Beyond the Embarrassingly Parallel

New Languages, Compilers, and Runtimes for Big-Data Processing

Madan Musuvathi

Microsoft Research

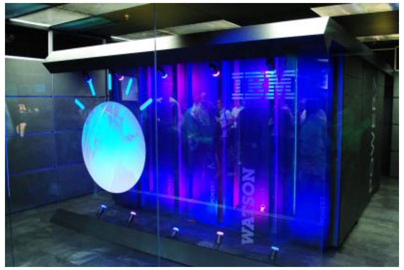
Joint work with

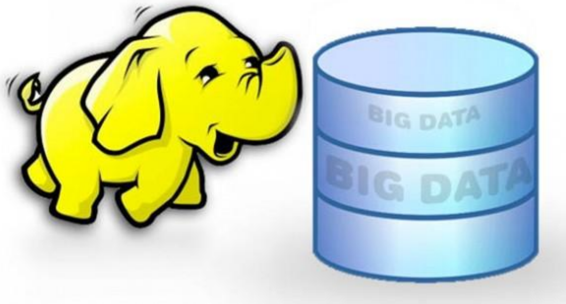
Mike Barnett (MSR), Saeed Maleki (MSR), Todd Mytkowicz (MSR)

Yufei Ding (N.C.State), Daniel Lupei (EPFL), Charith Mendis (MIT),

Mathias Peters (Humboldt Univ.), Veselin Raychev (EPFL)

parallelism





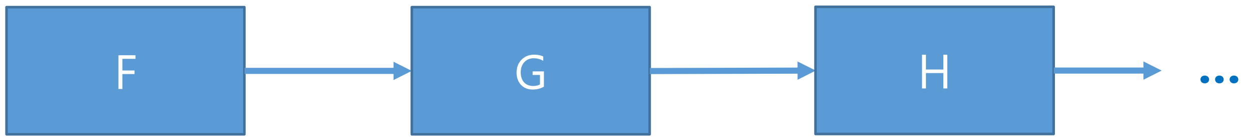
parallelism

=

independent computation

can we parallelize
dependent computation?

“Inherently sequential” code is common



log processing

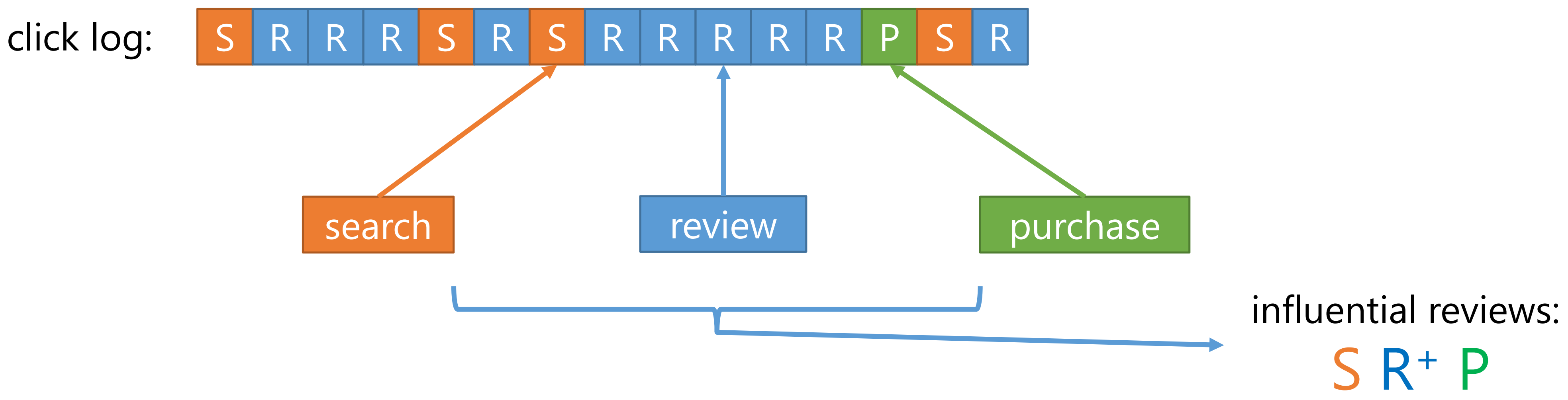
event-series pattern matching

machine learning algorithms

dynamic programming

…

Running example: processing click logs



problem: count influential reviews in the log

Running example: processing click logs

click log: S R R R S R S R R R R R P S R

influential reviews:

$S\ R^{+}\ P$

```
bool search_done = false; int num_reviews = 0; int sum = 0;

for each record in input
   switch record.type:
   case SEARCH:     if (!search_done) { num_reviews = 0;
                                        search_done = true; }
   case REVIEW:     num_reviews++;

   case PURCHASE:   if (search_done) { search_done = false;
                                       sum += num_reviews; }
```

Loop carried state

Extracting parallelism from dependent computations

S	R	R	P	R	S	R	R	R	R	R	R	P	R	P

```
for each record in input
    switch record.type:
    case SEARCH:    if (!search_done) { num_reviews = 0;
                                        search_done = true; }
    case REVIEW:    num_reviews++;

    case PURCHASE:  if (search_done) { search_done = false;
                                       sum += num_reviews;  }
```

(true, 1, 2)


```
for each record in input
    switch record.type:
    case SEARCH:    if (!search_done) { num_reviews = 0;
                                        search_done = true; }
    case REVIEW:    num_reviews++;

    case PURCHASE:  if (search_done) { search_done = false;
                                       sum += num_reviews;  }
```


(false, 0, 0)

(false, 1, 8)

// loop-carried state:
// (search_done, num_reviews, sum)

Extracting parallelism from dependent computations

S R R P R S R

```
for each record in input
    switch record.type:
    case SEARCH:    if (!search_done) { num_reviews = 0;
                                        search_done = true; }
    case REVIEW:    num_reviews++;

    case PURCHASE:  if (search_done) { search_done = false;
                                       sum += num_reviews;  }
```

(false, 0, 0)

(true, 1, 2)

// loop-carried state:
// (search_done, num_reviews, sum)

R R R R R P R P

(sd, nr, s)

```
for each record in input
    switch record.type:
    case SEARCH:    if (!search_done) { num_reviews = 0;
                                        search_done = true; }
    case REVIEW:    num_reviews++;

    case PURCHASE:  if (search_done) { search_done = false;
                                       sum += num_reviews;  }
```

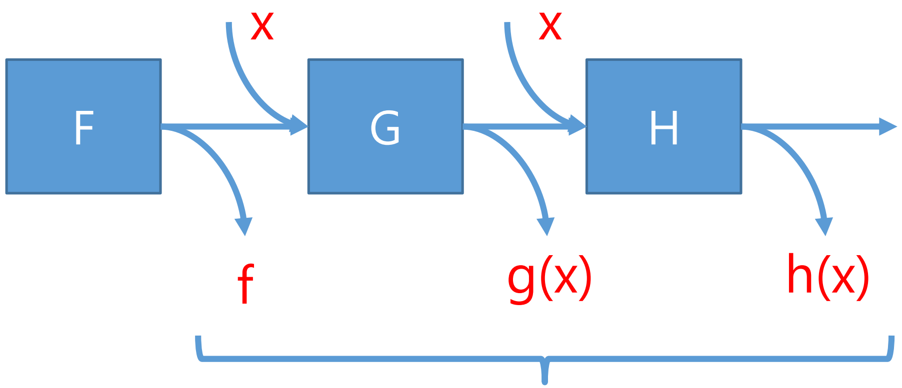
summary = F(sd, nr, s)

F(sd, nr, s) = (false, nr+6, sd ? s+nr+5 : s)

output = F(true, 1, 2)
= (false, 1, 8)

Recipe for breaking dependences

1. replace dependences with symbolic unknowns
2. compute symbolic summaries in parallel
3. combine symbolic summaries



output = h(g(f))

success depends on

1. fast symbolic execution
2. generation of concise summaries

research challenges :

1. identifying “compressible” computation
2. using domain-specific structure
3. automating the parallelization

Successful applications of this methodology

finite-state machines [ASPLOS '14]
- regular expression matching, Huffman decoding, …
- 3x faster on a single core, linear speedup on multiple cores

dynamic programming [PPoPP '14, TOPC '15, ICASSP '16]
- linear speedup beyond the previous-best software Viterbi decoder
- 7x speedup over state-of-the-art speech decoder

large-scale data processing [SOSP '15]
- automatically parallelizable language for temporal analysis

relational databases
- optimize sessionization & windowed aggregates
- 10x improvement over SQL server

machine learning
- parallel stochastic gradient descent

part 2 of the talk

part 1 of the talk

Auto-Parallelization Across Dependences

Large-scale data processing

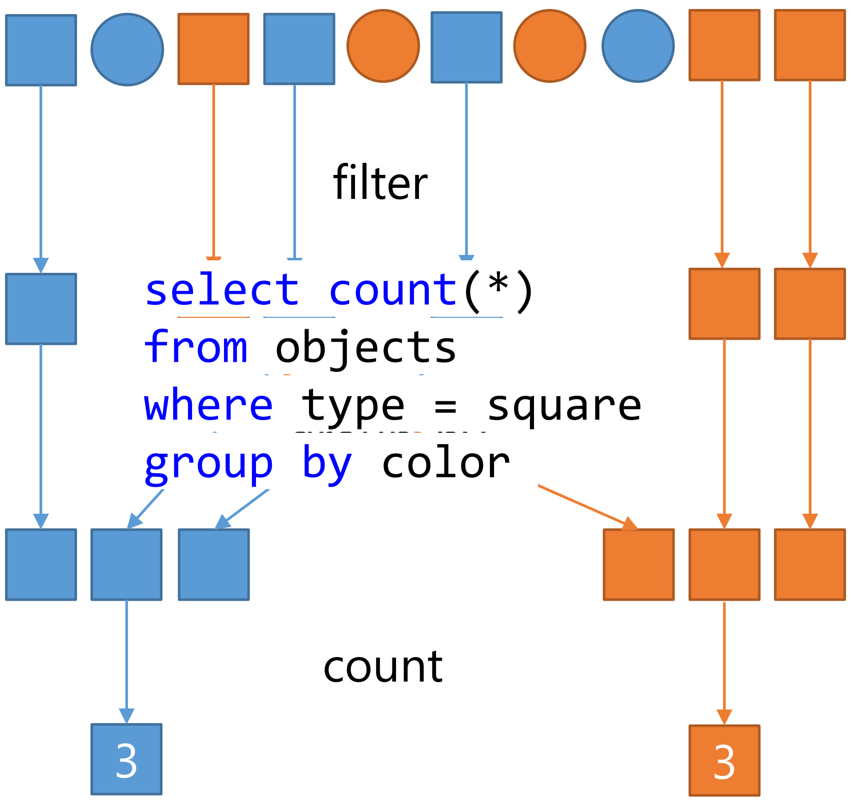
Relational abstractions for data processing

map, reduce, join, filter, group-by

expressive, simple, and declarative

automatically parallelizable

decades of work on optimizations



Forces pushing beyond relational abstractions

queries today = relational skeleton + non-relational logic

embarrassingly parallel
optimized

not parallel
not optimized

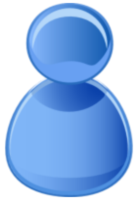
temporal, iterative, stateful
- log analysis
- sessionization
- machine learning

Map-Reduce example

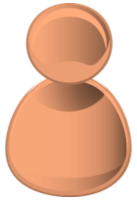
weblog

S	R	R	S	P	R	R	P	S	R	R	S	P	R	S	P	S	R	R	S	P	R	R	P

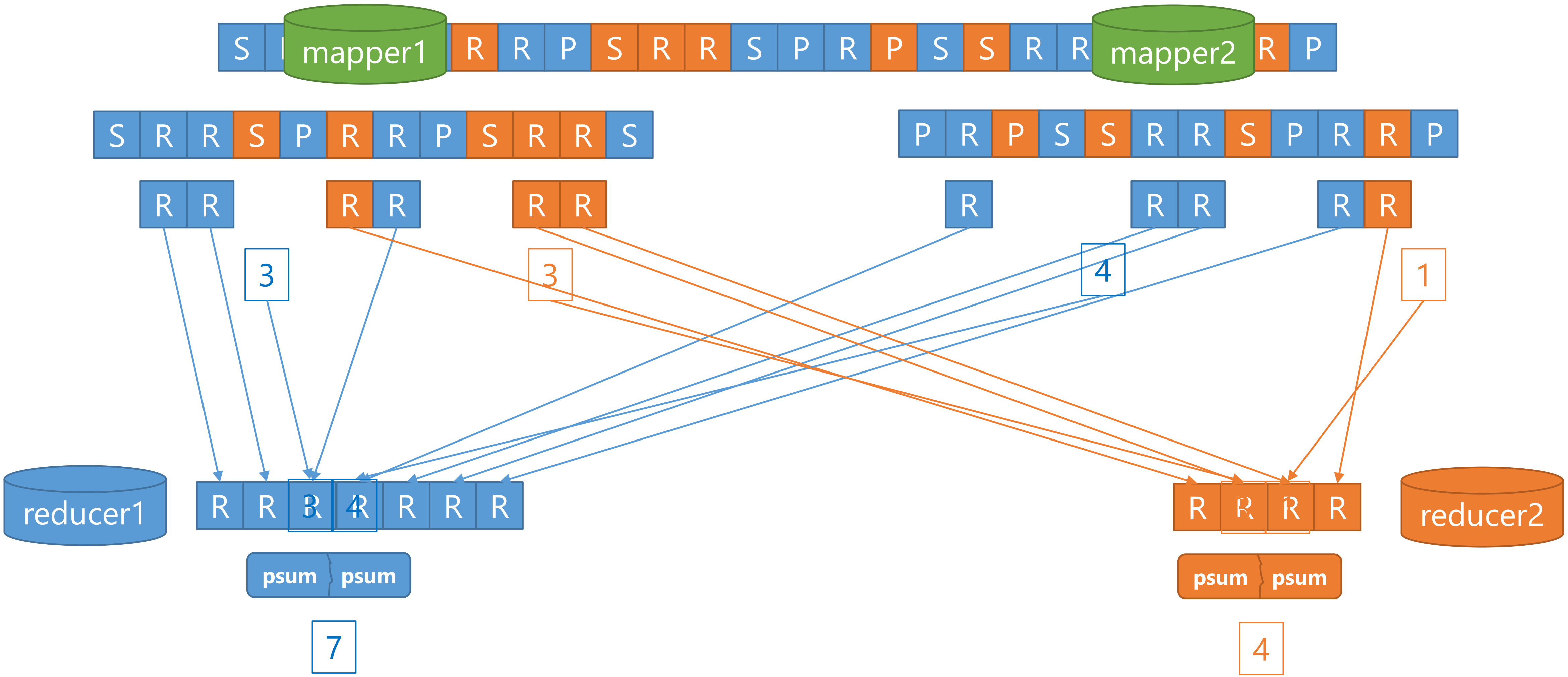
users can:



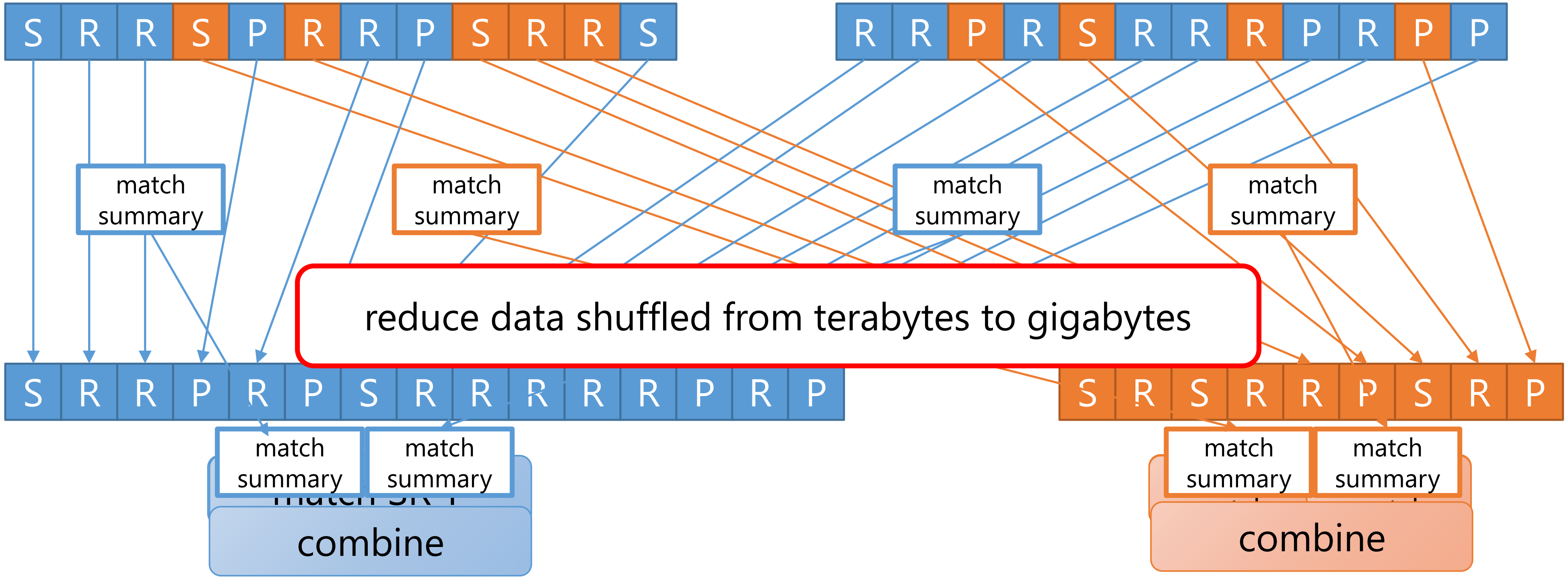
S	search	S
R	review	R
P	purchase	P



Count the number of reviews read per user



Count influential reviews (SR$^+$P) per user



SymPLE [SOSP ‘15]

a language for specifying nonrelational parts of data-processing queries

- a subset of C++

automatically parallelize sequential code

expose additional parallelism to query optimizer

up to 2 orders of magnitude efficiency improvement

Count influential reviews

```
bool search_done = false;
int num_reviews = 0;
int sum = 0;

for each record in input
   switch record.type:
   case SEARCH:    if (!search_done) { num_reviews = 0;
                                       search_done = true; }
   case REVIEW:    num_reviews++;

   case PURCHASE:  if (search_done) { search_done = false;
                                      sum += num_reviews;  }
```

Count influential reviews

```
SymBool search_done = false;
SymInt num_reviews = 0;
SymInt sum = 0;

for each record in input
   switch record.type:
   case SEARCH:    if (!search_done) { num_reviews = 0;
                                       search_done = true; }
   case REVIEW:    num_reviews++;

   case PURCHASE:  if (search_done) { search_done = false;
                                      sum += num_reviews;   }
```

user uses symbolic data types for loop carried state

overloaded operators encode efficient symbolic decision procedures for generating symbolic summaries

Computing max in parallel

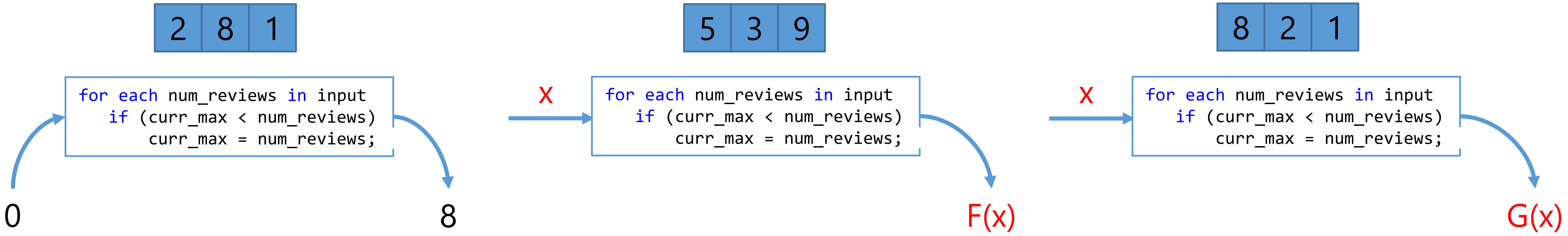
```
SymInt curr_max = 0;

for each num_reviews in input
   if (curr_max < num_reviews)
       curr_max = num_reviews;
```

max is, of course, associative

but this is not apparent from code

SymPLE can parallelize this code

Parallelize by breaking dependences



output = G(F(8))

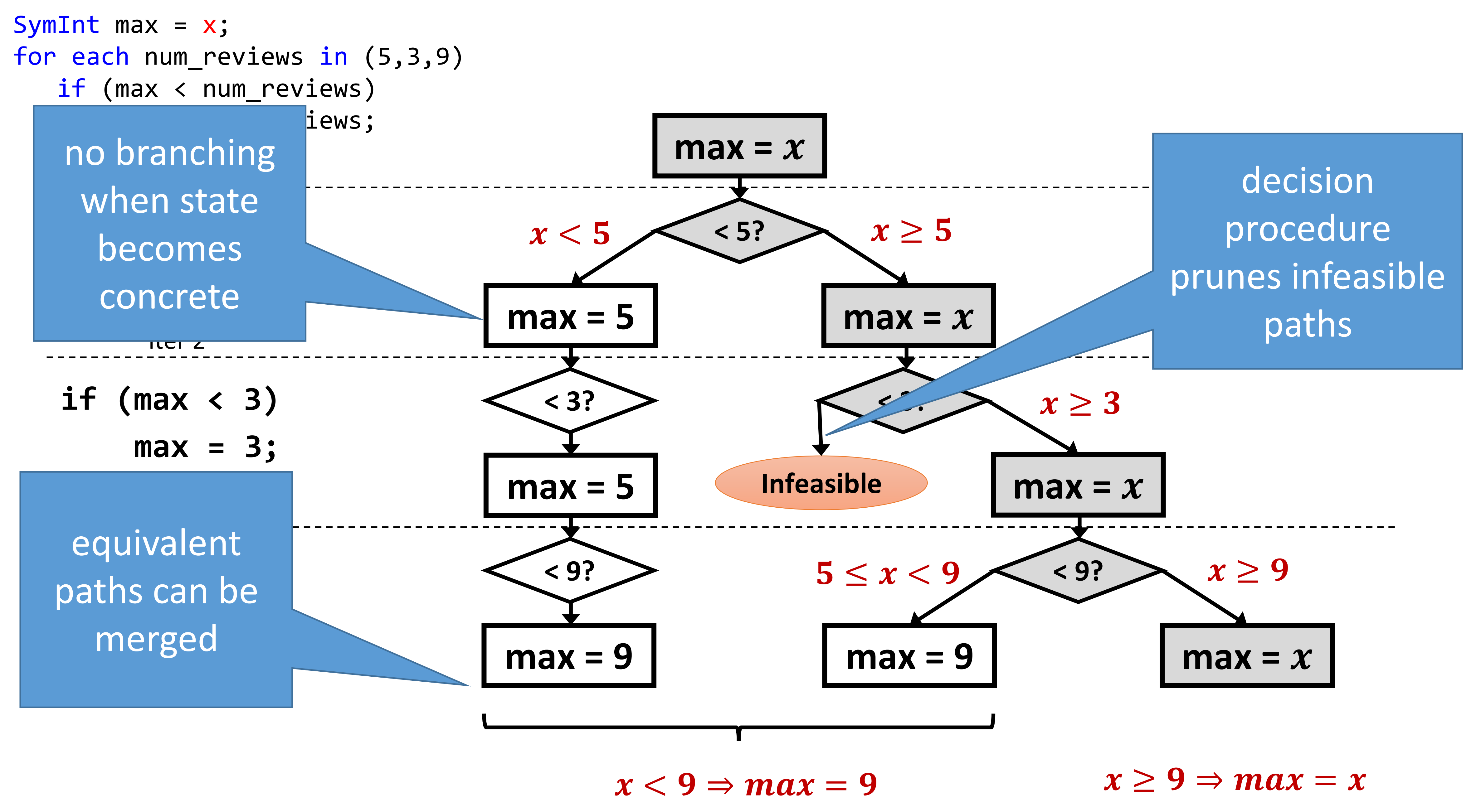
Parallelize by breaking dependences

5	3	9

x

```
for each num_reviews in input
   if (curr_max < num_reviews)
       curr_max = num_reviews;
```

F(x)



SymInt max = x;
for each num_reviews in (5,3,9)
if (max < num_reviews)
iews;
no branching when state becomes concrete
decision procedure prunes infeasible paths
max = x
< 5?
$x < 5$
$x \geq 5$
max = 5
max = x
if (max < 3)
max = 3;
< 3?
$x \geq 3$
max = 5
Infeasible
max = x
equivalent paths can be merged
< 9?
< 9?
$5 \leq x < 9$
$x \geq 9$
max = 9
max = 9
max = x
$x < 9 \Rightarrow max = 9$
$x \geq 9 \Rightarrow max = x$

Parallelize by breaking dependences

2	8	1

```
for each num_reviews in input
   if (curr_max < num_reviews)
      curr_max = num_reviews;
```

0

8

5	3	9

x

```
for each num_reviews in input
   if (curr_max < num_reviews)
      curr_max = num_reviews;
```

$$x < 9 \Rightarrow \text{curr\_max} = 9$$
$$x \geq 9 \Rightarrow \text{curr\_max} = x$$

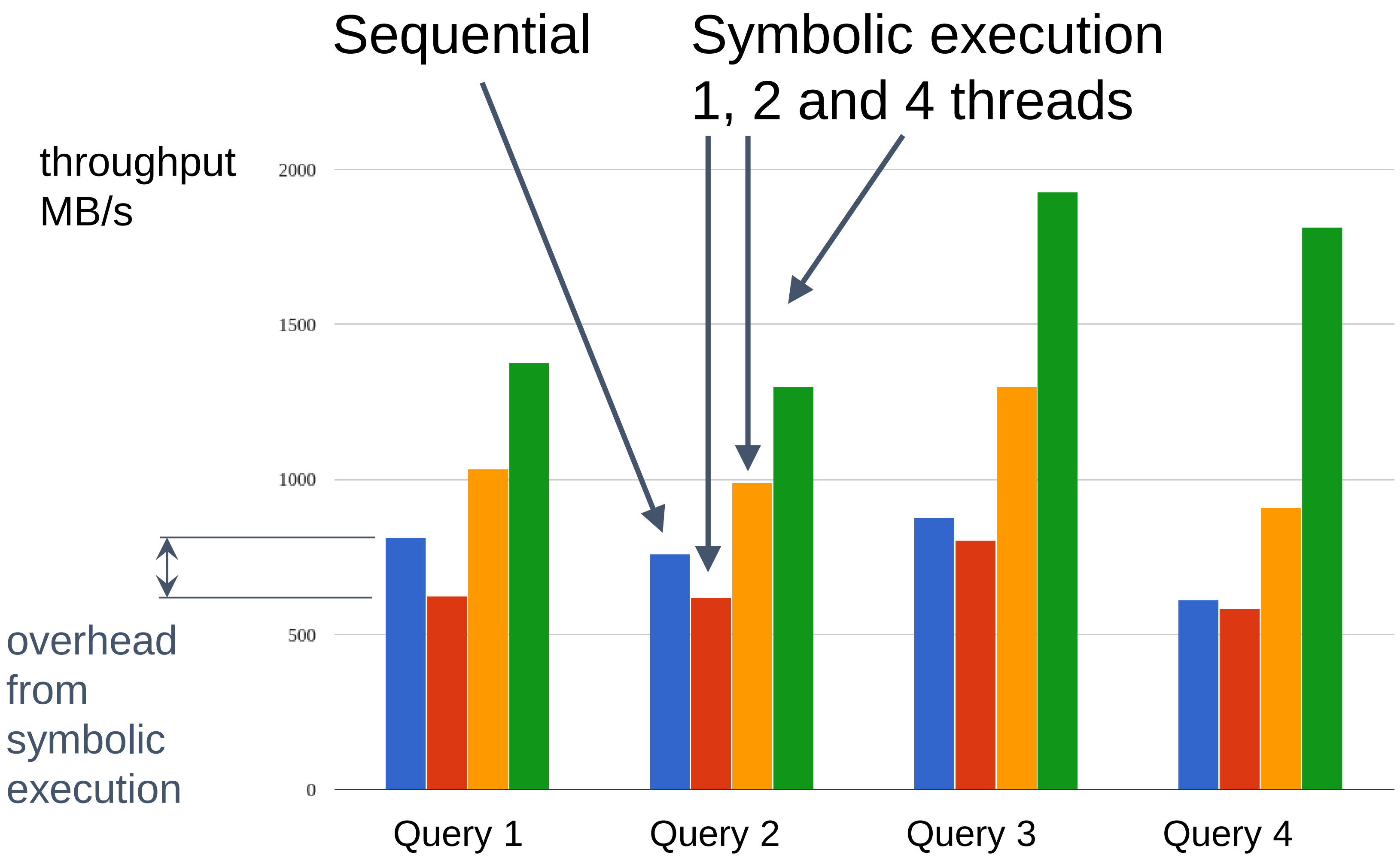
8	2	1

x

```
for each num_reviews in input
   if (curr_max < num_reviews)
      curr_max = num_reviews;
```

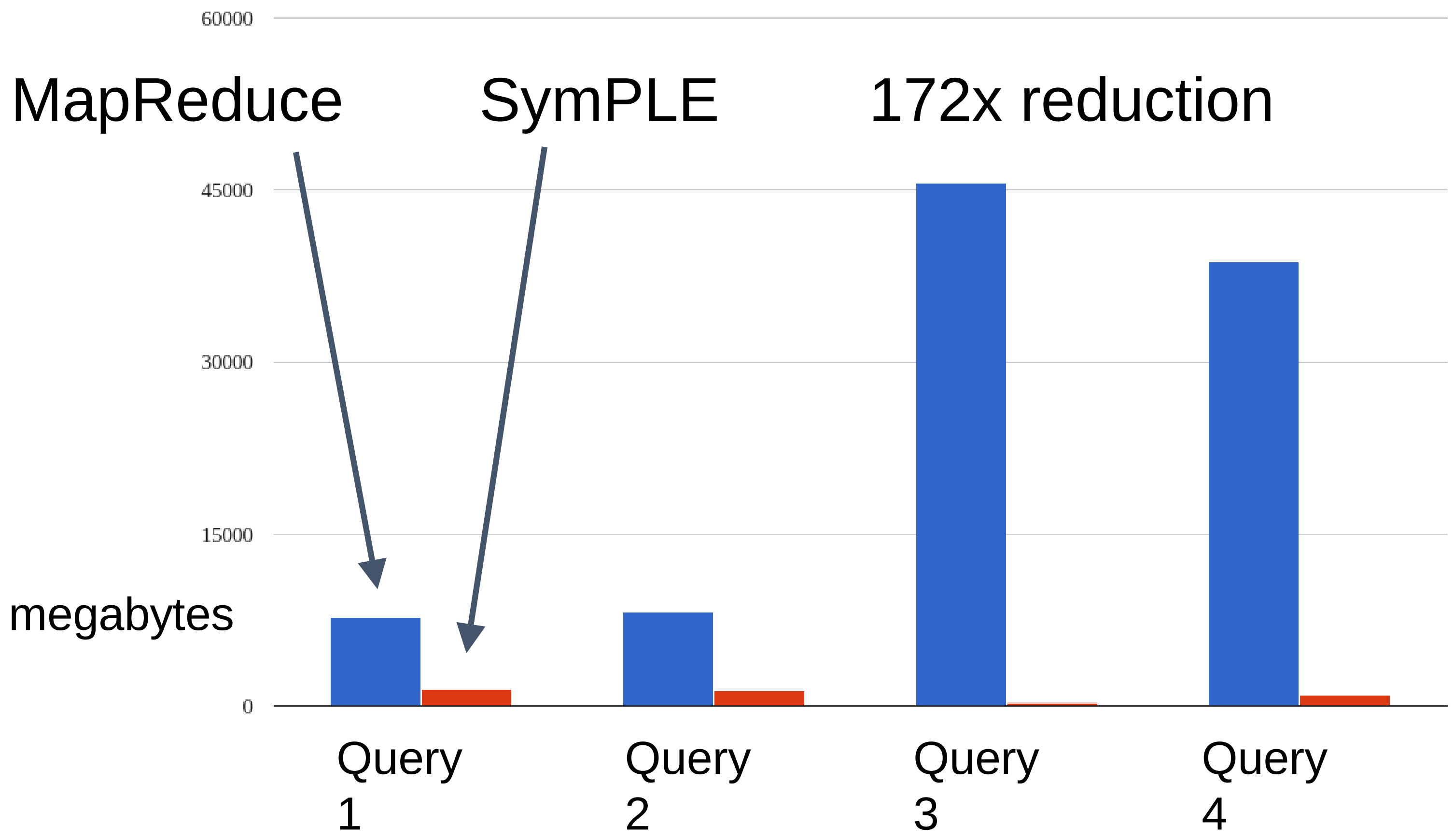
$$x < 8 \Rightarrow \text{curr\_max} = 8$$
$$x \geq 8 \Rightarrow \text{curr\_max} = x$$

Single machine throughput



Reduction in data movement

data shuffled from mappers to reducers



Challenge

can we develop new abstractions for future data-processing needs?

- move beyond embarrassingly parallel
- automatically parallelizable

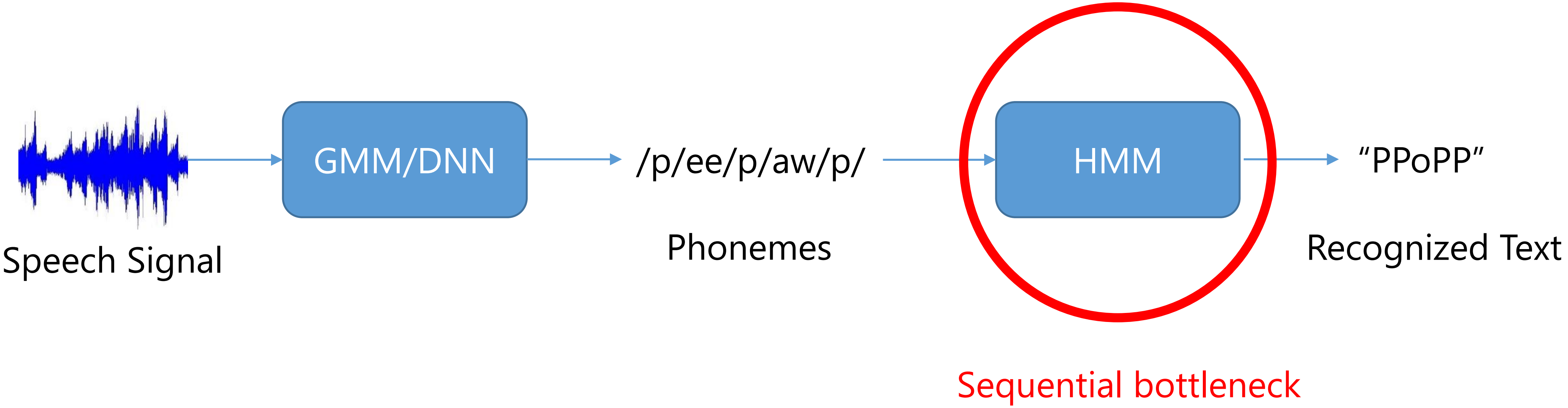
perform whole query optimizations

- unify relational and non-relational parts
- extract filters, project unused parts of data, …

Manual Parallelization Across Dependences

Dynamic Programming

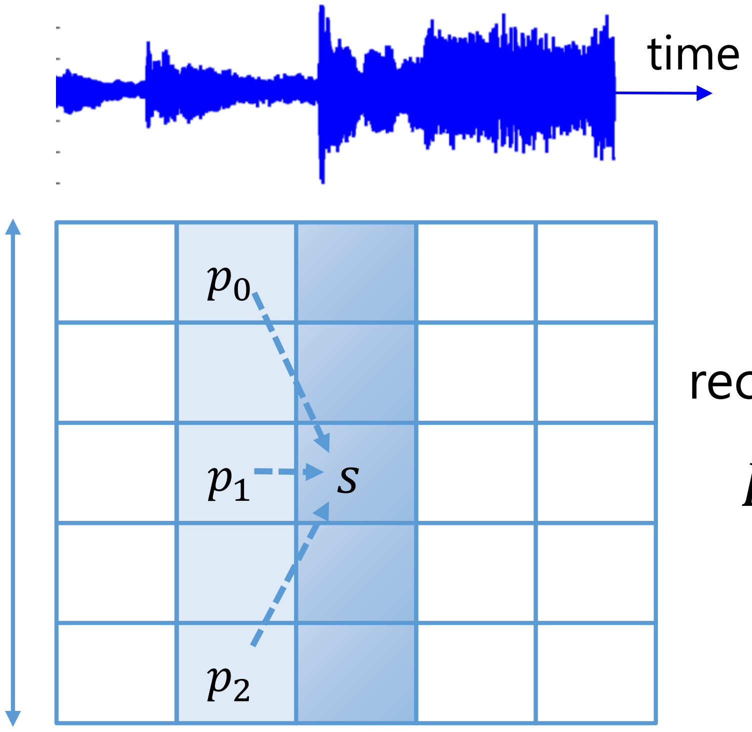
Speech decoders



Viterbi algorithm for Hidden Markov Models (HMM)

finds the most likely sequence of hidden states that explain an observation

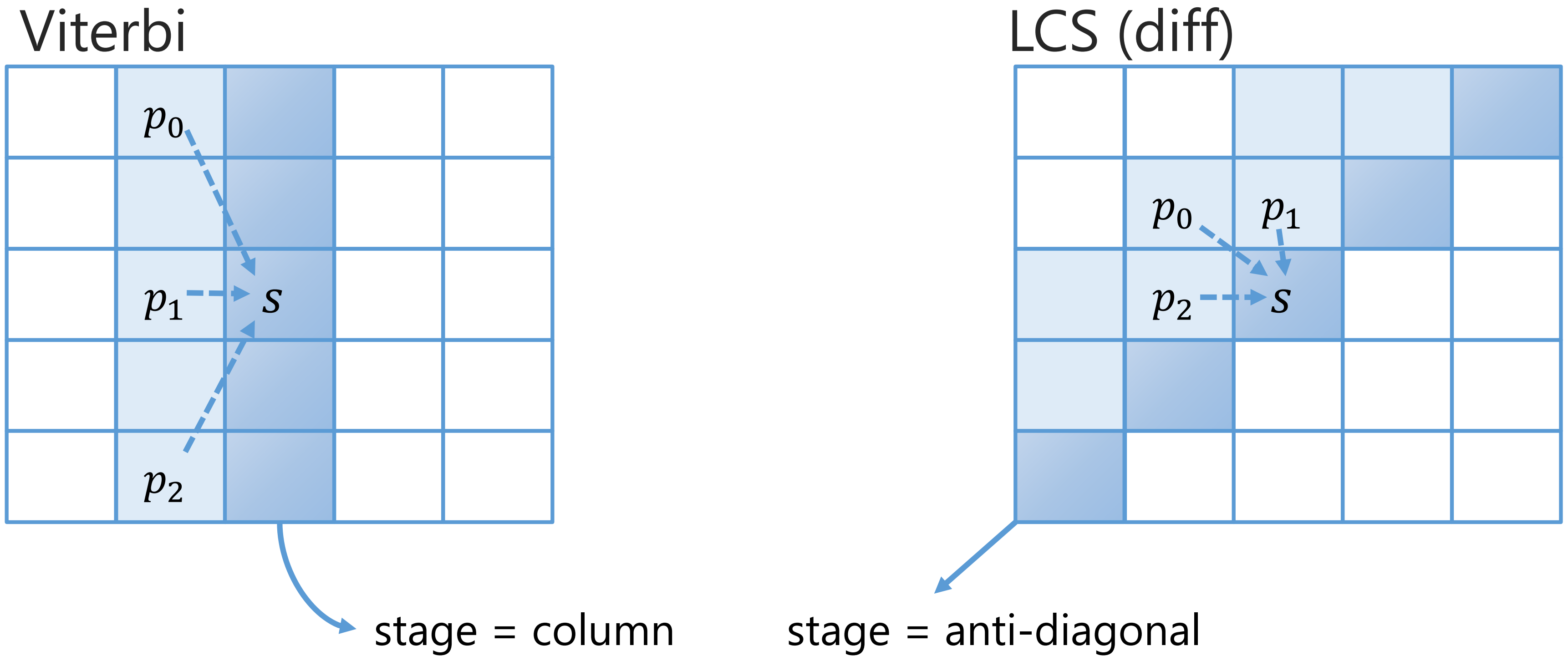
hidden states
=
language model
states



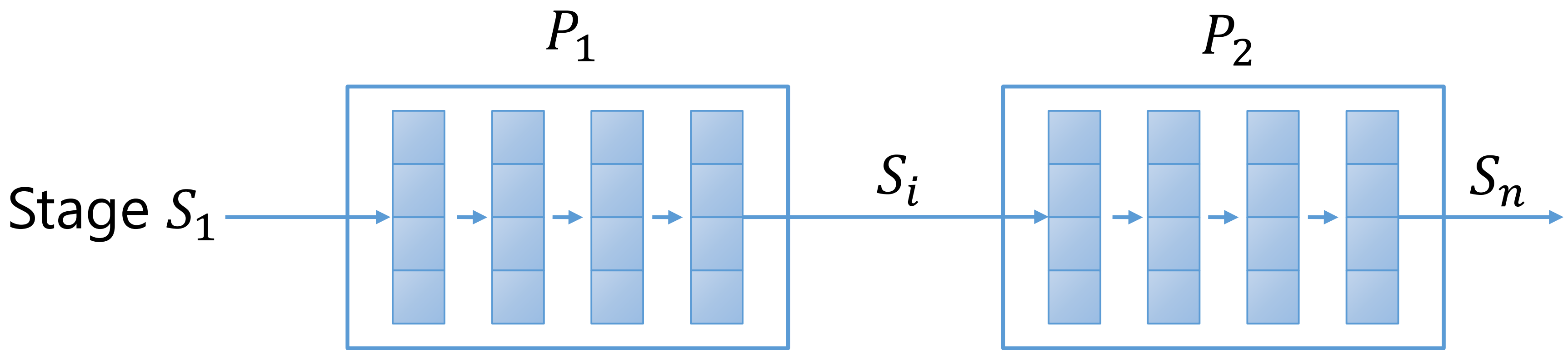
recurrence equation :

$$P_t(s) = \max_{p \in pred(s)} P_{t-1}(p) + TP_t(p \to s)$$

Dynamic programming computes a sequence of stages



Our focus: parallelization across stages

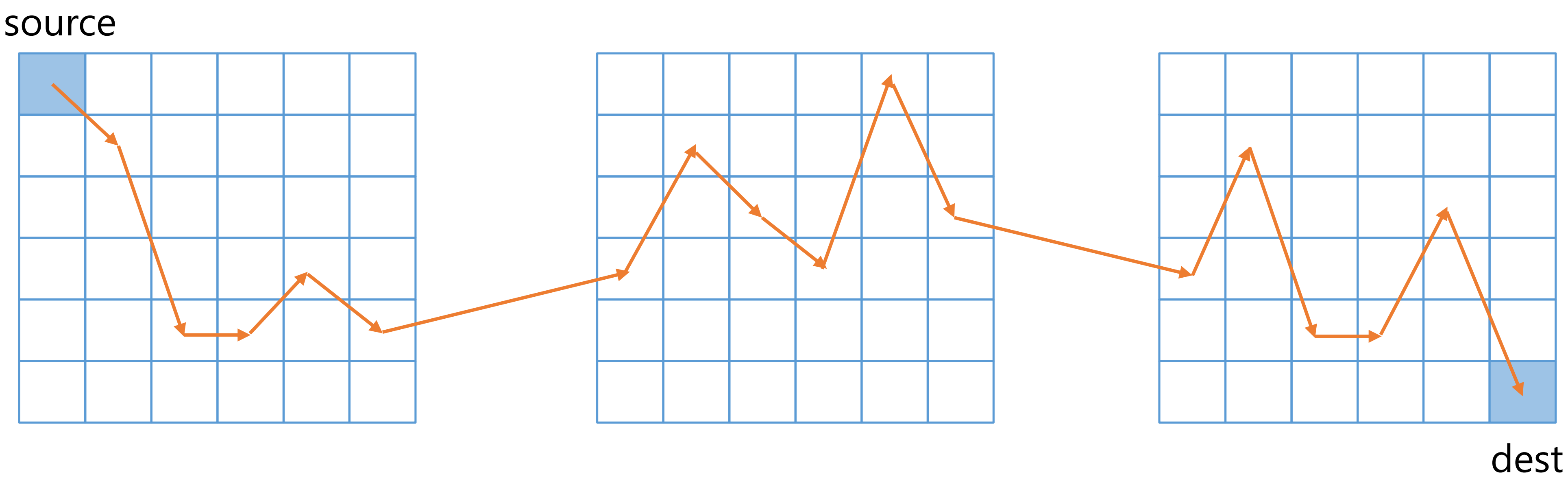


$$S_t[i] = \max_j \left(S_{t-1}[j] + c_{t,i,j}\right)$$

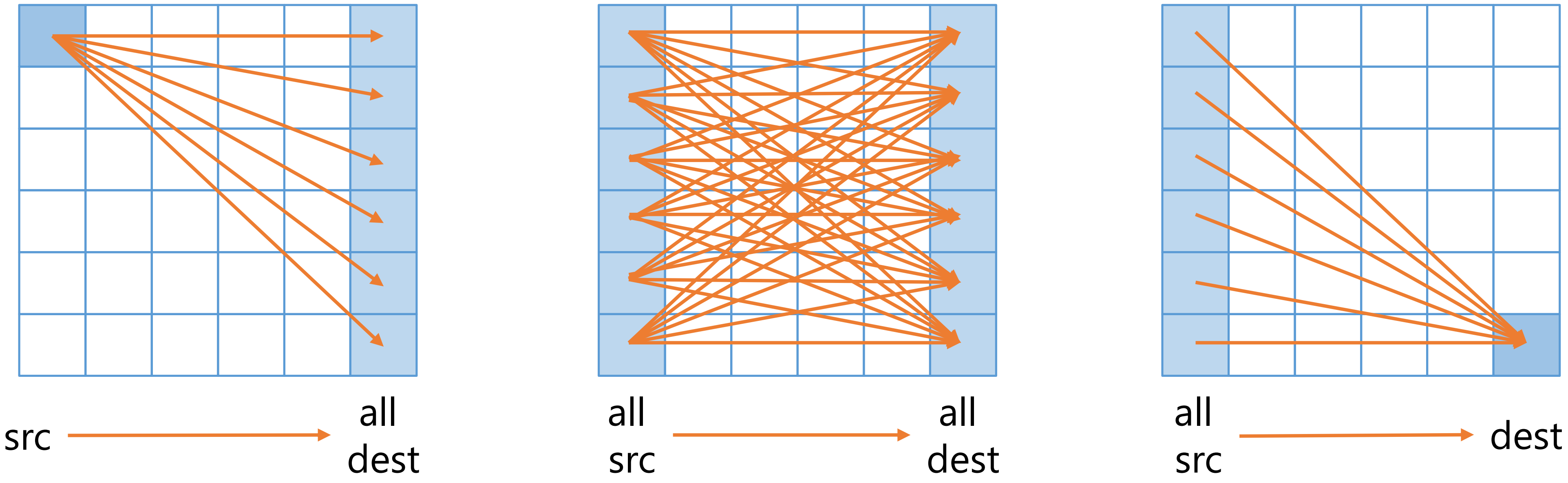
$$\overrightarrow{S_t} = \mathrm{A} \odot \overrightarrow{S_{t-1}}$$

where $\odot$ is matrix multiplication in tropical semiring

Solution in terms of finding shortest-paths

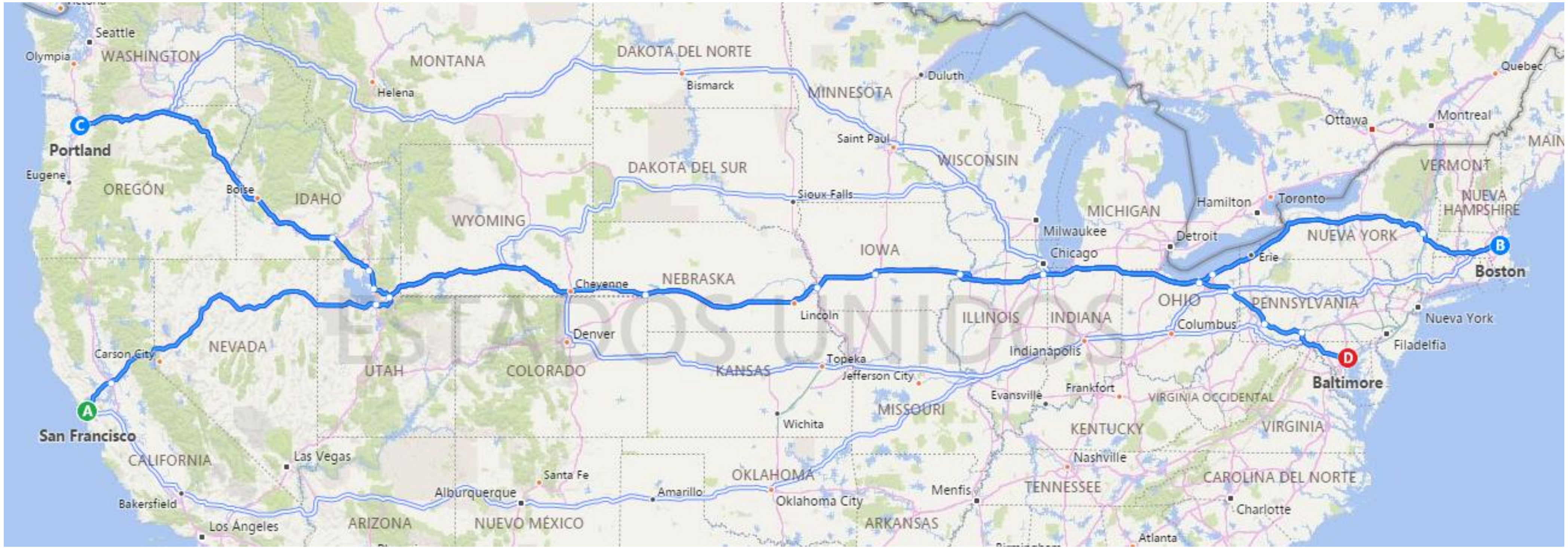


Solution in terms of finding shortest-paths

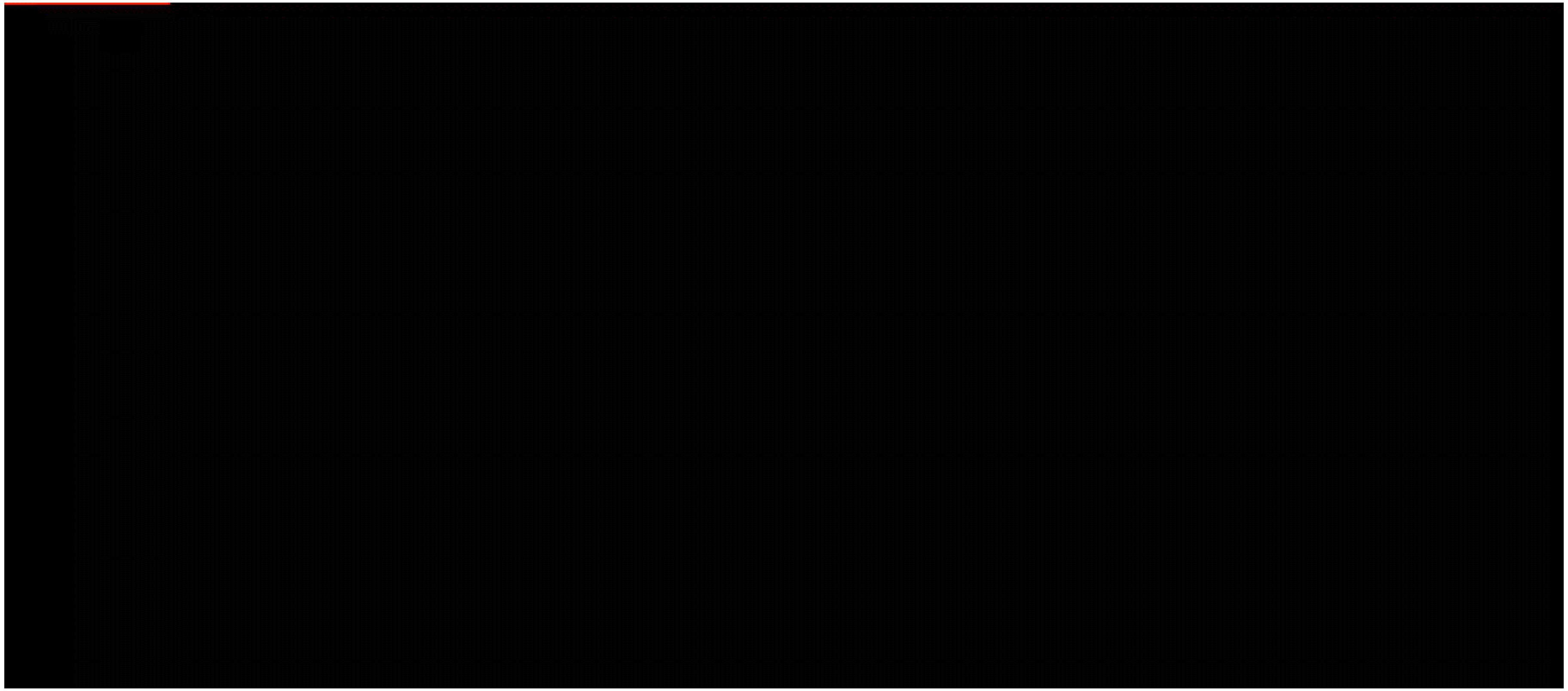


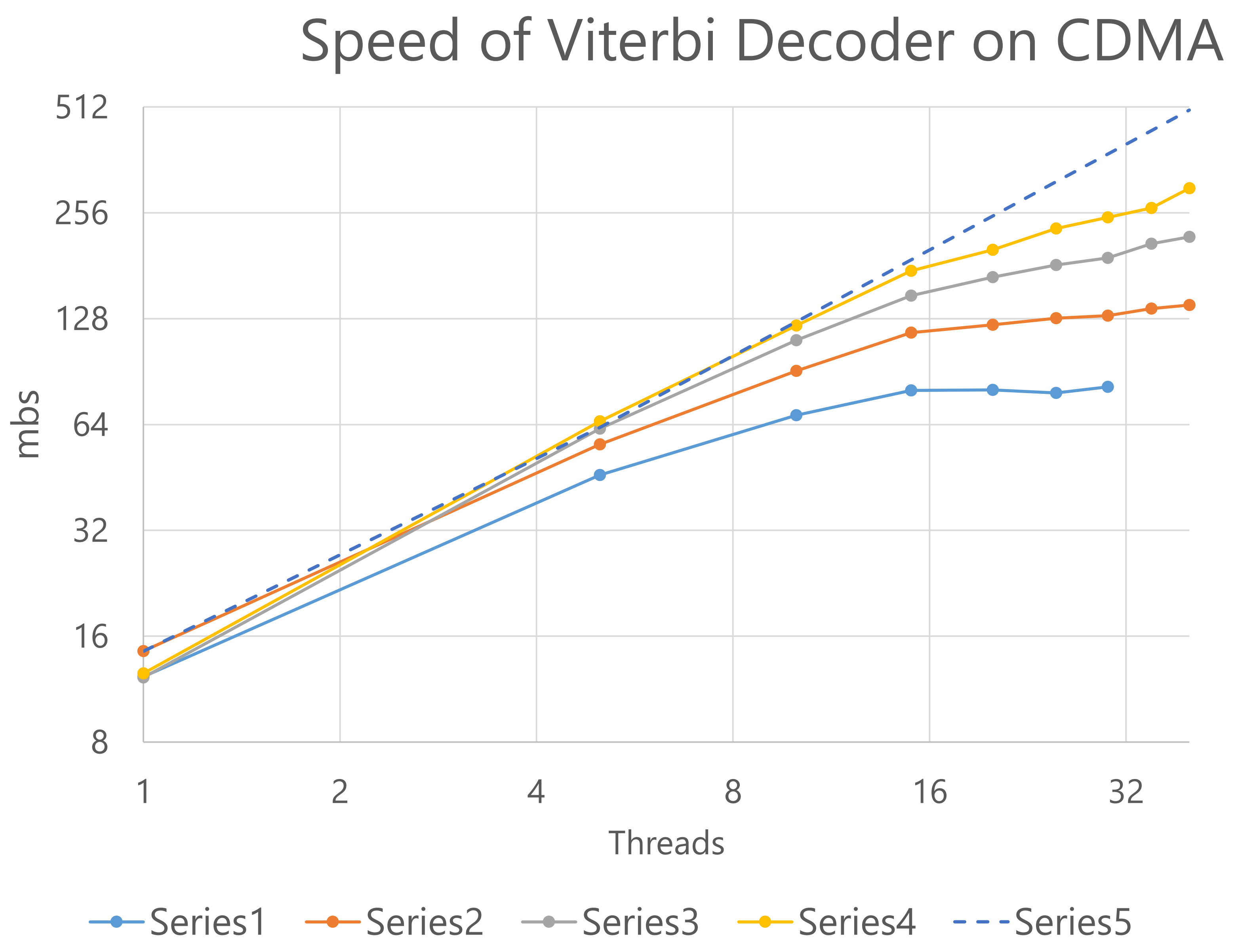
parallelization cost = size of stages

Shortest paths converge to optimal routes



Convergence in LCS





Speed of Viterbi Decoder on CDMA
512
256
128
64
32
16
8
mbs
1
2
4
8
16
32
Threads
Series1
Series2
Series3
Series4
Series5

Summary

“inherently sequential” ⇒ “embarrassingly parallel”