On efficient Hessian computation using Edge Pushing algorithm in Julia

Feng Qiang
Argonne National Laboratory

Joint work with
Cosmin G. Petra (LLNL), Joey Huchette (MIT), Miles Lubin (MIT), Mihai Anitescu (ANL)

AD2016
September 13rd, 2016
Outline

- Background and motivation
  - Julia language
  - JuMP, an algebraic modelling language in Julia
  - JuMP’s AD and current challenge

- The Edge Pushing (EP) implementation in Julia
  - Introduce the EP algorithm
  - The recursive expression for Hessian computation, and our modified version
  - Choice of different data structures, time vs. space
  - Results from abstract
  - More code optimization and new results
Julia

- Fresh approach for technical computing (http://julialang.org/)
- User friendly and syntax is similar to Matlab
  - Dynamic language with interactive command-line Read-eval-print loop
- C-like performance.
  - Just-In-Time compilation and generate native assembly code
- Open source with a large and fast growing community behind
- Runs on workstations, clusters, cloud and HPC platforms
JuMP – an Algebraic Modelling Language in Julia

- Optimization problem

\[
\begin{align*}
\min & \quad f(x) \\
\text{s.t} & \quad g(x) \leq 0
\end{align*}
\]

- Provide closed mathematical form for to express \( f(x), g(x) \)
- For domain specialists to quickly specifying the problem without knowledge about optimization algorithms/software and computing
- Automatic computation of the \( f(x), g(x), \nabla f(x), \nabla^2 f(x), \nabla^2 g(x) \)
  - using Automatic Differentiation (AD)
- Developed by collaborators at MIT (Miles Lubin, Iain Dunning, Joey Huchette)
AD in a nutshell

- Function $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$, can be represented by composite elementary functions
  - e.g. $f(x) = \phi_3(\phi_2(\phi_1(x)))$

- Forward accumulation
  - Applying chain rule right $\rightarrow$ left
    
    $$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial v_3} \left( \frac{\partial \phi_3}{\partial v_2} \left( \frac{\partial \phi_2}{\partial v_1} \frac{\partial \phi_1}{\partial x} \right) \right)$$
  - $n$ forward sweep to compute $\nabla f(x) : \mathbb{R} \rightarrow \mathbb{R}^n$
  - One direction for each sweep

- Reverse accumulation
  - Reverse mode: applying chain rule left $\rightarrow$ right
    
    $$\frac{\partial f}{\partial x} = \left( \frac{\partial f}{\partial v_3} \frac{\partial \phi_3}{\partial v_2} \right) \frac{\partial \phi_1}{\partial x}$$
  - Compute $\nabla f(x) : \mathbb{R} \rightarrow \mathbb{R}^n$ : one forward sweep + one reverse sweep.
  - Less computational cost and using $O(\#\text{function evaluations})$ storage.
JuMP’s AD and current challenge

- Reverse mode for gradient computation
- Apply forward on Reverse mode for computing Hessian-vector product.
  - Jacobian of \( \nabla f(x) \), is \( \nabla^2 f(x) \)
  - \( \nabla^2 f(x)d \) is the directional derivative of \( \nabla f(x) \)
- Coloring algorithm is used for computing sparse Hessian
  - Finding the smallest number of Hessian-vector products needed to recover nonzero entries in \( \nabla^2 f(x) \)

- Coloring is a NP-Hard problem
  - Extremely time consuming
  - Logistic regression model: 
    \[ \min_\theta \lambda \sum_{i=1}^{N} \theta_i^2 + \sum_{i=1}^{M} \log(1 + \frac{1}{e^{y_i \sum_{j=1}^{N} x_{ij} \theta_j}}) \]
    \( \theta \) independent variable, 
    \( \lambda, x, y \) are model parameters
  - M=2 N=8k, takes 3554 seconds to compute the number of colors.

- Question: can we avoid the coloring?
  - Yes, use the Edge_Pushing algorithm by Gower.
### Edge Pushing algorithm

- **Input:** function $f(x)$ represented by a list of elementary functions $\phi_i, i \in \{1, \cdots, l\}$

- **Output:** $f''(x) = PWP^T$

Pushing edges: accumulate the second order derivative between $i$’s precedent and $i$’s children

Creating edges: accumulate the second order derivative among $i$’s children

Updating adjoints: for $i = \{1, \ldots, l\}$ do
  
  foreach $p$ such that $p \leq i$ and $w_{pi} \neq 0$ do
    
    if $p \neq i$ then
      
      foreach $j \prec i$ do
        
        if $j = p$ then
          
          $w_{pp} = 2 \frac{\partial \phi_i}{\partial v_p} w_{pi}$
        
        else
          
          $w_{jp} = 2 \frac{\partial \phi_i}{\partial v_j} w_{pi}$
        
        end
      
    else
      
      foreach unordered pair $j, k$ such that $j, k \prec i$ do
        
        $w_{jk} = \frac{\partial^2 \phi_i}{\partial v_k \partial v_j} w_{ii}$
      
      end
    
  end

end

end

end

foreach unordered pair $j, k$ such that $j, k \prec i$

$w_{jk} = \tilde{v}_i \frac{\partial^2 \phi_i}{\partial v_k \partial v_j}$

end

foreach $j \prec i$ do

$\tilde{v}_j + = \tilde{v}_i \frac{\partial \phi_i}{\partial v_j}$

end
Edge Pushing (EP)

- At each node, EP
  - applies second-order chain rule, and incrementing the derivatives between variables with nonlinear relationship.
  - tracks dependences of the gradients (adjoint variables).

- EP is a reverse mode algorithm

- Hessian is computed
  - One forward sweep + one reverse sweep
  - Sparsity and Symmetry are exploited automatically
  - Memory access pattern is crucial for an efficient implementation.

**Question: what data structure for storing $w$?**

- At each node $i$, we need to access list of edges with endpoint at $i$ and their weights $w_{pi}$;
- and we need to update $w_{jk}$, where $j, k$ can be $i$’s children or its precedent;
Data structures for $w$ in Julia

- Dictionary of dictionaries, *e.g.* Dict{Int, Dict{Int, Float64}}
  - Only one entry for each nonzero in Hessian
  - Should give $O(1)$ read/write, but, slow in real world.
  - Memory is managed by Julia’s GC.

- Vector of $n + l$ dictionaries, *e.g.* Vector{Dict{Int, Float64}}
  - Some dictionary could be empty (linear variables).
  - Julia’s Vector can be pre-allocated, and then accessing with @inbound macro to disable bound checking.
  - Faster than previous one, but we still use Julia’s dictionary (*i.e.*, paying cost for GC).

- Question: can we do it without Julia’s dictionary?
The recursive definition for Hessian evaluation

- The recursive expression of the Hessian algorithm (Wang, Gebremedhin and Pothen, 2016)

\[ \forall i \in \{l, \ldots, 1\}, \forall (k, j) \in S_i \times S_i, \]
\[ H_i(k, j) = H_{i+1}(k, j) + \frac{\partial \phi_i}{\partial v_j} H_{i+1}(i, k) + \frac{\partial \phi_i}{\partial v_k} H_{i+1}(i, j) + \frac{\partial \phi_i}{\partial v_j} \frac{\partial \phi_i}{\partial v_k} H_{i+1}(i, i) + \bar{v}_i \frac{\partial^2 \phi_i}{\partial v_j \partial v_k} \]

where, the Live variable sets \( S_i (i = \{l, \ldots, 1\}) \), also defined recursively by
\[ S_i = \{S_{i+1}\backslash\{i\}\} \cup \{j | v_j < v_i\}. \]

- Our modification

\[ \forall i = \{l, \ldots, 1\}, \ H^P_i(k, j) = \emptyset \ and \ \forall (k, j) \in S_i \times S_i, \]

\[ H^P_i(k, j) \left\{ \begin{array}{l}
H^P_i(k, j) \cup \{\frac{\partial \phi_i}{\partial v_j} \cdot h : h \in H^P_{i+1}(i, k)\}, \\
H^P_i(k, j) \cup \{\frac{\partial \phi_i}{\partial v_k} \cdot h : h \in H^P_{i+1}(i, j)\}, \\
H^P_i(k, j) \cup \{\frac{\partial \phi_i}{\partial v_j} \frac{\partial \phi_i}{\partial v_k} \cdot h : h \in H^P_{i+1}(i, i)\} \cup \{\bar{v}_i \cdot \frac{\partial^2 \phi_i}{\partial v_j \partial v_k} : \bar{v}_i \neq 0\}, \\
v_j < v_i, v_k \neq v_i, \\
v_j < v_i, v_k < v_i, \\
v_j < v_i, v_k < v_i. 
\end{array} \right. \]

- Allow duplicate entries to be appended into the set \( H^P \)
- Store all terms that contribute to \( H_i(k, j) \), then \( H(k, j) = \sum_{h \in H^P_i(k, j)} h \)
Our proposed data structure for $H^p$

- Vector of $n + l$ vectors for $H^p$, and inner vector contains pairs (row, value).
- Similar to the compressed sparse column (CSC) storage, but allowing duplicate entries
- Removed overhead for using Julia’s dictionary.
- ~3x faster than vector of $n + l$ dictionaries.
Results for Hessian of arrowhead structure

- Function expression: \[ \sum_{i=1}^{N} \left[ \cos \left( \sum_{j=1}^{K} x_{i+j} \right) + \sum_{j=1}^{K} (x_i + x_j)^2 \right] \]

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>Coloring(s)</th>
<th>Hessian use coloring(s)</th>
<th>Hessian use EP(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>8</td>
<td>1.47</td>
<td>0.07</td>
<td>0.13</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
<td>5.93</td>
<td>0.14</td>
<td>0.26</td>
</tr>
<tr>
<td>16000</td>
<td>8</td>
<td>25.36</td>
<td>0.25</td>
<td>0.51</td>
</tr>
<tr>
<td>32000</td>
<td>8</td>
<td>141.20</td>
<td>0.49</td>
<td>1.08</td>
</tr>
<tr>
<td>64000</td>
<td>8</td>
<td>689.42</td>
<td>0.97</td>
<td>2.10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>Coloring(s)</th>
<th>Hessian use coloring(s)</th>
<th>Hessian use EP(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20000</td>
<td>2</td>
<td>16.03</td>
<td>0.02</td>
<td>0.13</td>
</tr>
<tr>
<td>20000</td>
<td>4</td>
<td>23.81</td>
<td>0.08</td>
<td>0.28</td>
</tr>
<tr>
<td>20000</td>
<td>8</td>
<td>43.78</td>
<td>0.30</td>
<td>0.65</td>
</tr>
<tr>
<td>20000</td>
<td>16</td>
<td>93.97</td>
<td>1.14</td>
<td>1.68</td>
</tr>
<tr>
<td>20000</td>
<td>32</td>
<td>179.81</td>
<td>4.10</td>
<td>5.72</td>
</tr>
</tbody>
</table>
Results for Hessian of random sparsity structure

- Function expression: \[ \sum_{i=1}^{N} \left( (x_i - 1)^2 + \prod_{j \in \text{rand.set}_i(N,K)} x_j \right) \]

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>Coloring(s)</th>
<th>Hessian use coloring(s)</th>
<th>Hessian use EP(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>4</td>
<td>0.05</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>4000</td>
<td>8</td>
<td>0.22</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>4000</td>
<td>16</td>
<td>2.57</td>
<td>1.05</td>
<td>0.27</td>
</tr>
<tr>
<td>4000</td>
<td>32</td>
<td>38.17</td>
<td>17.97</td>
<td>1.01</td>
</tr>
<tr>
<td>4000</td>
<td>64</td>
<td>254.29</td>
<td>119.83</td>
<td>4.36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>Coloring(s)</th>
<th>Hessian use coloring(s)</th>
<th>Hessian use EP(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>32</td>
<td>4.06</td>
<td>2.32</td>
<td>0.21</td>
</tr>
<tr>
<td>2000</td>
<td>32</td>
<td>3.56</td>
<td>7.18</td>
<td>0.44</td>
</tr>
<tr>
<td>4000</td>
<td>32</td>
<td>41.08</td>
<td>18.05</td>
<td>0.98</td>
</tr>
<tr>
<td>8000</td>
<td>32</td>
<td>94.47</td>
<td>38.77</td>
<td>1.93</td>
</tr>
<tr>
<td>16000</td>
<td>32</td>
<td>193.54</td>
<td>66.13</td>
<td>4.11</td>
</tr>
</tbody>
</table>
More code optimization for Julia EP implementation

- Unnecessary alias for independent nodes in the computational graph.
  - Removes the alias and uses variable indices from JuMP directly.
  - ~2x improvements in execution time,
  - Reduced memory usage as well

- Julia imposes some overhead for the Pair struct.
  - Separate Vector{Vector{Pair{Int,Float64}} with Vector{Vector{Int}}} and Vector{Vector{Float64}}.
  - ~5x improvement in execution time

- The optimization work give us ~10x improvement in execution time.
Results after code optimization (Hessian of arrowhead structure)

- Function expression:
  \[
  \sum_{i=1}^{N} \left[ \cos \left( \sum_{j=1}^{K} x_{i+j} \right) + \sum_{j=1}^{K} (x_i + x_j)^2 \right]
  \]

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>Hessian use EP(s)</th>
<th>Hessian use coloring(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>8</td>
<td>0.01</td>
<td>0.07</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
<td>0.02</td>
<td>0.14</td>
</tr>
<tr>
<td>16000</td>
<td>8</td>
<td>0.04</td>
<td>0.25</td>
</tr>
<tr>
<td>32000</td>
<td>8</td>
<td>0.09</td>
<td>0.49</td>
</tr>
<tr>
<td>64000</td>
<td>8</td>
<td>0.18</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>K</td>
<td>Hessian use EP(s)</td>
<td>Hessian use coloring(s)</td>
</tr>
<tr>
<td>-----</td>
<td>---</td>
<td>-------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>20000</td>
<td>2</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>20000</td>
<td>4</td>
<td>0.03</td>
<td>0.08</td>
</tr>
<tr>
<td>20000</td>
<td>8</td>
<td>0.06</td>
<td>0.30</td>
</tr>
<tr>
<td>20000</td>
<td>16</td>
<td>0.14</td>
<td>1.14</td>
</tr>
<tr>
<td>20000</td>
<td>32</td>
<td>0.45</td>
<td>4.10</td>
</tr>
</tbody>
</table>
Results after code optimization (Hessian of random sparsity structure)

- Function expression: \( \sum_{i=1}^{N} \left( (x_i - 1)^2 + \prod_{j \in \text{rand}\_set_i(N,K)} x_j \right) \)

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>Hessian use EP</th>
<th>Hessian use coloring</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>4</td>
<td>0.00</td>
<td>0.03</td>
</tr>
<tr>
<td>4000</td>
<td>8</td>
<td>0.01</td>
<td>0.09</td>
</tr>
<tr>
<td>4000</td>
<td>16</td>
<td>0.03</td>
<td>1.05</td>
</tr>
<tr>
<td>4000</td>
<td>32</td>
<td>0.16</td>
<td>17.97</td>
</tr>
<tr>
<td>4000</td>
<td>64</td>
<td>1.04</td>
<td>119.83</td>
</tr>
<tr>
<td>16000</td>
<td>32</td>
<td>0.68</td>
<td>66.13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N</th>
<th>K</th>
<th>Hessian use EP</th>
<th>Hessian use coloring</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>32</td>
<td>0.04</td>
<td>2.32</td>
</tr>
<tr>
<td>2000</td>
<td>32</td>
<td>0.08</td>
<td>7.18</td>
</tr>
<tr>
<td>4000</td>
<td>32</td>
<td>0.16</td>
<td>18.05</td>
</tr>
<tr>
<td>8000</td>
<td>32</td>
<td>0.33</td>
<td>38.77</td>
</tr>
<tr>
<td>16000</td>
<td>32</td>
<td>0.68</td>
<td>66.13</td>
</tr>
</tbody>
</table>
# Results for dense Hessian

- **Function expression:**
  \[ \sum_{i=1}^{N} \theta_i^2 + \sum_{j=1}^{M} \log(1 + \frac{1}{e^{\sum_{j=1}^{N} \theta_j}}) \]

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Coloring</th>
<th>Hessian use coloring</th>
<th>Hessian use EP</th>
<th>Hessian use AMPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
<td>55.26</td>
<td>0.47</td>
<td>0.26</td>
<td>0.016</td>
</tr>
<tr>
<td>1</td>
<td>4000</td>
<td>448.72</td>
<td>2.11</td>
<td>1.12</td>
<td>0.056</td>
</tr>
<tr>
<td>1</td>
<td>8000</td>
<td>3552.11</td>
<td>8.88</td>
<td>4.57</td>
<td>0.220</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Coloring</th>
<th>Hessian use coloring</th>
<th>Hessian use EP</th>
<th>Hessian use AMPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2000</td>
<td>55.36</td>
<td>0.50</td>
<td>0.511</td>
<td>0.024</td>
</tr>
<tr>
<td>2</td>
<td>4000</td>
<td>450.97</td>
<td>2.19</td>
<td>2.10</td>
<td>0.064</td>
</tr>
<tr>
<td>2</td>
<td>8000</td>
<td>3554.23</td>
<td>9.35</td>
<td>9.21</td>
<td>0.288</td>
</tr>
</tbody>
</table>
Analysis the overhead using Julia

- We implement the “recovery” phase using C++.
  - Iterating the vector of vectors data structure, and
  - copying entries of Hessian into solver’s buffer.
  - C++ code compiled with `-O3` and `unroll-loops`

- Test with function expression
  \[
  \min_\theta \sum_{i=1}^{N} \theta_i^2 + \sum_{i=1}^{M} \log(1 + \frac{1}{e^{\sum_{j=1}^{N} \theta_j}})
  \]

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>Recovery julia (s)</th>
<th>Recovery c++ (s)</th>
<th>Julia overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2000</td>
<td>0.04</td>
<td>0.04</td>
<td>0%</td>
</tr>
<tr>
<td>1</td>
<td>4000</td>
<td>0.14</td>
<td>0.14</td>
<td>0%</td>
</tr>
<tr>
<td>1</td>
<td>8000</td>
<td>0.57</td>
<td>0.53</td>
<td>7.5%</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>0.07</td>
<td>0.07</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>4000</td>
<td>0.29</td>
<td>0.27</td>
<td>7.4%</td>
</tr>
<tr>
<td>2</td>
<td>8000</td>
<td>1.13</td>
<td>1.04</td>
<td>8.6%</td>
</tr>
</tbody>
</table>

- Indicating <10% overhead in Julia implementation
Future Work

- Further code optimization
- Explore more for the data structure used by Julia’s EP implementation
- Study heuristics for when to use Coloring or EP.
- Integrate EP into JuMP’s AD