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

```cpp
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; }
```

Influential reviews: $S R^+ P$

Loop carried state
Extracting parallelism from dependent computations

```
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:
// (search_done, num_reviews, sum)

```
(true, 1, 2)
```

```
(false, 0, 0)
```

```
(false, 1, 8)
```
Extracting parallelism from dependent computations

```
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

success depends on
1. fast symbolic execution
2. generation of concise summaries

output = h( g( f( x ) ) )

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 1 of the talk

part 2 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

```sql
select count(*)
from objects
where type = 'square'
group by color
```
Forces pushing beyond relational abstractions

queries today = \textbf{relational skeleton} + \textbf{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 P

users can:

search

review

purchase
Count the number of reviews read per user
Count influential reviews ($SR^+P$) per user

reduce data shuffled from terabytes to gigabytes
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
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; }
Computing max in parallel

```cpp
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

\[
\begin{align*}
\text{for each } & \text{num_reviews in input} \\
   & \text{if (curr_max < num_reviews)} \\
   & \text{curr_max = num_reviews;}
\end{align*}
\]
SymInt max = x;
for each num_reviews in (5,3,9)
    if (max < num_reviews)
        max = num_reviews;

if (max < 5)
    max = 5;
if (max < 3)
    max = 3;
if (max < 9)
    max = 9;

Infeasible

no branching when state becomes concrete

equivalent paths can be merged

decision procedure prunes infeasible paths

$x < 9 \Rightarrow \text{max} = 9$

$x \geq 9 \Rightarrow \text{max} = x$
Parallelize by breaking dependences
Single machine throughput

<table>
<thead>
<tr>
<th>Query</th>
<th>Throughput (MB/s)</th>
<th>Overhead from symbolic execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query 1</td>
<td>1200</td>
<td>500</td>
</tr>
<tr>
<td>Query 2</td>
<td>1500</td>
<td>700</td>
</tr>
<tr>
<td>Query 3</td>
<td>1800</td>
<td>900</td>
</tr>
<tr>
<td>Query 4</td>
<td>2000</td>
<td>1100</td>
</tr>
</tbody>
</table>
Reduction in data movement

Data shuffled from mappers to reducers

MapReduce vs. SymPLE

172x reduction

Megabytes

Query 1
Query 2
Query 3
Query 4
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

Speech Signal → GMM/DNN → /p/ee/p/aw/p/ Phonemes → HMM → “PPoPP” Recognized Text

Sequential bottleneck
Viterbi algorithm for Hidden Markov Models (HMM)

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

recurrence equation:

\[ P_t(s) = \max_{p \in \text{pred}(s)} P_{t-1}(p) + TP_t(p \rightarrow s) \]
Dynamic programming computes a sequence of stages

<table>
<thead>
<tr>
<th>Viterbi</th>
<th>LCS (diff)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_0$</td>
<td>$p_0$</td>
</tr>
<tr>
<td>$p_1$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>stage = column</td>
<td>stage = anti-diagonal</td>
</tr>
</tbody>
</table>

- $V$ iterbi
- LCS (diff)
Our focus: parallelization across stages

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

\[ \overrightarrow{S_t} = A \odot \overrightarrow{S_{t-1}} \]

where \( \odot \) is matrix multiplication in tropical semiring
Solution in terms of finding shortest-paths

source

dest
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

Threads

mbs

Series1
Series2
Series3
Series4
Series5
Summary

“inherently sequential” ⇒ “embarrassingly parallel”