All you need is monoid

Monoidal aggregators

Michał J. Gajda

Introduction

Different computation models have received significant attention as a way to decrease complexity of parallel computation.

We propose a tested in practice monoidal aggregator/splitoid model, and provide simulations suggesting that it is much more efficient in practice.

Monoid model

```
class Monoid m where
  mappend :: m → m → m
  mempty  :: m
```

In order to divide tasks into smaller ones, we could use CoMonoid that is dual to given Monoid:

```
class CoMonoid m where
  split :: m →      (m,m)
```

By duality we mean that *split* is inverse of monoid multiplication $\diamond$.

We can now call f homomorphism, or compatible with CoMonoid, when for $(x, y) = split\ xy$ the equality $f\ xy = f\ x \diamond f\ y$ holds. That is we can compute the same function on parts, and then merge with $\diamond$, or just directly without splitting.

But we want to balance the task over n computing capabilities, we want a stronger property of always being able to split into that many balanced parts. CoMonoid only gives us ability to split into two parts, possibly unbalanced.

In order to divide tasks into smaller ones, we use weaker Splitoid:

```
class Monoid m ⇒ Splitoid m where
  split :: Int  → α → Vector α
```

Note that first argument must be ≥ 1. The additional laws for Splitoid would be:

$$
\begin{aligned}
&\mathit{length}\ \ (\mathit{split}\ i\ x) = i\\
&\mathit{mconcat}\ (\mathit{split}\ i\ x) = x
\end{aligned}
$$

Here $\mathit{mconcat} = \mathit{fold}\ \mathit{mappend}$.

Alternative formulation (**edward-divide-and-conquer?**) is available in cases that the different subtasks are of different types:

$$
\begin{aligned}
&\textbf{class}\ \ \textsc{Contravariant}\ f\\
&\quad \Rightarrow\ \textsc{Divisible}\ \ \ f\ \textbf{where}\\
&\quad \mathit{divide} :: (\alpha \to (\beta\ \ , \gamma)) \to f\,\beta \to f\,\gamma \to f\,\alpha\\
&\quad \mathit{conquer} :: f\ \ \alpha
\end{aligned}
$$

This approach allows us freedom to divide into tasks of a different type.

Monoidal aggregator

We also define *aggregator* as a triple of preprocessing, postprocessing steps, and a monoid: $\mathit{preprocessing} \to \mathit{monoid} \to \mathit{postprocessing}$. This will allow us to describe a wider class of operations (like all SQL aggregation functions[1], or batch gradient descent). All parallelism here is confined in monoid step.

```
data Agg m a b =
  forall m. Agg {
              pre    :: a -> m
            , append :: m -> m -> m
            , post   :: m -> b
            }
```

Tensor multiplication on monoids and aggregators

$$
\begin{aligned}
&(><) :: \ \ \textsc{Agg}\ m\ \alpha\ \beta \to \textsc{Agg}\ m\ \alpha\ \gamma \to \textsc{Agg}\ m\ \alpha\ (\beta, \gamma)\\
&\textsc{Agg}\ \ \{\ \mathit{pre}\ \ \ = \mathit{preA}\\
&\qquad ,\ \mathit{append} = \mathit{appendA}\\
&\qquad ,\ \mathit{post}\ \ \ = \mathit{postA}\ \}\ ><\\
&\ \ \textsc{Agg}\ \{\ \mathit{pre}\ \ \ = \mathit{preA}\\
&\qquad ,\ \mathit{append} = \mathit{appendA}\\
&\qquad ,\ \mathit{post}\ \ \ = \mathit{postA}\ \}\ \ =\\
&\ \ \textsc{Agg}\ \{\ \mathit{pre}\ \ \ = \mathit{preA}\ \ \&\&\&\ \ \mathit{preB}\\
&\qquad ,\ \mathit{append} = \lambda\,(\mathit{a1}\ ,\ \mathit{b1})\ (\mathit{a2}\ ,\ \mathit{b2}) \to (\mathit{a1}\ `\mathit{mappendA}`\ \mathit{a2},\ \mathit{b1}\ `\mathit{mappendB}`\ \mathit{b2})\\
&\qquad ,\ \mathit{post}\ \ \ = \lambda\,(\alpha,\ \beta) \to (\mathit{postA}\ \ \alpha\,,\ \mathit{postB}\ \ \beta)\\
&\qquad \}
\end{aligned}
$$

Preprocessing of input

The aggregators are well behaving *profunctors*(**profunctor?**):

[1] Assuming exact arithmetic.

```
instance FUNCTOR      AGG m α where
  fmap f  AGG { pre,   append ,  post   } =
          AGG { pre,   append ,  post   = f . post }

instance PROFUNCTOR AGG m where
  premap f AGG { pre           , append, post } =
           AGG { pre = pre . f, append, post }
```

Conditional

Chunking

Prefiltering

Serial algorithms

We will describe serial algorithms as left folds:

```
serial = (λ(finalState, finalOutput) → finalOutput)
       . foldl seqStep initialState

seqStep :: ( BOUNDED state
           , ENUM    state)
         ⇒ ( state, output) → input → (state, output)
```

We note that given small enough state space *state*, and *output write-only* monoid, we may be able to make it monoidal by translating states from (*state*, *output*) into mapping from initial state to final result: (*state* :-> (*state*, *output*)).

While finite automata usually have multiple states, it may be also possible to constrain input monoid to *choke points*, where automaton only accepts a smaller number of states.

Basically we recover for a fold making decisions over a finite spaces of states s, and with write-only output, we recover $O(n * s)$ parallel algorithm, where s is a number of states.

For example CSV parser has only 2 states at the end of line, PDB parser has only one state. That means that parallel PDB parser that first splits input into line would do no extra bookkeeping, and CSV parser would only take a constant 2x overhead, with opportunity to parallelize without limits.

Function	Preprocessing	Monoid	Postprocessing	Notes
SUM (SQL 2016)	id	Sum = (NUMBER, +, 0)	id	
COUNT(SQL 2016)	(_ -> 1)	Count = (INT, +, 0)	id	
MIN(SQL 2016)	id	$(a, min, +\infty)$	null if $+\infty$, *id* otherwise	
MAX(SQL 2016)	id	$(a, max, -\infty)$	null if $-\infty$, *id* otherwise	
AVG(SQL 2016)	$id * x(\lambda\_ -> 1)$	Sum >< Count	$\lambda(s, \gamma) \rightarrow s \ / \ \gamma$	
VAR(TSQL 2005)	$(\lambda x -> (x^2, x, 1))$	Sum >< Sum >< Count	$\lambda(x2, x, c) -> x2/c - (x/c)^2$	
STDDEV (TSQL 2005)	$(\lambda x -> (x^2, x, 1))$	Sum >< Sum >< Count	$\lambda(x2, x, c) -> sqrt(x2/c - (x/c)^2)$	
STRING_AGG (TSQL 2005)	id	(STRING, concat, "")		
GROUPING (TSQL 2005)	1 if grouping, 0 otherwise	(BOOL, *and*, *true*)	id	
GROUPING_ID (TSQL 2005)	level of grouping	(a, lastNonNull, null)	fromJust	
Combinatorial optimization(Athougies 2018)				
Inefficient f	*id*	$(\alpha, \diamond, [])$	f	Arbitrary function $f ::$ MONOID $m \Rightarrow m \rightarrow \beta$
State transducer(Kmett 2017)	*id*	$(St \mapsto (St, Out), bind_s tate, [])$	*lookup initialState*	Overhead: O(|STATE|), split at *choke points*
CSV parser(Ge et al. 2019; Stehle and Jacobsen 2020)	*id*	$(Bool \mapsto Out), bind_s tate, [])$	*lookup* FALSE	Merging is O(AST_DEPTH), split at EOLs

Function	Preprocessing	Monoid	Postprocessing	Notes
hPDB parser(Gajda 2013)	*id*	(VECTOR^5, *merge_nested*, []^5)	fill_holes	Merging is O(AST_DEPTH)=5, split at EOLs
CFG parser	*id*	(STATE:->(STATE,ASTWITHHOLES) → ASTWITHHOLES), *bind_state*, [])		Merging is O(AST_DEPTH), split at *choke points*

```
type (:->) = Data.Map.Map
m1 `bind_{state}` m2 =
  Data.Map.fromList [   let (s2, o2) ← Data.Map.lookup s1 m2
                        in (s1i, (s2, o1 ◇ o2))
                      | (s1i, (s1o, o1)) ← Data.Map.toList m1 ]
```

All TransactSQL function with exception of CHECKSUM_AGG can be implemented as aggregators[2].

Also, all relational algebra operations can be implemented with aggregators.

Related work

While traditional functional programming community used *foldr* and *foldl* to describe reductions on the lists, the more general operation arisen $fold :: \text{Monoid } m \Rightarrow (\alpha \rightarrow m) \rightarrow [\alpha] \rightarrow m$. *fold* can be applied to lists, but also trees and most other collections(Paterson 2005). It was observed that this more general operation allows for more efficient execution of most algorithms(Newton 2020).

Functional programming community has also long noticed that popular representations of the String type as list of characters or array frequently list inefficient $O(n^2)$ algorithms when used for printing long documents. Thus it was replaced with Builder data structures that allow monoidal $\diamond$ operation (string concatenation) to be executed efficiently whether most of the input lies on left or right side of the operator.

Alternative models

Map reduce model(Dean and Ghemawat 2004) uses similar formulation of reduction, but without mathematical rigour. It also does not allow splitting at any stage of computation.

Most reviews of models of paralellization shy from the actual simulation [@..], even if there is evidence that some models perform poorly in practice (like piping parsers in BioJava(Yates et al. 2012) benchmarked (Gajda 2013; **c-pdb-parser?**)...).

The closest work to ours is empirical comparison of efficiency of different reduction patterns in serial setting (Newton 2020).

[2]We do not have enough data on checksum algorithm to decided whether it could also be implemented as aggregator.

Experiments

In order to abstract and erase differences between the cost of computation, and cost of transmission, we model all components as computation time, using time-to-completion model used to model latency of distributed systems (**network-latency?**).

We model the following components of the cost of computation:

- cost of initial computation of size n as $O(n)$, $O(n^2)$ or $O(n^3)$
- cost of merging the results as $O(n)$, $O(n)$, assuming that merging has no greater cost than computation

We consider the following computation models:

1. Directed acyclic graph decomposition of the task (Kreps, Narkhede, and Rao 2011; Armbrust et al. 2015).
2. Pipeline decomposition of the task (Yates et al. 2012).
3. Static monoidal reduction (Gajda 2013).
4. Dynamic monoidal reduction (**csv?**).
5. Work stealing.

We also provide some examples of the real-life tasks that were implemented in different ways to support validity of our approach: ...

Armbrust, Michael, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. 2015. "Scaling Spark in the Real World: Performance and Usability." VLDB.

Athougies, Travis. 2018. "Monadic Structure of Combinatorial Optimization." 2018. https://travis.athougies.net/posts/2018-04-23-combinatorial-optimization.html.

Dean, Jeffrey, and Sanjay Ghemawat. 2004. "MapReduce: Simplified Data Processing on Large Clusters." In *OSDI'04: PROCEEDINGS OF THE 6th CONFERENCE ON SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*. USENIX Association.

Gajda, Michal J. 2013. "hPDB - Haskell Library for Processing Atomic Biomolecular Structures in Protein Data Bank Format." *BMC Research Notes* 6 (1): 483. https://doi.org/10.1186/1756-0500-6-483.

Ge, Chang, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. "Speculative Distributed CSV Data Parsing for Big Data Analytics." In *Proceedings of the 2019 International Conference on Management of Data*, 883–99. SIGMOD '19. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3299869.3319898.

Kmett, Edward. 2017. "Monoidal Parsing." https://www.youtube.com/watch?v=Txf7swrcLYs.

Kreps, Jay, Neha Narkhede, and Jun Rao. 2011. “Kafka: A Distributed Messaging System for Log Processing.” NetDB’11.

Newton, Jim. 2020. “Performance Comparison of Several Folding Strategies.” Trends in Functional Programming. 2020. https://www.lrde.epita.fr/wiki/Publications/newton.20.tfp.

Paterson, Ross. 2005. “Data.foldable.” GHC Basic libraries. 2005. http://hackage.haskell.org/package/base-4.14.1.0/docs/Data-Foldable.html.

SQL. 2016. “Information Technology — Database Languages — SQL — Part 1: Framework (SQL/Framework).” ISO/IEC 9075-1:2016.

Stehle, Elias, and Hans-Arno Jacobsen. 2020. “ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data.” *Proc. VLDB Endow.* 13 (5): 616–28. https://doi.org/10.14778/3377369.3377372.

TSQL. 2005. “Transact-SQL User’s Guide Adaptive Server ® Enterprise.”

Yates, Andrew, Spencer E. Bliven, Peter W. Rose, Peter V. Troshin, Mark Chapman, Jianjiong Gao, Hock Koh, et al. 2012. “BioJava: An Open-Source Framework for Bioinformatics in 2012.” In *Page 10 of 11 Original Article Database, Vol. 2013, Article ID Bat051, Doi:10.1093/Database/Bat051*. https://doi.org/doi:10.1093/database/bat051.