Programming Models

Languages and Technologies

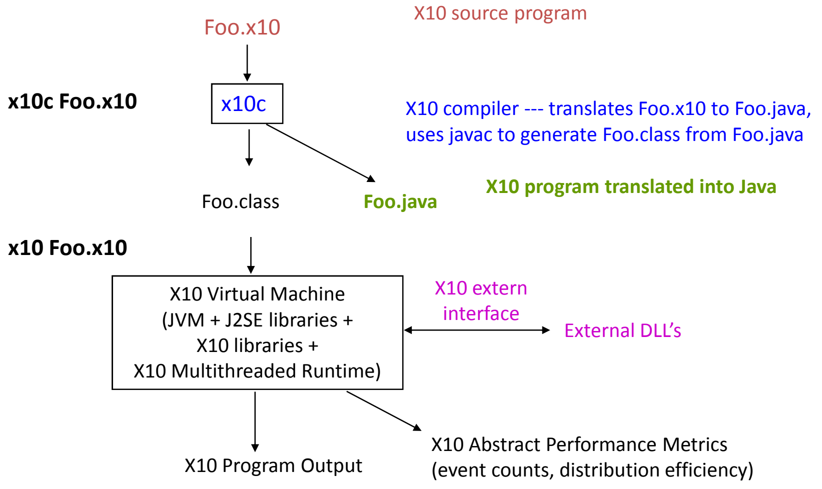
Contents

- Advanced languages for multicore systems
  - Programming languages for Non-Uniform Cluster Computers
    - IBM X10
  - Programming languages for embedded systems
    - Models of computations
    - Streamit

What is X10?

- X10 is a new experimental language developed in the IBM PERCS project as part of the DARPA program on High Productivity Computing Systems (HPCS)
- X10's goal is to provide a new parallel programming model and its embodiment in a high level language that:
  1. is more productive than current models,
  2. can support higher levels of abstraction better than current models, and
  3. can exploit the multiple levels of parallelism and nonuniform data access that are critical for obtaining scalable performance in current and future HPC systems,

Current X10 Environment



Future X10 Environment

TOOLS (e.g. Eclipse)

Very High Level Languages (VHLL's),

Domain Specific Languages (DSL's)

Implicit parallelism,

Implicit data distributions

X10 Libraries

X10 High Level Language

X10 places --- abstraction of explicit control & data distribution

X10 Deployment

Mapping of places to nodes in HPC Parallel Environment

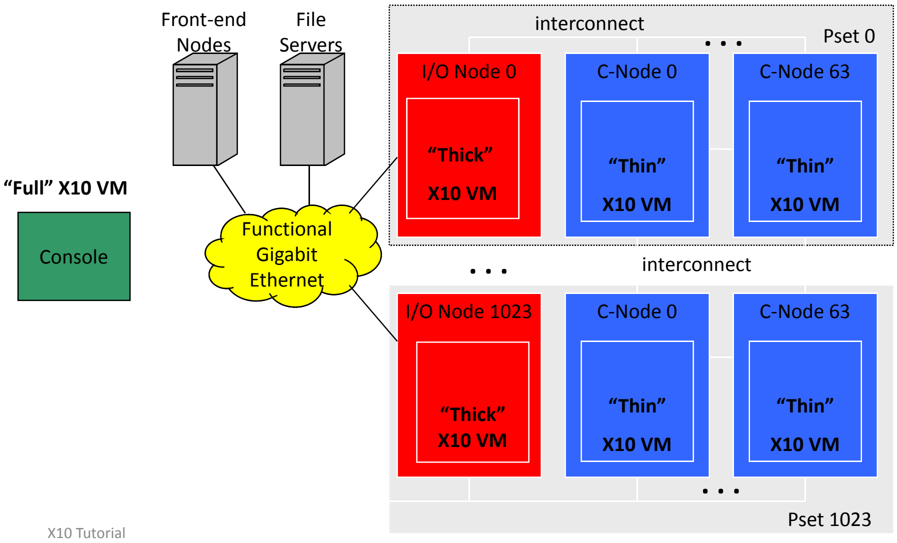
HPC Runtime Environment

(Parallel Environment, MPI, LAPI, …)

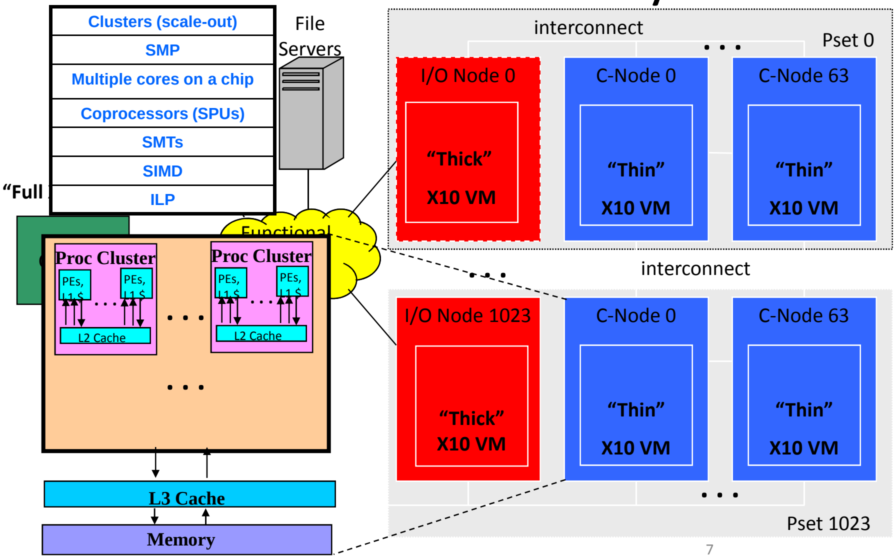
Primitive constructs for parallelism, communication, and synchronization

Operating system

Future X10 Environment: Targeting Scalable HPC Parallel Systems



Future X10 Environment: Targeting Scalable HPC Parallel Systems



X10 vs. Java

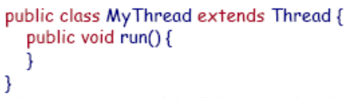
- X10 is an extended subset of Java
  - Notable features removed from Java
    - Concurrency --- threads, synchronized, etc.
    - Java arrays – replaced by X10 arrays
  - Notable features added to Java
    - Concurrency – async, finish, atomic, future, force, foreach, ateach, clocks
    - Distribution --- points, distributions
    - X10 arrays --- multidimensional distributed arrays, array reductions, array initializers,
    - Serial constructs --- nullable, const, extern, value types
- X10 supports both OO and non-OO programming paradigms

X10 vs. Java

- X10 developers think Java will be a suitable platform for High performance computing by 2010
- X10 addresses the main limitation of Java for NUCC systems: the notion of *single uniform heap*
  - Introduction of the *Partitioned Global Address Space (PGAS)*
  - Programmers control which objects and activities are co-located using the concept of *places*
- X10 addresses another limitation of Java: heavyweight mechanism for managing threads and messages
  - Introduction of asynchronous activities

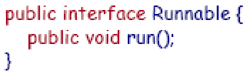
Java: Concurrency

- Java has a predefined class java.lang.Thread which provides the mechanism by which threads are created



```
public class MyThread extends Thread {
   public void run() {
   }
}
```

- However to avoid all threads having to be subtypes of Thread, Java also provides a standard interface



```
public interface Runnable {
   public void run();
}
```

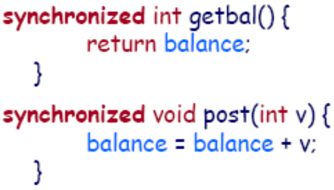
- Hence, any class which wishes to express concurrent execution must implement this interface and provide the run method
- Threads do not begin their execution until the start method in the Thread class is called

Java: Synchronization

- All the interleavings of the threads are NOT acceptable correct programs
- Java provides synchronization mechanism to restrict the interleavings
- Synchronization serves two purposes:
  - **Ensure safety for shared updates**
    - Avoid **race conditions**
  - **Coordinate actions of threads**
    - Parallel computation
    - Event notification

Java: Mutual Exclusion

- Prevent more than one thread from accessing *critical section at a given time*
  - Once a thread is in the critical section, no other thread can enter that critical section until the first thread has left the critical section.
  - No interleavings of threads within the critical section
  - **Serializes access to section**



```
synchronized int getbal() {
        return balance;
  }
synchronized void post(int v) {
        balance = balance + v;
  }
```

Java: Atomicity

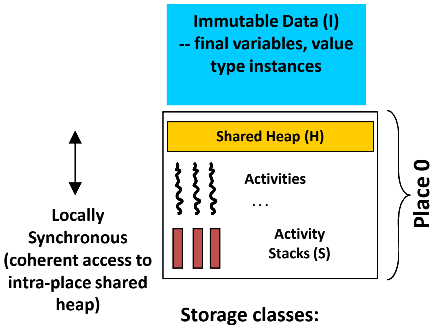
- The synchronized keyword takes as a parameter an object whose lock the system needs to obtain before it can continue

```
synchronized (acc) {
        if (acc.getbal() + val > 0)
                acc.post(val);
        else
                throw new Exception();
        out.print("your balance is " + acc.getbal());
}
```

X10: Places

- A collection of resident mutable data objects and activities operating on the data
- The number of places is fixed at compile time
- Places are virtual
  - Mapping to physical locations is done in the deployment step
  - Objects and activities do not migrate across places but places can migrate across physical locations
  - An activity in a place can spawn activities on remote places
- Within a place, activities operate on memory in a *sequentially consistent* fashion
- Inter-place data access follows a weak ordering semantic

X10 Programming Model (Single Place)



Immutable Data (I) -- final variables, value type instances

Shared Heap (H)

Activities

…

Activity Stacks (S)

Place 0

Locally Synchronous (coherent access to intra-place shared heap)

Storage classes:

- **Immutable Data (I)**
- **Shared Heap (H)**
- **Activity Stacks (S)**

- Activity = lightweight thread
  - Main program starts as single activity in Place 0
- Single Place Memory model
  - No coherence constraints needed for I and S storage classes
  - Guaranteed coherence for H storage class --- all writes to same shared location are observed in same order by all activities
  - Largest deployment granularity for a single place is a single SMP

In order to be accessed by different activities variables must be declared as “final”

Basic X10 (Single Place)

Core constructs used for intra-place (shared memory) parallel programming:

- Async = construct used to execute a statement in parallel as a new activity
- Finish = construct used to check for global termination of statement and all the activities that it has created
- Atomic = construct used to coordinate accesses to shared heap by multiple activities
- Future = construct used to evaluate an expression in parallel as a new activity
- Force = construct used to check for termination of future

async statement

- async <stmt>
  - Parent activity creates a new child activity to execute <stmt> in the same place as the parent activity
  - An async statement returns immediately – parent execution proceeds immediately to next statement
  - Any access to parent's local data must be through final variables
    - Similar to data access rules for inner classes in Java
- Example

```
public class TutAsync {
  const boxedInt oddSum=new boxedInt();
  const boxedInt evenSum=new boxedInt();
  public static void main(String[] args) {
    final int n = 100;
    async for (int i=1 ; i<=n ; i+=2 ) oddSum.val += i;
    for (int j=2 ; j<=n ; j+=2 ) evenSum.val += j;
```

Variable n must be declared as final --- its value is passed from parent to child activity

finish statement

- finish <stmt>
  - Execute <stmt> as usual, but wait until all activities spawned (transitively) by <stmt> have terminated before completing the execution of finish S
  - finish traps all exceptions thrown by activities spawned by S, and throws a wrapping exception after S has terminated.
- Example:

```
   . . .
  finish {
    async for (int i=1 ; i<=n ; i+=2 ) oddSum.val += i;
    for (int j=2 ; j<=n ; j+=2 ) evenSum.val += j;
 }
```

Atomic statements & methods

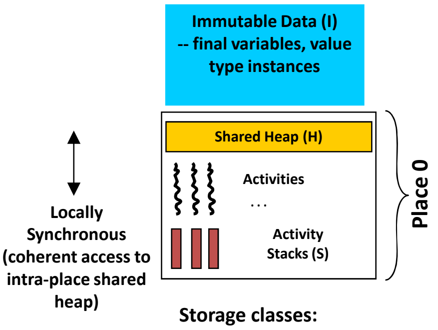
- atomic <stmt>, atomic <method-decl>
- An atomic statement/method is *conceptually* executed in a single step, while other activities are suspended
  - Note: programmer does not manage any locks explicitly
- An atomic section may not include
  - Blocking operations
  - Creation of activities
- Example:

```
finish {
     async for (int i=1 ; i<=n ; i+=2 ) {
         double r = 1.0d / i ; atomic rSum += r;
     }
     for (int j=2 ; j<=n ; j+=2 ) {
         double r = 1.0d / j ; atomic rSum += r;
     }
 }
 System.out.println("rSum = " + rSum);
```

foreach loop (Parallel iteration)

- The X10 foreach loop is similar to the pointwise for loop, except that each iteration executes in parallel as a new asynchronous activity i.e.,
  - "foreach (point p : R) S" is equivalent to "for (point p : R) async S"
- As before, finish can be used to wait for termination of all foreach iterations
  - finish foreach (point[i,j] : [0:M-1,0:N-1]) . . .
- Allowing a single foreach construct to span multiple dimensions makes it convenient to write parallel matrix code that is independent of the underlying rank and region e.g.
  - foreach (point p : A.region) A[p] = f(B[p], C[p], D[p]) ;
- Multiple foreach instances may accesses shared data in the same place ➔ use finish, atomic, force to avoid data races

Limitations of using a Single Place



Immutable Data (I) -- final variables, value type instances

Shared Heap (H)

Activities

. . .

Activity Stacks (S)

Place 0

Locally Synchronous (coherent access to intra-place shared heap)

Storage classes:

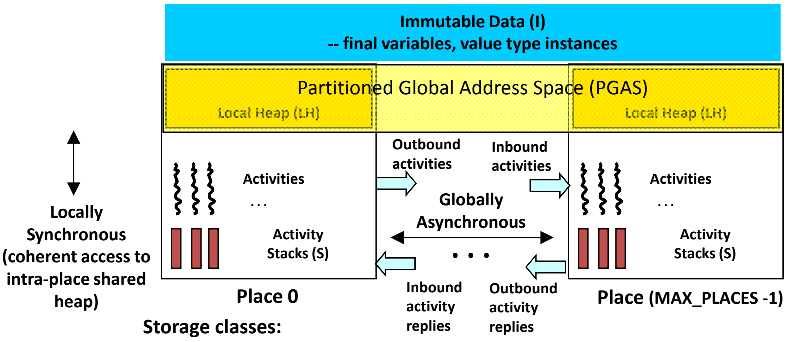
- **Immutable Data (I)**
- **Shared Heap (H)**
- **Activity Stacks (S)**

- Largest deployment granularity for a single place is a single SMP
  - Smallest granularity can be a single CPU or even a single hardware thread
- Single SMP is inadequate for solving problems with large memory and compute requirements
- X10 solution: incorporate multiple places as a core foundation of the X10 programming model

➔ Enable deployment on large-scale clustered machines, with integrated support for intra-place parallelism

---

Scalable X10: using multiple places



Immutable Data (I) -- final variables, value type instances

Partitioned Global Address Space (PGAS)

Local Heap (LH)

Local Heap (LH)

Activities

. . .

Activity Stacks (S)

Outbound activities

Inbound activities

Globally Asynchronous

. . .

Inbound activity replies

Outbound activity replies

Activities

. . .

Activity Stacks (S)

Place 0

Place (MAX_PLACES -1)

Locally Synchronous (coherent access to intra-place shared heap)

Storage classes:

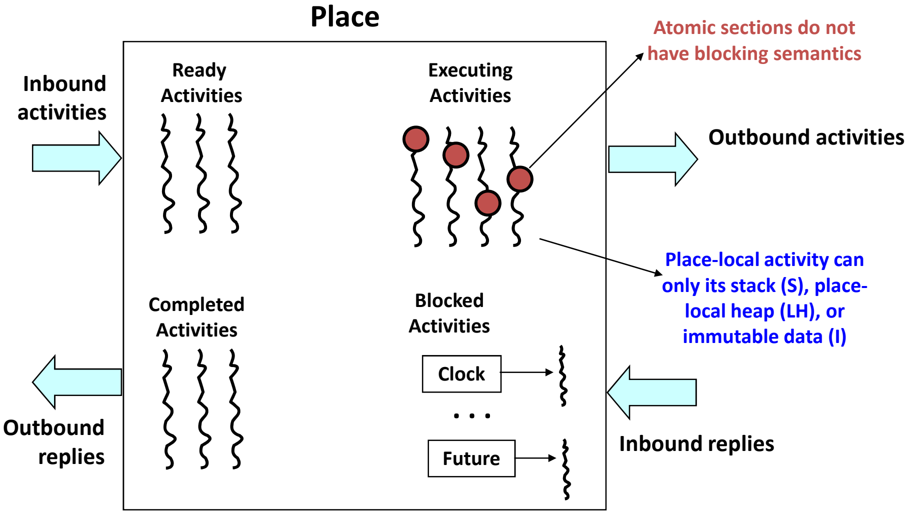
- **Immutable Data (I)**
- **PGAS**
  - **Local Heap (LH)**
  - **Remote Heap (RH)**
- **Activity Stacks (S)**

- Place = collection of activities & objects
  - Activities and data objects do not move after being created
- Scalar object, O -- maps to a single place specified by O.location
- Array object, A – may be local to a place or distributed across multiple places, as specified by A.distribution

Locality Rule

- *Any access to a mutable (shared heap) datum must be performed by an activity located at the place as the datum*
- Inter-place data accesses can only be performed by creating remote activities (with weaker ordering guarantees than intra-place data accesses)

Activity Execution within a Place


Place

Inbound activities

Ready Activities

Executing Activities

Atomic sections do not have blocking semantics

Outbound activities

Place-local activity can only its stack (S), place-local heap (LH), or immutable data (I)

Completed Activities

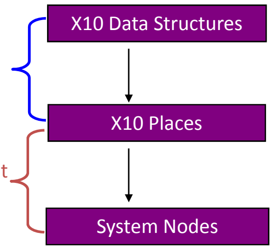
Blocked Activities

Outbound replies

Inbound replies

Place Management

- place.MAX_PLACES = total number of places
  - Default value is 4
  - Can be changed by using the -NUMBER_OF_LOCAL_PLACES option in x10 command
- place.places = Set of all places in an X10 program(see java.lang.Set)
- place.factory.place(i) = place corresponding to index i
- here = place in which current activity is executing
- <place-expr>.toString() returns a string of the form “place(id=99)”
- <place-expr>.id returns the id of the place



X10 language defines mapping from X10 objects to X10 places, and abstract performance metrics on places

Future X10 deployment system will define mapping from X10 places to system nodes; not supported in current implementation

Inter-place communication using async and future

- Question: how to assign A[i] = B[j], when A[i] and and B[j] may be in different places?
- Answer #1 --- use nested async’s!

```
finish async ( B.distribution[j] ) {
  final int bb = B[j];
  async ( A.distribution[i] ) A[i] = bb;
}
```

- Answer #2 --- use future-force and an async!

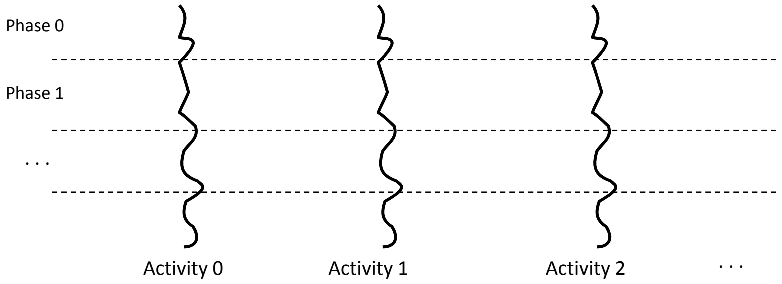
```
final int b = future (B.distribution[j]) { B[j] }.force();
finish async ( A.distribution[i] ) A[i] = b;
```

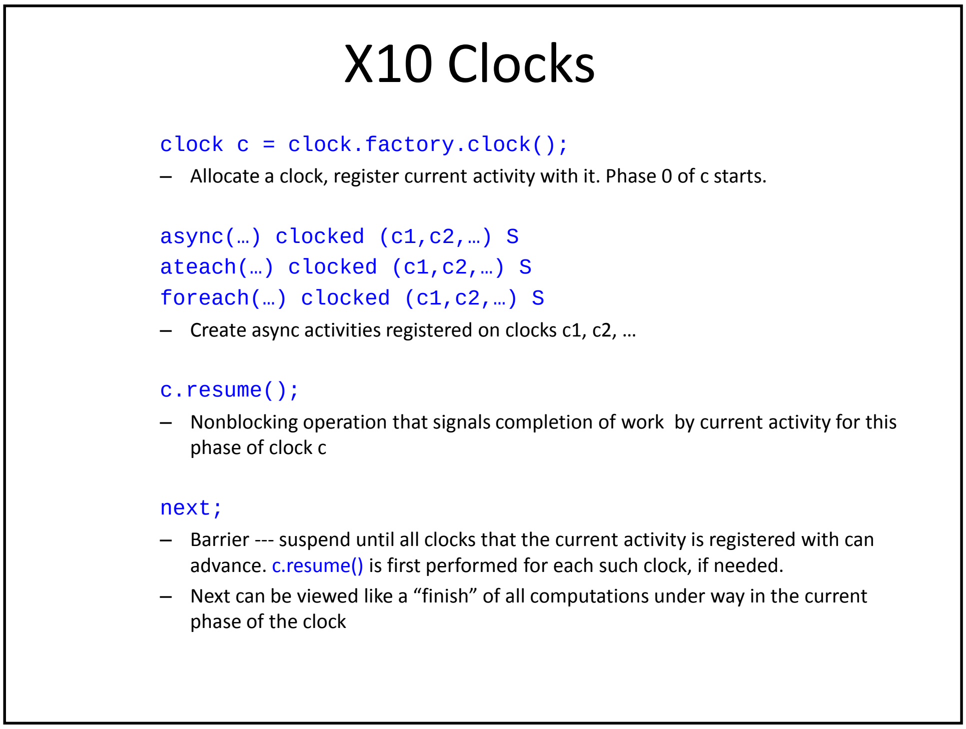
ateach loop (distributed parallel iteration)

- The X10 ateach loop is similar to the foreach loop, except that each iteration executes in parallel at a place specified by a distribution
  - “ateach (point p : D) S ” is equivalent to “for (point p : D.region) async (D[p]) S”
- As before, finish can be used to wait for termination of all ateach iterations
  - “finish ateach(point[i] : dist.factory.unique()) S” creates one activity per place, as in an SPMD computation
  - ateach is a convenient construct for writing parallel matrix code that is independent of the underlying distribution e.g.,
    - ateach (point p : A.distribution) A[p] = f(B[p], C[p], D[p]) ;

X10 clocks: Motivation

- Activity coordination using finish and force() is accomplished by checking for activity termination
- However, there are many cases in which a producer-consumer relationship exists among the activities, and a “barrier”-like coordination is needed without waiting for activity termination
  - The activities involved may be in the same place or in different places





X10 Clocks

`clock c = clock.factory.clock();`

- Allocate a clock, register current activity with it. Phase 0 of c starts.

`async(…) clocked (c1,c2,…) S`
`ateach(…) clocked (c1,c2,…) S`
`foreach(…) clocked (c1,c2,…) S`

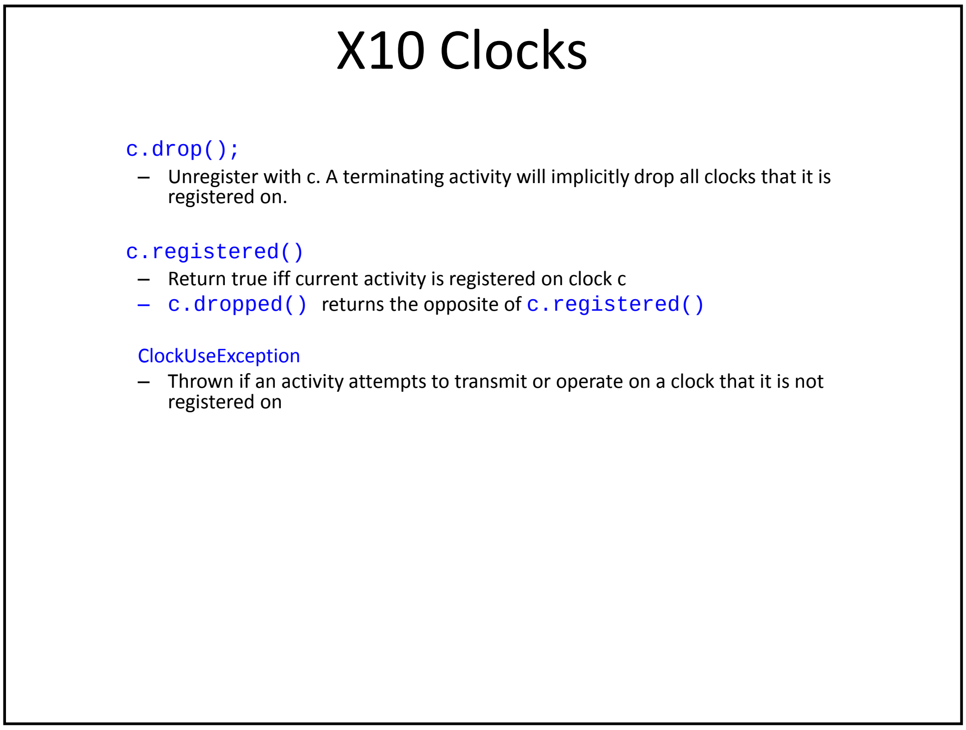
- Create async activities registered on clocks c1, c2, …

`c.resume();`

- Nonblocking operation that signals completion of work by current activity for this phase of clock c

`next;`

- Barrier --- suspend until all clocks that the current activity is registered with can advance. c.resume() is first performed for each such clock, if needed.
- Next can be viewed like a “finish” of all computations under way in the current phase of the clock



X10 Clocks

`c.drop();`

- Unregister with c. A terminating activity will implicitly drop all clocks that it is registered on.

`c.registered()`

- Return true iff current activity is registered on clock c
- `c.dropped()` returns the opposite of `c.registered()`

ClockUseException

- Thrown if an activity attempts to transmit or operate on a clock that it is not registered on

Example

Example of transmitting clock from parent to child

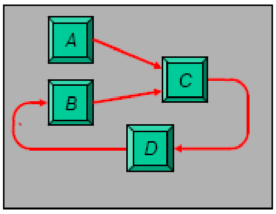
```
finish async {
   final clock c = clock.factory.clock();
   foreach (point[i]: [1:N]) clocked (c) {
      while ( true ) {
         int old_A_i = A[i]; int new_A_i = Math.min(A[i],B[i]);
         if ( i > 1 ) new_A_i = Math.min(new_A_i,B[i-1]);
         if ( i < N ) new_A_i = Math.min(new_A_i,B[i+1]);
         A[i] = new_A_i;
         next;
         int old_B_i = B[i]; int new_B_i = Math.min(B[i],A[i]);
         if ( i > 1 ) new_B_i = Math.min(new_B_i,A[i-1]);
         if ( i < N ) new_B_i = Math.min(new_B_i,A[i+1]);
            B[i] = new_B_i;
         next;
         if ( old_A_i == new_A_i && old_B_i == new_B_i ) break;
      } // while
   } // foreach
/ finish async
```

NOTE: exiting from while loop terminates activity for iteration i, and automatically deregisters activity from clock

---

Models of Computations for Embedded Systems

- A *Model of computation is a formal representation* of the operational semantics of networks of functional blocks describing the computations
- MoC is related to an application or an architecture
  - A mapping is required

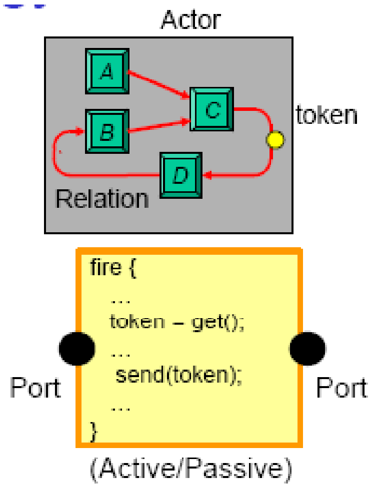


Language Styles

- **Finite versus infinite state**
  - Some models assume that an infinite number of states can exist; other models are finite-state
- **Control versus data**
  - Many programming languages have been developed for control-intense applications such as protocol design
  - Similarly, many other programming languages have been designed for data intense applications such as signal processing
- **Sequential versus parallel**
  - Many languages have been developed to make it easy to describe parallel programs in a way that is both intuitive and formally verifiable
  - However, programmers still feel comfortable with sequential programming when they can get away with it

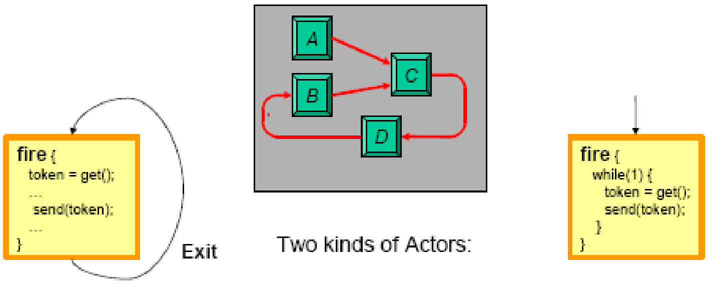
Terminology

- Actor
  - Encapsulates part of the functionality of a design
- Relation
  - Actors are connected with each other using relations
- Token
  - A quantum of information
  - Represents a communication signal
- Firing
  - Internal computation
  - Communication with other actors



```
fire {
   ...
   token – get();
   ...
    send(token);
   ...
}
```

Active/Passive Actors



```
fire {
  token = get();
  ...
  send(token);
  ...
}
```

Two kinds of Actors:

```
fire {
  while(1) {
    token = get();
    send(token);
  }
}
```

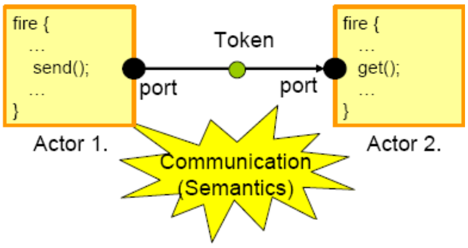
Passive Actor:

- Scheduler needed
  - Schedule ABBCD
- A firing needs to terminate
- Fire-and-exit behavior

Active Actor:

- Schedules itself
- A firing typically does not terminate
  - Endless while loop
- Process behavior

Communication between Actors



```
fire {
  ...
  send();
  ...
}
```

```
fire {
  ...
  get();
  ...
}
```

Data Type of the Token

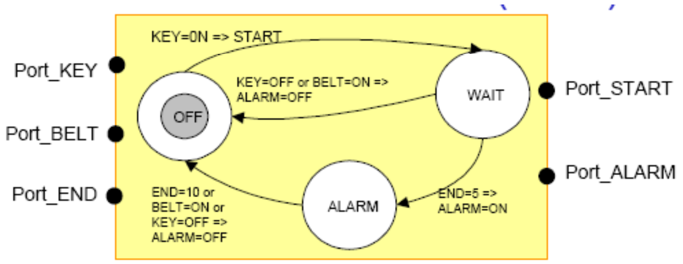
- Integer, Double, Complex
- Matrix, Vector
- Record

Way exchange takes place

- Buffered
- Timed
- Synchronized

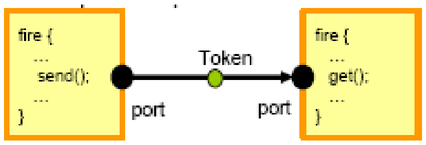
Finite State Machines

- More efficient way to describe sequential control
- Formal semantics which allows for verifying various properties like safety, liveness, and fairness
- FSM may only have one state active at the time
- FSM has only a finite number of states



Dumb Wiring Models

- Network of concurrent executing actors
  - passive Actors
  - Communication is unbuffered
- A model progresses as a sequence of "ticks."
- Computation and Communication is instantaneous within a tick.
- At a tick, the values of the registers are defined by state update equations



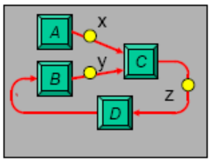
$$D' = C(A(), B(D))$$

Synchronous/Reactive Models

- Network of concurrent executing actors
  - passive Actors
  - Communication is unbuffered
- A model progresses as a sequence of “ticks.”
- Computation and Communication is instantaneous within a tick.
- At a tick, the signals are defined by a fixed point equation:
- Characteristics of SR Models
  - Tightly Synchronized
  - Control intensive systems

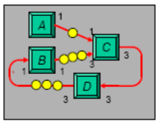
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f_A(1) \\ f_b(z) \\ f_c(x,y) \end{bmatrix}$$

Fixed point equation

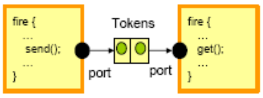


Synchronous Dataflow

- Network of concurrent executing actors
  - passive Actors
  - Communication is buffered
- A model progresses as a sequence of “iterations.”
- A “firing rule” determines the firing condition of an actor.
- At each firing, a fixed number of tokens is consumed and produced
- Characteristics of SDF
  - Compile time analyzable
  - Memory/Schedule/Speed



Schedule: **ABBBC**

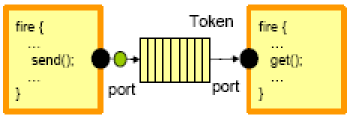


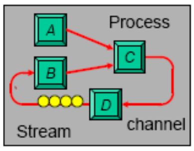
```
fire {
  ...
   send();
  ...
}
```

```
fire {
  ...
  get();
  ...
}
```

Process Networks

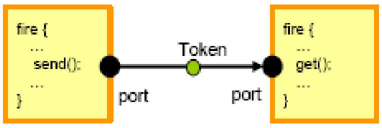
- Network of concurrent executing processes
  - active Actors
  - Communicate over unbounded FIFOs
- Performing some operation, a blocking read or a non-blocking write
- Characteristics of Process Networks
  - Deterministic Execution
  - Doesn't impose a particular schedule
  - (Dynamic) Dataflow

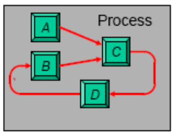




Communicating Sequential Processes

- Network of concurrent executing processes
  - active Actors
  - Communicate by rendezvous
- Reads block until a blocking read or a non-blocking write
- Characteristics of CSP
  - Inherently non-deterministic execution
  - Formalization of Agent/Repository



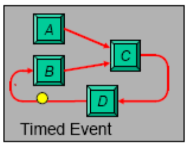


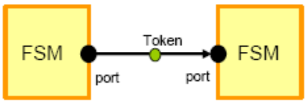
Hierarchical Composition

- Each model of computation has advantages and disadvantages
  - Ease of representation
  - Formally provable properties
  - Computational completeness
  - Concurrency vs. Sequentiality
- Combine models of computation hierarchically to balance those tradeoffs:
  - Preserve formal properties in composition
  - Proper abstraction

Codesign Finite State Machine

- Network of concurrent executing actors
  - Passive Actors
  - Synchronous locally
  - Asynchronous globally
- An "event" causes the evaluation (firing) of a FSM
- Characteristics of CFSM
  - Compile time analyzable
  - Reactive systems



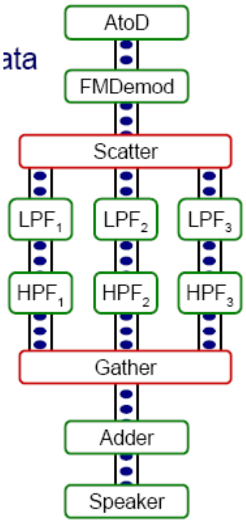


Notes

- Not all Models of Computation are concurrent
  - Good representations of sequential operations and state can be just as important as representing concurrency
- No model of computation makes all the right design tradeoffs…
  - Less structured models of computation sometimes easier to use and sometimes more difficult…
- The semantics of models of computation actually say very little about "implementation"
  - Although in many cases there are known good ways of implementing them

Streamit

- For programs based on streams of data
  - Audio, video, DSP, networking, and cryptographic processing kernels
- Examples: HDTV editing, radar tracking, microphone arrays, cell phone base stations, graphics
- Several attractive properties
  - Regular and repeating computation
  - Independent filters with explicit communication
  - Task, data, and pipeline parallelism

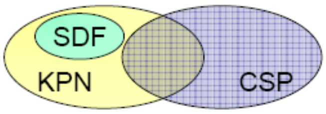


Streaming Models of Computation

- Many different ways to represent streaming
  - Do senders/receivers block?
  - How much buffering is allowed on channels?
  - Is computation deterministic?
  - Can you avoid deadlock?
- Three common models:
  - Kahn Process Networks
  - Synchronous Dataflow
  - Communicating Sequential Processes

Streaming Models of Computation

	Communication Pattern	**Buffering**	**Notes**
Kahn process networks (KPN)	Data-dependent, but deterministic	Conceptually unbounded	- UNIX pipes - Ambric (startup)
Synchronous dataflow (SDF)	Static	Fixed by compiler	- Static scheduling - Deadlock freedom
Communicating Sequential Processes (CSP)	Data-dependent, allows non-determinism	None (Rendesvouz)	- Rich synchronization primitives - Occam language



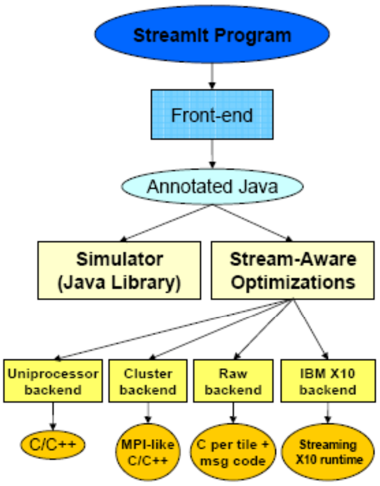
space of program behaviors

What is Streamit

- A high-level, architecture-independent language for streaming applications
  - Improves programmer productivity (vs. Java, C)
  - Offers scalable performance on multicores
- Based on *synchronous dataflow*, with dynamic extensions
  - Compiler determines execution order of filters
  - Many aggressive optimizations possible

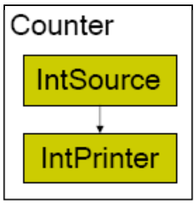
The Streaming Project

- **Applications**
  - DES and Serpent [PLDI 05]
  - MPEG-2 [IPDPS 06]
  - SAR, DSP benchmarks, JPEG, …
- **Programmability**
  - StreamIt Language (CC 02)
  - Teleport Messaging (PPOPP 05)
  - Programming Environment in Eclipse (P-PHEC 05)
- **Domain Specific Optimizations**
  - Linear Analysis and Optimization (PLDI 03)
  - Optimizations for bit streaming (PLDI 05)
  - Linear State Space Analysis (CASES 05)
- **Architecture Specific Optimizations**
  - Compiling for Communication-Exposed Architectures (ASPLOS 02)
  - Phased Scheduling (LCTES 03)
  - Cache Aware Optimization (LCTES 05)
  - Load-Balanced Rendering (Graphics Hardware 05)



Example: A Simple Counter

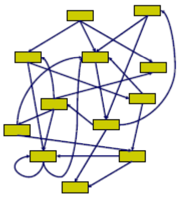
```
void->void pipeline Counter() {
    add IntSource();
    add IntPrinter();
}
void->int filter IntSource() {
    int x;
    init { x = 0; }
    work push 1 { push (x++); }
}
int->void filter IntPrinter() {
    work pop 1 { print(pop()); }
}
```



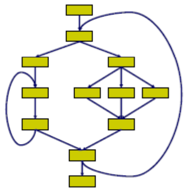
```
% strc Counter.str -o counter
% ./counter -i 4
0
1
2
3
```

Representing Streams

- Conventional wisdom: streams are graphs
  - – Graphs have no simple textual representation
  - – Graphs are difficult to analyze and optimize
- Insight: stream programs have structure

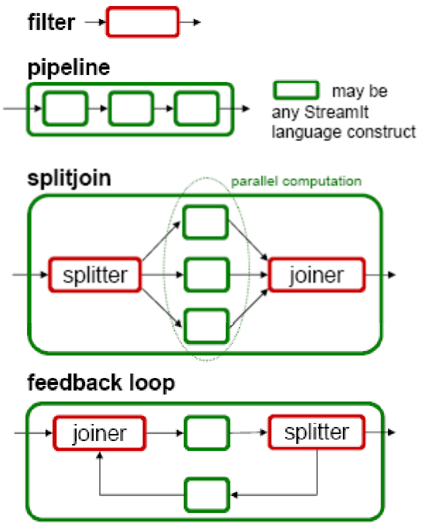


unstructured



structured

Structured Streams



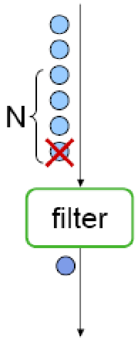
- Each structure is single-input, single-output
- Hierarchical and composable

Filter Example: Low Pass Filter

```
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;

    init {
        weights = calcWeights(freq);
    }

    work peek N push 1 pop 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



Low Pass Filter in C

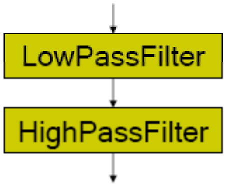
```
void FIR(
  int* src,
  int* dest,
  int* srcIndex,
  int* destIndex,
  int srcBufferSize,
  int destBufferSize,
  int N) {

  float result = 0.0;
  for (int i = 0; i < N; i++) {
    result += weights[i] * src[(*srcIndex + i) % srcBufferSize];
  }
  dest[*destIndex] = result;
  *srcIndex = (*srcIndex + 1) % srcBufferSize;
  *destIndex = (*destIndex + 1) % destBufferSize;
}
```

- FIR functionality obscured by buffer management details
- Programmer must commit to a particular buffer implementation strategy

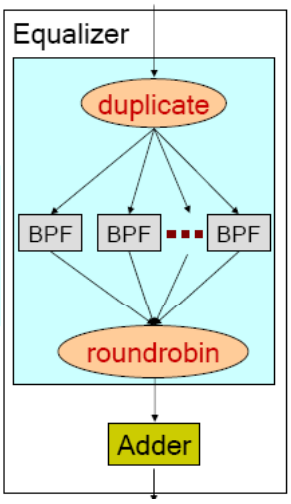
Pipeline Example: Band Pass Filter

```
float→float pipeline BandPassFilter (int N,
                                     float low,
                                     float high) {
   add LowPassFilter(N, low);
   add HighPassFilter(N, high);
}
```



Split-Join Example: Equalizer

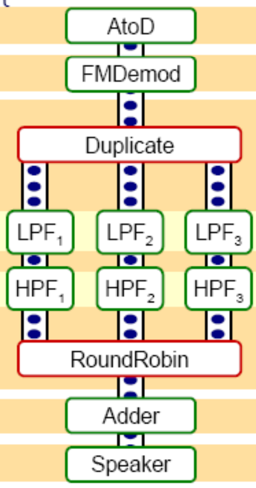
```
float→float pipeline Equalizer (int N,
                                float lo,
                                float hi) {
   add splitjoin {
      split duplicate;
      for (int i=0; i<N; i++)
         add BandPassFilter(64, lo + i*(hi - lo)/N);
      join roundrobin(1);
   }
   add Adder(N);
}
```



Building Larger Programs: FM Radio

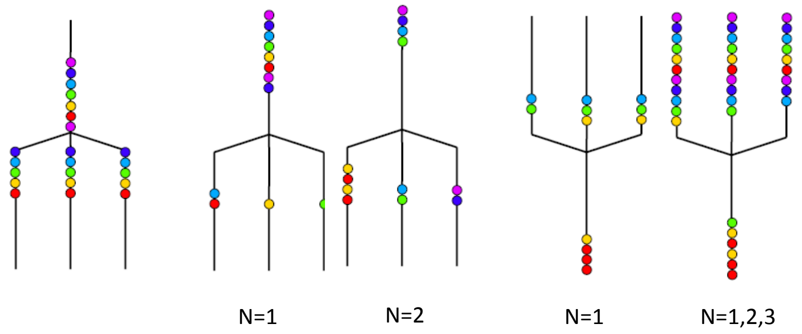
```
void->void pipeline FMRadio(int N, float lo, float hi) {
   add AtoD();
   add FMDemod();
   add splitjoin {
    split duplicate;
    for (int i=0; i<N; i++) {
      add pipeline {
         add LowPassFilter(lo + i*(hi - lo)/N);

         add HighPassFilter(lo + i*(hi - lo)/N);
      }
    }
    join roundrobin();
   }
   add Adder();
   add Speaker();
}
```

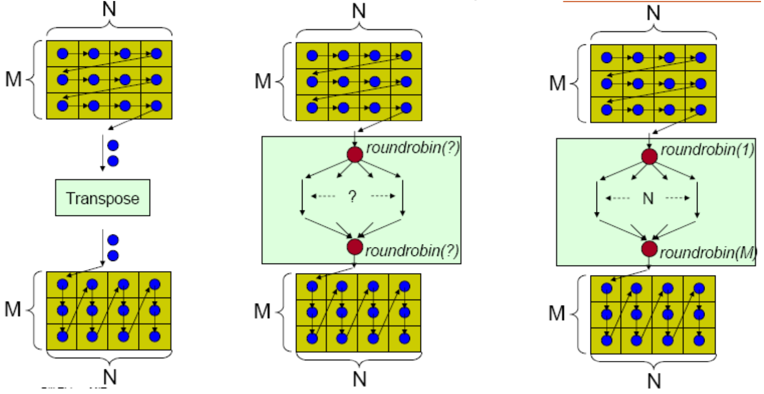


SplitJoin Options

- Split duplicate
- Split roundrobin (N)
- Join roundrobin (N)



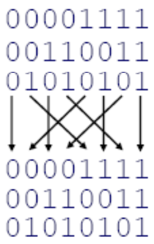
Matrix Transpose

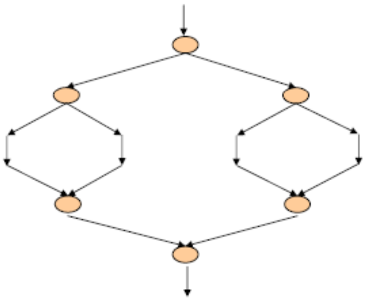


```
float->float splitjoin Transpose (int M,
                                  int N) {
   split roundrobin(1);
   for (int i = 0; i<N; i++) {
      add Identity<float>;
   }
   join roundrobin(M);
}
```

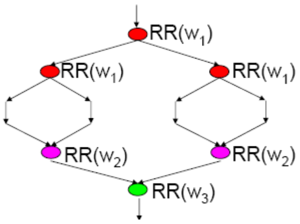
Bit Reversed Ordering

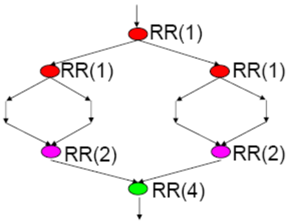
- Many FFT algorithms require a bit-reversal stage
- If item is at index *n (with binary digits b0 b1 … bk),* then it is transferred to reversed index bk… b1 b0
- For 3-digit binary numbers:





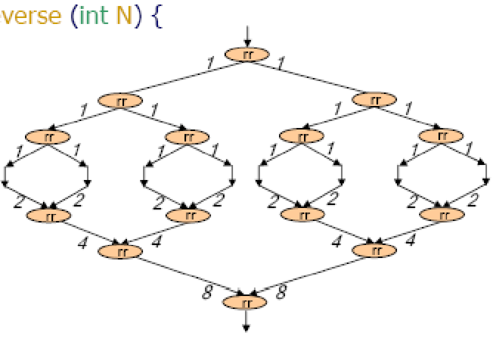
Bit Reversed Ordering





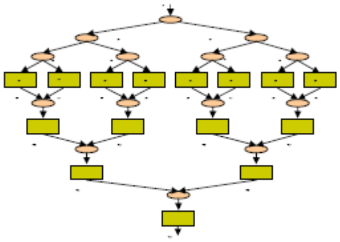
Bit Reversed Ordering

```
complex->complex pipeline BitReverse (int N) {
   if (N==2) {
      add Identity<complex>;
   } else {
      add splitjoin {
         split roundrobin(1);
         add BitReverse(N/2);
         add BitReverse(N/2);
         join roundrobin(N/2);
      }
   }
}
```

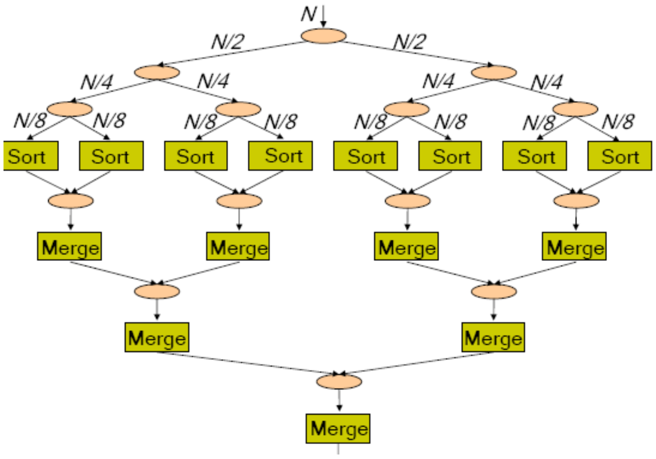


N-Element Mergesort

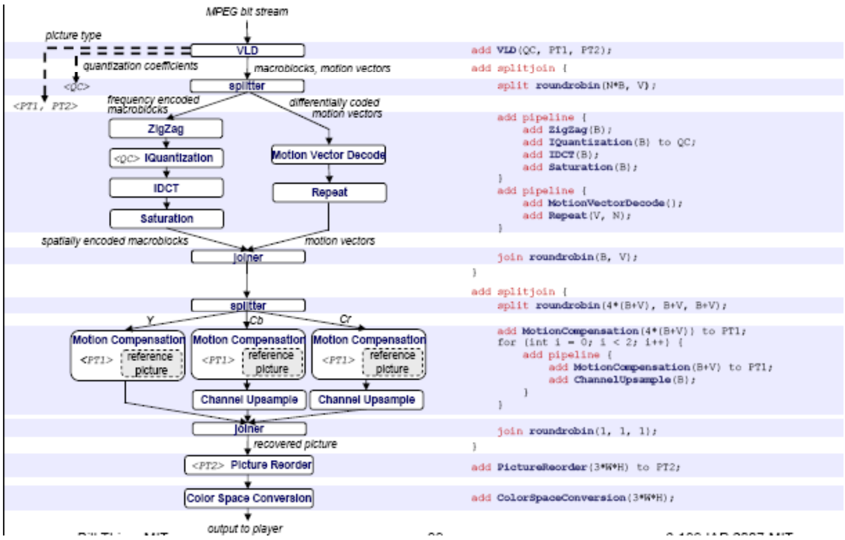
```
int->int pipeline MergeSort (int N) {
   if (N==2) {
      add Sort(N);
   } else {
      add splitjoin {
         split roundrobin(N/2);
         add MergeSort(N/2);
         add MergeSort(N/2);
         join roundrobin(N/2);
      }
   }
   add Merge(N);
}
```



N-element Mergesort (3-level)



MPEG-2 Decoder



```
add VLD(QC, PT1, PT2);
add splitjoin {
    split roundrobin(N*B, V);

    add pipeline {
        add ZigZag(B);
        add IQuantization(B) to QC;
        add IDCT(B);
        add Saturation(B);
    }
    add pipeline {
        add MotionVectorDecode();
        add Repeat(V, N);
    }

    join roundrobin(B, V);
}
add splitjoin {
    split roundrobin(4*(B+V), B+V, B+V);

    add MotionCompensation(4*(B+V)) to PT1;
    for (int i = 0; i < 2; i++) {
        add pipeline {
            add MotionCompensation(B+V) to PT1;
            add ChannelUpsample(B);
        }
    }

    join roundrobin(1, 1, 1);
}
add PictureReorder(3*W*H) to PT2;

add ColorSpaceConversion(3*W*H);
```