Automating distributed partial aggregation

Presenter: Guangzhe Gao
Background: MapReduce

Map(Key, Value) → Shuffle → Process

- Key1:V11 → V11 → Key1:Result1
- Key2:V21 → V12 → Key2:Result2
- Key1:V12
- Key2:V22 → V21 → V22
Bottleneck

- Network I/O from data nodes to the aggregate nodes.
- Means high delay when shuffling data to one machine
- Can’t avoid if there is aggregation measure: e.g. SUM, COUNT, AVERAGE ...
- How about aggregate what we have and only pass the partial aggregate result to the network?
Solution: aggregate intermediate results, then transmit partial result.

**Motivation**

High level layout
Partial Aggregation

• Why: saves I/O time than aggregate everything finally in one shot.
• Old Approach to change to partial aggregation:
  • Reduce function becomes:
    - Reduce
    - Combine
• Need a way to judge partial aggregation feasibility.
Tool for aggregation optimization

- Automatic verification of partial aggregation applicability
- Change normal reducer code to more efficient scheme (initial reduce, combine, final-reduce)
- Decomposable: distributed programs eligible for partial aggregation.
- Decomposable Properties:
  
  1. Reduce functions loops through records in a group using accumulator.
  2. Initial/final-reduce function is the rearrangement of original reduce function code. Only combine function need to be made.
  3. Reduce function decomposability depends on algebraic properties of combine function and accumulator.

- Claim: the necessary and sufficient condition of reduce function’s decomposability: is determined by accumulator’s commutativity; also, combine function is the inverse function of accumulator.
- Prove and verify (using program) the above claim is the work of the paper.
Combiner Difficulty

- Combining tree may be different, but the result must be same.
- For the graph example below, \( C(C(x_3, x_2), C(x_1, x_4)) \) or \( C(C(x_1, x_4), C(x_2, x_3)) \) should yield same result.

![Combiner Diagram]

- The above right picture is an sample reducer. (emits key and summation plus 10)
- Commutativity of accumulator is verified by making sure for two row sets \( x, y \), \( C(x, y) = C(y, x) \)

```csharp
public IEnumerable<Row> Reduce(RowSet input, Row outputRow) {
    int sum = 0;
    bool isFirst = true;
    foreach (Row row in input.Rows) {
        if (isFirst) {
            row[0].CopyTo(outputRow[0]);
            isFirst = false;
            sum = 10 + row[1].Integer;
        } else sum += row[1].Integer;
    }
    outputRow[1].Set(sum);
    yield return outputRow;
}
```
Combiner based on new Initial Reducer

```java
public Row InitialReduce(RowSet input, Row outputRow) {
    int sum = 0;
    bool isFirst = true;
    foreach (Row row in input.Rows) {
        if (isFirst) {
            row[0].CopyTo(outputRow[0]);
            isFirst = false;
            sum = 10 + row[1].Integer;
        } else sum += row[1].Integer;
    }
    outputRow[1].Set(sum);
    outputRow[2].Set(isFirst);
    return outputRow;
}
```

```java
public Row Combine(Row x, Row y) {
    if (x[2].Boolean) {
        x[0] = y[0];  // for outputRow[0]
        x[1] = y[1];  // for sum
        x[2] = false; // for isFirst
    } else x[1] += y[1].Integer - 10;
    return x;
}
```

```java
public Row FinalReduce(Row x, Row output) {
    output[0] = x[0];
    output[1] = x[1];
    return output;
}
```
Reducer Decomposibility?

• Even if we have commutativity, how does this mean decomposability of reduce function (which means partial aggregatable)?

• Also, above optimization is easy for human, but not for computer. Complex optimization will be infeasible for human.

• Things need to proof:
  
  • **Combiner** exists <-> **accumulator** and reduce function commutative.
  
  • General inverse function of **accumulator** -> construction of **combiner**.
  
  • Both will be proved later on.
Accumulator decomposibility Definition Preliminaries

\[ F(s, \varepsilon) = s \text{ and } F(s, \langle x \rangle \oplus X) = F(F(s, x), X). \]

- Just means accumulator F can do element-wise accumulation.
- If Solution space is defined as \( S = \{ s | s = F(s_0, X), X \in I^* \} \) I is input domain
- We have for all input sequences \( X, Y \in I^* \), then \( F(F(s, X), \hat{Y}) = F(s, X \oplus Y) \)
- Result of reduce function \( R = \text{Results of P(InitialReduce, Combine, and FinalReduce)} \rightarrow R \) is decomposable.
Accumulator decomposibility Definition

**Definition 1** (Decomposability). An accumulator $F$ in reduce function $R$ with the initial solution $s_0$ is decomposable, if and only if there exists a function $C$ such that the following four requirements are satisfied:

1. For any two input sequences $X_1, X_2 \in I^*$,
   \[
   F(s_0, X_1 \oplus X_2) = C(F(s_0, X_1), F(s_0, X_2)).
   \] (1)

2. $F$ is commutative: for any two input sequences $X_1, X_2 \in I^*$, $F(s_0, X_1 \oplus X_2) = F(s_0, X_2 \oplus X_1)$;

3. $C$ is commutative: for any two solutions $s_1, s_2 \in S$, i.e., $C(s_1, s_2) = C(s_2, s_1)$;

4. $C$ is associative: for any three solutions $s_1, s_2, s_3 \in S$, i.e., $C(C(s_1, s_2), s_3) = C(s_1, C(s_2, s_3))$.

We say that $C$ is the decomposed combiner of $F$.

• (1) means accumulator can wave the hand and throw work to combiner. Other parts just follow previous definitions and proof. Note that requirement 2. implies all other requirements (necessary and sufficient). Will be shown later.
Reduction Decomposability $\iff$ Accumulator commutativity

**Theorem 1 (Informally).** Reducer $R$ is decomposable if and only if the corresponding accumulator $F$ is commutative. The decomposed combiner $C$ is uniquely determined by $F$.

- The author only gives proof outline in this paper.
- First proof is to show: **Lemma 1.** Given a combiner $C$ that satisfies Equation 1, $C$ is commutative and associative if and only if accumulator $F$ is commutative.

\[
F(s_0, X_1 \oplus X_2) = C(F(s_0, X_1), F(s_0, X_2)). \tag{1}
\]

- Recall Definition 1, condition 2: $F$ is commutative: for any two input sequences $X_1, X_2 \in I^*$, $F(s_0, X_1 \oplus X_2) = F(s_0, X_2 \oplus X_1)$;
- consider input as single element rather than sequence, we have $C$ commutative. Similar reasoning for associative.
Lemma 2. \( F \) is commutative if and only if for any solutions \( s \in S \), and any two input values \( x, y \in I \), \( F(s, xy) = F(s, yx) \).

- Lamma 2 is more like definition so not proof given.

- \( F \) is commutative: for any two input sequences \( X_1, X_2 \in I^* \), \( F(s_0, X_1 \oplus X_2) = F(s_0, X_2 \oplus X_1) \); implies exist combiner \( C \) satisfies

\[
F(s_0, X_1 \oplus X_2) = C(F(s_0, X_1), F(s_0, X_2)).
\]  \( \text{(1)} \)

- Proof: given \( s_1 = F(s_0, X) \), \( s_2 = F(s_0, Y) \) \( C(s_1, s_2) \) can be defined as \( \tilde{F}(s_0, X \oplus Y) = F(s_1, Y) \)

- How to do mapping?
General Inverse Function

- Inverse function $\mathcal{H}_F = \{ H | \forall s \in S. F(s_0, H(s)) = s \}$
- H need not to be one-on-one according to the construction.
- We define derived combiner $C_H(s_1, s_2) = F(s_1, H(s_2))$ for each H.
- Now need to show if F commutative, derived containers produce same result.
- The intuition in paper: If same aggregate output for two input sequences with initial solution value,
- The two input sequences can always generate same output for arbitrary solution value.
Lemma 3. Given an accumulator $F$ that is commutative and two input sequences $X, Y \in I^*$, if there is a $s_0$ such that $F(s_0, X) = F(s_0, Y)$, then $F(s, X) = F(s, Y)$ holds true for any $s \in S$.

\[ F(s, X) = F(F(s_0, Z), X) = F(s_0, Z \oplus X) \]
\[ = F(s_0, X \oplus Z) = F(F(s_0, X) \oplus Z) = F(F(s_0, Y) \oplus Z) \]
\[ = F(s_0, Y \oplus Z) = F(s_0, Z \oplus Y) = F(F(s_0, Z), Y) = F(s, Y) \]

• Finally: the proof is done: for any commutative accumulator $F$, the combiner $C$ can be generated using any general inverse function $H$ of $F$.

• Formal restatement of everything:

**Lemma 4.** If an accumulator $F$ is commutative, then for any $H, H' \in \mathcal{H}_F$, $C_H \equiv C_{H'}$.

**Theorem 2.** Reducer $R$ is decomposable if and only if the corresponding accumulator $F$ is commutative. The decomposed combiner $C$ is uniquely determined by $F$, and takes the form $C(s_1, s_2) = F(s_1, H(s_2))$ where $H$ is any inverse function of $F$. 
Decomposability Verification

• From above lemmas and theorems, only need to show $F(s, xy) = F(s, yx)$ for all $s, x, y$.

• But $F$ is a program?

• Parse the language code.

```
F ::= f[x, ..., x].F, ..., F; s; return e, ..., e
e ::= x | e op a e | n
s ::= x := e | x, ..., x := f(e, ..., e) | s; s
    | if (p) then s else s | skip
p ::= e op r e | true | false
```

• $f$: function name, $F, ..., F$: nested function definition,

• $e$: expression (variable $x$, constant $n$, binary operator)

| $F_l = F(s, xy)$ | $f_l[s, x, y].F, s_1 := f(s, x); s_2 := f(s_1, y); return s_2$ |
| $F_r = F(s, yx)$ | $f_r[s, x, y].F, s_1 := f(s, y); s_2 := f(s_1, x); return s_2$ |
Path Formula

\[ \phi_{F}^{o_1, \ldots, o_n} = \bigvee \left( \bigwedge_{i \in I} p_{ij} \wedge \bigwedge_{j=1}^{n} o_j = e_{ij} \right) \]

Index Set
(of input
and output)

Path Formula is true if \{X_1 \ldots X_n, O_1 \ldots O_m\} evaluate to true. Meaning \( F(X_1 \ldots X_n) = O_1 \ldots O_m \)
Convert program to formulae

- Symbolic execution.
- Look a lot like compiler parsers. If interested refer to extra slides.
- Now decomposability verification becomes SMT satisfiability problem:

\[ \forall x,y. F(s, xy) = F(s, yx) \text{ if and only if } \phi_{F_1, \ldots, F_n} \land \phi_{F_1', \ldots, F_n'} \land (\bigvee_{i=1}^{n} o_i \neq o'_i) \text{ is not satisfiable.} \]

- i.e. not output different case.

\[
F_{acc} = acc[row_0, sum, isFirst, x_0, x_1].
\]

\[
\text{if (isFirst = 1) then } \{
row_0 := x_0;
isFirst := 0;
sum := 10 + x_1;
\}
\text{else } \{
sum := sum + x_1;
\};
\text{return row}_0, sum, isFirst
\]

Separate by cases (isFirst) and output
• Given decomposable accumulator $F$, combiner $C$ that satisfies $C_H(s_1, s_2) = F(s_1, H(s_2))$ is difficult to construct.

• $H$ is general inverse function of $F$.

• Greedy approach:

  • most accumulators of decomposable reducer are in three categories:

  1. **Counting** aggregation over an input sequence that is only determined by the length of the sequence;
  2. **State machine** aggregation that essentially simulates a state machine with a limited number of states; and
  3. **Single input** aggregation over an input sequence that can be simulated by aggregating over one input record.
Counting category

**Definition 2** (Counting category). An accumulator $F$ belongs to the counting category if and only if, $F(s_0, X) = F(s_0, Y)$ holds for any two input sequences $X, Y \in I^*$, $|X| = |Y|$.  

- In other words only size matters

**Lemma 5.** An commutative accumulator $F$ belongs to the counting category if and only if, for any two input records $x, y \in I$, $F(s_0, x) = F(s_0, y)$ holds true.

```c
input(s1, s2);
s = s0; r = s1;
while (s<s2) {
    s = F(s, 0);
    r = F(r, 0);
}
return r;
```

r accumulates $|s2| - |s0|$ zeros, so set H be of this number zeros
State Machine category

- Finite state machine defined by accumulator?
- BFS with depth(states #) threshold T.
- Explore all possible states and store transition table in Sol, so H can computed based on lookup
- C(S1, S2) has T^2 combinations

\[ F_{sm} = sm[s, x]. \]
if \((s = -1)\) then
  if \((x > 100)\) then \(s := 0;\) else \(s := 2;\)
else if \((s = 0)\) then
  if \((x > 100)\) then \(s := 0;\) else \(s := 1;\)
else if \((s = 2)\) then
  if \((x > 100)\) then \(s := 1;\) else \(s := 2;\)
return \(s\)

Sol:
(0, -1, 101), (2, -1, 100), and (1, 0, 100).

Combiner

```
input(s1, s2)
if (s1 == -1 and s2 == -1) then return -1;
...
else if (s1 == 2 and s2 == 0) then return 1;
...```
**Lemma 6.** An accumulator $F$ belongs to single input category if and only if, for any two input records $x, y \in I$, there exists an input record $z$ such that $F(s_0, xy) = F(s_0, z)$.

- Novel idea: Eliminate the need of $z$.
- Can eliminate if satisfies the partial function that requires $z$.

\[ \forall x_0, x_1, y_0, y_1. \exists z_0, z_1. z_0 = x_0 \land 10 + z_1 = 10 + x_1 + y_1 \land 0 = 0 \]

- Can be rewritten as:

\[ \forall x_0, x_1, y_0, y_1. \exists z_0, z_1. z_0 = x_0 \land z_1 = 10 + x_1 + y_1 - 10 \land 0 = 0 \]

- Two $Z$'s in the left, so requirement is:

\[ \forall x_0, x_1, y_0, y_1. 0 = 0 \]
Evaluation

- It is a prototype

- Total jobs 4,429
- Baseline of auto partial aggregation: 183 Jobs use partial aggregation, 28 of them (15.3%) may be incorrect.
- Remaining 4246 jobs, 261 are true positives. (manually checked)

- Performance gain (in reduction):
  - Time (61.6%) 165 sec -> 64 sec
  - Space: (99.98%) 7.99GB -> 1.22MB
  - 62.4% latency, 76% in network IO
Time cost and failure (not very useful)

<table>
<thead>
<tr>
<th>#</th>
<th>∀sxy</th>
<th>∀xy</th>
<th>#</th>
<th>∀sxy</th>
<th>∀xy</th>
<th>#</th>
<th>∀sxy</th>
<th>∀xy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.08</td>
<td>0.04</td>
<td>7</td>
<td>0.31</td>
<td>0.29</td>
<td>13</td>
<td>0.17</td>
<td>0.07</td>
</tr>
<tr>
<td>2</td>
<td>0.11</td>
<td>0.04</td>
<td>8</td>
<td>2.54</td>
<td>0.31</td>
<td>14</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
<td>0.06</td>
<td>9</td>
<td>0.06</td>
<td>0.03</td>
<td>15</td>
<td>0.05</td>
<td>0.02</td>
</tr>
<tr>
<td>4</td>
<td>0.20</td>
<td>0.10</td>
<td>10</td>
<td>0.05</td>
<td>0.02</td>
<td>16</td>
<td>0.05</td>
<td>0.02</td>
</tr>
<tr>
<td>5</td>
<td>0.09</td>
<td>0.04</td>
<td>11</td>
<td>0.02</td>
<td>0.02</td>
<td>17</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>6</td>
<td>0.10</td>
<td>0.08</td>
<td>12</td>
<td>0.08</td>
<td>0.04</td>
<td>18</td>
<td>0.17</td>
<td>0.17</td>
</tr>
</tbody>
</table>

Table 1. Performance of our prototype’s decomposability verification. The ∀sxy and ∀xy columns show running time to verify the decomposability using ∀sxy.F(s, xy) = F(s, xy) and ∀xy.F(s0, xy) = F(s0, xy) respectively. All times are reported in seconds with stars meaning that verification failed.

<table>
<thead>
<tr>
<th>#</th>
<th>Type</th>
<th>Time</th>
<th>#</th>
<th>Type</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>0.03</td>
<td>7</td>
<td>SI</td>
<td>1.10</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>0.04</td>
<td>8</td>
<td>SM</td>
<td>0.77</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0.06</td>
<td>9</td>
<td>C</td>
<td>0.02</td>
</tr>
<tr>
<td>4</td>
<td>C+SI</td>
<td>0.31</td>
<td>10</td>
<td>C</td>
<td>0.02</td>
</tr>
<tr>
<td>5</td>
<td>SI</td>
<td>0.08</td>
<td>11</td>
<td>SI</td>
<td>0.05</td>
</tr>
<tr>
<td>6</td>
<td>C+SI</td>
<td>0.09</td>
<td>12</td>
<td>C</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 2. Performance of our prototype’s combiner synthesis. The type column shows which kinds of techniques are used to synthesize the combiner; the time column shows the running time in seconds, including both the times to check technique validity and to generate combiner code; and the stars after Reducer 21 and Reducer 22 mean that combiner synthesis failed.
Thank you!
• Symbolic Execution:

\[
\begin{align*}
\sigma(e) &= e \\
S-Var : &\quad x \in \text{Vars} \\
\sigma(x) &= \sigma(x) \\
S-Cnst : &\quad \sigma(n) = n \\
\sigma(\ell_i) &= \ell_i', i = 1, 2 \\
S-Aop : &\quad e = \ell_1 \sigma_p, \ell_2' \\
\sigma(\ell_1 \sigma_p, \ell_2) &= e \\
S-TrPlx : &\quad v = \text{true or } v = \text{false} \\
\sigma(v) &= v \\
\sigma(\ell_i) &= \ell_i', i = 1, 2 \\
S-Rep : &\quad \phi = \ell_1 \sigma_p, \ell_2' \\
\sigma(\ell_1 \sigma_p, \ell_2) &= \phi
\end{align*}
\]

\[
\begin{align*}
\Sigma \vdash (\mathcal{M}, s) &\rightarrow \mathcal{M}' \\
S-Seq : &\quad \Sigma \vdash (\mathcal{M}, s_1) \rightarrow \mathcal{M}' \quad \Sigma \vdash (\mathcal{M}', s_2) \rightarrow \mathcal{M}'' \\
&\quad \Sigma \vdash (\mathcal{M}, s_1; s_2) \rightarrow \mathcal{M}'' \\
S-Assign : &\quad \mathcal{M}' = \{(\phi, \sigma'), (\phi, \sigma) \in \mathcal{M} \land \sigma' = \sigma[x \mapsto \sigma(e)]\} \\
&\quad \Sigma \vdash (\mathcal{M}, x := e) \rightarrow \mathcal{M}' \\
S-Skip : &\quad \Sigma \vdash (\mathcal{M}, \text{skip}) \rightarrow \mathcal{M} \\
S-If : &\quad \Sigma \vdash (\mathcal{M}, \text{if}(p) \text{then } s_1 \text{ else } s_2) \rightarrow \mathcal{M}' \\
&\quad \Sigma \vdash (\mathcal{M}, s_1) \rightarrow \mathcal{M}_i, i = 1, 2 \\
&\quad \mathcal{M}' = \{(\phi \land \sigma(p), \sigma) | (\phi, \sigma) \in \mathcal{M}_1\} \\
&\quad \cup \{(\phi \land \sigma(\lnot p), \sigma) | (\phi, \sigma) \in \mathcal{M}_2\} \\
&\quad Y^* = \{(\text{name}(F_j) \rightarrow F_j | j = 1, \ldots, m)\} \\
&\quad \Sigma^* \vdash (\mathcal{M}^*, s) \rightarrow \mathcal{M}^{**} \\
&\quad \mathcal{M}' = \{(\phi \land \phi_1, \sigma_1') | (\phi, \sigma_1) \in \mathcal{M} \land (\phi_2, \sigma_2) \in \mathcal{M}^{**} \\
&\quad \land \sigma' = \sigma_1[x \mapsto \sigma_2(e)]\} \\
S-Proc : &\quad \Sigma \vdash (\mathcal{M}, x_1, \ldots, x_n := f(e_1, \ldots, e_m)) \rightarrow \mathcal{M}'
\end{align*}
\]

Figure 6. Symbolic Execution for Path Formula

**Figure 5.** Operational Semantics