Multicore Implementation of LDPC Decoders based on ADMM Algorithm

Imen DEBBABI\textsuperscript{1}, Nadia KHOUJA\textsuperscript{1}, Fethi TLILI\textsuperscript{1}, Bertrand LE GAL\textsuperscript{2} and Christophe JEGO\textsuperscript{2}

\textsuperscript{1} - SUP’COM, GRESCOM Lab, University of Carthage, Tunisia

\textsuperscript{2} - Bordeaux-INP, IMS-lab., CNRS UMR 5218 University of Bordeaux, France
The LP decoding for LDPC codes
Introduction to LDPC codes

- LDPC codes are well-known Error Correction Codes working on blocks,
  - K information bits;
  - \(N\) transmitted values,
  - \((N-K)\) redundant values,

- The LDPC code structure is defined by a H matrix,
  - Provides VN/CN involved in parity equations,
  - Visually represented as a Tanner graph.

- State-of-the-art works for LDPC decoding are based on MP algorithm;
  - Propagate message between CNs and VNs,
  - MP algorithm is iterative.

\[
H = \begin{pmatrix}
V_0 & V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 \\
C_0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
C_1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
C_2 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
C_3 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1
\end{pmatrix}
\]

Tanner graph representation.
During the last decade, lots of works focused on LDPC codes. For instance:

- Find an « efficient » **SPA approximation**,
  - SPA algorithm is efficient but complex to implement,
  - MS, OMS, NMS, 2NMS, lambda-min, ANMS, etc.

- Reduce **computation complexity** through different computation schedules,
  - Flooding, TDMP, conditional activation, etc.

- **Efficient implementation** of LDPC decoders,
  - Hardware (ASIC, FPGA) for efficiency,
  - Software (CPU & GPU) for flexibility.

**Linear Programming (LP) approach for LDPC decoding** is a « recent » way.
LP decoding of LDPC codes

- Linear programming formulation of LDPC decoding problem,
  - First, proposed by in [1],
  - Huge memory & computation complexities,
  - Limited to very short frames (N << 200),

- Interesting FER performance
  - Especially in Error floors (Even against SPA),
  - ML certificate when frame is successfully decoded (not decoded otherwise).

- Lower complexity formulation,
  - Initial LP ADMM algorithm [2],
  - Good FER performance ADMM-l2 against SPA [3],
  - Reduced complexity s-ADMM-l2 [4]

- LP LDPC decoding is affordable for implementation purpose.

Linear programming formulation of LDPC decoding,
- First, proposed by in [1],
- Huge memory & computation complexities,
- Limited to very short frames (< 200 bits),

Interesting FER performance
- Even against SPA algorithm,
- ML certificate when frame is successfully decoded (not decoded otherwise).

Lower complexity formulation,
- Initial LP ADMM algorithm [2],
- Improved ADMM-l2 against SPA [3],
- Computation complexity reduction [4],

LP LDPC decoding becomes now realistic for implementation purpose.

Fig. 1. FER comparison of ADMM-l2 penalized decoders with SPA decoders on AWGN channel.


The ADMM decoding algorithm
Formulation of the ADMM decoding algorithm

- The ADMM algorithm is a MP-based formulation of the LP problem,
  - Proposed in [2] and correction improved in [3],
  - Traditional flooding schedule,
  - The key element is the Euclidian projection;
  - Formulation maintains LP properties,

- Based on 4 distinct kernels
  - Kernel 1, initializes the decoder;
  - Kernel 2, processes all VNs;
  - Kernel 3, processes all CNs;
  - Kernel 4, takes hard decision;

- Kernels 2 and 3 are iterated k times (# iterations)
  - Computation complexity is located there;

Algorithm 1 Flooding based ADMM - l2 Algorithm.

1: KERNEL 1: Initialization
2: ∀ j ∈ J, i ∈ Nc(j) : z_j^(0)^i = 0.5, λ_j^(0)^i = 0
3: ∀ i ∈ I : n_i = \frac{\lambda_i}{\mu}
4: for all k = 1 \rightarrow q when stop criterion = false do
5:   KERNEL 2: For all variable nodes in the code
6:     for all i ∈ I, j ∈ Nv(i) do
7:         t_i^(k) = \sum_{j ∈ N_v(i)} (z_j^(k-1)^i - λ_j^(k-1)^i)
8:         L_i → j = \Pi_{[0,1]}^i (\frac{1}{dv_i-2\alpha} (t_i^(k) - n_i - \frac{n_i}{\alpha}))
9:     end for
10:   KERNEL 3: For all check nodes in the code
11:     for all j ∈ J, i ∈ Nc(j) do
12:         z_j → i = \Pi_{P_{dec_j}}^{(k)} (\rho L_i → j + (1-\rho) z_j → i + \lambda_j → i)
13:         λ_j → i = λ_j → i + \rho L_i → j + (1-\rho) z_j → i - z_j → i
14:     end for
15:   KERNEL 4: Hard decisions from soft-values
16:     ∀ i ∈ I : \hat{c}_i = \left( \sum_{j ∈ N_v(i)} L_i → j \right) > 0.5


The ADMM algorithm has a flooding-based formulation of the LP problem,
- Proposed in [2] and correction improved in [3],
- Traditional flooding schedule,
- Based on Euclidian projection;
- Formulation maintains LP properties,

Based on 4 distinct kernels
- Kernel 1, initializes the decoder;
- Kernel 2, processes all VN;
- Kernel 3, processes all CN;
- Kernel 4, takes hard decision;

Kernels 2 and 3 are iterated k times (# iterations)
- Decoding computation complexity is located there;

Algorithm 1 Flooding based ADMM - $l_2$ Algorithm.

1: **Kernel 1**: Initialization
2: $\forall j \in J, i \in N_c(j) : z_j^{(0)} = 0.5, \lambda_j^{(0)} = 0$
3: $\forall i \in I : n_i = \frac{2}{\mu}$
4: **for all** $k = 1 \rightarrow q$ **when** stop criterion = false **do**
5: **Kernel 2**: For all variable nodes in the code
6: **for all** $i \in I, j \in N_v(i)$ **do**
7: $t_i^{(k)} = \sum_{j \in N_v(i)} (z_j^{(k-1)} - \lambda_j^{(k-1)})$
8: $L_i^{(k)} = I[0,1](\frac{1}{d_{v,i} - 2\frac{\alpha}{\mu}} (t_i^{(k)} - n_i - \frac{\alpha}{\mu}))$
9: **end for**
10: **Kernel 3**: For all check nodes in the code
11: **for all** $j \in J, i \in N_c(j)$ **do**
12: $z_j^{(k)} = I[P_{dc_j}] (\rho L_i^{(k)} + (1-\rho)z_j^{(k-1)} + \lambda_j^{(k-1)})$
13: $\lambda_j^{(k)} = \lambda_j^{(k-1)} + \rho L_i^{(k)} + (1-\rho)z_j^{(k-1)} - z_j^{(k)}$
14: **end for**
15: **end for**
16: **Kernel 4**: Hard decisions from soft-values
17: $\forall i \in I : \hat{c}_i = \left( \sum_{j \in N_v(i)} L_i \rightarrow j \right) > 0.5$


The VN and CN computation kernels

One broadcasted message

$$\gamma_i = \frac{\left( \sum (\lambda_j + z_j) - \frac{LLR_i}{\mu} \right) - \frac{\alpha}{\mu}}{\text{deg}_V N - \frac{2\alpha}{\mu}}$$

Two « messages » per VN

$$\omega_i = \rho \times \lambda_j^{(k-1)} + \left(1 - \rho\right) L_{j \rightarrow i} \stackrel{(k-1)}{=\lambda_j^{(k-1)}}$$

$$z = \prod_{d_{cj}} (\omega)$$

$$\lambda_{j \rightarrow i} = \omega_i - z_i$$

$$L_{j \rightarrow i}^{(k)} = \left(z_j^{(k)}\right)_i - \left(\lambda_j^{(k)} \right)_i$$
The VN and CN processing kernels

One broadcasted message

$$\gamma_i = \frac{\left( \sum (\lambda_j + z_j) - \frac{LLR_i}{\mu} \right) - \frac{\alpha}{\mu}}{\deg_{VN} - \frac{2\alpha}{\mu}}$$

Two « messages » per VN

$$\omega_i = \rho \times L_{i\rightarrow j}^k + (1 - \rho)z_j^{(k-1)} + \lambda_j^{(k-1)}$$

$$z = \prod_{P_{dcj}} (\omega)$$

$$\lambda_{j\rightarrow i}^k = \omega_i - z_i$$

$$L_{j\rightarrow i}^{(k)} = (z_j^{(k)})_i - (\lambda_j^{(k)})_i$$
The « Euclidian projection » task

- Euclidian projection operation is not trivial at all,
  - Lots of arithmetic operations,
  - 4 conditional statements, that break computation parallelism,
  - Many sequential sections exist due to data dependencies between computations,

- Except arithmetic operations,
  - Data clipping in [0.0, 1.0] range,
  - Data sorting (deg_cn) required twice,
  \{	ext{sorted values, initial positions}\} = \text{SORT}(\text{values})

- It is already the simplified version of the Euclidian projection…
  - Less straightforward than Min-Sum algorithm,

Algorithm 2 Projection to the convex polytope.

1: function \text{PROJECTION}(x_j:\text{float values})
2: \text{if } \forall j \in [0,d_c[, x_j \leq 0 \text{ then}
3: \text{return } \{0,0,\ldots,0\}
4: \text{else if } \forall j \in [0,d_c[, x_j > 1 \text{ then}
5: \text{return } \{1,1,\ldots,1\}
6: \text{end if}
7: \{x^r,p^r\} = \text{Sort in Ascending Order and Store Positions } (x)
8: x^rc = \text{clamp}(x^r, [0,1])
9: cp = \sum_{i=0}^{d_c-1} x_i^rc
10: f = \lfloor cp \rfloor - \lfloor cp \rfloor \mod 2
11: sc = \sum_{i=0}^{f} x_i^rc - \sum_{i=f+1}^{d_c-1} x_i^rc
12: \text{if } sc \leq r \text{ then}
13: \text{return reorder}({x^rc,p^r})
14: \text{end if}
15: \forall j \in [0,d_c[, y_j = \begin{cases} (x_j^rc - 1) & \text{if } j \leq f \\ -x_j^rc & \text{otherwise} \end{cases}
16: \{y^r,p^r\} = \text{Sort in Ascending Order and Store Positions } (y)
17: \text{Set } \beta_{\max} = \frac{1}{2}(y_{f+1}^r - y_{f+2}^r)
18: \text{Construct a set of breakpoints } B = \{y_i^r | 0 \leq i \leq d_c-1; 0 \leq y_i^r \leq \beta_{\max}\}
19: \forall j \in [0,d_c[, y_j^r(\beta) = \begin{cases} \text{clamp} (y_j^r - \beta, [0,1]) & \text{if } j \leq f \\ \text{clamp} (y_j^r + \beta, [0,1]) & \text{otherwise} \end{cases}
20: \text{March through the breakpoints to find } i \mid \sum_{j=0}^{d_c-1} y_j^r(\beta) \leq r
21: \text{Find } \beta_{\text{opt}} \in [\beta_{i-1}, \beta_i] \text{ by solving Equation (4.28) in [39]}
22: \text{return reorder} (y^r(\beta_{\text{opt}}), p^r)
23: \text{end function}
Comparison with traditional LDPC decoding algorithms

<table>
<thead>
<tr>
<th></th>
<th>MSA</th>
<th></th>
<th></th>
<th>ADMM [30]</th>
<th>ADMM-(l_2) [37]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>add &amp; sub</strong></td>
<td>VN</td>
<td>CN</td>
<td>VN</td>
<td>VN</td>
<td>VN</td>
</tr>
<tr>
<td></td>
<td>2(d_v) - 1</td>
<td>2(d_v) - 1</td>
<td>2(d_v)</td>
<td>2(d_v) + 2</td>
<td>2(d_v) + 2</td>
</tr>
<tr>
<td><strong>multiply &amp; div</strong></td>
<td></td>
<td></td>
<td>4(d_c)</td>
<td>1</td>
<td>2(d_c)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2(d_c)</td>
<td></td>
</tr>
<tr>
<td><strong>arctan</strong></td>
<td></td>
<td></td>
<td>9(d_c)</td>
<td>2</td>
<td>2(d_c)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6(d_c)</td>
<td></td>
</tr>
<tr>
<td><strong>min, max, abs, xor, cmp</strong></td>
<td>9(d_c)</td>
<td>6(d_c)</td>
<td>2(d_c)</td>
<td>1</td>
<td>1(d_c)</td>
</tr>
<tr>
<td><strong>projection</strong></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1(d_c)</td>
</tr>
<tr>
<td><strong>Memory access</strong></td>
<td>2(d_v) + 1</td>
<td>2(d_c)</td>
<td>2(d_v) + 1</td>
<td>2(d_v) + 2</td>
<td>5(d_c)</td>
</tr>
<tr>
<td><strong>Memory reads</strong></td>
<td>2(d_c)</td>
<td>2(d_c)</td>
<td>2(d_v) + 1</td>
<td>2(d_v) + 1</td>
<td>3(d_c)</td>
</tr>
<tr>
<td><strong>Memory writes</strong></td>
<td>2(d_c)</td>
<td>2(d_c)</td>
<td>2(d_v) + 1</td>
<td>2(d_v) + 1</td>
<td>3(d_c)</td>
</tr>
</tbody>
</table>

From a decoding point of view CN processing consume more than 80% of the execution time

Execution time profiling of a « naive » ADMM software implementation (% of the total decoding time)

<table>
<thead>
<tr>
<th>Code</th>
<th>SNR=1.5dB</th>
<th></th>
<th></th>
<th>SNR=2dB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VN</td>
<td>CN</td>
<td>Proj.</td>
<td>Sort</td>
</tr>
<tr>
<td>576 \times 288</td>
<td>15</td>
<td>85</td>
<td>53</td>
<td>38.5</td>
</tr>
<tr>
<td>1152 \times 288</td>
<td>14</td>
<td>86</td>
<td>60</td>
<td>45</td>
</tr>
<tr>
<td>2304 \times 1152</td>
<td>15</td>
<td>86</td>
<td>54</td>
<td>36</td>
</tr>
<tr>
<td>2640 \times 1320</td>
<td>15</td>
<td>85</td>
<td>52</td>
<td>38</td>
</tr>
<tr>
<td>4000 \times 2000</td>
<td>15</td>
<td>85</td>
<td>51</td>
<td>38</td>
</tr>
</tbody>
</table>

Execution time profiling obtained thanks to X. Liu open-source C++ ADMM decoder sites.google.com/site/xishuoliu/codes.
Comparison with traditional LDPC decoding algorithms

Amount of computations involved in VN/CN processing for different LDPC decoding algorithms

<table>
<thead>
<tr>
<th></th>
<th>MSA</th>
<th>SPA</th>
<th>ADMM [30]</th>
<th>ADMM-l₂ [37]</th>
</tr>
</thead>
<tbody>
<tr>
<td>add &amp; sub</td>
<td>VN</td>
<td>CN</td>
<td>VN</td>
<td>CN</td>
</tr>
<tr>
<td></td>
<td>2d_v + 1</td>
<td>2d_v - 1</td>
<td>2d_v</td>
<td>4d_c</td>
</tr>
<tr>
<td>multiply &amp; div</td>
<td></td>
<td>4d_c</td>
<td>2d_v + 2</td>
<td>4d_c</td>
</tr>
<tr>
<td>arctan1/−1</td>
<td></td>
<td>2d_c</td>
<td>2d_c</td>
<td>2d_c</td>
</tr>
<tr>
<td>min, max, abs, xor, cmp</td>
<td>9d_c</td>
<td>6d_c</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>projection*</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Memory access</td>
<td>2d_v + 1</td>
<td>2d_v</td>
<td>2d_v + 2</td>
<td>5d_c</td>
</tr>
<tr>
<td>Memory reads</td>
<td>2d_v</td>
<td>2d_v</td>
<td>2d_v + 1</td>
<td>3d_c</td>
</tr>
<tr>
<td>Memory writes</td>
<td>2d_v</td>
<td>2d_v</td>
<td>2d_v + 1</td>
<td>2d_c</td>
</tr>
</tbody>
</table>

Euclidian projection is more than 60% of the CN processing time

Execution time profiling of a « naive » ADMM software implementation (% of the total decoding time)

<table>
<thead>
<tr>
<th>Code</th>
<th>SNR=1.5dB</th>
<th>SNR=2dB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VN</td>
<td>CN</td>
</tr>
<tr>
<td>576 x 288</td>
<td>15</td>
<td>85</td>
</tr>
<tr>
<td>1152 x 288</td>
<td>14</td>
<td>86</td>
</tr>
<tr>
<td>2304 x 1152</td>
<td>15</td>
<td>86</td>
</tr>
<tr>
<td>2640 x 1320</td>
<td>15</td>
<td>85</td>
</tr>
<tr>
<td>4000 x 2000</td>
<td>15</td>
<td>85</td>
</tr>
</tbody>
</table>

Execution time profiling obtained thanks to X. Liu open-source C++ ADMM decoder sites.google.com/site/xishuoliu/codes.
Comparison with traditional LDPC decoding algorithms

### Amount of computations involved in VN/CN processing for different LDPC decoding algorithms

<table>
<thead>
<tr>
<th></th>
<th>MSA</th>
<th>SPA</th>
<th>ADMM [30]</th>
<th>ADMM-(l_2) [37]</th>
</tr>
</thead>
<tbody>
<tr>
<td>add &amp; sub</td>
<td>VN</td>
<td>CN</td>
<td>VN</td>
<td>CN</td>
</tr>
<tr>
<td>multiply &amp; div</td>
<td>2(d_v) - 1</td>
<td>2(d_v) - 1</td>
<td>4(d_c)</td>
<td>2(d_v) + 2</td>
</tr>
<tr>
<td>arctan(1/) - 1</td>
<td></td>
<td></td>
<td>2(d_c)</td>
<td></td>
</tr>
<tr>
<td>min, max, abs, xor, cmp</td>
<td>9(d_c)</td>
<td>6(d_c)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>projection*</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Memory access</td>
<td>2(d_v) + 1</td>
<td>2(d_c)</td>
<td>2(d_v) + 1</td>
<td>5(d_c)</td>
</tr>
<tr>
<td>Memory reads</td>
<td>2(d_v) + 1</td>
<td>2(d_c)</td>
<td>2(d_v) + 2</td>
<td>5(d_c)</td>
</tr>
<tr>
<td>Memory writes</td>
<td></td>
<td></td>
<td></td>
<td>5(d_c)</td>
</tr>
</tbody>
</table>

### Execution time profiling of a « naive » ADMM software implementation (% of the total decoding time)

<table>
<thead>
<tr>
<th>Code</th>
<th>SNR=1.5dB</th>
<th>SNR=2dB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VN</td>
<td>CN</td>
</tr>
<tr>
<td>576 x 288</td>
<td>15</td>
<td>85</td>
</tr>
<tr>
<td>1152 x 288</td>
<td>14</td>
<td>86</td>
</tr>
<tr>
<td>2304 x 1152</td>
<td>15</td>
<td>86</td>
</tr>
<tr>
<td>2640 x 1320</td>
<td>15</td>
<td>85</td>
</tr>
<tr>
<td>4000 x 2000</td>
<td>15</td>
<td>85</td>
</tr>
</tbody>
</table>

Both data sorting task consumes 80% of the Euclidian projection time.

---

Execution time profiling obtained thanks to X. Liu open-source C++ ADMM decoder sites.google.com/site/xishuoliu/codes.
Software implementation of the ADMM-l2 decoding algorithms
Features of targeted multi-core architecture (Intel Core-i7)

- Work focuses on multicore (Intel x86),
  - Efficient as (or more than) GPUs for ECCs [5, 6],
- Two parallel programming features,
  - SIMD programming model (Single Instruction, Multiple Data),
  - SPMT/MPMT programming model (Single Program, Multiple Threads),
- Targeted INTEL Core-i7 device:
  - SIMD => 8 floats can be processed per cycle;
  - SPMT => 4 physical processor cores
- Implementation challenges,
  - Take advantage of parallelization features (usage rate of SIMD and SPMT) cores;
  - Minimize computation complexity and memory footprint.

The parallelism levels available for SIMD parallelization

An « easy » parallelization is possible inside CN and VN elements. For instance, compute all in/out messages in parallel using SIMD feature. However, efficiency depends on CN/VN degree.

A « more complex » parallelization is also possible across CN and VN. For instance, execute the same computations with data from 8 different CNs. Needs an offline computation and message reordering.

An another « quite easy » parallelization way consists in decoding multiple frames in parallel with SIMD feature. However, complex conditional statements in Euclidian projection discard this approach for SIMD parallelization.
The first (naive) decoder implementation

- In 1st implementation parallelization was performed inside CNs/VNs,

- For VN elements,
  - Semi-parallel sum of message input messages,
  - Seq. message generations,

- For CN elements,
  - Semi-parallel \( \omega \)i computations from messages,
  - Semi-parallel Euclidian projection,
  - Semi-parallel message generation,

- Speed-up the processing but,
  - Usage rate of SIMD unit is lower than 100%,
    - VN degree usually in \{2, 3, 4, 6\},
    - CN degree usually in \{6, 7, 8, 11, 12\},
  - Some processing parts (e.g. sorting) generate or process scalar results and cannot be parallelized.

\[
\gamma_i = \frac{\sum (\lambda_j + z_j) - \frac{LLR_i}{\mu}}{\deg_{VN} \frac{2\alpha}{\mu}}
\]

\[
\omega_i = \rho \times L_{i \rightarrow j}^k + (1 - \rho)z_j^{(k-1)} + \lambda_j^{(k-1)}
\]

\[
z = \Pi_{\omega \rightarrow j} (\omega)
\]

\[
\lambda_{j \rightarrow i} = \omega_i - z_i
\]

\[
L_{j \rightarrow i}^{(k)} = (z_j^{(k)})_i - (\lambda_j^{(k)})_i
\]
The second (improved) decoder implementation

- In 2nd implementation parallelization inside and **across** CNs/VNs,

- For VN elements,
  - Fully-// sum of message input messages,
  - Fully-// message generations,

- For CN elements,
  - Fully-// \( \omega_i \) computation and message,
  - Semi-parallel Euclidian projection,
    ✓ Fully-// 1st data sorting (done before projection),
  - Fully-// message generation,

- Speed-up the processing but,
  ✓ Usage rate of SIMD unit is often equal to 100%,
  ✓ Some processing parts remain un-parallelized,

\[
\gamma_i = \left( \sum (\lambda_j + z_j) - \frac{LLR_i}{\mu} \right) - \frac{\alpha}{\mu} \\
\text{deg}_{VN} - \frac{2\alpha}{\mu}
\]

\[
\omega_i = \rho \times L_{i \rightarrow j}^k + (1 - \rho)z_{j}^{(k-1)} + \lambda_{j}^{(k-1)} \\
z = \Pi_{P_{de}} (\omega) \\
\lambda_{j \rightarrow i}^k = \omega_i - z_i \\
L_{j \rightarrow i}^{(k)} = (z_{j}^{(k)})_i - (\lambda_{j}^{(k)})_i
\]
Common optimizations for the parallelization approaches

The both sort processing that are sequential tasks were optimized in terms of latency. Selection of the best data sorting algorithm according to the need (value, position).

Algorithm 2 Projection to the convex polytope.

```
function Projection(p^v) { float value)
1: if yj \in [0, d_j], x_j \leq 0 then
2: return \{0, 0, \ldots, 0\}
3: else if yj \in [0, d_j], x_j \geq 1 then
4: return \{1, 1, \ldots, 1\}
5: end if
6: \{x^*, p^*\} = Sort in Ascending Order and Store Positions (x)
7: x^* = clamp(x^*, [0, 1])
8: c_p = \sum_{y=1}^{d_j} x_{yj}
9: f = (c_p - |c_p|) \mod 2
10: s_c = \sum_{y=1}^{d_j} x_{yj} - \sum_{y=1}^{d_j} x_{yj}^*
11: if s_c \leq r then
12: return \text{round}(\{x^*, p^*\})
13: end if
14: \forall yj \in [0, d_j], yj = \left\{ \begin{array}{ll}
\left(\frac{x_{yj}^* - 1}{-2r_j} \right) & \text{if } j \leq f \\
\left(\frac{x_{yj}^* + 1}{2r_j} \right) & \text{otherwise}
\end{array} \right.
15: \\{y^*, p^*\} = Sort in Ascending Order and Store Positions (y)
16: \beta_{\text{breakp}} = \frac{1}{2}(y_{d_j} + y_{d_j+1})
17: \\{\beta_{\text{breakp}}, \ldots, \beta_{\text{breakp}}\} = \text{Sort in Ascending Order (\{\beta_{\text{breakp}}\})}
18: \forall yj \in [0, d_j], yj = \left\{ \begin{array}{ll}
\text{clamp}(y_{yj}^* - \beta, [0, 1]) & \text{if } j \leq f \\
\text{clamp}(y_{yj}^* + \beta, [0, 1]) & \text{otherwise}
\end{array} \right.
19: \beta_{\text{breakp}} = \left\{ \beta_{\text{breakp}} \right\} + \left\{ \sum_{y=1}^{d_j} y_{yj}^* \right\} \leq r
20: \{y^*, p^*\} = \text{Sort in Ascending Order (\{y^*, p^*\})}
21: \{\beta_{\text{breakp}} \right\} \leq r
22: \text{return \text{round}(y^*, p^*)}
23: end function
```

Euclidian projection was implemented and accelerated thanks to SIMD feature, however:
- Reach only a partial SIMD usage (degc is often < SIMD width);
- Requiress horizontal computations that are slow in SIMD mode.
- Parts cannot be parallelized using SIMD (scalar or sequential processing).
The parallelism levels available for SPMD parallelization

- INTEL Core-i7 has many physical cores having each a SIMD unit,

- Processing different VN/CN in //,
  - Necessitate costly synchronization at runtime,
  - Reduce the decoder throughput compared to a single thread implementation.

- Processing different frames in //,
  - No synchronization required during decoding,
  - Easily scalable to other multicore targets,
  - Increase memory footprint (cache misses),

ADMM LDPC decoder 1

One decoder per physical core
Experiments
The targeted platform for experiments (a laptop computer)

- **Evaluation platform,**
  - ✔️ INTEL Haswell Core-i7 4960HQ CPU,
  - ✔️ 4 Physical Cores (PC) and 4 Logical Cores (LC),
  - ✔️ Turbo boost @3.6GHz when one core is switched on 3.4GHz otherwise.
  - ✔️ 256 KB of L2 cache, 6 MB of L3 cache,

- **Software decoders are compiled with Intel C++ compiler 2016,**

- **Experimental setup,**
  - ✔️ IEEE 802.16e (2304 × 1152 and 576 × 288),
  - ✔️ 200 decoding iterations are executed (max.),
  - ✔️ 32b floating point data format.
Measure of the ADMM-l2 decoder throughputs

Evaluation on a single processor core

Throughput increases according to the SNR value thanks to the stopping criterion

Throughputs reach about 3Mbps@2.0dB and up to 6Mbps@4.0dB for both codes

Low throughputs for low SNR values due to the high number of executed iterations

Evaluation on P processor cores

Throughputs scale quite well with the amount of physical processor cores [1 => 4]
xP speed-up are not strictly reached due to L3 cache pollution between processor cores

8 core experiment shows that logical cores slightly improve the decoding throughput

Fig. 4. Average number of iterations Vs throughput evolution (a) 2304 × 1152 LDPC code (b) 576 × 288 LDPC code.

Fig. 3. ADMM-l2 optimized decoder measured throughputs wrt the number of threads (a) 2304 × 1152 code (b) 576 × 288 code.
Measure of the ADMM-l2 decoder throughputs

Evaluation on a single processor core

Throughput increases according to the SNR value thanks to the stopping criterion

Throughputs reach about 3Mbps@$2.0$dB and up to 6Mbps@$4.0$dB for both codes

Low throughputs for low SNR values due to the 200 decoding iterations

Evaluation on P processor cores

Throughputs scale quite well with the amount of physical processor cores $[1 \rightarrow 4]$

xP speed-up are not strictly reached due to L3 cache pollution between processor cores

8 core experiment shows that logical cores slightly improve the decoding throughput

---

**Fig. 3.** ADMM-l2 optimized decoder measured throughputs wrt the number of threads (a) $2304 \times 1152$ code (b) $576 \times 288$ code.

**Fig. 4.** Average number of iterations Vs throughput evolution (a) $2304 \times 1152$ LDPC code (b) $576 \times 288$ LDPC code.
Conclusion & Future works
Current work conclusion

- ADMM-l2 algorithm is of great interest due to its high correction performances,

- ADMM-l2 is composed of massively parallel computations,
  - Flooding schedule makes parallelization quite straightforward,

- ADMM-l2 has a high-computation complexity of the CN kernels,
  - Mainly due to Euclidian projection,

- Throughput performances are honorable on x86 target for medium SNR values.

Continuous research effort to reach higher throughputs for a large set of applications!

Sources in open-source: [http://github.com/blegal](http://github.com/blegal)
Since the submission … & future works

- Reducing the decoding computation complexity,
  - Layered scheduling technique (horizontal [7] or vertical [8]),
  - Simplifying the Euclidian projection processing ???

- Switching to many-core devices ?
  - More computation parallelism but other hardware constraints to manage:
    - Instruction replay,
    - Memory latency, etc.

- Switching to hardware design ?
  - ADMM works well with float values not yet with fixed-point ones…