Generate, Test, and Aggregate
A Calculational Framework for Programming with MapReduce

Zhenjiang Hu

National Institute of Informatics, Japan

February 7, 2011
(Joint Work with Kento Emoto, Sebastian Fischer)
The Knapsack Problem

*Fill a knapsack with items, each of certain value and weight. Maximize total value of packed items without exceeding weight restriction of knapsack.*
The Knapsack Problem

Fill a knapsack with items, each of certain value and weight. Maximize total value of packed items without exceeding weight restriction of knapsack.

... in parallel!
The Knapsack Problem

Fill a knapsack with items, each of certain value and weight. Maximize total value of packed items without exceeding weight restriction of knapsack.

... in parallel!

... with MapReduce!
The Knapsack Problem, in parallel

5 kg max
[...,(2000, 1), (3000, 3), ..., (4000, 3), ...]
The Knapsack Problem, in parallel

5 kg max
[... (2000, 1), (3000, 3), ... (4000, 3), ...

Divide.

[... (2000, 1), (3000, 3), ...] [... (4000, 3), ...]
The Knapsack Problem, in parallel

5 kg max
[...,(2000, 1), (3000, 3), ..., (4000, 3), ...]

Divide.

[...,(2000, 1), (3000, 3), ...]  [...,(4000, 3), ...]

Conquer.

(2000, 1), (3000, 3)  (4000, 3)
The Knapsack Problem, in parallel

5 kg max

[...,(2000, 1), (3000, 3), ...,(4000, 3), ...]

Divide.

[...,(2000, 1), (3000, 3), ...] [...,(4000, 3), ...]

Conquer.

(2000, 1), (3000, 3) (4000, 3)

Combine.

???
The Knapsack Problem, summarized

- naïve divide and conquer does not work
- unclear how to compute \textit{knapsack} function in parallel
The Knapsack Problem, summarized

- naïve divide and conquer does not work
- unclear how to compute \textit{knapsack} function in parallel

\textit{very common in practice!}
The Knapsack Problem, summarized

- naïve divide and conquer does not work
- unclear how to compute \textit{knapsack} function in parallel

\textbf{very common in practice!}

- many smart hard to understand algorithms
- for \textit{specific} problems
- often resemble each other
- yet, no general methodology!
aggregate \circ \textit{filter} \ test \circ \textit{generate}
Generate, Test, and Aggregate (GTA)

**Generator**
- generates solution candidates
- e.g., all sublists of items to put in knapsack

**Test**
- filters admissible candidates
- e.g., items with total weight ≤ 5kg

**Aggregator**
- computes summary of solutions
- e.g., maximum of all total values
GTA as a Parallel Programming Framework

- general methodology for parallel programming
- supports wide class of problems
- programs are efficient automatically
  - can be developed modularly
  - and extended incrementally
GTA as a Parallel Programming Framework

- general methodology for parallel programming
- supports wide class of problems
- programs are efficient automatically
  - can be developed modularly
  - and extended incrementally

by imposing reasonable constraints on G, T, A
Requirements on G, T and A

Generator: \([a] \rightarrow \mathcal{L}[a]\)
- in D&C form: list homomorphism
- e.g., subs, segs, inits, tails, perms, parts

Test: \([a] \rightarrow \text{Bool}\)
- in D&C form: almost list homomorphism
- e.g., allEvens, increasing, noEqElements

Aggregator: \(\mathcal{L}[a] \rightarrow R\)
- a semiring homomorphism
- e.g., maxSum, firstSolution, allSolutions
Outline

1. GTA: Generate, Test, Aggregate
2. GTA Specification
3. Parallelization of GTA
4. Conclusion
List Homomorphism

A function \( h : [a] \rightarrow M \) is a list homomorphism if it is defined in the following form:

\[
\begin{align*}
    h [] &= e \\
    h [x] &= f \cdot x \\
    h (xs \oplus ys) &= h \cdot xs \oplus h \cdot ys
\end{align*}
\]

where \((R, \oplus, e)\) forms a monoid, and \(f\) is a given function.
List Homomorphism

A function \( h : [a] \rightarrow M \) is a list homomorphism if it is defined in the following form:

\[
\begin{align*}
   h \left[ \right] & = e \\
   h \left[ x \right] & = f \times \\
   h \left( xs \oplus ys \right) & = h \times xs \oplus h \times ys
\end{align*}
\]

where \((R, \oplus, e)\) forms a monoid, and \(f\) is a given function.

\[
\begin{align*}
   sum \left[ \right] & = 0 \\
   sum \left[ n \right] & = n \\
   sum \left( ms \oplus ns \right) & = sum \times ms + sum \times ns
\end{align*}
\]
List Homomorphism

A function $h : [a] \rightarrow M$ is a list homomorphism if it is defined in the following form:

\[
\begin{align*}
    h [\ ] &= e \\
    h [x] &= f \ x \\
    h (xs \oplus ys) &= h xs \oplus h ys
\end{align*}
\]

where $(R, \oplus, e)$ forms a monoid, and $f$ is a given function.

\[
\begin{align*}
    \text{sum} [\ ] &= 0 \\
    \text{sum} [n] &= n \\
    \text{sum} (ms \oplus ns) &= \text{sum} ms + \text{sum} ns
\end{align*}
\]

Key to parallelization with MapReduce!
A function \( h : \mathbb{L}[a] \rightarrow R \) is a **semiring homomorphism** if it is defined in the following form:

\[
\begin{align*}
    h \mathbb{L} & = \iota_\oplus \\
    h \mathbb{L}[\mathbb{L}] & = \iota_\otimes \\
    h \mathbb{L}[x] & = f \times \\
    h (b \oplus b') & = h b \oplus h b' \\
    h (b \otimes b') & = h b \otimes h b'
\end{align*}
\]

where \((R, (\oplus, \iota_\oplus), (\otimes, \iota_\otimes))\) forms a semiring.
A function $h : \mathbb{L}[a] \rightarrow R$ is a **semiring homomorphism** if it is defined in the following form:

$$
\begin{align*}
    h \mathbb{L} & = \mathbb{I} \oplus \\
    h \mathbb{L} xs & = h' xs \\
    h (b \uplus b') & = h b \oplus h b' \\
    h' [] & = \mathbb{I} \otimes \\
    h' [x] & = f x \\
    h' (xs \uplus ys) & = h' xs \otimes h ys
\end{align*}
$$

where $(R, (\oplus, e1), (\otimes, e2))$ forms a semiring.
The Knapsack Specification

\[ knapsack = \text{maxvalue} \circ \text{filter} \ ((\leq 5) \circ \text{weight}) \circ \text{subs} \]
The Generator

\[ \text{subs} [1, 2, 3] = \left[ [], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3] \right] \]
The Generator

\[ \text{subs} \{1, 2, 3\} = \emptyset, [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3] \]

\textit{subs} is monoid homomorphism:

\[
\begin{align*}
\text{subs} \; \emptyset & = \emptyset \\
\text{subs} \; [x] & = \emptyset, [x] \\
\text{subs} \; (xs + ys) & = \text{subs} \; xs \times + \text{subs} \; ys
\end{align*}
\]
The Test

\[ test = (\leq 5) \circ weight \]

\[ weight [] = 0 \]

\[ weight [(v, w)] = w \]

\[ weight (xs \oplus ys) = weight xs + weight ys \]
The Aggregator

\[
\begin{align*}
\text{maxvalue} \{ \} & = -\infty \\
\text{maxvalue} \{ [] \} & = 0 \\
\text{maxvalue} \{ [(v, w)] \} & = v \\
\text{maxvalue} (b \uplus b') & = \text{maxvalue} b \uparrow \text{maxvalue} b' \\
\text{maxvalue} (b \times_+ b') & = \text{maxvalue} b + \text{maxvalue} b'
\end{align*}
\]
One More: Generator – Polymorphic over Semiring

\[
\begin{align*}
subs[\ ] &= \epsilon[\ ]
\\
subs[x] &= \epsilon[\ ] \uplus \epsilon[x]
\\
subs(xs \uplus ys) &= subs xs \times_{+} subs ys
\\
\downarrow
\\
subs_{\oplus, \otimes} f[\ ] &= \iota_{\otimes}
\\
subs_{\oplus, \otimes} f[x] &= \iota_{\otimes} \oplus f x
\\
subs_{\oplus, \otimes} f(xs \uplus ys) &= subs_{\oplus, \otimes} xs \otimes subs_{\oplus, \otimes} ys
\\
subss &= subs_{\uplus, \times_{+}} (\lambda x \rightarrow \epsilon[x])
\end{align*}
\]
Outline

1. GTA: Generate, Test, Aggregate
2. GTA Specification
3. Parallelization of GTA
   - Semiring Fusion
   - Filter Embedding
4. Conclusion
Our Theorems

Inefficient Specification

\[
\text{maxvalue} \circ \text{filter} ((\leq 5) \circ \text{weight})
\]
\[
\circ \text{subs}_{\cup, \times} (\lambda x \rightarrow \lbrack x \rbrack)\]
\[
= \{ \text{Filter Embedding} \}
\]
\[
\text{postprocess}_5 \circ \text{maxvalue}_5
\]
\[
\circ \text{subs}_{\cup, \times} (\lambda x \rightarrow \lbrack x \rbrack)\]
\[
= \{ \text{Semiring Fusion} \}
\]
\[
\text{postprocess}_5
\]
\[
\circ \text{subs}_{
7, +} (\lambda (v, w) \rightarrow \text{maxvalue}_5 \lbrack (v, w) \rbrack)\]

Efficient Implementation
Semiring Fusion

\[ \text{aggregate} \circ \text{generate} \oplus, \otimes (\lambda x \rightarrow \mathbb{L}[x]\mathbb{I}) \]
\[ = \{ \text{Free Theorem} \} \]
\[ \text{generate} \oplus, \otimes (\lambda x \rightarrow \text{aggregate} \mathbb{L}[x]\mathbb{I}) \]

(if aggregate is semiring homomorphism)

Similar to short-cut fusion

- can transform exponential into linear algorithm
- avoids intermediate blowup via distributivity
aggregate(S,⊕,⊗) ◦ filter hom(M,⊙) = aggregate(S^M,⊕_M,⊗_M)

Here,

$$S^M = \{ \{ f_m \}_{m \in M} | f_m \in S \}$$

$$(f \oplus_M f')_m = f_m \oplus f'_m$$
$$(f \otimes_M f')_m = \bigoplus_{k,l \in M \atop k \circ l = m} (f_k \otimes f'_l)$$
Filter Embedding

\[ \text{aggregate} \circ \text{filter} \ (\text{ok} \circ \text{hom}) = \text{postprocess} \circ \text{aggregate}' \]

- \text{aggregate}' new semiring homomorphism
- based on \text{aggregate} and \text{hom}
- \text{postprocess} maps back to original semiring
- based on \text{ok}

moves test out of the way to allow semiring fusion
The Result Program for Knapsack

\[ knapsack = \]

\[ postprocess_5 \]

\[ \circ \, \text{subs}^+, \lambda (v, w) \rightarrow \maxvalue_5 \preceq [(v, w)] \]
The Result Program for Knapsack

\[
\text{knapsack} = \text{postprocess}_5 \\
\quad \circ \text{subs}^{\uparrow, +} (\lambda (v, w) \rightarrow \text{maxvalue}_5 \sqcap \left[ (v, w) \right])
\]

Semiring of 7-tuples, max value for every weight:

\[
\text{postprocess}_5 (v_0, v_1, v_2, v_3, v_4, v_5, v_6) = \\
v_0 \uparrow v_1 \uparrow v_2 \uparrow v_3 \uparrow v_4 \uparrow v_5
\]
The Result Program for Knapsack

\[ \text{knapsack} = \]
\[ \text{postprocess}_5 \]
\[ \circ \text{subs}^\uparrow,\uparrow (\lambda(v, w) \rightarrow \text{maxvalue}_5 \downarrow[(v, w)]\uparrow) \]

Semiring of 7-tuples, max value for every weight:

\[ \text{postprocess}_5 (v_0, v_1, v_2, v_3, v_4, v_5, v_6) = \]
\[ v_0 \uparrow v_1 \uparrow v_2 \uparrow v_3 \uparrow v_4 \uparrow v_5 \]

Initially, associate value to its weight

\[ \text{maxvalue}_5 \downarrow[(4000, 3)]\uparrow = \]
\[ (\infty, \infty, \infty, 4000, \infty, \infty, \infty) \]
Lifted Semiring Operations

Pointwise maximum:

\[(v_0, ..., v_6) \uparrow^7 (v'_0, ..., v'_6) = (v_0 \uparrow v'_0, ..., v_6 \uparrow v'_6)\]

Combine values with corresponding weight:

\[(v_0, v_1, v_2, v_3, ...) +^7 (v'_0, v'_1, v'_2, v'_3, ...) = (v_0 + v'_0, (v_0 + v'_1) \uparrow (v_1 + v'_0), (v_0 + v'_2) \uparrow (v_1 + v'_1) \uparrow (v_2 + v'_0), (v_0 + v'_3) \uparrow (v_1 + v'_2) \uparrow (v_2 + v'_1) \uparrow (v_3 + v'_0), ...)]
Performance

- linear in length of item list
- quadratic in maximum weight (pseudo polynomial)
  - with one processor: $O(nw^2)$
  - with $p$ processors: $O((p + \frac{n}{p})w^2)$

- algorithm generalizes to other programs in generate-test-and-aggregate form
- performance too
Other Applications and Generalization

Practical Applications:
- inferring states of hidden Markov model
- maximum segment sums, maximum prefix sums, ...
- incremental refinement through additional filters

Generalization to other Types:
- not only lists
- other input types: easy
- other intermediate types: possible too
Outline

1. GTA: Generate, Test, Aggregate
2. GTA Specification
3. Parallelization of GTA
4. Conclusion
Conclusion

- methodology for developing parallel programs
- generalizes existing techniques
- covers wide class of problems
- supports modular, incremental development
- intuitive API (in Haskell), automatic efficiency (in GHC)