Parallelizing XML Pipelines

Daniel Zinn

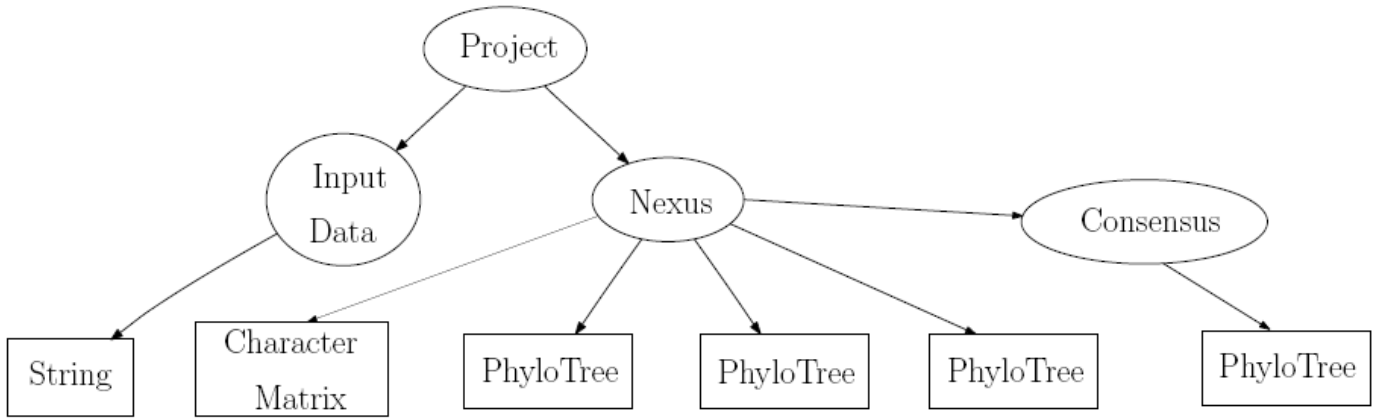
As part of a guest lecture at UC Davis
for ECS265 Fall'09

XML Processing Pipelines

- XML as folder structures for scientific data
  - Nested, labeled Collections
  - Data in binary “domain-specific” format inside collections

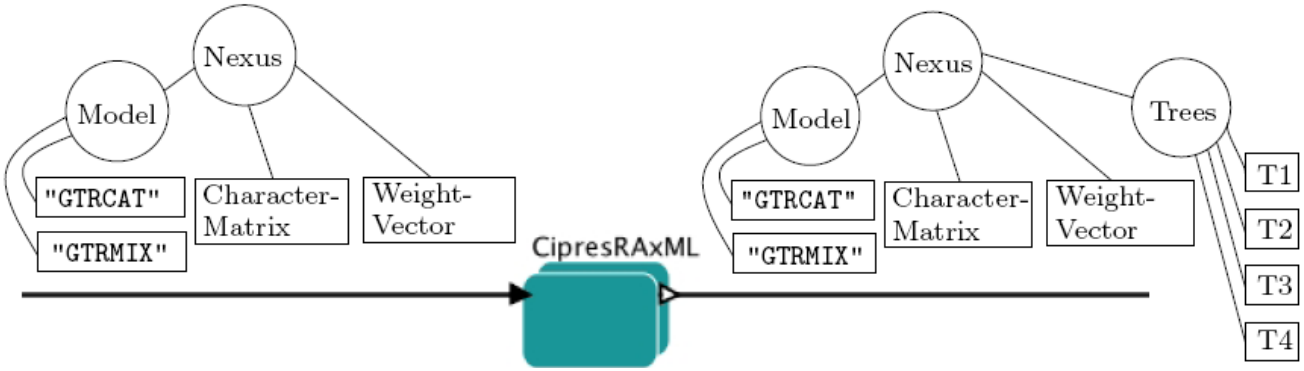
Pipeline Examples:

- Image conversion pipeline (we’ll use here)
- Phylogenetics pipeline

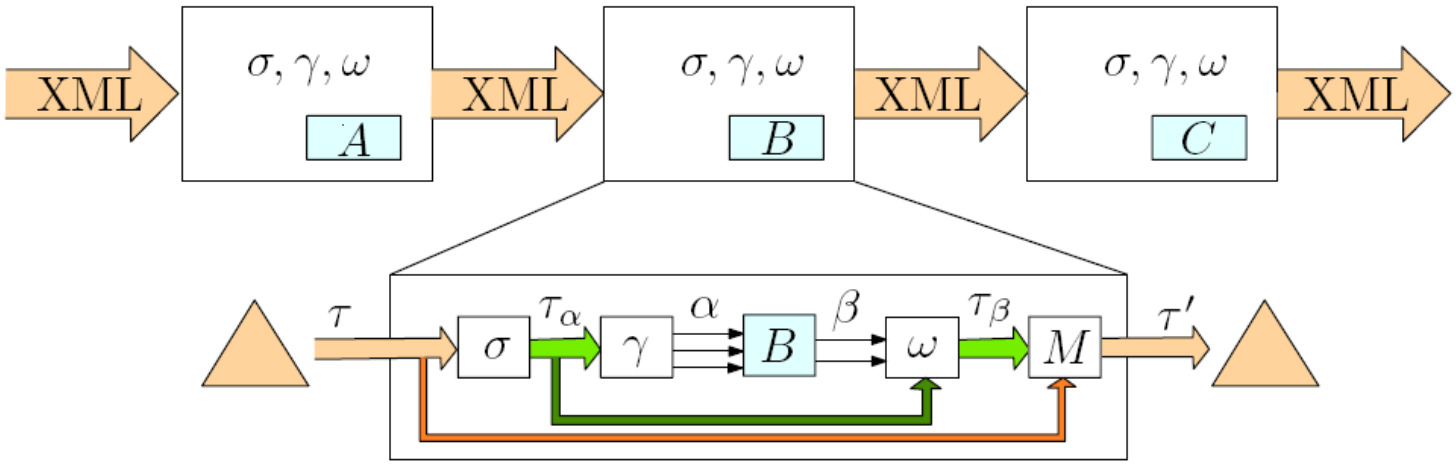


XML Pipelines – Special Case of VDAL

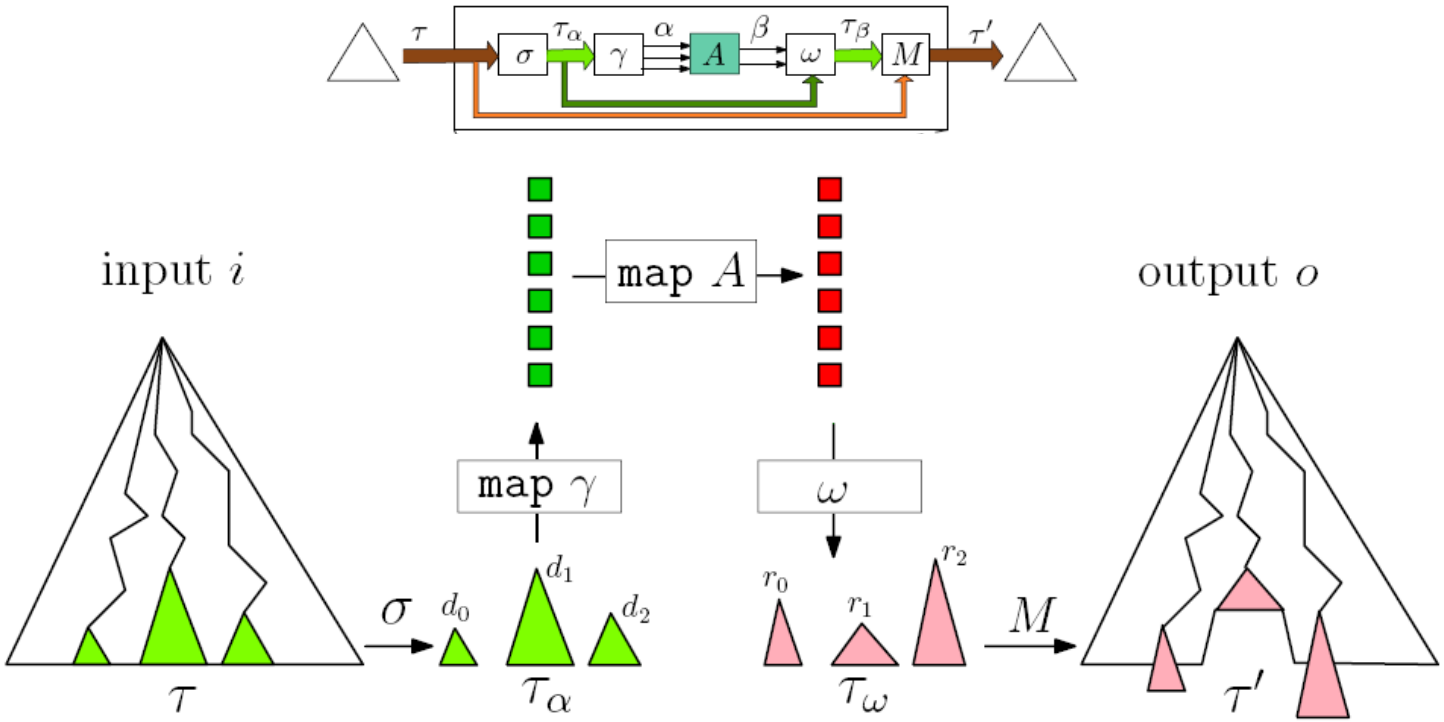
- Example Step in Phylo-Pipeline:



- VDAL = Virtual Data Assembly Lines



VDAL Execution Semantics



Scope σ – to select scope of actor invocation
Input assembler γ – to create inputs for A
Write expression ω – to write results back into data stream

Andy Warhol – Pop Art

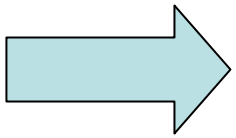
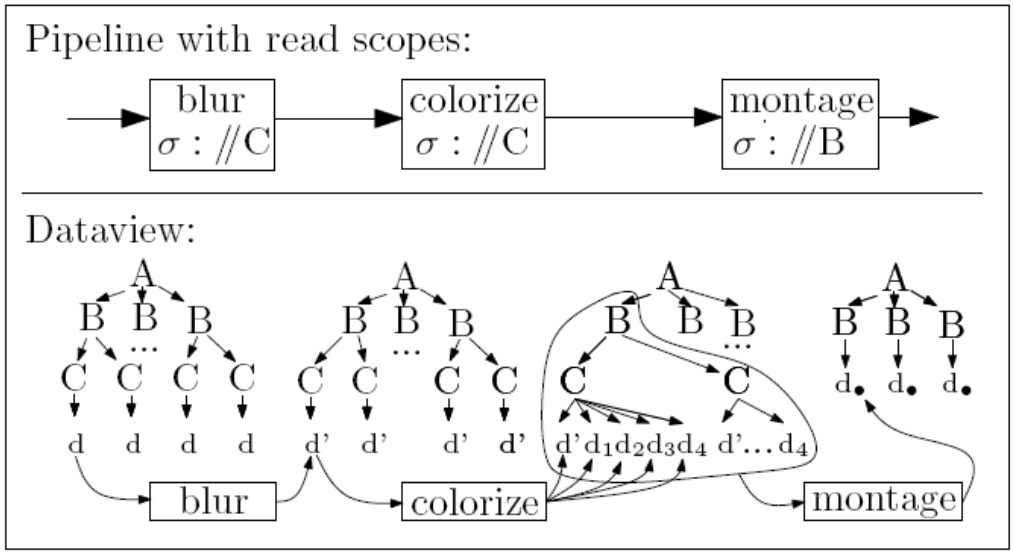
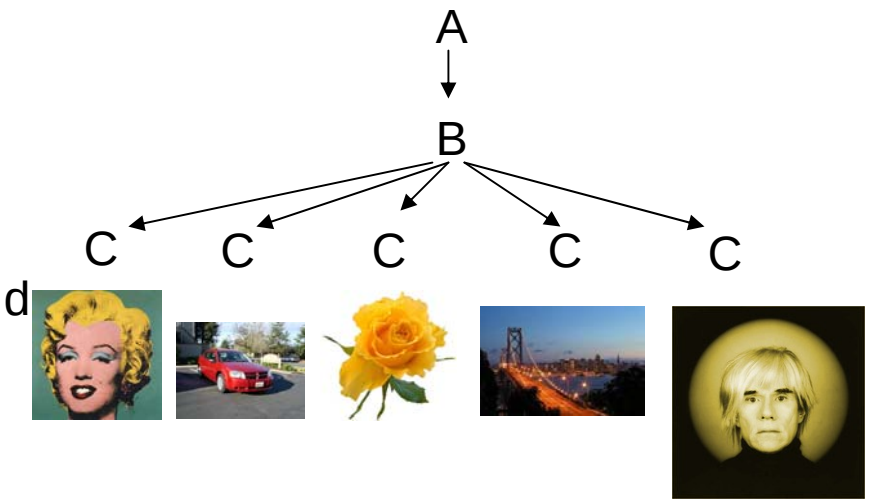
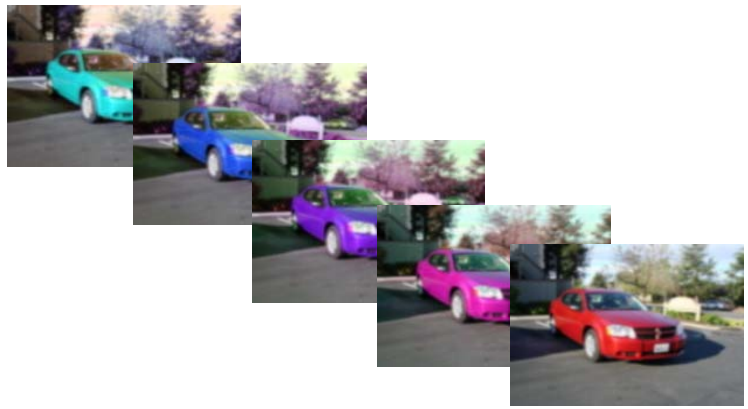
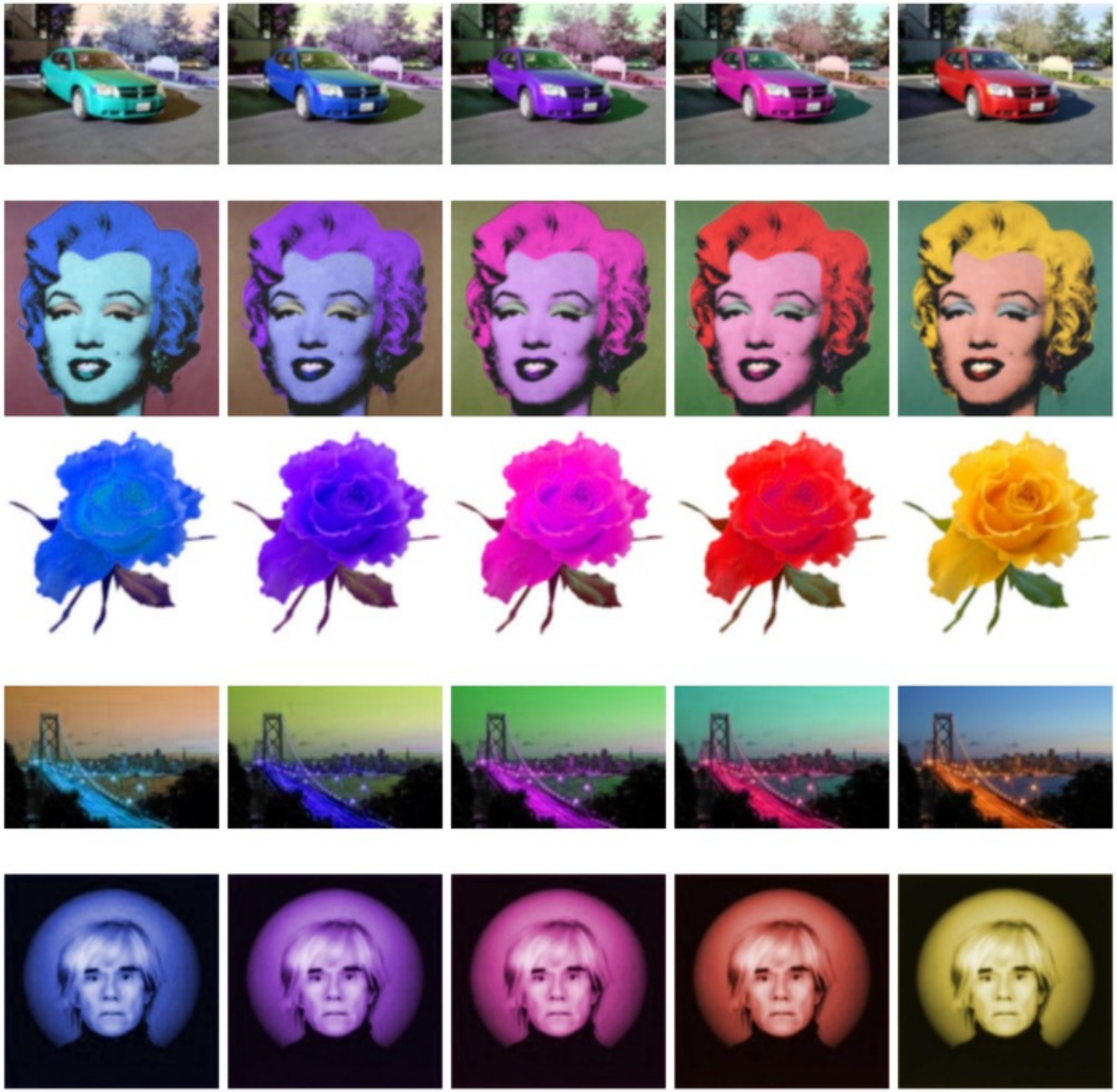


Image Processing Example





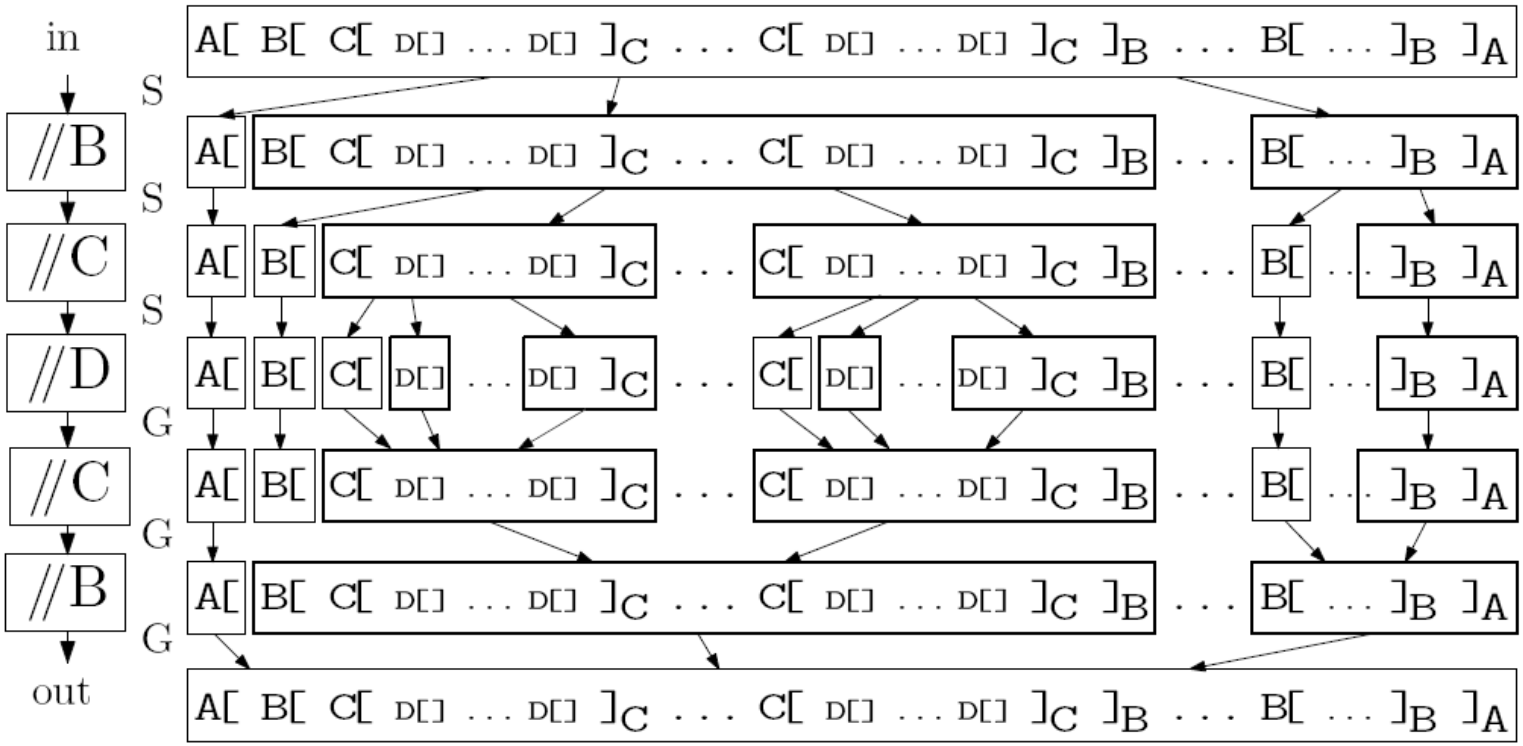




Goal: Parallelize Execution over Scope

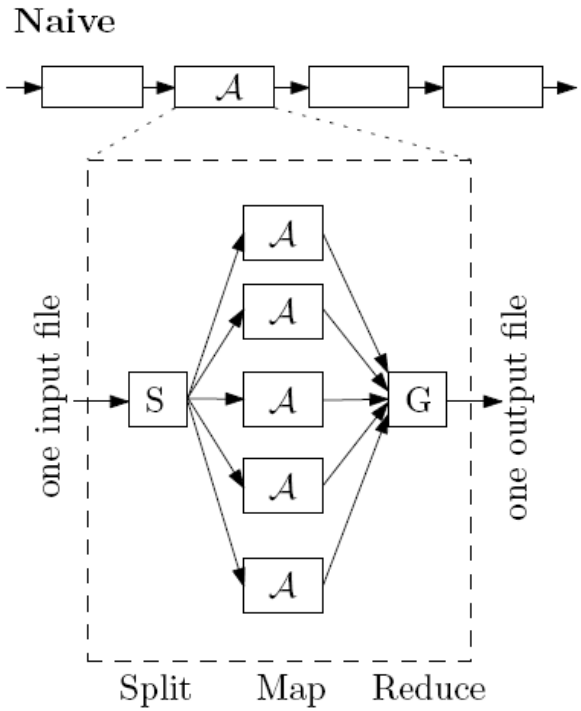
- Independent work pieces (scopes)
- But scopes vary from step to step in the pipeline

→ How to efficiently split the data for each step?



Approaches: Naive

- 1 Naive -- Split for each step from scratch



Key: Integer
Token := XOpen | XClose | Data
Value: TList := **List of** Token

Approaches: Naive

- 1 Naive

```
1  SplitNaive: TList input, XPath σ → [(Integer, TList)]
     int i := 0; TList splitOut := [ ]
3    FOREACH token IN input DO
       IF (token opens new scope match with σ) AND
5           (splitOut ≠ [ ]) THEN
         EMIT (i, splitOut) // one split for each scope match
7        i++; splitOut := [ ]
       splitOut.append(token)
9    EMIT (i, splitOut)

11 MapNaive: Integer s, TList val → [(Integer, TList)]
     val' := 𝒜(val) // execute pipeline task
13   EMIT (s, val')

15 ReduceNaive: Integer s, [TList] vals → [(Integer,TList)]
     TList output := [ ]
17   WHILE vals.notEmpty() DO
       output.append(vals.getValue()) // collapse to single value
19   EMIT (0, output)
```

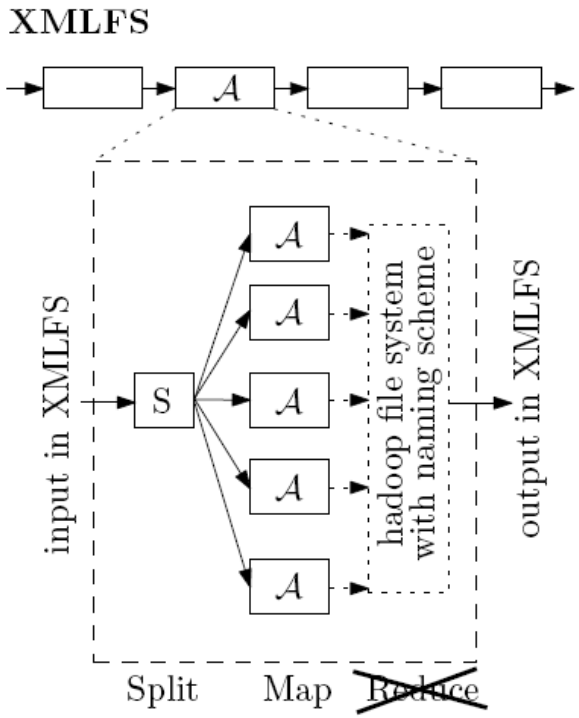
Figure 5: Split, Map, Reduce for Naive strategy.

Key: Integer
Token := XOpen | XClose | Data
Value: TList := **List of** Token

+ easy to implement
+ small keys
- serial split and join

Approach: XML FS

- 2. XMLFS -- Merge data in HDFS



Encoding for XML Data in FS

- Collections – directories
- Data – files

→ Need naming scheme for collections and data

  - Stable insertions and deletions
  - Short string representation
  - Efficient comparisons

- Each Mapper needs to know where the data should be stored

Adding information to keys and values...

- Simple decimal IDs as naming scheme
  - Decimal numbers form a dense space
  - Optimizations: Use base “maxint”; Start with gaps

- XMLFS Data Structures

	ID	:=	**List of** Long
	IDXOpen	:=	**Record{** id: ID, t: XOpen**}**
	IDData	:=	**Record{** id: ID, t: Data**}**
	IDToken	:=	IDXOpen | IDData | XClose
Key:	XKey	:=	**Record{** start: ID, end: ID, lp: TIDList**}**
Value:	TIDList	:=	**List of** IDToken

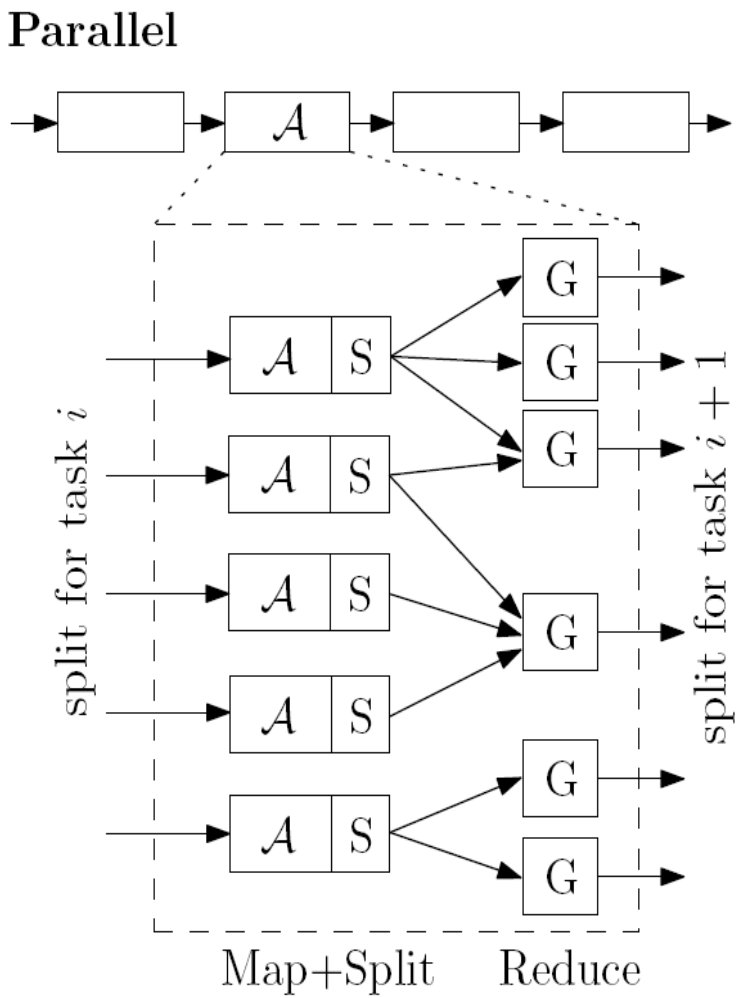
Drawbacks of XMLFS

- Splitting in serial
- Merging is done by file-system
  - Single master holds FS index
  - Not build for vast amount of directories / small files
  - Essentially a bottleneck again
- Next Strategy: Try to do all in parallel

Approach: Parallel

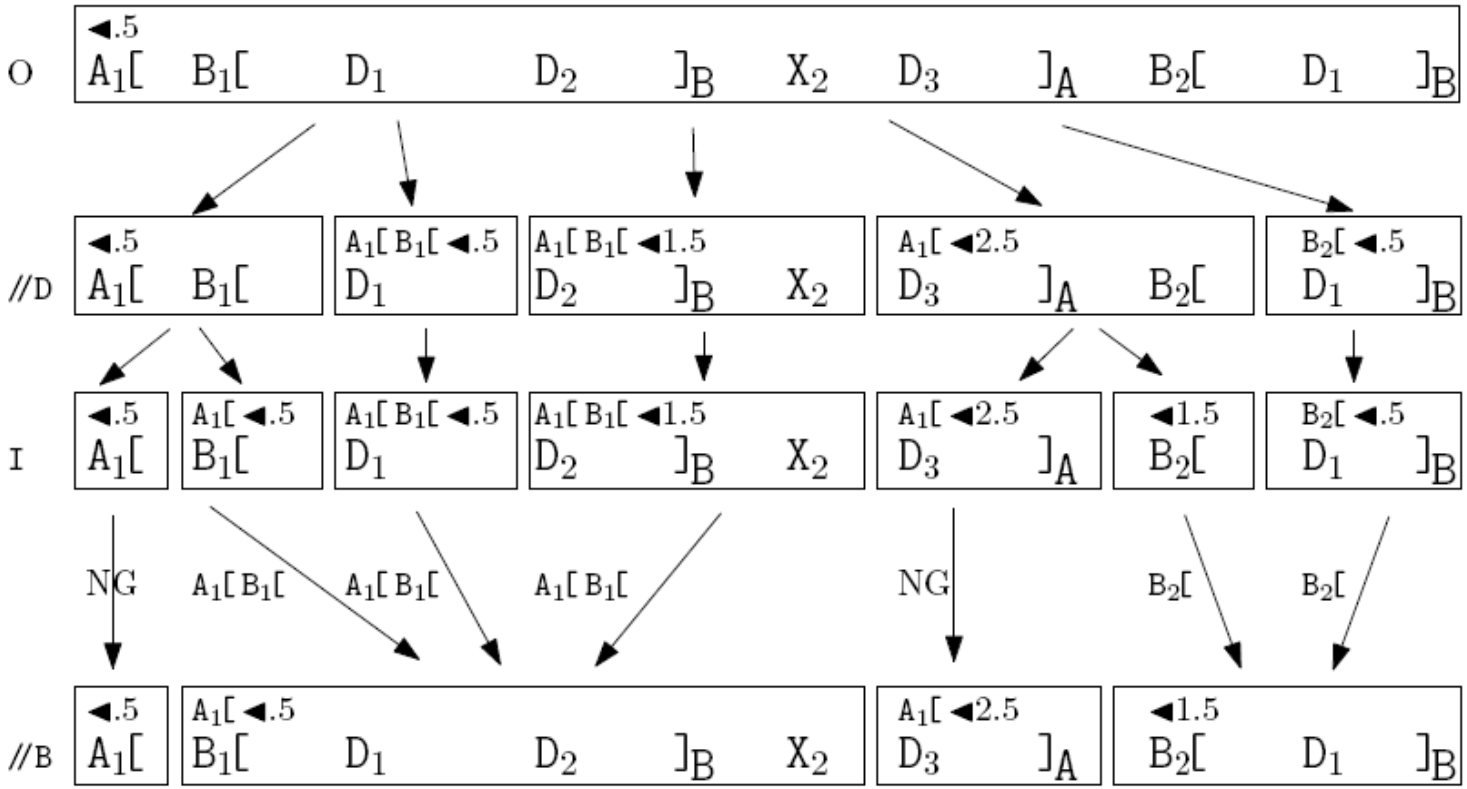
- Exploit existing partitioning for consecutive MRs
- How does the scope change?
  - Refinement (//B → //C)
  - More coarse-grained (//C → //B)
  - Unrelated (//C → //D)
  - Same scope – can we keep grouping?
- Re-grouping is done in a two-step process:
  1. Splitting (if multiple scopes in current packet)
  2. Merging (if scope scattered over multiple packets)

Parallel Schematics



- Map: A and split
- Reduce: combine fragments

Parallel Grouping



GroupBy := **Record{** group: Bool, gpath: TIDList **}**

Key: PKey := **Record of** XKey **and** GroupBy

Value: TIDList := **List of** IDToken

Parallel Grouping

```
1  MapParallel: SKey key, TIDList val → [(SKey, TIDList)]
     IF (key.lp / val[0]) matches scope σ
3      val' := A(val)
     List of (SKey, TIDList) outlist;
5    // split according to the scope σ' of the following step
     outlist := Split(val', σ', key.start, key.end, key.lp)
7    FOREACH (key,fragment) ∈ outlist DO
       EMIT(key, fragment);
```

Map: Actor + Splitting

Framework: Group and Sort

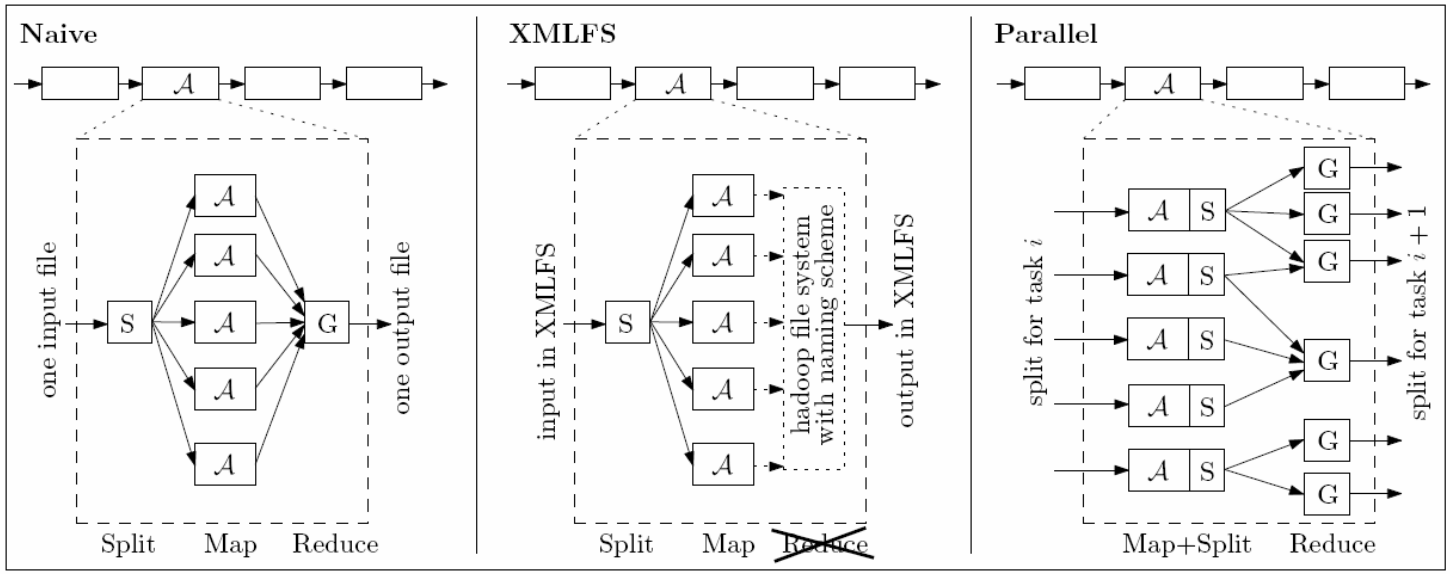
```
1  GroupCompare: SKey keyA, SKey keyB → { <, =, > }
     IF (keyA.group AND keyB.group) THEN
3      // group based on grouping−path
       RETURN LexicCompare(keyA.gpath, keyB.gpath)
5    ELSE
       // don't group (returns < or > for two different fragments)
7      RETURN SortCompare(keyA, keyB)

9  SortCompare: SKey keyA, SKey keyB → { <, =, > }
     // always lexicographically compare "leading path ⊕ start"
11   RETURN LexicCompare( keyA.lp ⊕ keyA.start,
                          keyB.lp ⊕ keyB.start )
```

```
ReduceParallel: SKey key, [TIDList] vs → [(SKey, TIDList)]
  TIDList out := [ ]
  WHILE (val := vs.next())
    out.append(val);
    key.end := val.end // set end in key to end of last fragment
  EMIT(key, out)
```

Reduce: Combine fragments

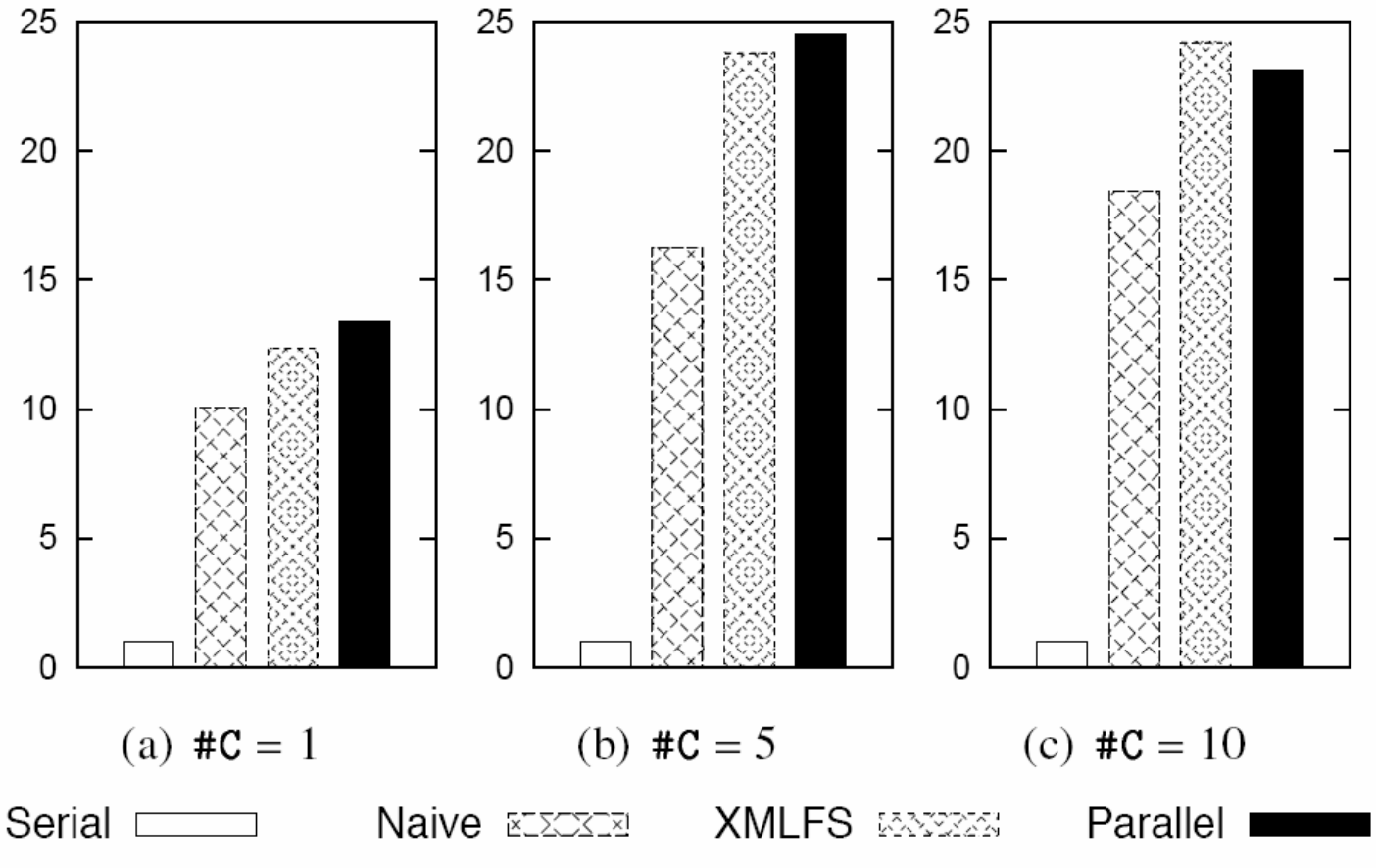
Comparison of Strategies



	Naive	**XMLFS**	**Parallel**
Data	XML File	File system representation	Key-value pairs
Split	Centralized	Centralized	Parallel
Group	Centralized by one reducer	Via file system + naming No shuffle, no reduce	Parallel by reducers
Key-Structure	One integer	Leading path with Ids	Leading path with Ids and grouping information
Value-Structure	SAX-elements	SAX-elements with XMLIds	SAX-elements with XMLIds

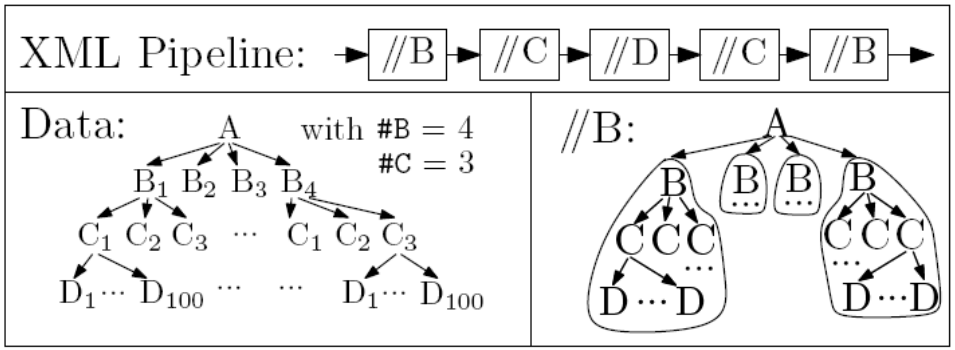
Figure 11: Main differences for compilation strategies

Performance Measurements I

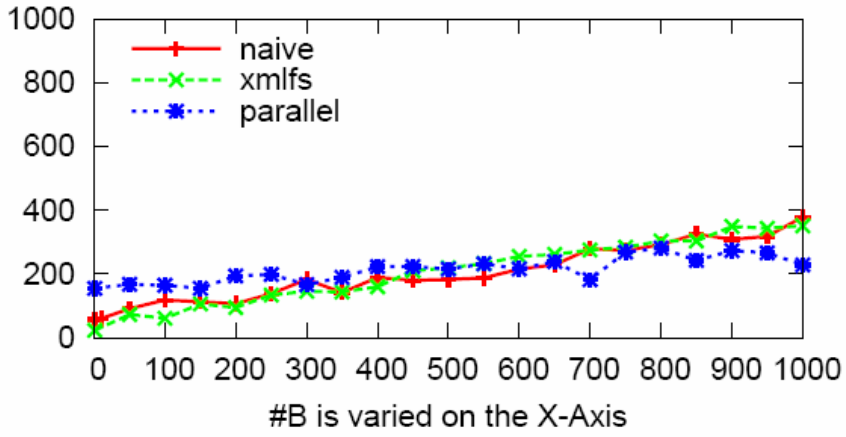


Results on image-transformation pipeline
(#B = 200; each image 2MB)

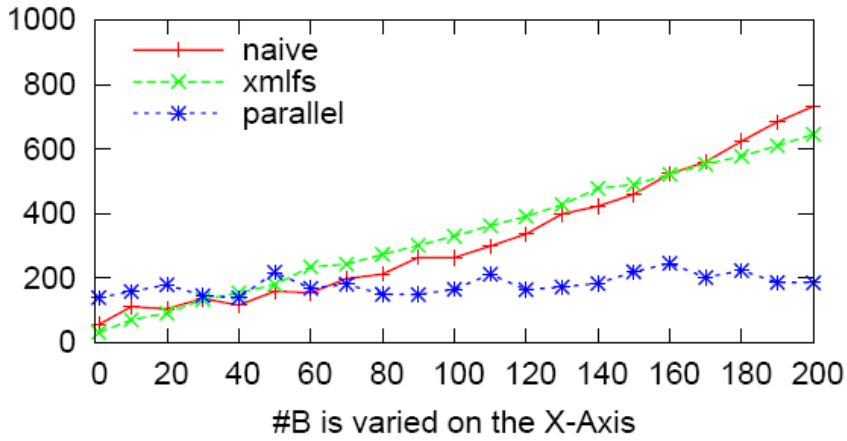
Performance Measurements II



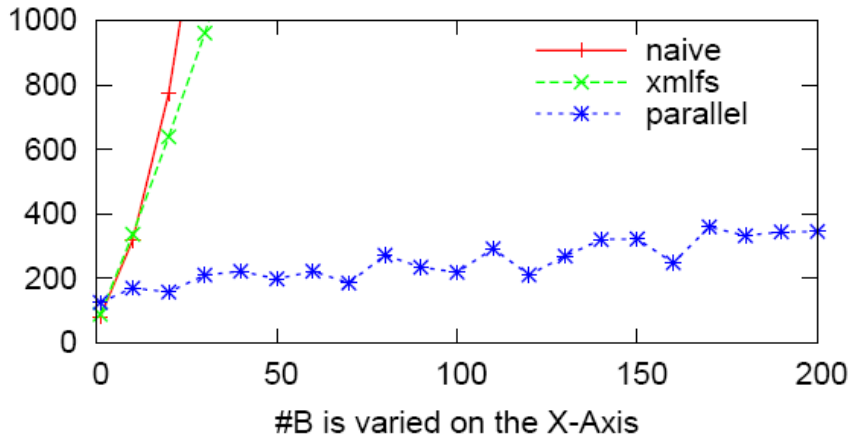
Benchmarks grouping and regrouping



(a) #C = 1



(b) #C = 10



(c) #C = 100

Thank you.

Questions?