Multithreaded Architectures
Lecture 4 of 4

Supercomputing ’93 Tutorial

Friday, November 19, 1993

Portland, Oregon

Rishiyur S. Nikhil

Digital Equipment Corporation
Cambridge Research Laboratory

Copyright (c) 1992, 1993 Digital Equipment Corporation

The Software Story
Overview of Lecture 4

- **The software view: shared memory model**
- **Compatibility with existing parallel software**
- **New languages and their implementations**
- **Extensions to existing languages, implementation**
- **Resource management issues**

The Software View

Multithreaded architectures:

- **Have physically distributed memories, for scalability reasons**
- **However, they all support a single, global (machine−wide) address space**

  (The J−Machine provides global addressing through its support for global object identifiers)

- **All of them are targeted to support:**

  A Shared Memory Programming Model

Software Compatibility

Software developed for Massively Parallel Architectures that are not globally addressed, such as

- **MIMD: Intel iPSCs, nCube, CM−5, Transputer arrays, ...**
- **SIMD: MasPar, DECmpp, CM−2, ...**

is expected to work at least as well on multithreaded architectures, because:

- **Much cheaper communication**
- **Message−passing libraries easily simulated, efficiently**

Such programs include

- **Programs with explicit message−passing**
- **SIMD/SPMD programs:**

  Fortran D
  Fortran 90
  HPF (High Performance Fortran)

Locality in these programs: suit cache coherent architectures beautifully (DASH, KSR−1, Alewife)

The Software Story

- "Automatic Parallelization"
  - Not main emphasis for multithreaded archs. Too hard a problem, for more dynamic parallelism
  - Can be used profitably, where applicable
- New languages:
  - Id
  - Concurrent Smalltalk
- Extensions to existing languages:
  - Fortran, C, Lisp
  - Methods to create new threads, terminate threads
  - Data structures with fine−grain synchronization (synchronization on individual elements)
- Truly "data parallel" programs:
  - No restriction on data parallel operations
  - No restriction on nesting (compositionality)

Multithreaded Architectures

New Languages

The Id Programming Language

Implicit parallelism, declarative style and data−driven execution permit the easy expression of abundant, fine−grain parallelism.

- **Asynchronous, parallel procedure calls and loops**
- **Parallel argument evaluation**

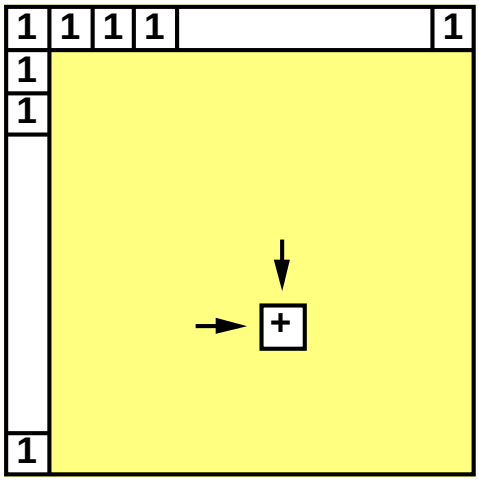
Atomic data structures: implicit synchronization on every field (every field has FULL/EMPTY state; readers automatically block on empty fields).

I−Structures: **Write−once data structures. Reads can be issued before the write.**

M−Structures: **Updatable data structures. Reads and writes alternate, despite out−of−order issuing.**

The "Wavefront" Program

(abstracted from SOR, radiation transport, ...)

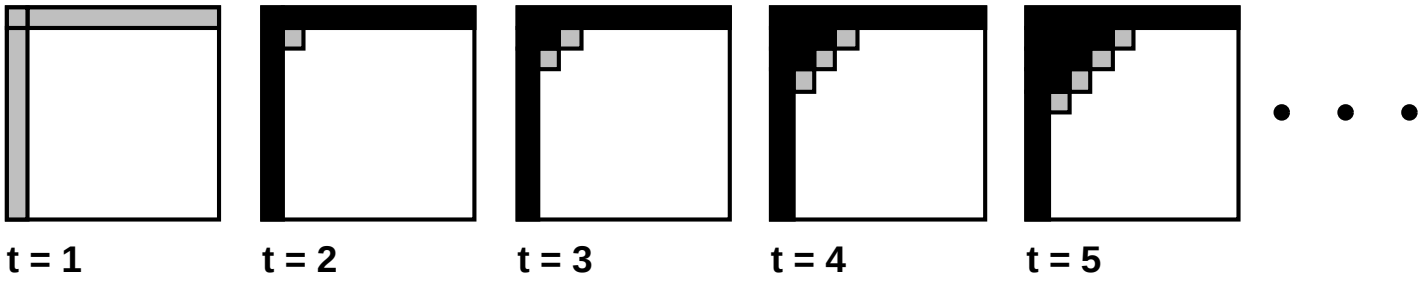


- **Left, top boundary have known values.**
- **Interior points contain function of left, top neighbors**

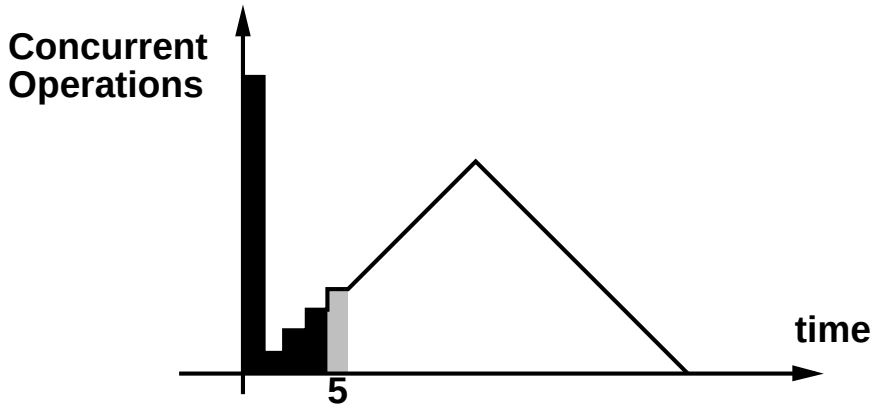
Solution in Id:

```
{ A = I_matrix (1,N),(1,N);
  A[1,1] = 1;
  {for i <- 2 to N do
     A[i,1] = 1;
     A[1,i] = 1;

     {for j <- 2 to N do
        A[i,j] = A[i-1,j]+ A[i,j-1]}}
In
  A }
```

Parallelism in Wavefront



Parallelism profile:



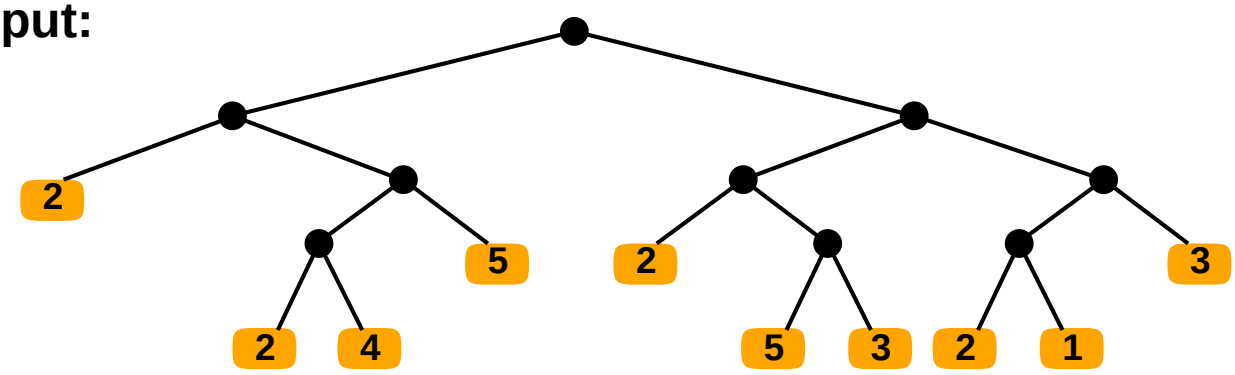
This degree of parallelism is difficult to express and schedule explicitly.

More complex variations :

- **Irregular shape**
- **Non−uniform density**
- **3D, instead of 2D**

Histogram of leaves of a tree

Input:



Output:

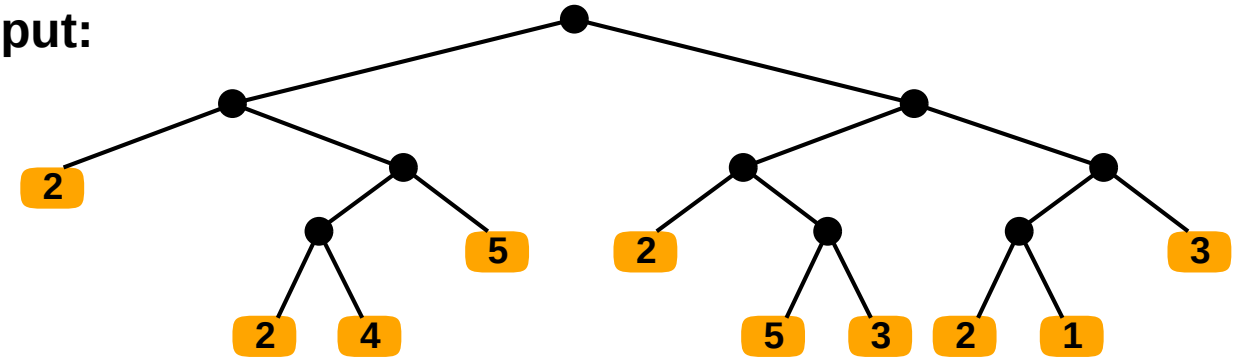
1	4	2	1	2

Solution using pure functional programming style:

```
type tree = Leaf int | Node tree tree ;


def incr A i = {array (1,5)
                 | [i] = A[i]+1
                 | [j] = A[j]  || j <- 1 to 5 where j <> i};



def traverse (Leaf i)   H = incr H i

 |  traverse (Node L R) H = traverse L (traverse R H);



def histogram T = traverse T {array (1,5)
                               | [j] = 0 || j <- 1 to 5};
```

Histogram of leaves of a tree

Input:



Output:

1	4	2	1	2

Id program using M−structures:

```
type tree = Leaf int | Node tree tree ;

def histogram T = {  H  =  {M_array (1,5) of
                                  [j] = 0 || j <- 1 to 5};
                     traverse T H
                     ---
                   In
                     H } ;

def traverse (Leaf i)   H =  { H![i] = H![i] + 1 }
 |  traverse (Node L R) H =  { traverse L H;
                               traverse R H } ;
```

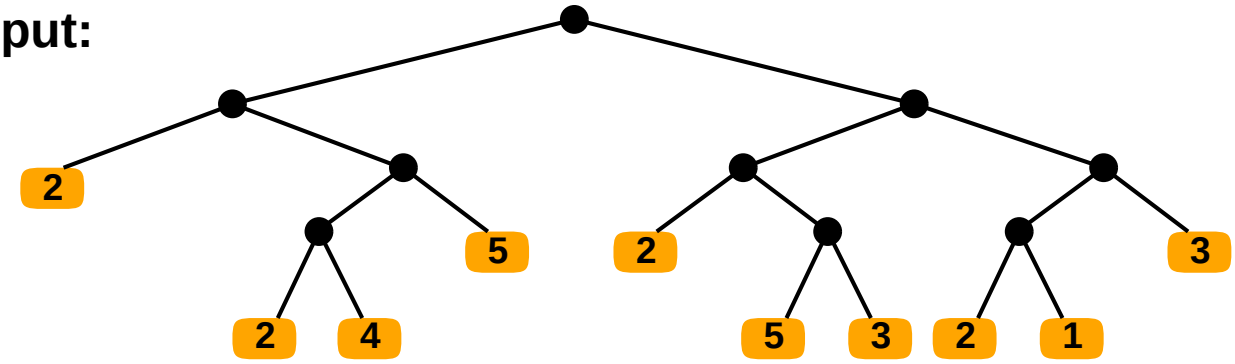
Sequentialize

Wait till FULL, Read, Set EMPTY

Write, Set FULL

Fringe of a tree

Input:

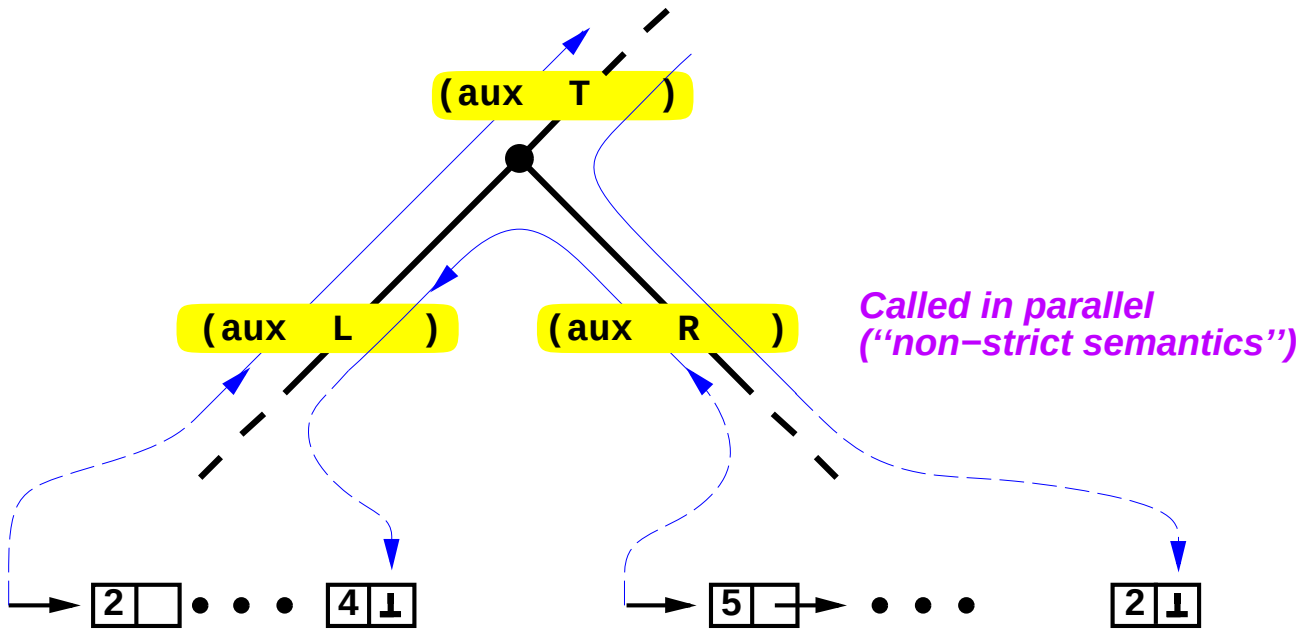


Output:


```
type tree = Leaf int | Node tree tree ;

def fringe T = aux T Nil ;

def aux (Leaf j)   list    = (j : list)
 |  aux (Node L R) list    =  aux  L  (aux  R  list) ;
```

Called
in parallel
("non−strict semantics")

Fringe of a tree



"⊥" is an "empty" location

- **Strict, call−by−value semantics permits *no* parallelism at all (sequential traversal)**
- **Eager, non−strict semantics permits *exponential* parallelism**

"Look and Feel" of Id

To first order, programming in Id is like programming in Lisp / Scheme / ML / Haskell :

- **"Mostly functional" style**
- **Higher−order functions**
- **User−defined types:**
  - **Recursive types**
  - **Polymorphism (Milner−style)**
  - **Static type checking, type inference**
- **Dynamically allocated objects**
  - **Automatic storage management**

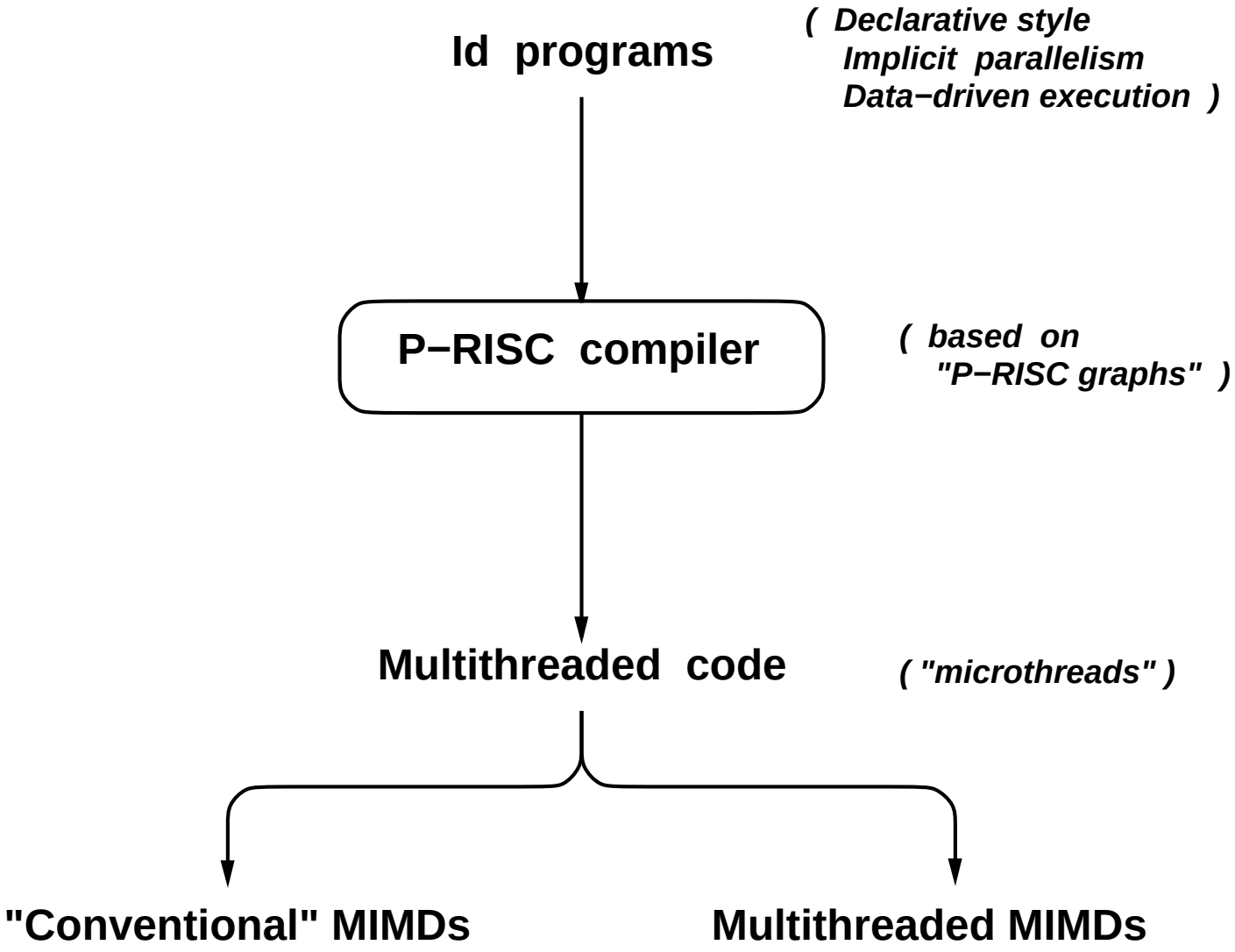
The principal (and pervasive) differences:

(w.r.t. Lisp/ Scheme/ ML/ Haskell)

- ***Parallel semantics***
- ***Fine grain synchronizing data structures***

Implementing Id

P−RISC compiler (Nikhil, DEC CRL):



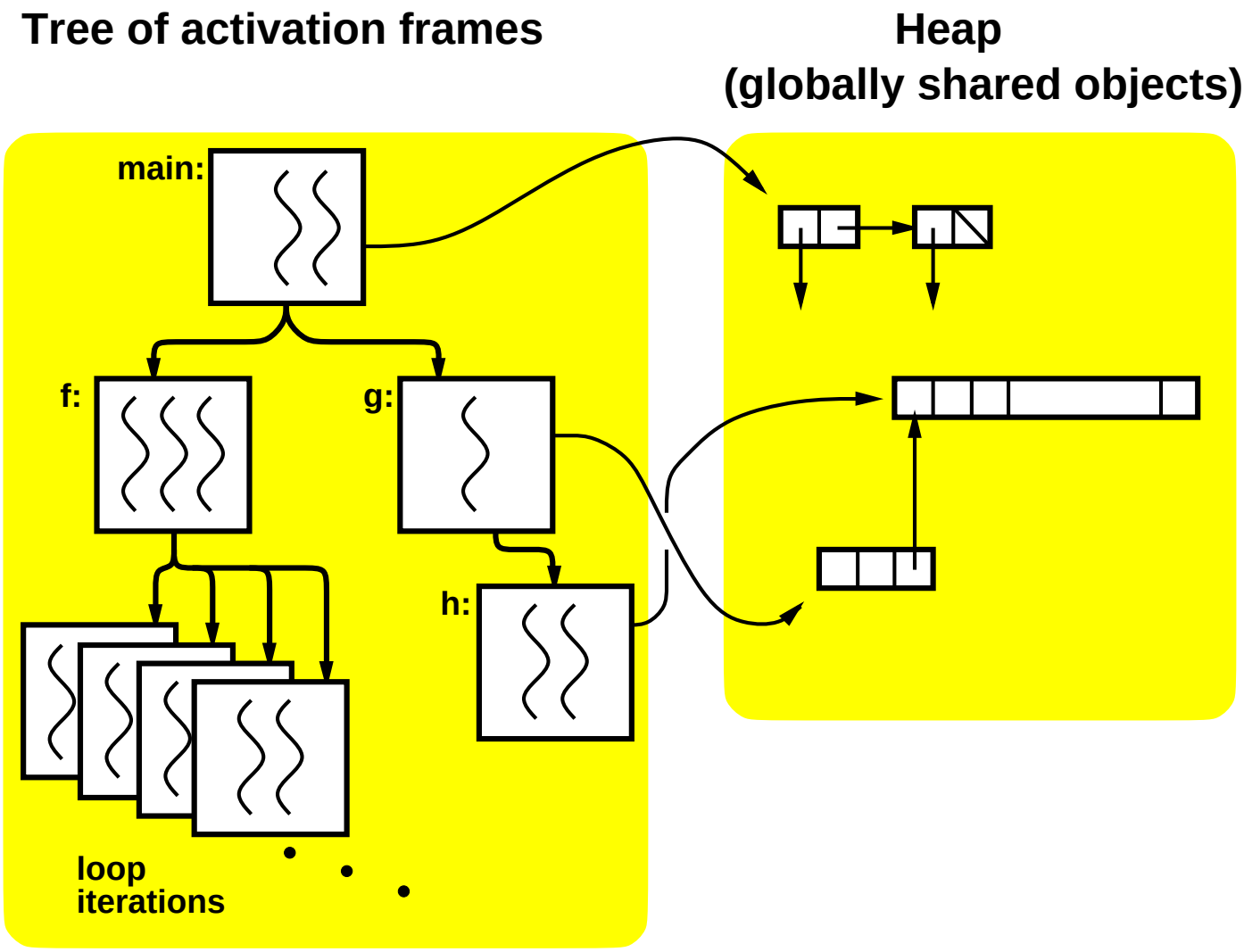
Other Id compilers:

- **MIT Id Compiler: Targeted to TTDA first, now Monsoon**
- **Berkeley TAM Compiler: similar to P−RISC compiler**
- **Sandia Id Compiler: targeted to Sandia Epsilon**

P−RISC Graphs

- **Structured control flow graphs**
- **Parallelism**
  - **Program represented as collection of "microthreads"**
  - **Explicit instructions for forking and joining microthreads**
  - **Microthreads never suspend; all potentially long−latency operations (communication, synchronization) are split−− one thread initiates the operation, another thread continues with the result.**
- **Locality model**
  - **Explicit movement of data across memory heirarchy**
    - **Registers**
    - **Local memory**
    - **Non−local memory**

P−RISC: Runtime Model



Frame: ■ Local to a node

Thread: ■ Local to a frame

Heap objects: ■ May span nodes

P–RISC Runtime Structures: Locality

- **Each frame resides entirely within single node (heap objects may span node boundaries)**
- **Access from a thread to current frame:**
  - **Always local (usual loads and stores)**
- **Access to other frames, and to heap objects:**
  - **Potentially non–local (long latency)**
  - **Potentially involves synchronization**
  - **Split into two threads:**
    - **One thread initiates the action**
    - **Another thread continues with the result**

P–RISC Linkage Instructions

Used for:

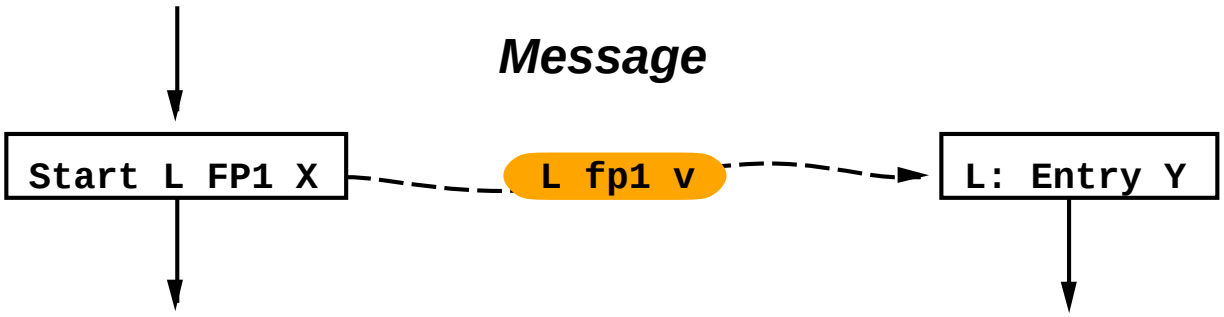
- **Initiating a thread in a different frame**

  (procedure calls and returns, from one loop iteration to the next, etc.)

- **Transmitting values to different frames**

  (arguments, results)

Code:

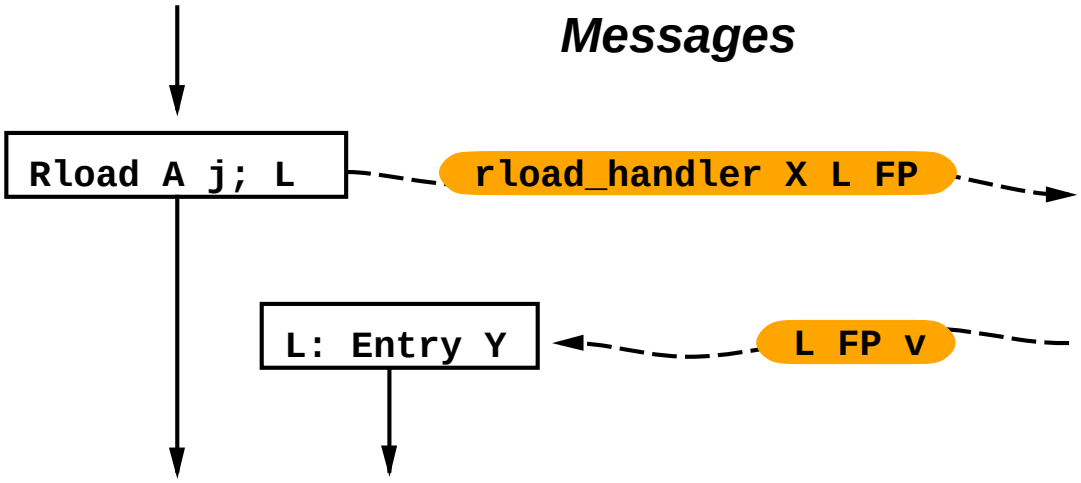


Frames:

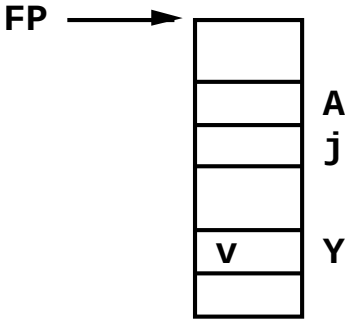
P–RISC Heap Access Instructions

Example: `A[j]`

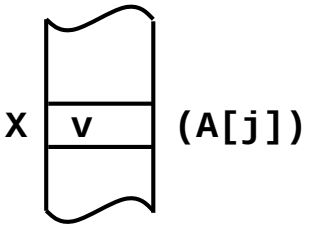
Code:



Frame on node i:

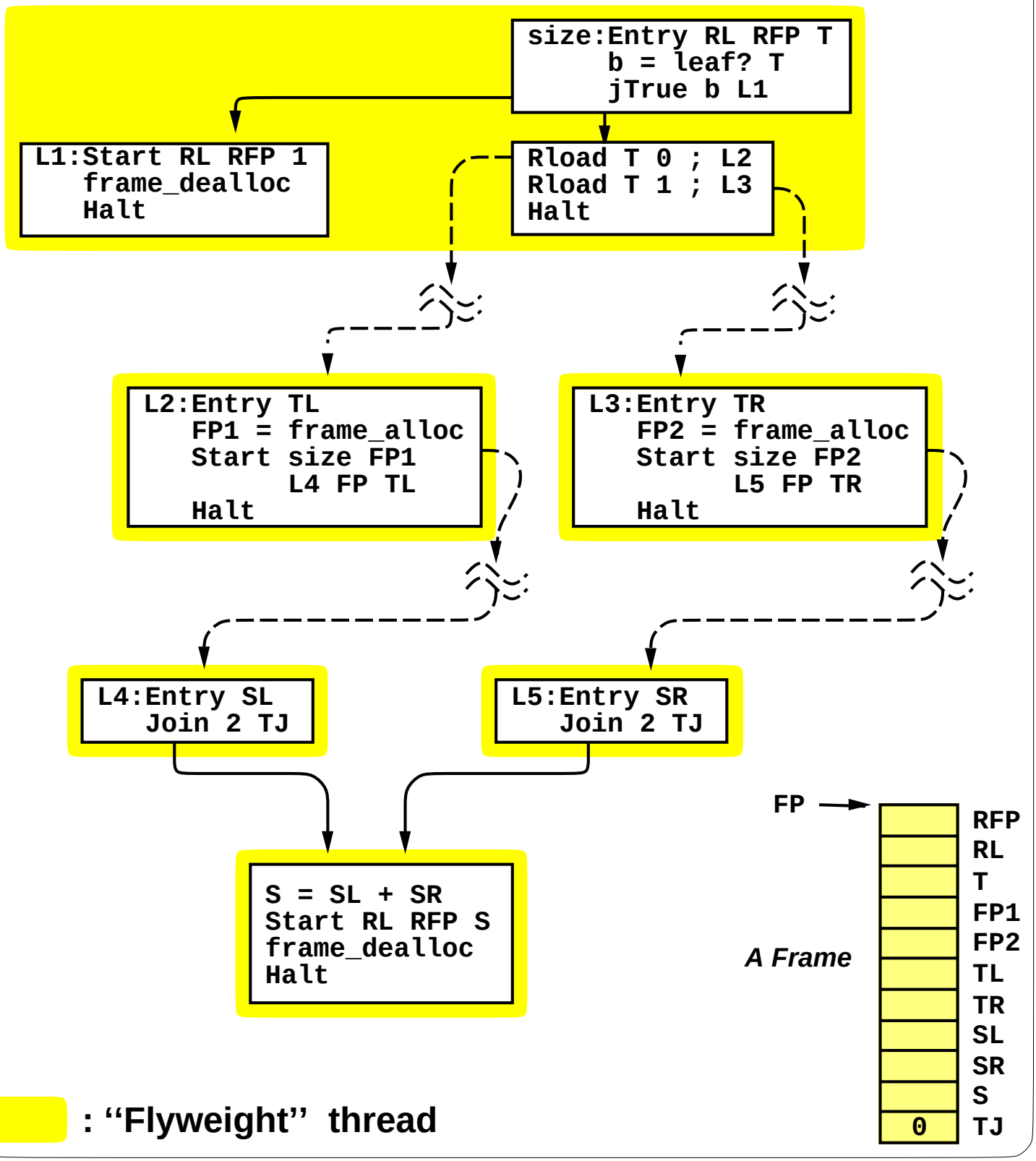


Heap location on node j



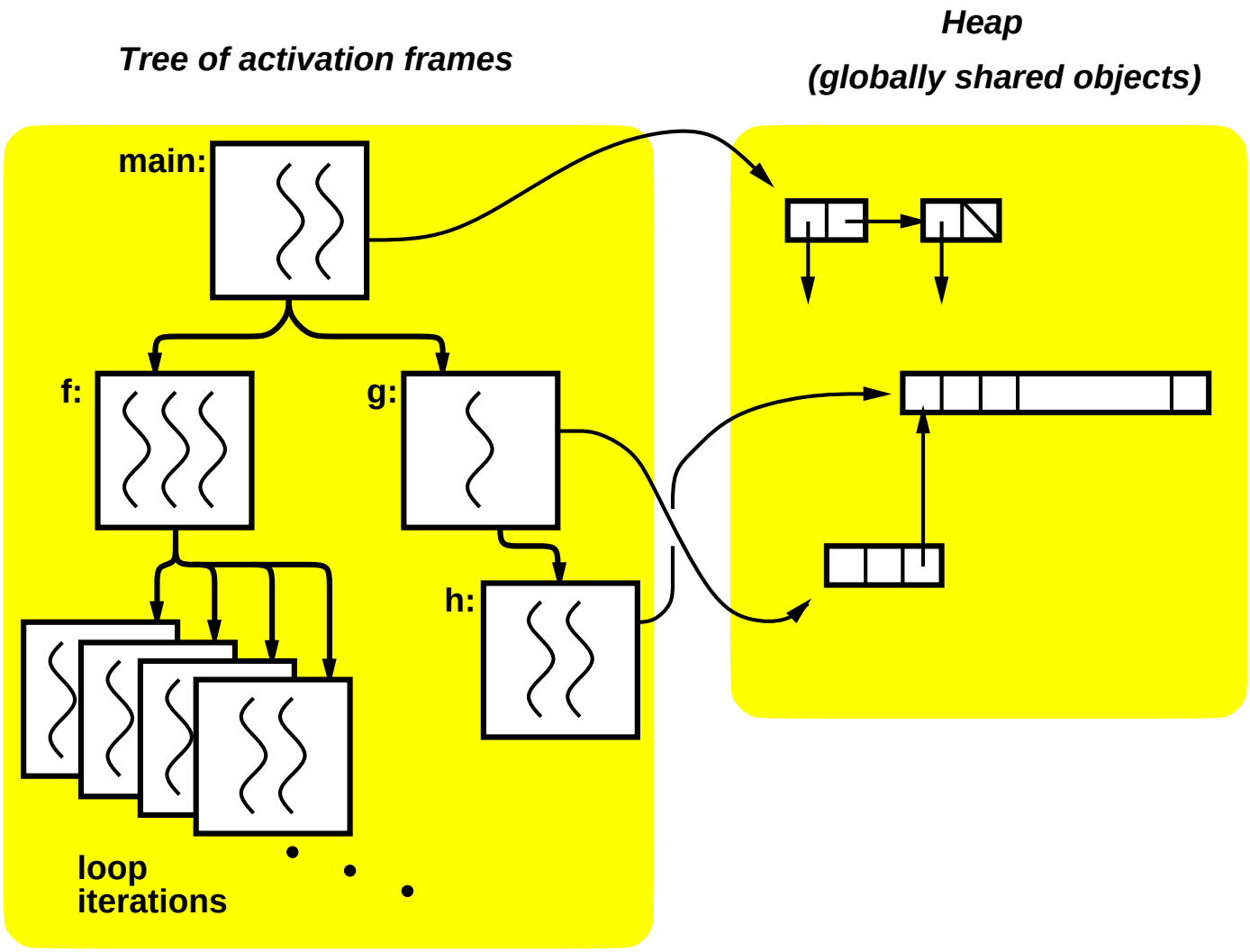
An Example Program and P–RISC Graph

```
def size T = if (leaf? T) then 1
             else size(T.left) + size(T.right)
```



P−RISC Compiler: Runtime System

Directly implements programming model:



Thread scheduling

- In−line code, entirely (no OS involvement)
  Feasible because threads never suspend
- Attempt to execute threads in one frame before switching to another frame, for better locality

CST (Concurrent Smalltalk)

- **MIT J−Machine project**
- **Example program: counting leaves of a binary tree (from [Horwat, Chien and Dally, 1989])**
  - **Class declarations:**

```
(class  node   (object)   left  right  tree−node?)
```

class name *superclass* *instance variables*

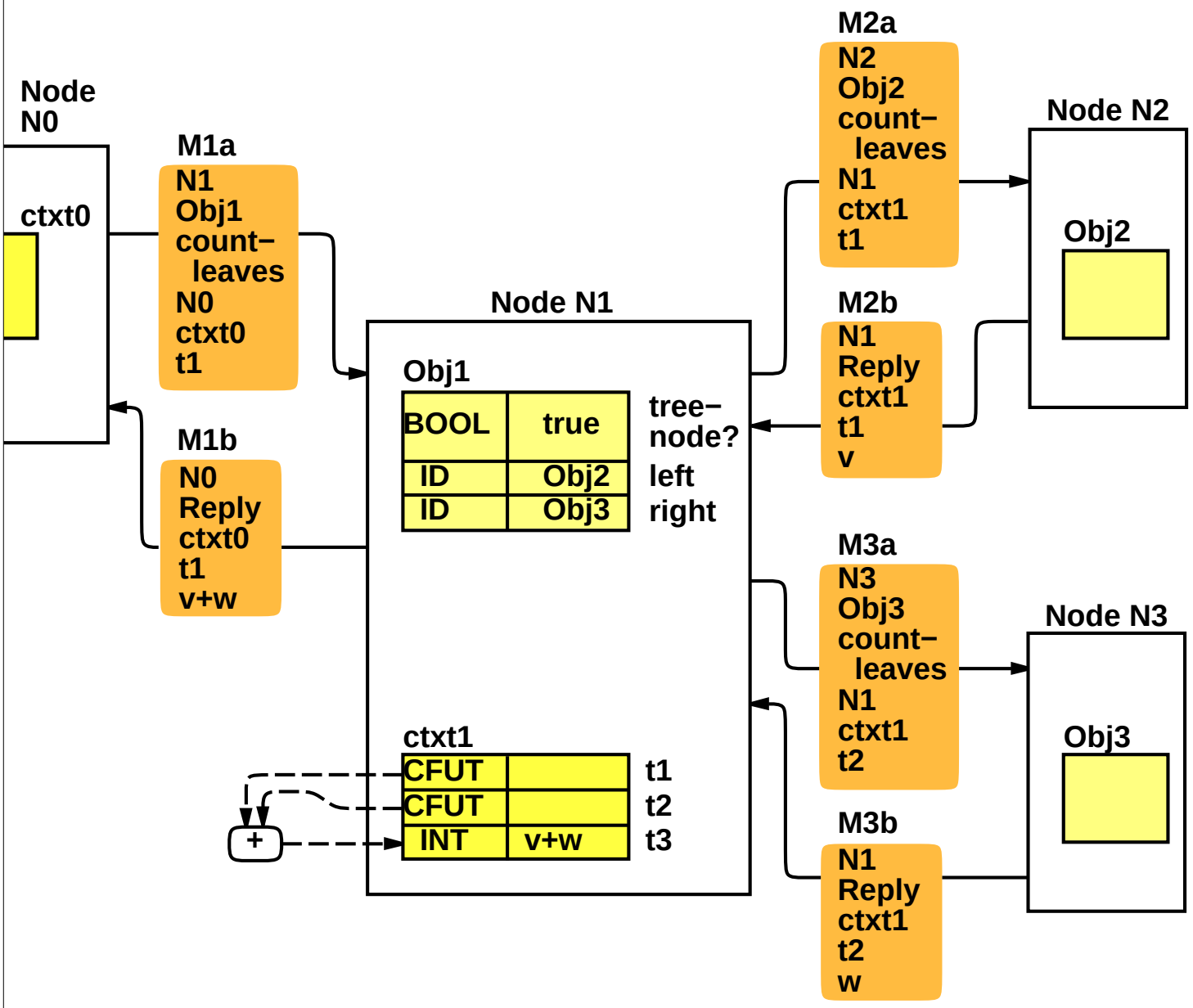
  - **Method declarations:**

class name *method name* *args* *local vars*

```
(method   node  count−elements    ()      ()
   (if  tree−node?    (+   (count−elements  left)
                           (count−elements  right))
                      1))
```

body

CST (Concurrent Smalltalk) Implementation



+ operation traps on attempt to read slot with CFUT type tag; ctxt1 is enqueued on CFUT slot; awakened by Reply message for that slot

ctxt1 allocated on arrival of M1a

M2a, M3a sent in parallel

M2b, M3b arrive in any order

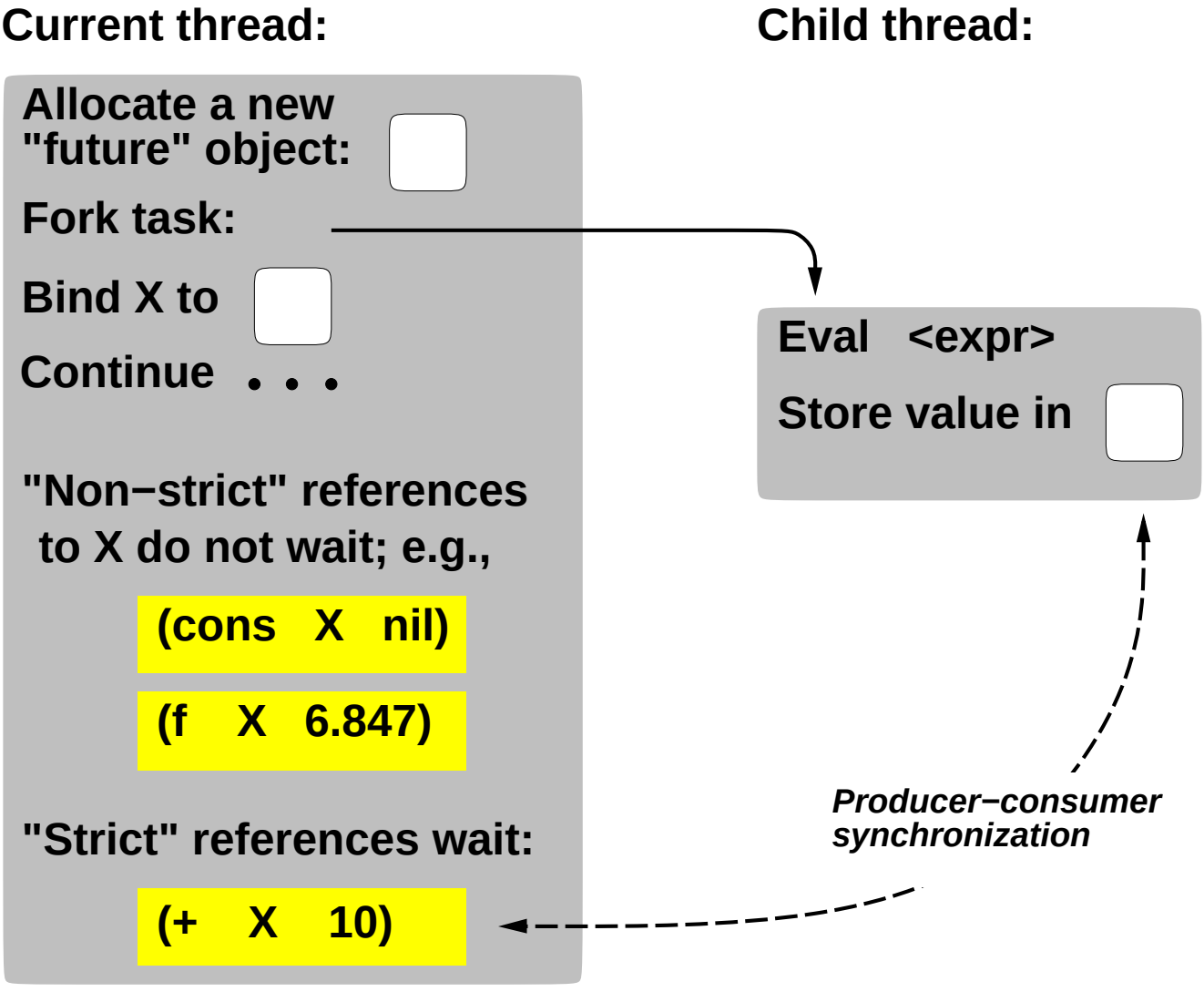
Multithreaded Architectures

Extensions to Existing Languages

Mul−T FUTUREs

- **Roots:**
  - **Lisp => Scheme (MIT) => T (Yale) (all sequential)**
  - **Multilisp at MIT: Scheme + FUTURE [Halstead@MIT]**
  - **Mul−T: T + FUTURE**
  - **Principal language for Alewife**
- **Suppose we have:**

```
(let   ((X  (future   <expr>))
    . . .
```



Mul−T FUTUREs

- Communication between parent and child tasks:
  - Through the "future" object (implicit synchronization)
  - Free variables in <expr> (in child task) refer to locations in parent task.

    Accesses to these variables are unsynchronized. However, Lisp style generally discourages updating variables (set!)

  - Communication also occurs through synchronizing data structures (in Alewife Mul−T)

- Example program: counting leaves of a binary tree

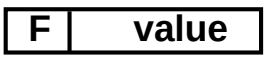
```
(define (count−leaves tree)
   (if (leaf? tree)
       1
       (let ( (X (FUTURE (count−leaves (left tree))))
              (Y         (count−leaves (right tree))))
         (+ X Y))))
```

  - Note: parallelism can grow exponentially

FUTUREs: Implementation

- **Exactly the "Synchronizing Loads" problem!**
- **Use FULL/EMPTY bits**
- **Simple (inefficient) implementation (HEP/Tera/Alewife):**
  - **"Future" object: reference to an EMPTY word**

  - **Non−strict references pass around address**
  - **Strict references LOAD from the address**
  - **LOAD while EMPTY fails, retry until FULL**
  - **Child task writes value, marks it FULL**

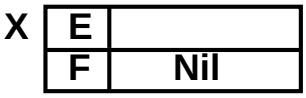


  - **Problem: busy−waiting, can waste resources**

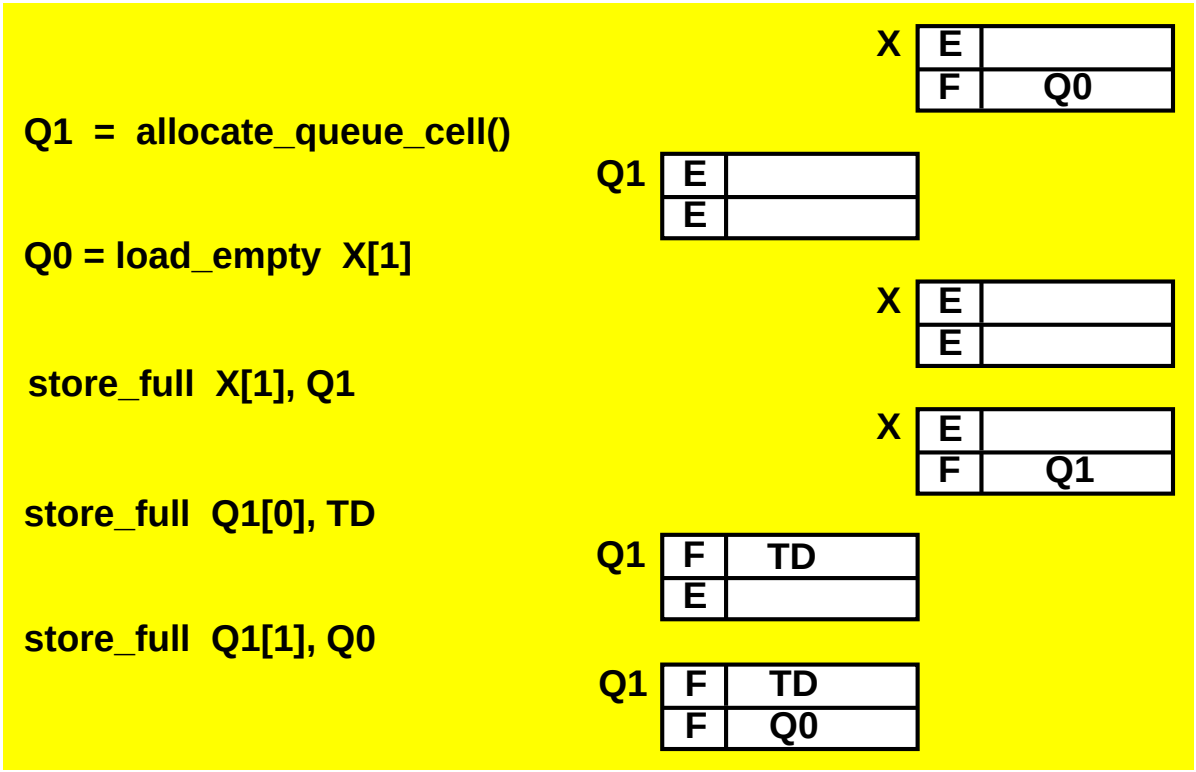
FUTUREs: Implementation

- **Better implementation (HEP/Tera/Alewife):**
  - **"Future" object: reference to pair of words, value cell, thread queue:**

    (In Tera: combine into 1 word, using a Trap bit)



  - **LOAD X[0] on EMPTY fails, may retry a few times, finally traps. Trap handler saves thread state in a thread descriptor TD, enqueues it on X[1]:**



  - **Child task writes value into X[0], marks it FULL, resumes all threads in X[1] queue**
  - **Busy waiting on X[1] acceptable, because locked for very short time**

FUTUREs: Implementation

- **DASH, KSR−1 (no FULL/EMPTY bits): implementation is "heavier":**
  - **"Future" object: variable + locked semaphore**
  - **Producer writes value, unlocks semaphore**
  - **Each consumer attempts to lock semaphore, read, unlock semaphore**

    (semaphore implementation can take care of blocking vs. busy−waiting, etc.)

    Locking/unlocking cost paid even after FUTURE has been "determined"

    (whereas in HEP/TERA/Alewife, once FUTURE object is FULL, subsequent accesses are as fast as ordinary LOADs)

FUTUREs: Implementation

- **TTDA, Manchester Dataflow, Sigma−1, Monsoon and Hybrid: implementation of FUTURE is easiest:**
  - **FUTUREs: single producer, multiple consumer objects**
  - **Corresponds exactly to I−structure semantics**
  - **I−structures supported directly in hardware ("structure−memory", "I−structure memory" units)**
- **EM−4, *T, J−Machine: implementation of FUTURE achieved by implementing I−structure semantics in message handlers at node N2**
  - ***T: have to interpret FULL/EMPTY bits**
  - **EM−4: FULL/EMPTY bits on all locations**
  - **J−Machine: all words have type tag bits. FUT type tag is used for FUTURES (like EMPTY)**

FUTURE Implementations Communication Issues

Suppose consumer on node N1 attempts to read X, an undetermined FUTURE residing on node N2

DASH, KSR−1, Alewife, HEP, Tera: can involve multiple remote communications

- **HEP, Tera (N2 is a memory node):**
  - **Reply to N1 contains EMPTY status**
  - **N1 may retry a couple of times**
  - **Finally, N1 implements enqueuing of waiting thread, at X on N2**
- **Alewife, KSR−1:**
  - **Reply to N1 contains X, with EMPTY status**
  - **X is now cached at N1, so retrying and and enqueuing do not involve communication**
  - **However, if producer is on node N3, need more communication to invalidate X on N1, copy X to N3 and, later, copy X back to N1**
- **DASH:**
  - **Queue−based locks avoid multiple communications**
  - **However, N1 is stuck until X is unlocked**

FUTURE Implementations Communication Issues

Suppose a consumer on node N1 attempts to read X, an undetermined FUTURE residing on node N2

TTDA, Manchester Dataflow, Sigma−1, Monsoon, Hybrid, EM−4, *T and J−Machine:

Exactly one request message, one response message

Reason:

- **FULL/EMPTY testing performed at N2, by N2**
- **Request message carries a**

  complete virtual continuation

  (rather than a physical continuation), which is sufficient for:
  - **N2 to implement queueing**
  - **N1 to deschedule the requesting thread**

  without further interaction with each other

Other forms of FUTURE

Basic idea behind Multilisp's FUTURE adopted and adapted in various extensions to conventional imperative languages

- **In Tera C**
- **In Semi−C (Alewife project)**

"The future just ain't what it used to be"

FUTUREs in Semi−C

- MIT Alewife project
- Semi−C: highly influenced by Mul−T, shares much of the implementation
- Future expressions: C−syntax versions of Mul−T FUTURE expressions:

  future (<expr>)

  - <expr>: arbitrary C expression
  - Semantics identical to Mul−T
- Future statements: simply forks

  future { <stmt list> }

  - <stmt list>: an arbitrary C statement list
  - no "future" object

FUTUREs in Tera C

- "future": new type qualifier for variables
- Uses of "future" variables synchronize, implicitly
- New "future" statment:

```
future  <expr>  (  <parameter list> )  {
     <stmt list>
}
```

  - <expr> evaluates to a "future" variable
  - <stmt list>: arbitrary C statement list.

    Should execute `return v` which determines the future, with value v
  - <parameter list>: specifies interpretation of free variables in <stmt list>
    - Make local copy on starting thread, or
    - Refer to variable in parent

    Syntactic sugar, because it can be programmed explicitly

FUTUREs in Tera C

Example (from [Callahan and Smith, 1990])

```
/* Dot product of vectors x, y for  i <= j <= (n−1) */

double dot(double *x, double *y, int i, int n) {
    double sum, right;
    future double left;
    int half, j;

    if (n−i < 10) {      /* small range; do sequentially */
        sum = 0.0;
        for (j = i; j < n; j++) {   sum +=  x[j] * y[j]  }
    }

    else {               /* large range; parallel divconq */
        half  =  (i+n) / 2;
        future  left  (x,  y,  half,  n)  {
            return  dot (x, y, half, n);
        }
        right = dot (x, y, i, half);
        sum =  left  +   right;
    }
    return sum;
}
```

synchronizing access

Fine Grain Synchronized Data Structures

Language constructs for variables and data structures with implicit synchronization on individual elements

- **Single producer, multiple consumers:**
  - **I−structures in Id (arrays, records)**
  - **J−structures in Alewife Mul−T, Semi−C (arrays)**
  - **FUTUREs in Mul−T, Semi−C (vars)**
  - **FUTUREs in Tera C (vars, arrays, records)**
- **Multiple producers, multiple consumers:**
  - **M−structures in Id (arrays, records)**
  - **L−structures in Alewife Mul−T, Semi−C (arrays)**
  - **"Synchronized variables" in Denelcor HEP Fortran, Tera Fortran, Tera C (vars, arrays, records)**

The Denelcor HEP Software: Fortran extensions

Creating and destroying threads:

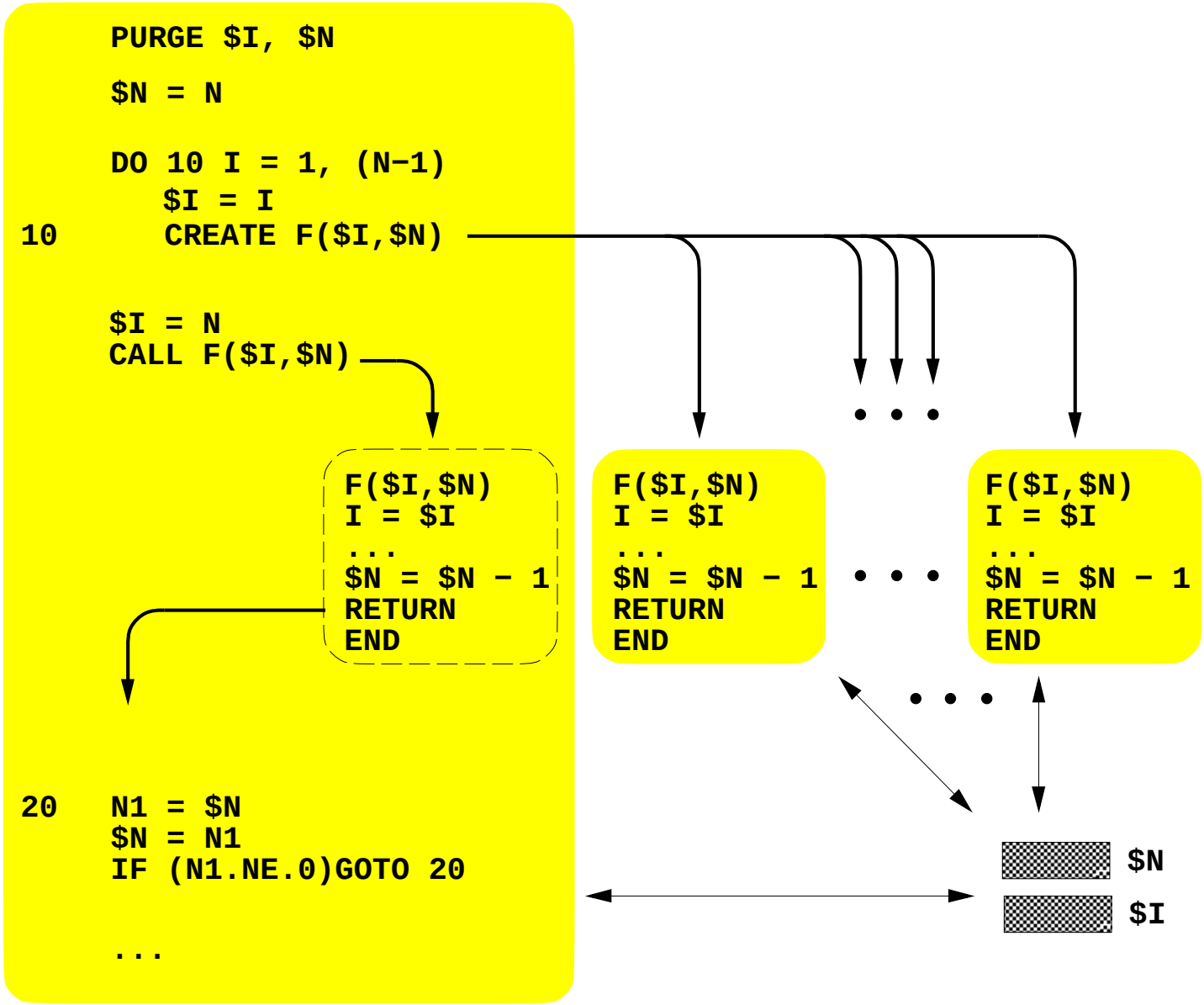
- **CREATE: like CALL, except forks the subroutine**
- **RESUME: in a CALLed subroutine, allows parent to continue (so, this is also a fork)**
- **RETURN: In a CALLed subroutine, returns as normal**
  In a CREATEed subroutine, kills the thread

Synchronization:

- **"Asynchronous variables" (names begin with "$")**
  - **`"... = $X"`** **Wait for full, read, set empty**
  - **`"$X = ..."`** **Wait for empty, write, set full**
  - **`"PURGE($X)"`** **Set empty**

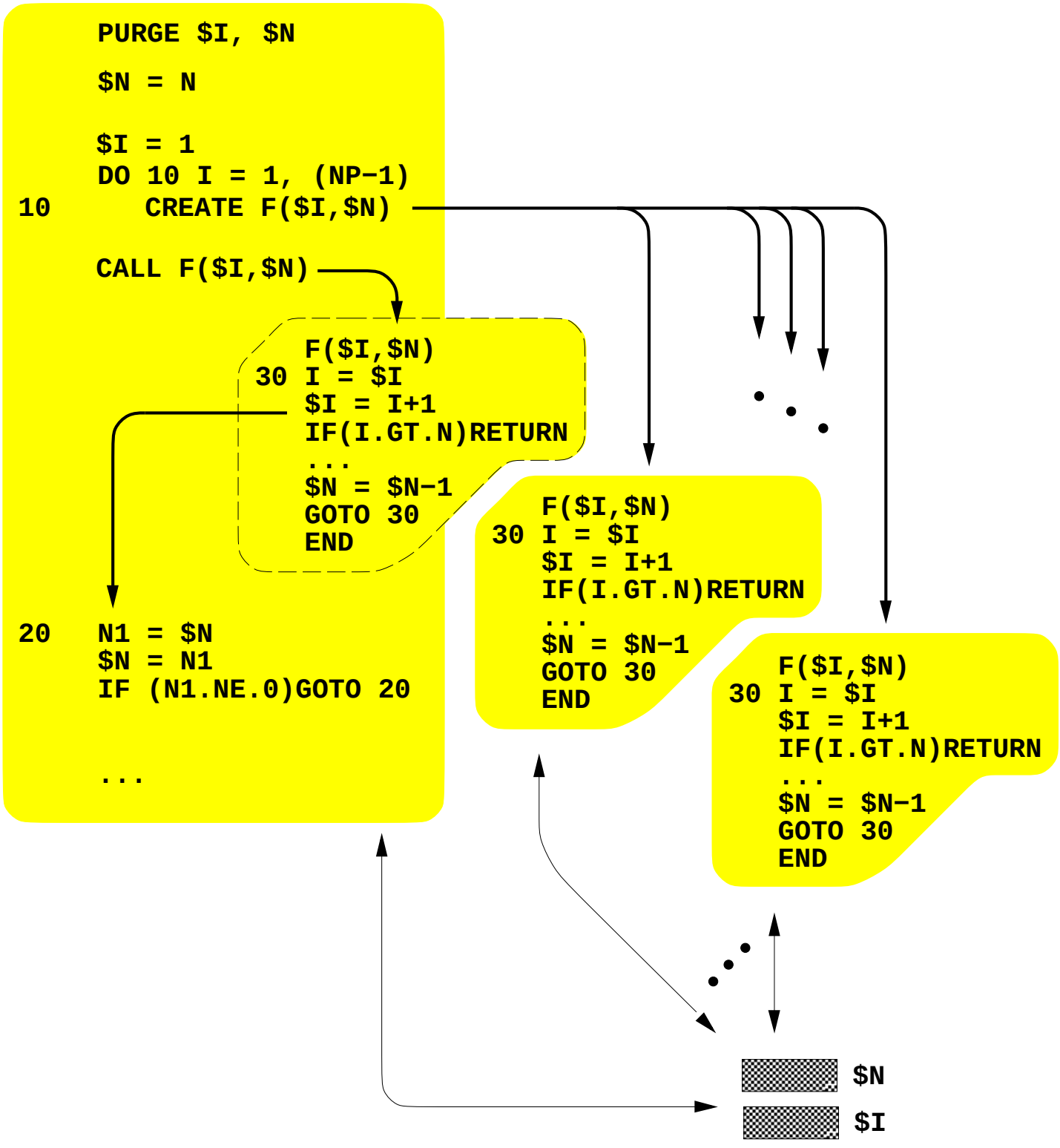
The Denelcor HEP Software: Fortran example

Parallel loop (DOALL):



The Denelcor HEP Software: Fortran example

Self−scheduling Parallel loop (DOALL):



Resource Management Granularity

Mul−T FUTURE implementation: ''lazy task creation'' [Mohr, Kranz, Halstead 1991]

- **When (FUTURE <expr>) is evaluated in thread T1, T1 evaluates <expr>, leaving continuation C on stack, enqueuing C in pool of available work**
- **Idle processors free to steal C, execute it**
- **If T1 completes <expr> before any other processor steals C, it executes C by itself**
  - **In this case, full overhead of forking a thread not paid (e.g., allocating a new stack)**
- **So,**
  - **If evaluation of <expr> is very fast, and/or there are no idle processors,**
  - **execution is sequential, as if <expr> was in−line (not inside FUTURE)**
- **Thus, self−regulating granularity:**
  - **Fine grain when resources available**
  - **Large grain otherwise**

Resource Management
Load Balancing: Interleaving

- **HEP, Sigma−1, Monsoon:**
  - **Global memory units have interleaved addresses**
- **KSR−1:**
  - **Slotted ring SE:0 can be interleaved 2 ways**
  - **Slotted ring SE:1 can be interleaved 1, 2 or 4 ways**
- **Tera, MIT/Motorola *T:**
  - **Programmable mapping tables allow sophisticated interleaving on a per−segment basis**
- **Compiler support:**
  - **Many compilers, especially for machines with caches (DASH/KSR−1/Alewife) pay attention to partitioning and "blocking" data, to improve locality and to avoid "false sharing"**

Resource Management
Load Balancing: Object Migration

- DASH, KSR−1, Alewife
  - "Automatic" due to distributed cacheing
- J−Machine: Supported through management of
  - Global "logical" object identifiers
  - Object directory, which is a kind of software−managed distributed cache manager
  - Hardware support for associative directory lookups
- Tera: Supported through
  - Implicit, invisible forwarding in memory units
- IBM Empire:
  - Hardware support for implicit, invisible forwarding
- Software solutions:
  - Object migration can be folded into the garbage collector, which moves objects anyway

Resource Management
Load Balancing: Process Migration

- Mostly software solutions

- Hardware support in Tera:
  - Elaborate set of hardware counters that measure resource usage
  - Can be used by OS to reallocate resources

- Hardware support in EM−4:
  - Each group of PEs maintains a "load factor buffer", computed from:
    - Number of messages in the input buffers
    - Number of free frames
    - Size of available heap memory
  - To invoke a function, a PE first sends a special message on a circuit through the network
  - The message reads the load factor buffers on the way, returning with the id of the PE group with minimum load
  - The PE invokes the function on that group

Resource Management Multiprogramming

- In principle, no special new problems are introduced by multithreading, w.r.t. multiprogramming
- Many of the research dataflow machines do not have (much) support for inter−program protection:
  - Manchester dataflow, Sigma−1, Monsoon, EM−4, J−Machine
  - The research focus has been on running a single program rapidly, on the entire machine
- Other machines (DASH, KSR−1, Alewife, HEP, Tera, *T) provide the usual inter−program protection mechanisms.
  - They may be time− or space−shared, as usual
  - The Tera system is the most ambitious in matching allocation of parallelism resources to programs based on dynamically changing parallel behavior

The Software Story
Final Comments

Ultimately, software is the raison d'etre of multithreading.

Multithreaded architectures are very exciting because, though implemented with distributed memories for scalability, they provide a (viable)

shared memory programming model

- **This should make it significantly easier to program parallel machines**
- **Hopefully, *general purpose programming* will now be able to benefit from parallel processing**
- **Hopefully, this will make parallel machines usable for the non−specialist**

We hope these efforts succeed,

Because a MIMD is a terrible thing to waste!

This page intentionally blank