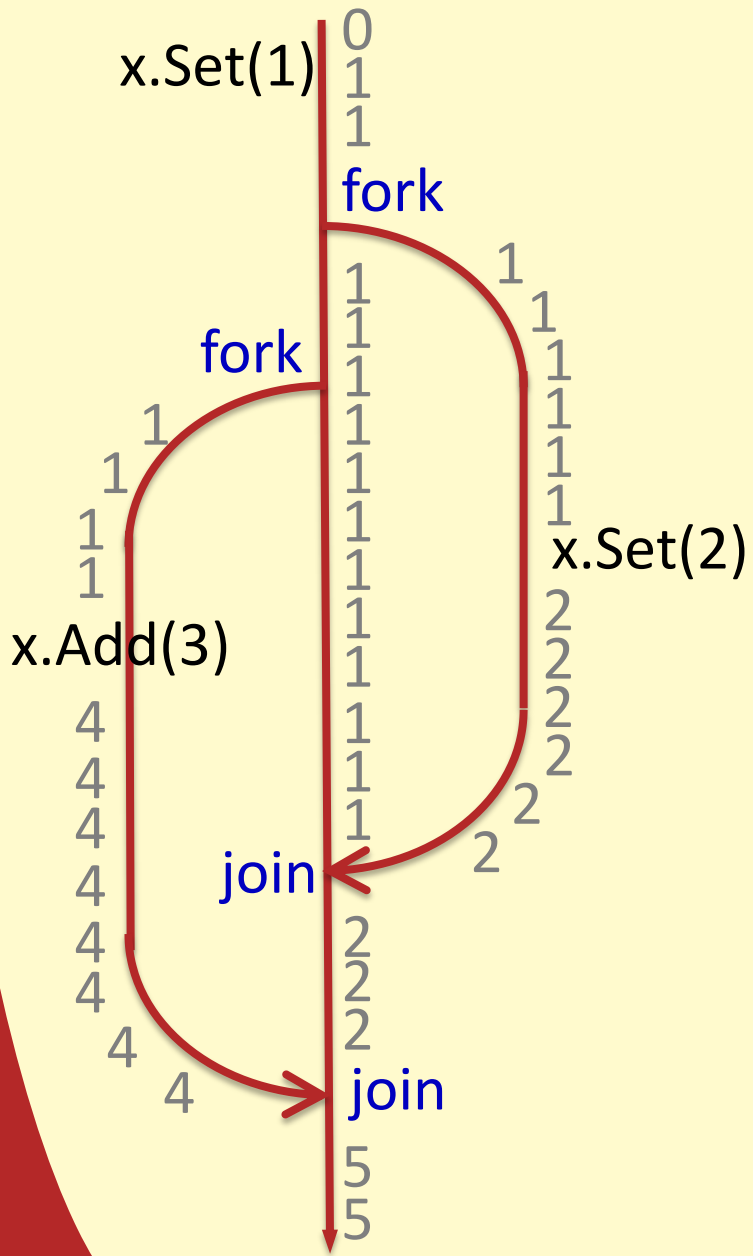
Conflicts on object Coordinates:

- **Reads and writes** all positions
- **Writes** some positions
- **Writes** some positions
- **Reads** all positions
- **Reads** all positions

# Fundamental Insights

- Intra-loop RW conflicts can be eliminated:
  Tasks do not need most recent version of state. What they need is a snapshot of the state, taken at the beginning of each iteration.
- Intra-loop WW conflicts can be eliminated:
  All we need to do is get the priority right (network task trumps conflict detection task trumps physics task)
- How to express this intent?
  Not equivalent to *any* sequential execution of the tasks.

# Concurrent Revisions Model

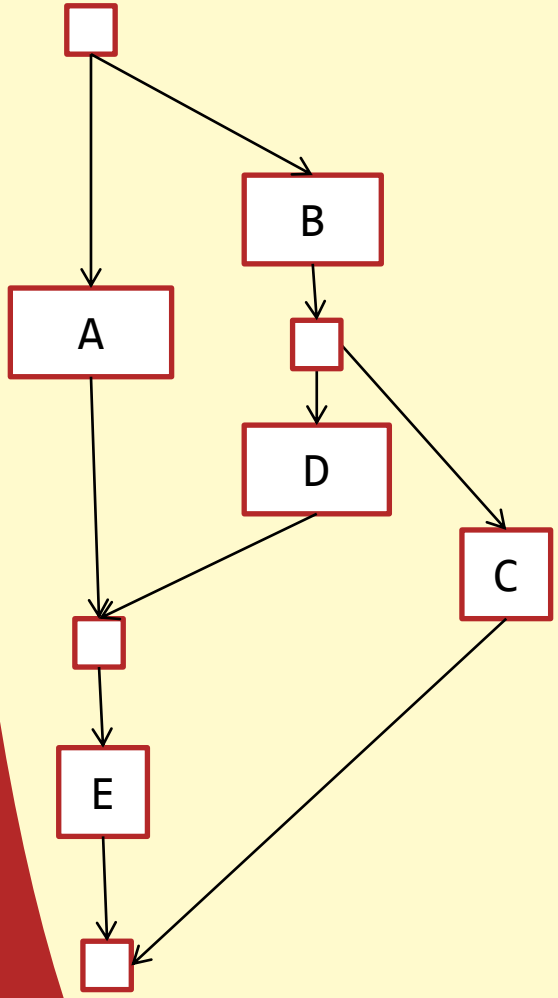



- fork and join revisions (lines with arrow at end)
- Revisions are isolated
  - fork copies all state
  - join replays updates along the arrow at the tip of the arrow
- Use named operations (add/set) instead of basic assignment

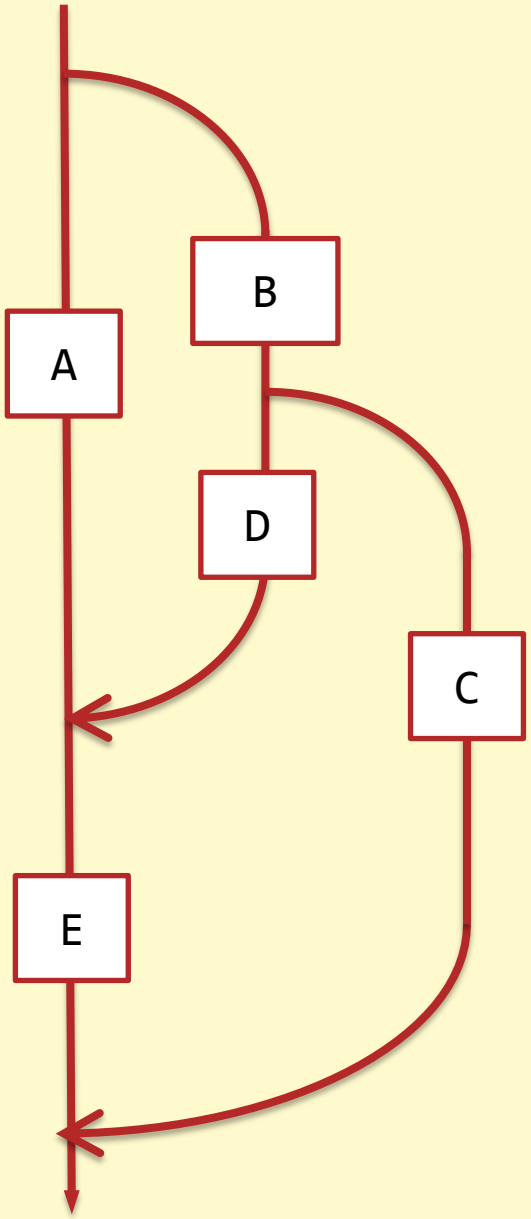
# Revision Diagrams vs. Task Dags

- Different look
  - Rounded corners
  - No boxes for fork, join
  - Arrow tip only at end of revision
- Less general
  - ONLY asymmetric fork, join
  - no symmetric parallel

# Side-By-Side Example

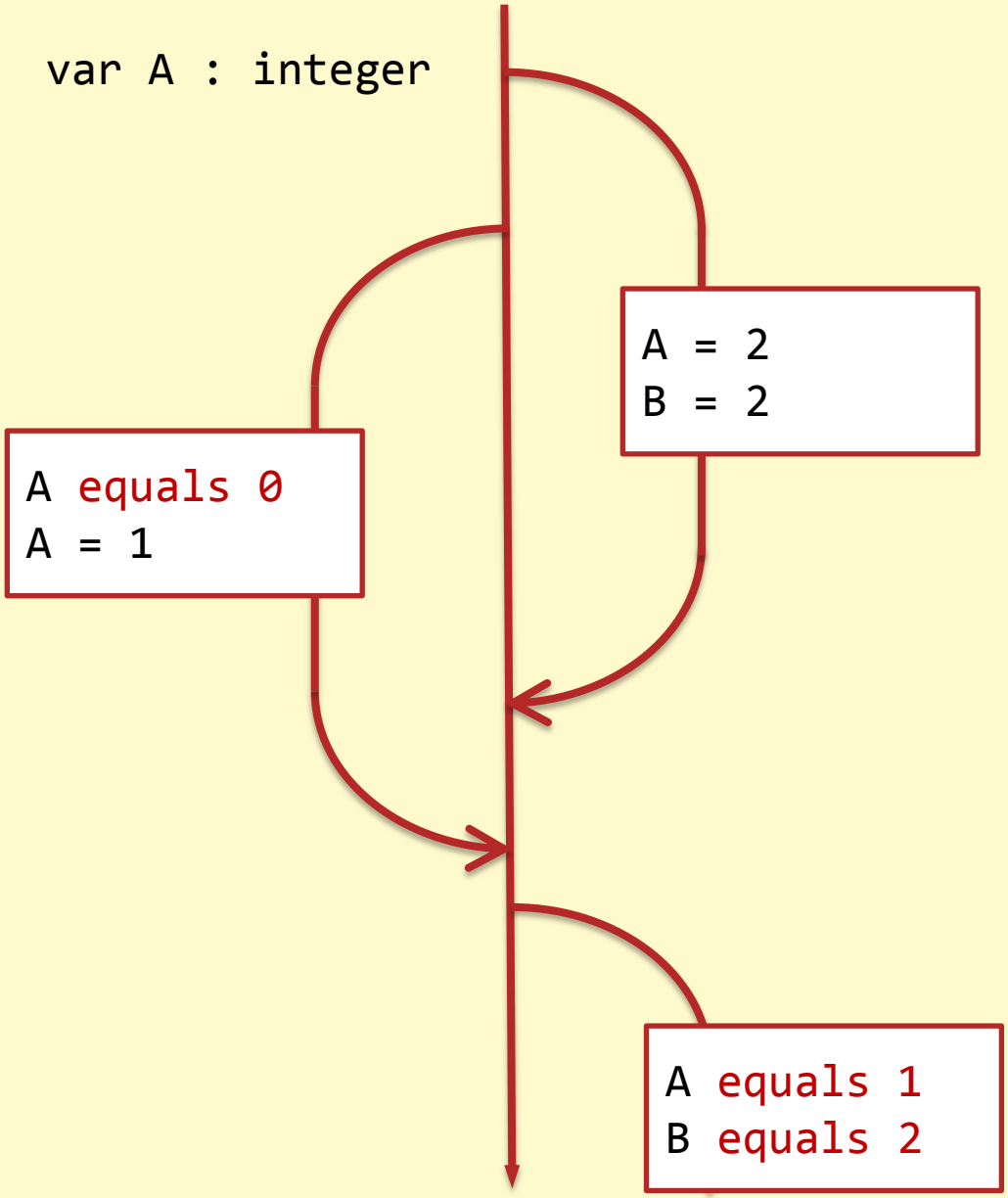



```
var t1,t2;

t1 = fork {
   A(),
   {
      B();
      t2 = fork {
       C();
      }
      D();
   }
}
join t1;
E();
join t2;
```

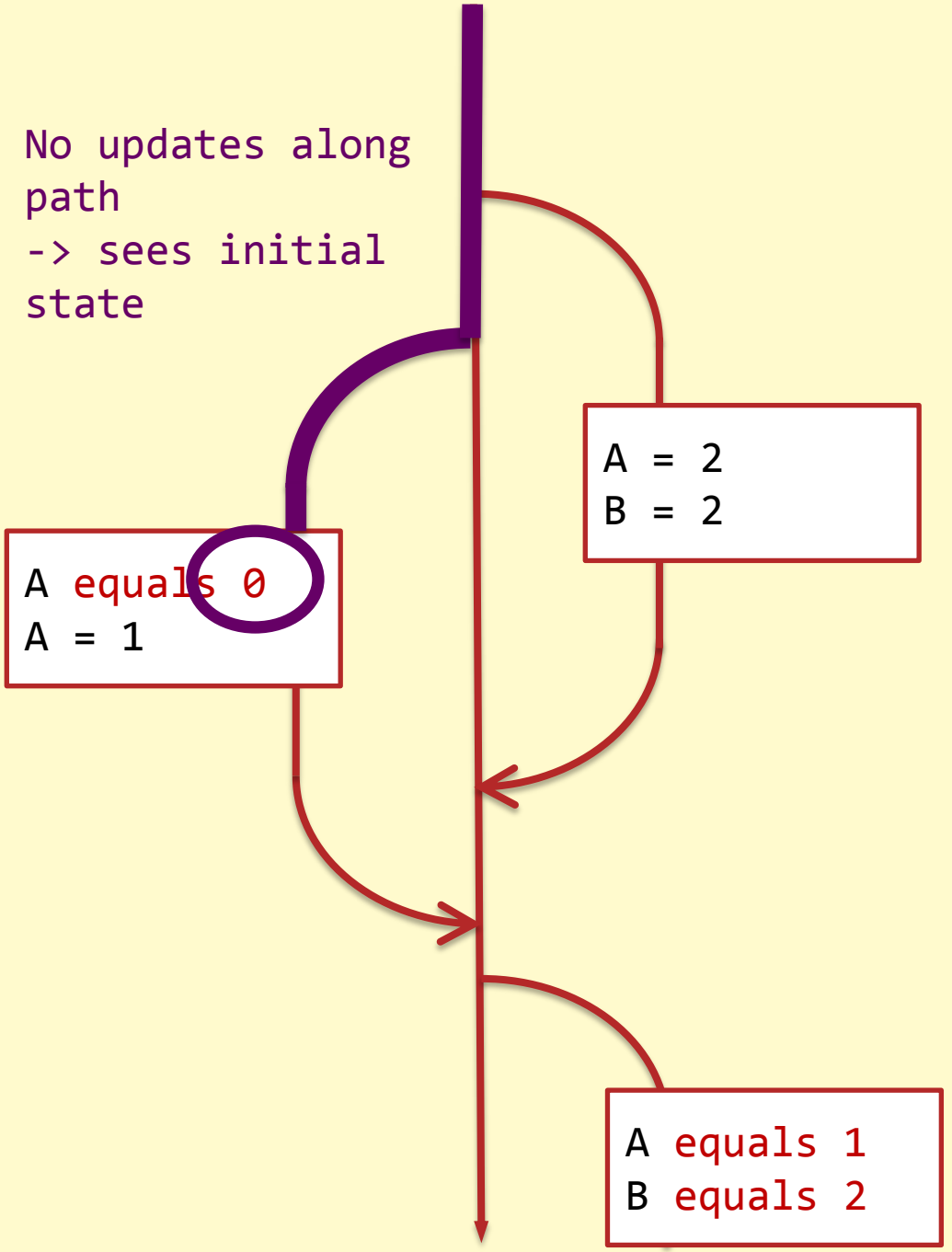


# Understanding Concurrent Revisions

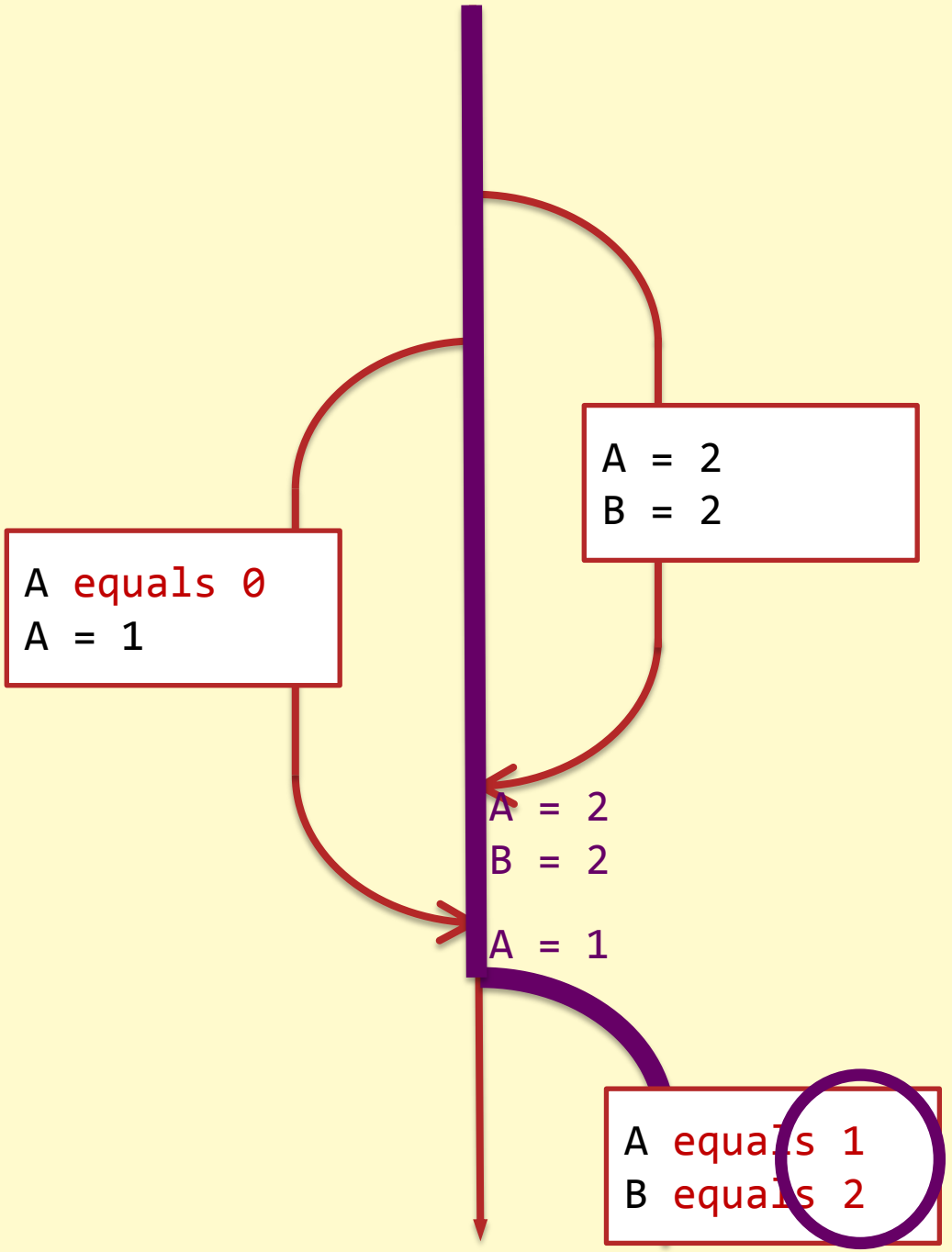
- Fork copies the current state.
- Join replays updates at the tip of arrow.



- State determined by sequence of updates along path from root



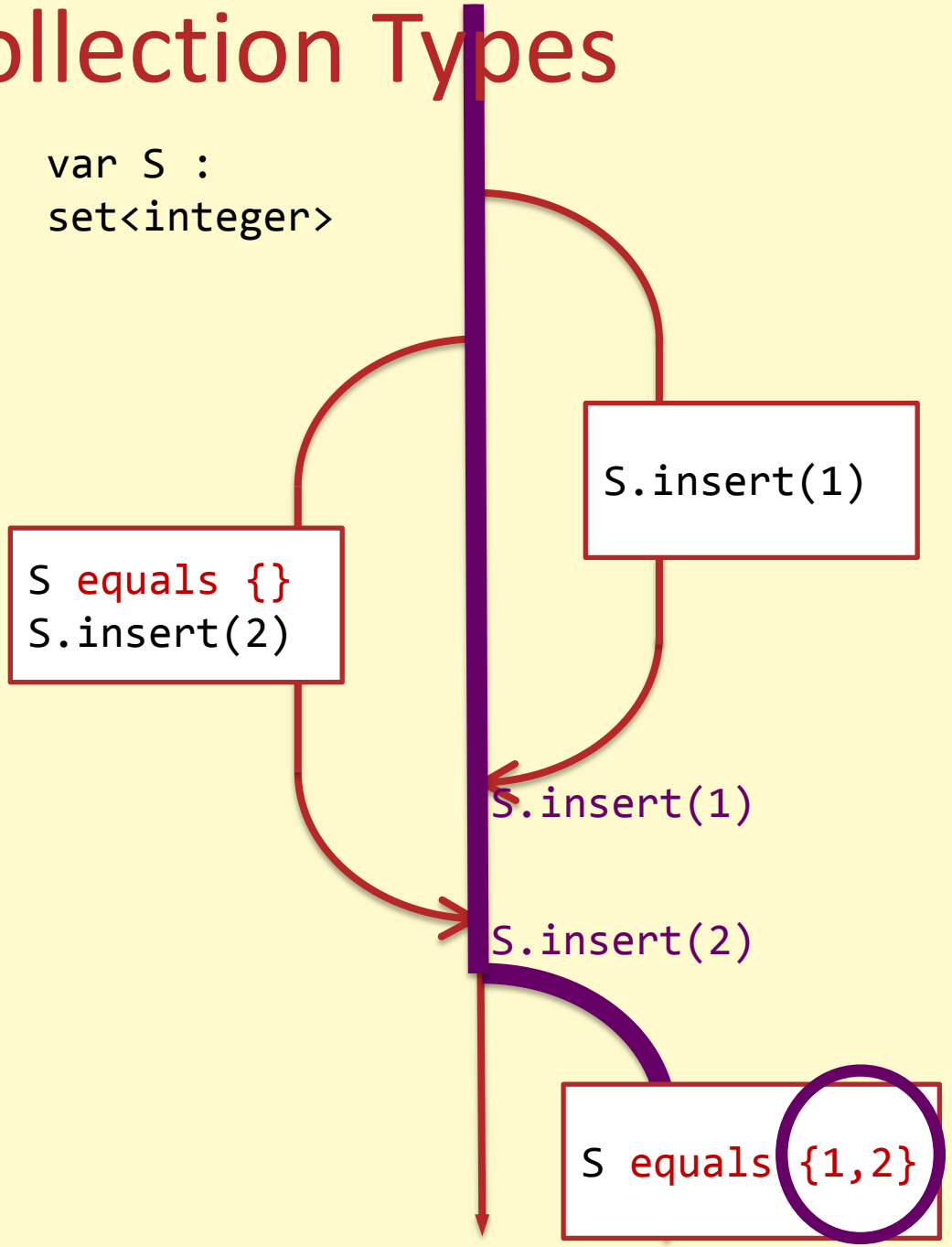
- State determined by sequence of updates along path from root
- Join replays updates at the tip of arrow.



# Can Support Collection Types

- State determined by sequence of updates along path from root
- Updates are now collection operations

Join replays updates at the tip of arrow.



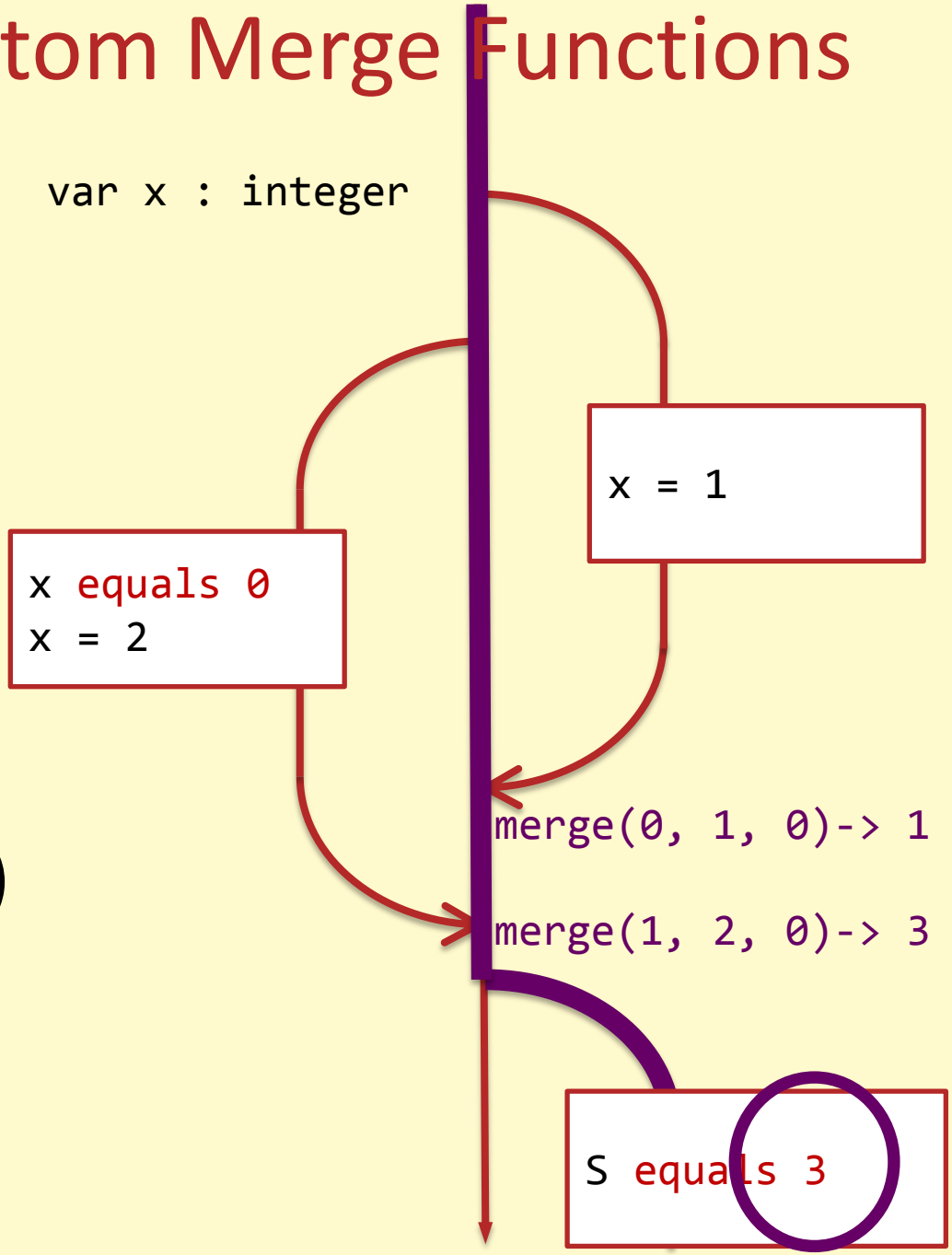
# Can Support Custom Merge Functions

- State determined by result of user-defined merge function
- For example

merge(cur, join, orig)

= cur + (join - orig)

Performs additive accumulation

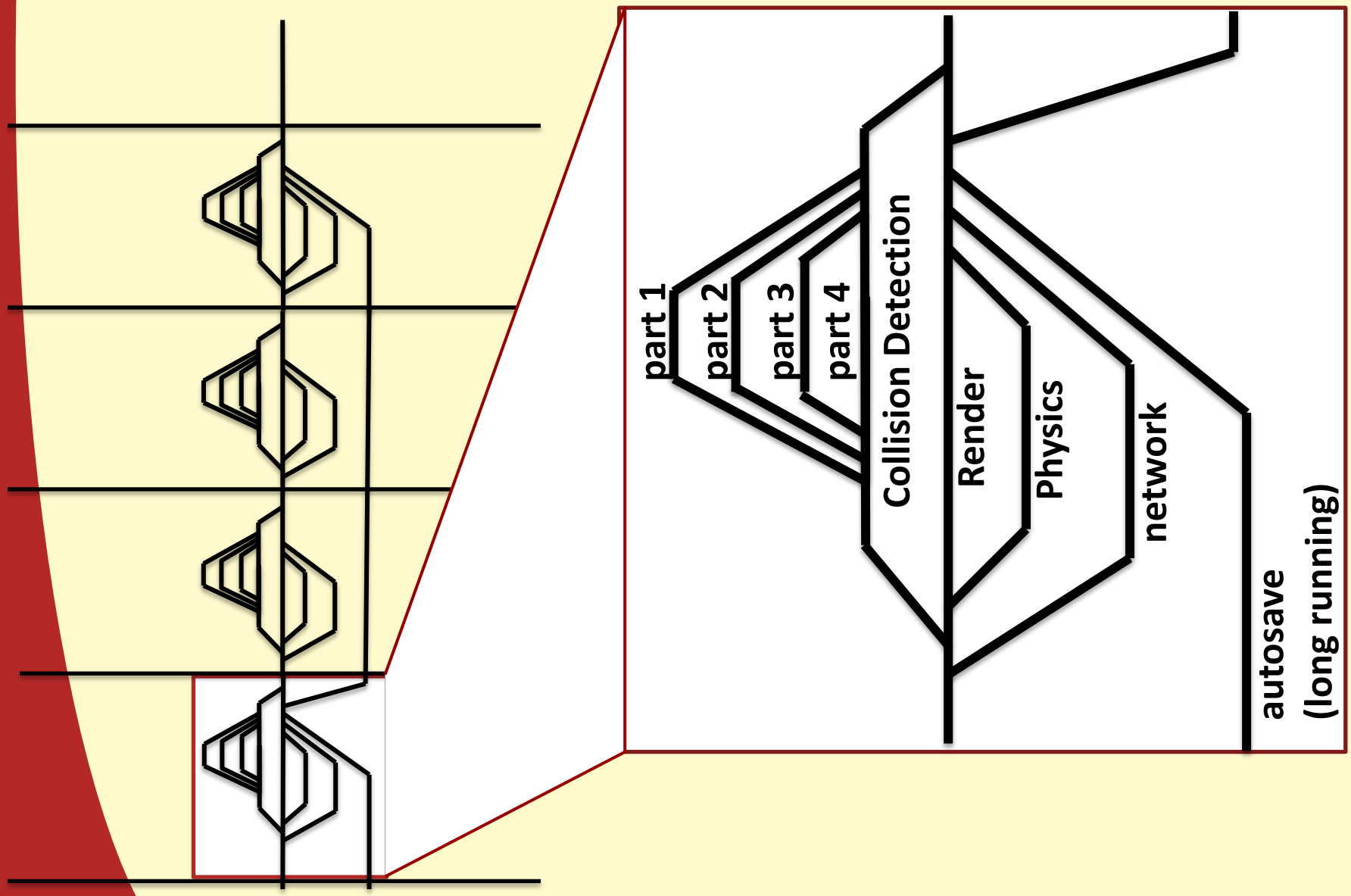


# Applying this idea

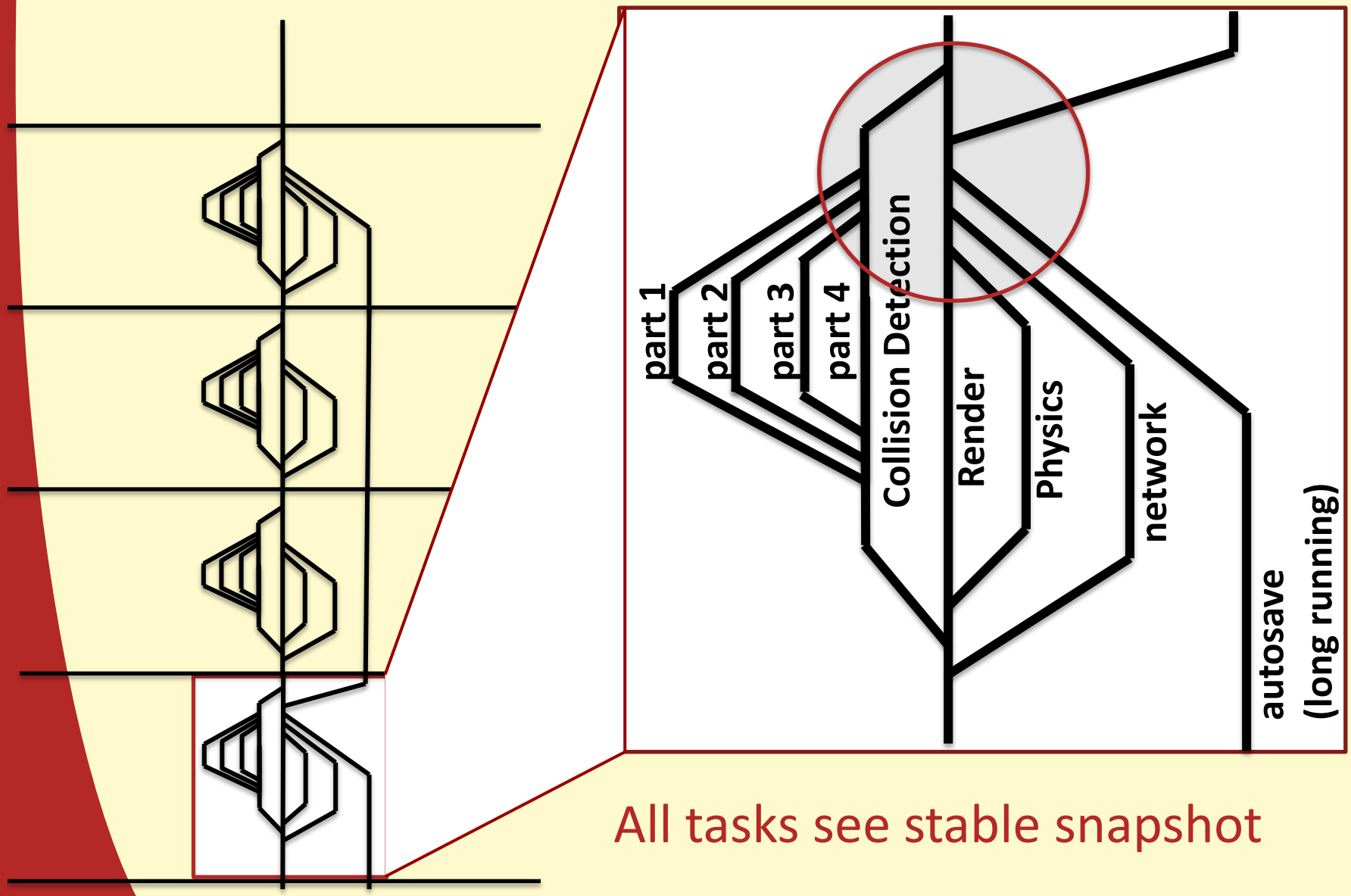
- So now, can we express the intended parallelization of the SpaceWars Loop?

  - Intra-loop RW conflicts can be eliminated:
    Tasks do not need most recent version of state. What they need is a snapshot of the state, taken at the beginning of each iteration.
  - Intra-loop WW conflicts can be eliminated:
    All we need to do is get the priority right (network task trumps conflict detection task trumps physics task)

# Revision Diagram of Parallelized Game Loop

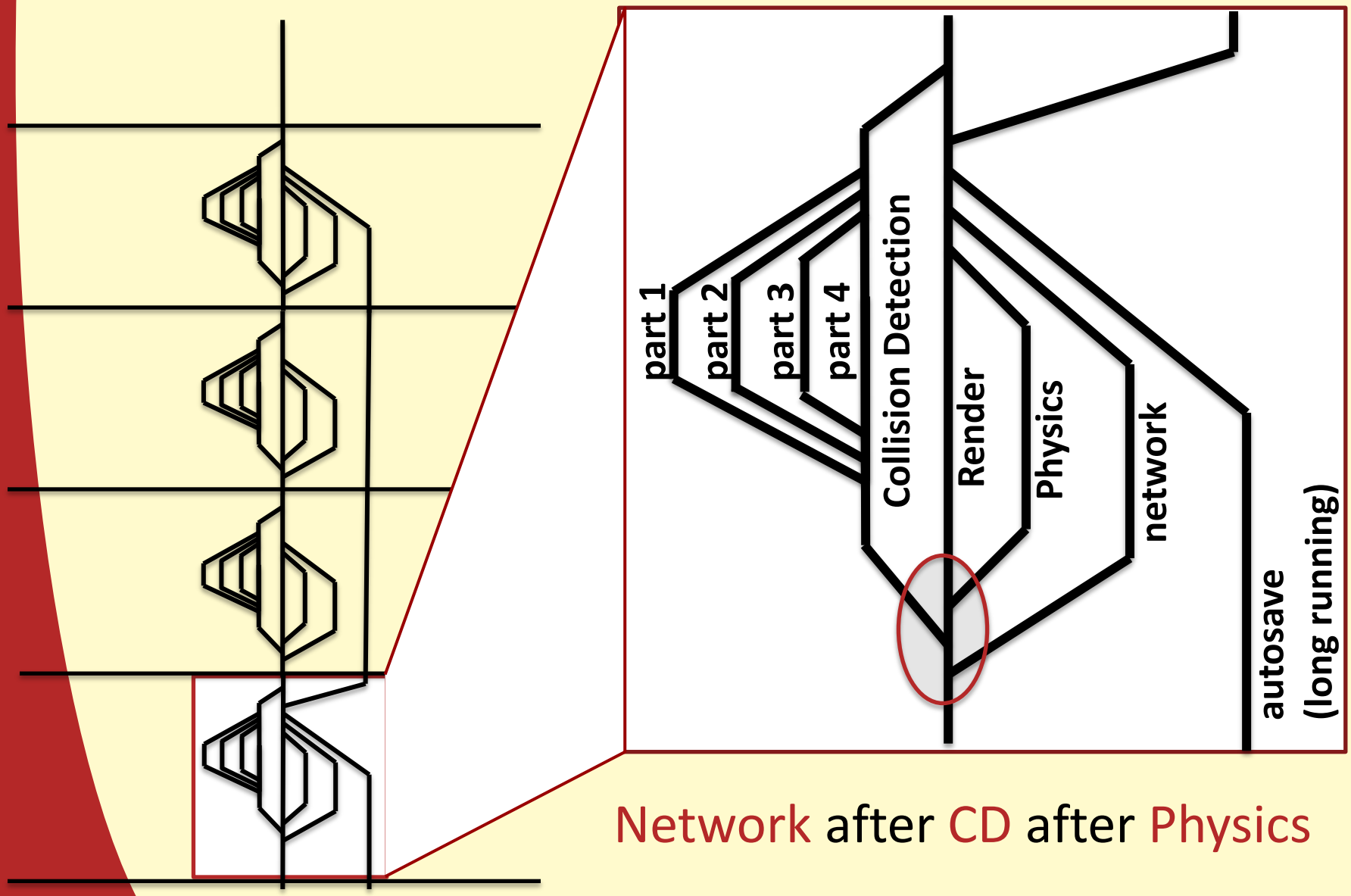


# Eliminated Read-Write Conflicts



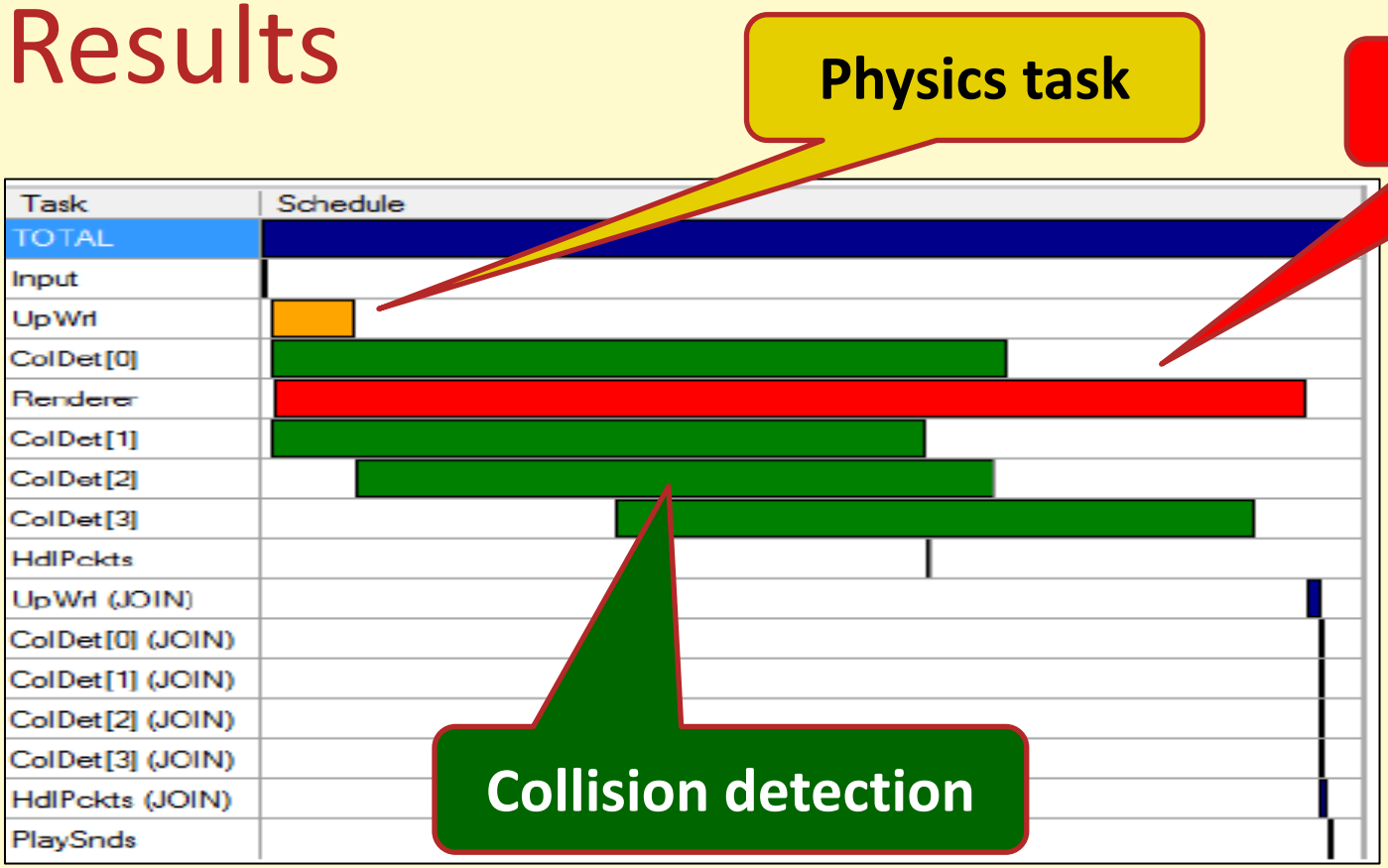

All tasks see stable snapshot

# Eliminated Write-Write Conflicts




Network after CD after Physics

# Results



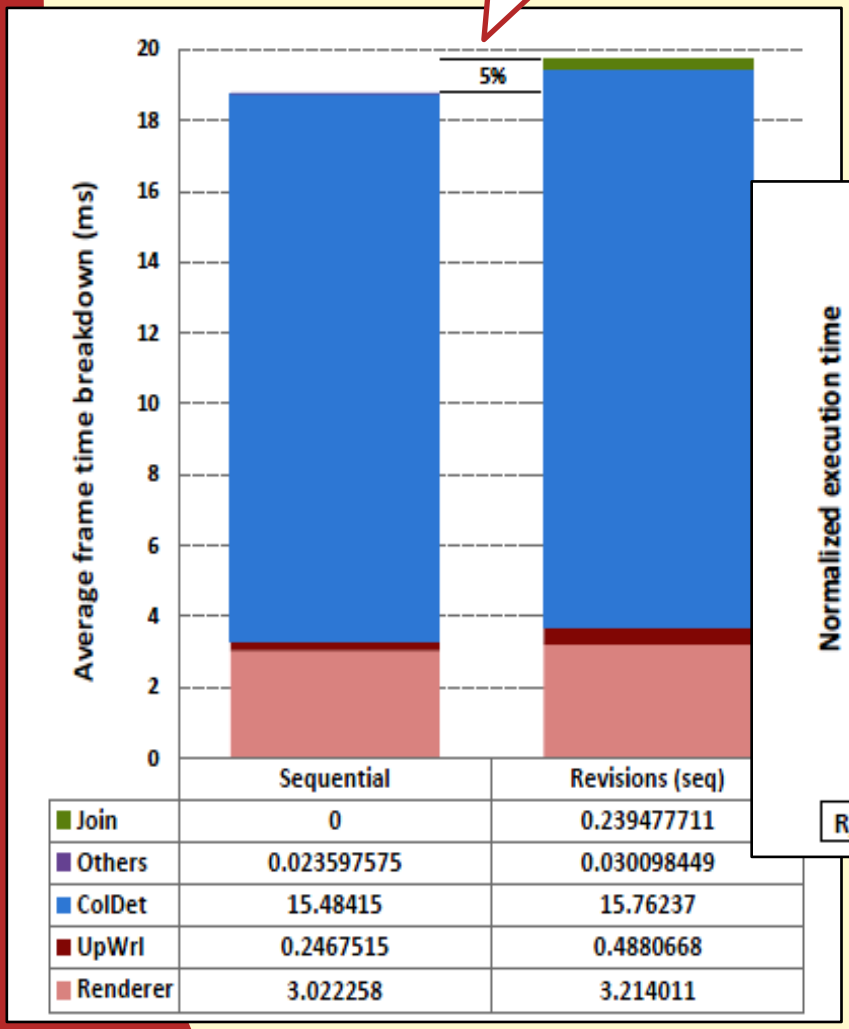

- Autosave now perfectly unnoticeable in background
- Overall Speed-Up: **3.03x on four-core**
  (almost completely limited by graphics card)

# Overhe

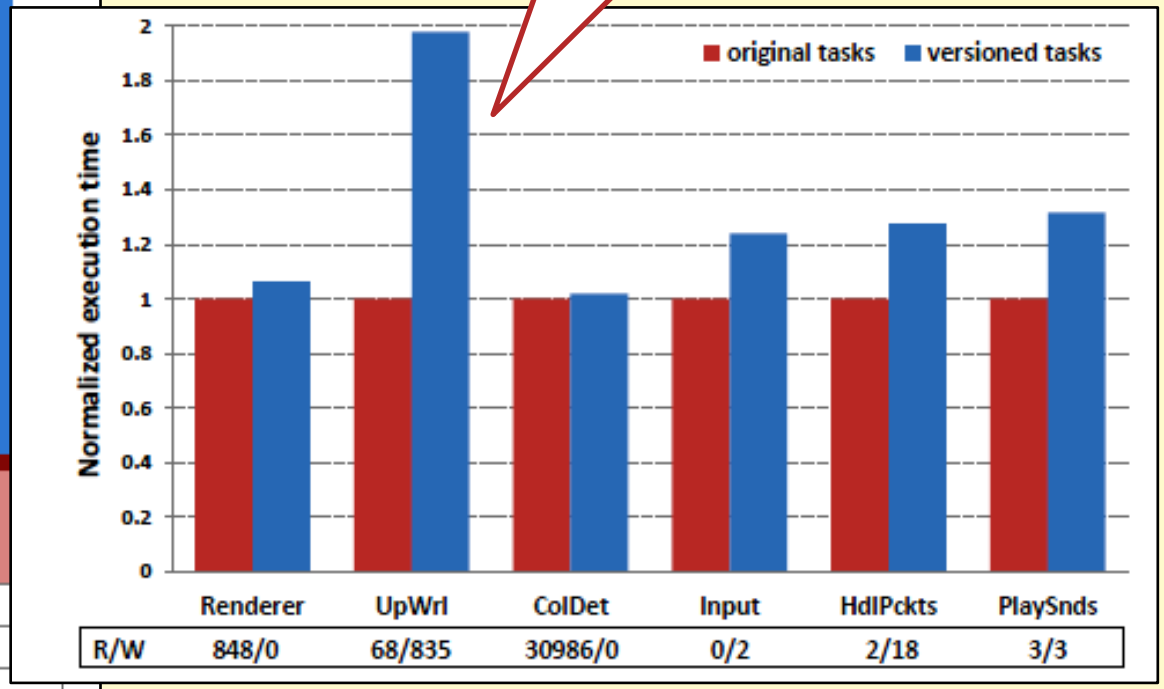
How much d[illegible] the copying and t[illegible]

Only a 5% slowdown in the sequential case

Some individual tasks slow down much more (i.e. physics simulation)




| | Sequential | Revisions (seq) |
|---|---|---|
| Join | 0 | 0.239477711 |
| Others | 0.023597575 | 0.030098449 |
| ColDet | 15.48415 | 15.76237 |
| UpWrl | 0.2467515 | 0.4880668 |
| Renderer | 3.022258 | 3.214011 |




| | Renderer | UpWrl | ColDet | Input | HdlPckts | PlaySnds |
|---|---|---|---|---|---|---|
| R/W | 848/0 | 68/835 | 30986/0 | 0/2 | 2/18 | 3/3 |

# PART III: A LOOK AT CODE

# The C# library & IL rewriter

- We have seen the programming model
  - versioned data
  - fork/join of revisions
- But what does it concretely look like?
  - Let's see some code.

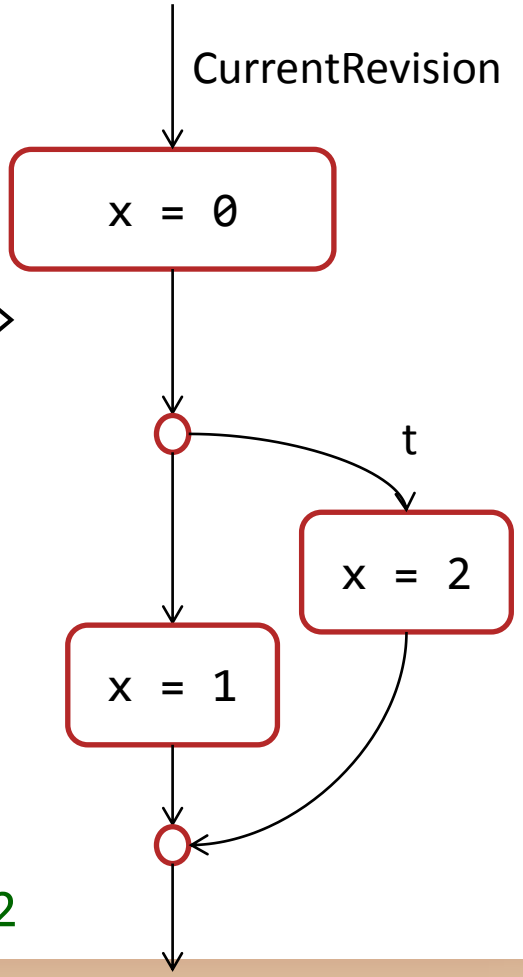
Or do it yourself: visit http://rise4fun.com and play

# Version Data

```
[Versioned] int z;
[Versioned] string s = "abc";

[Versioned, MergeWith(“AdditiveMerge”)]
int i = 0;

static int AdditiveMerge(int current,
                         int join,
                         int original)  {
  // add the values of the current and joined revision,
  // and subtract the original (or we would count it twice)
  return current + join - original;
}
```

# Using Simple Fork/Join

```
[Versioned] int x = 0;


RevisionTask t = CurrentRevision.Fork(() =>
                 {
                    x = 2;
                 });
X = 1;

CurrentRevision.Join(t);


Console.WriteLine("x = " + x ); // writes 2
```



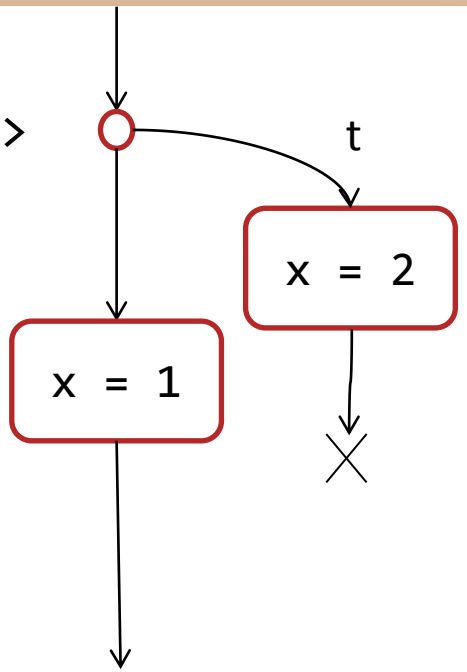
# Ordering Joins

```
RevisionTask a = CurrentRevision.Fork(() =>
                 {
                     x = 1;
                 });
RevisionTask b = CurrentRevision.Fork(() =>
                 {
                     x = 2;
                 });

CurrentRevision.Join(b);
CurrentRevision.Join(a);

Console.WriteLine("x = " + x ); // writes 1
```

# Abandoning Revisions

- We can abandon a revision (instead of join)
  - Discards the version once task completes
  - Is not a cancellation: task still runs to completion

```
RevisionTask t = CurrentRevision.Fork(() =>
                 {
                    x = 2;
                 });

x = 1;

CurrentRevision.Abandon(t);

Console.WriteLine("x = " + x ); // writes 1
```



# Example: Background Save

- If a task does only read from shared state, we can abandon it to avoid blocking on a join
- Example: background save operation
  - never blocks application
  - snapshot survives as long as it needs to

```
RevisionTask backgroundsave =
    CurrentRevision.Fork(() => { SaveStateToFile(); });

// don’t wait for task… let it finish on its own time
CurrentRevision.Abandon(backgroundsave);
```

ADVANCED SECTION – BONUS MATERIAL

# PART IV: FORMALIZATION & DETERMINACY

# Formal Semantics

- See paper in [ESOP 2011]
- Similar to the AME calculus by Abadi *et al.*
- Proof of determinism
- Formal correspondence to the Revision Diagrams
- Proof of the semi-lattice property
- Can be shown to generalize ‘snapshot isolation’

# Syntax

**Syntactic Symbols**

$$
\begin{aligned}
v \in \mathit{Val} &::= c \mid x \mid l \mid r \mid \lambda x.e \\
c \in \mathit{Const} &::= \text{unit} \mid \text{false} \mid \text{true} \\
l &\in \mathit{Loc} \\
r &\in \mathit{Rid} \\
x &\in \mathit{Var} \\
e \in \mathit{Expr} &::= v \\
&\quad \mid \; e\ e \mid (e\,?\,e : e) \\
&\quad \mid \; \text{ref}\ e \mid !e \mid e := e \\
&\quad \mid \; \text{rfork}\ e \mid \text{rjoin}\ e
\end{aligned}
$$

# State

$$LocalStore = Loc \rightharpoonup Val$$

# State

**Map from revision identifiers to revision state**
(no global shared state!)

$$GlobalState = Rid \rightharpoonup LocalState$$
$$LocalState = Snapshot \times LocalStore \times Expr$$

$$Snapshot = Loc \rightharpoonup Val$$

$$LocalStore = Loc \rightharpoonup Val$$

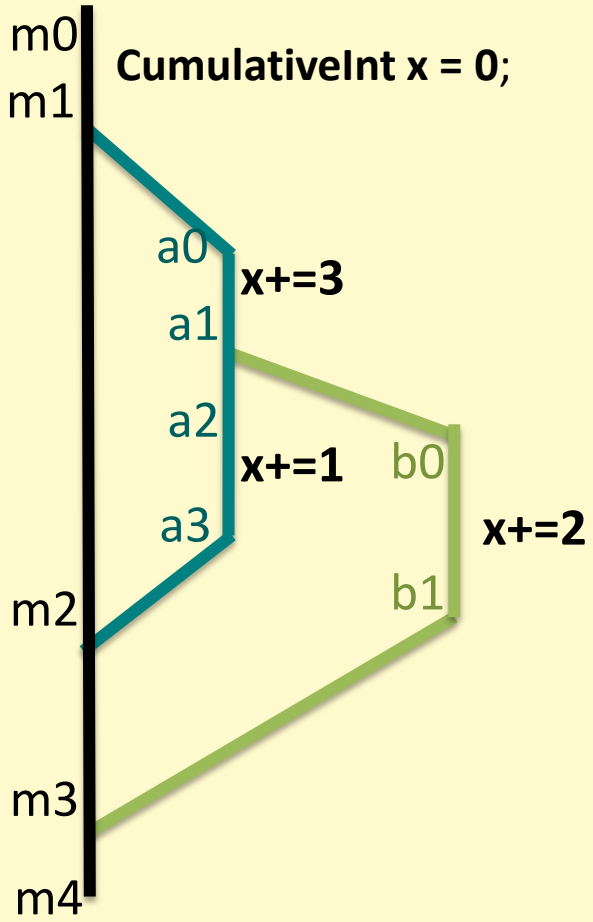
**"current program"**
(expression for func. lang., pc+stack for imp. lang)

**Snapshot of store**
Contains original value of all locations

**Writes performed**
Contains current value of all locations that were modified

# State

$$
\begin{aligned}
s \in \mathit{GlobalState} &= \mathit{Rid} \rightharpoonup \mathit{LocalState} \\
\mathit{LocalState} &= \mathit{Snapshot} \times \mathit{LocalStore} \times \mathit{Expr} \\
\sigma \in \mathit{Snapshot} &= \mathit{Loc} \rightharpoonup \mathit{Val} \\
\tau \in \mathit{LocalStore} &= \mathit{Loc} \rightharpoonup \mathit{Val}
\end{aligned}
$$




{ m->({}, {}, m0) }
{ m->({}, {x->0}, m1) }
{ m->({}, {x->0}, m2), a->({x->0}, {}, a0) }
{ m->({}, {x->0}, m2), a->({x->0}, {x->3}, a1) }
{ m->({}, {x->0}, m2), a->({x->0}, {x->3}, a2), b->({x->3}, {}, b0) }
{ m->({}, {x->0}, m2), a->({x->0}, {x->4}, a3), b->({x->3}, {}, b0) }
{ m->({}, {x->0}, m2), a->({x->0}, {x->4}, a3), b->({x->3}, {x->5}, b1) }
{ m->({}, {x->4}, m3), b->({x->3}, {x->5}, b1) }
{ m->({}, {x->6}, m4) }

# Execution Contexts

$$
\begin{aligned}
\mathcal{E} = {} & \Box \\
\mid {} & \mathcal{E}\ e \mid v\ \mathcal{E} \mid (\mathcal{E}\ ?\ e : e) \\
\mid {} & \mathsf{ref}\ \mathcal{E} \mid\ !\mathcal{E} \mid \mathcal{E} := e \mid l := \mathcal{E} \\
\mid {} & \mathsf{rjoin}\ \mathcal{E}
\end{aligned}
$$

# Expressions

$$
\begin{aligned}
v \in \mathit{Val} \quad & ::= c \mid x \mid l \mid r \mid \lambda x.e \\
c \in \mathit{Const} & ::= \mathsf{unit} \mid \mathsf{false} \mid \mathsf{true} \\
l \in \mathit{Loc} & \\
r \in \mathit{Rid} & \\
x \in \mathit{Var} & \\
e \in \mathit{Expr} & ::= v \\
& \quad \mid\ e\ e \mid (e\ ?\ e : e) \\
& \quad \mid\ \mathsf{ref}\ e \mid\ !e \mid e := e \\
& \quad \mid\ \mathsf{rfork}\ e \mid \mathsf{rjoin}\ e
\end{aligned}
$$

- An expression matches at most one redex

# Redexes

$$
\begin{aligned}
& \mathcal{E}[(\lambda x.e)\ v] \\
& \mathcal{E}[(\mathsf{true}\ ?\ e_1 : e_2)] \\
& \mathcal{E}[(\mathsf{false}\ ?\ e_1 : e_2)] \\
& \\
& \mathcal{E}[\mathsf{ref}\ v] \\
& \mathcal{E}[!l] \\
& \mathcal{E}[l := v] \\
& \\
& \mathcal{E}[\mathsf{rfork}\ e] \\
& \mathcal{E}[\mathsf{rjoin}\ r']
\end{aligned}
$$

# Operational Semantics

$$
\begin{array}{ll}
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\lambda x.e)\ v] \rangle) & \rightarrow_r\ s[r \mapsto \langle \sigma, \tau, \mathcal{E}[[v/x]e] \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{true}\ ?\ e_1 : e_2)] \rangle) & \rightarrow_r\ s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_1] \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{false}\ ?\ e_1 : e_2)] \rangle) & \rightarrow_r\ s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_2] \rangle] \\
 & \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v] \rangle) & \rightarrow_r\ s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[l] \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[!l] \rangle) & \rightarrow_r\ s[r \mapsto \langle \sigma, \tau, \mathcal{E}[(\sigma::\tau)(l)] \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[l := v] \rangle) & \rightarrow_r\ s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[\mathsf{unit}] \rangle] \\
 & \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rfork}\ e] \rangle) & \rightarrow_r\ s[r \mapsto \langle \sigma, \tau, \mathcal{E}[r'] \rangle][r' \mapsto \langle \sigma::\tau, \epsilon, e \rangle] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \langle \sigma', \tau', v \rangle) & \rightarrow_r\ s[r \mapsto \langle \sigma, \tau::\tau', \mathcal{E}[\mathsf{unit}] \rangle][r' \mapsto \bot] \\
s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \bot) & \rightarrow_r\ \epsilon
\end{array}
$$

$$
\begin{array}{rcl}
s \in \mathit{GlobalState} & = & \mathit{Rid} \rightharpoonup \mathit{LocalState} \\
\mathit{LocalState} & = & \mathit{Snapshot} \times \mathit{LocalStore} \times \mathit{Expr} \\
\sigma \in \mathit{Snapshot} & = & \mathit{Loc} \rightharpoonup \mathit{Val} \\
\tau \in \mathit{LocalStore} & = & \mathit{Loc} \rightharpoonup \mathit{Val}
\end{array}
$$

# Local Operations w/o side effects

$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\lambda x.e)\ v] \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[[v/x]e] \rangle]$$
$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{true}\ ?\ e_1 : e_2)] \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_1] \rangle]$$
$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{false}\ ?\ e_1 : e_2)] \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_2] \rangle]$$

For some revision $r$, with snapshot $\sigma$ and local modifications $\tau$

$$\begin{aligned} s \in \mathit{GlobalState} &= \mathit{Rid} \rightharpoonup \mathit{LocalState} \\ \mathit{LocalState} &= \mathit{Snapshot} \times \mathit{LocalStore} \times \mathit{Expr} \\ \sigma \in \mathit{Snapshot} &= \mathit{Loc} \rightharpoonup \mathit{Val} \\ \tau \in \mathit{LocalStore} &= \mathit{Loc} \rightharpoonup \mathit{Val} \end{aligned}$$

# Local Operations w/ side effects

$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v] \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[l] \rangle]$$
$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[!l] \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[(\sigma :: \tau)(l)] \rangle]$$
$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[l := v] \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[\mathsf{unit}] \rangle]$$

Pick a fresh location identifier

Read current state $\sigma :: \tau$ meaning the changes $\tau$ applied snapshot $\sigma$

Update changeset $\tau$

$$s \in \mathit{GlobalState} = \mathit{Rid} \rightharpoonup \mathit{LocalState}$$
$$\mathit{LocalState} = \mathit{Snapshot} \times \mathit{LocalStore} \times \mathit{Expr}$$
$$\sigma \in \mathit{Snapshot} = \mathit{Loc} \rightharpoonup \mathit{Val}$$
$$\tau \in \mathit{LocalStore} = \mathit{Loc} \rightharpoonup \mathit{Val}$$

# Fork and Join

On a **fork**, the snapshot of the new revision $r'$ is the current state: $\sigma :: \tau$

$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\text{rfork } e] \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[r'] \rangle][r' \mapsto \langle \sigma :: \tau, \epsilon, e \rangle]$$

$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\text{rjoin } r'] \rangle)(r' \mapsto \langle \sigma', \tau', v \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau :: \tau', \mathcal{E}[\text{unit}] \rangle][r' \mapsto \bot]$$

$$s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\text{rjoin } r'] \rangle)(r' \mapsto \bot) \rightarrow_r \epsilon$$

Trying to **join** a unknown revision identifier (i.e. revision has already been joined) transitions the whole system into a special error state

On a **join**, the writes of the joinee $r'$ take priority over the writes of the current revision: $\sigma :: \tau'$

$$\begin{aligned} s \in \mathit{GlobalState} &= \mathit{Rid} \rightharpoonup \mathit{LocalState} \\ \mathit{LocalState} &= \mathit{Snapshot} \times \mathit{LocalStore} \times \mathit{Expr} \\ \sigma \in \mathit{Snapshot} &= \mathit{Loc} \rightharpoonup \mathit{Val} \\ \tau \in \mathit{LocalStore} &= \mathit{Loc} \rightharpoonup \mathit{Val} \end{aligned}$$

# Determinacy

- Execution is determinate! That is, the final state of program execution does not depend on any scheduling decisions.

**Theorem 1 (Determinacy).** *Let $e$ be a program expression, and let $e \downarrow s$ and $e \downarrow s'$. Then $s \approx s'$.*

- With two caveats:
  - States could differ in identifiers used for revisions and locations. We say s ≈ s' if states s, s' are equivalent in this sense.
  - It is possible for some executions to diverge while others don't.

# Proving Determinacy (1/3)

- First: if we fix a particular revision r to take a step, the outcome is determinate

**Lemma 1 (Local Determinism).** *If* $s_1 \approx s_1'$ *and* $s_1 \rightarrow_r s_2$ *and* $s_1' \rightarrow_r s_2'$, *then* $s_2 \approx s_2'$.

- Because the expression representing the program can evaluate in at most one way
  - An expression matches at most one redex
  - Each redex matches a unique operational rule

# Proving Determinacy (2/3)

- Local Confluence: if two different revisions take a step, they both can take (zero or one) additional step to get to an equivalent state

**Lemma 2 (Strong Local Confluence).** *Let $s_1$ and $s'_1$ be reachable states that satisfy $s_1 \approx s'_1$. Then, if $s_1 \rightarrow_r s_2$ and $s'_1 \rightarrow_{r'} s'_2$, then there exist equivalent states $s_3 \approx s'_3$ such that both $s_2 \rightarrow^?_{r'} s_3$ and $s'_2 \rightarrow^?_r s'_3$.*

Reason: steps by different revisions do commute because they either

- Do not influence each other
- Are mutually exclusive (two revisions trying to join the same revision) thus triggering a transition to the error state.