Multicore Implementation of LDPC Decoders based on ADMM Algorithm

Imen DEBBABI[1], Nadia KHOUJA[1], Fethi TLILI[1],
Bertrand LE GAL[2] and Christophe JEGO[2]

1 - SUP'COM, GRESCOM Lab,
University of Carthage, Tunisia

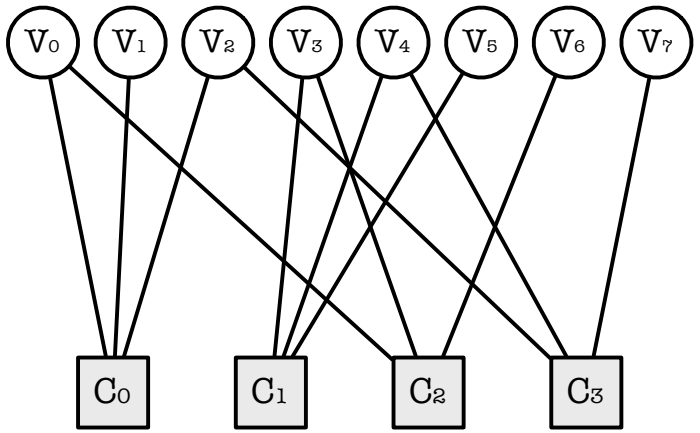
2 - Bordeaux-INP, IMS-lab., CNRS UMR 5218
University of Bordeaux, France

The LP decoding for LDPC codes

Introduction to LDPC codes

- LDPC codes are well-known Error Correction Codes working on blocs,
  - 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.

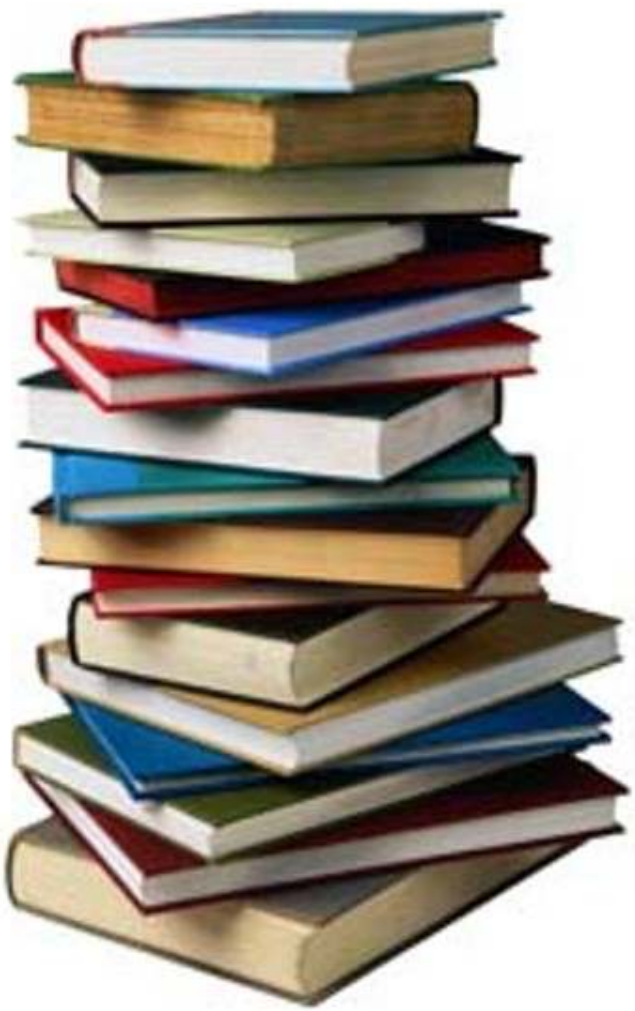
$$H = \begin{array}{c} \\ C_0 \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{cccccccc} V_0 & V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \end{array}$$



Tanner graph representation.

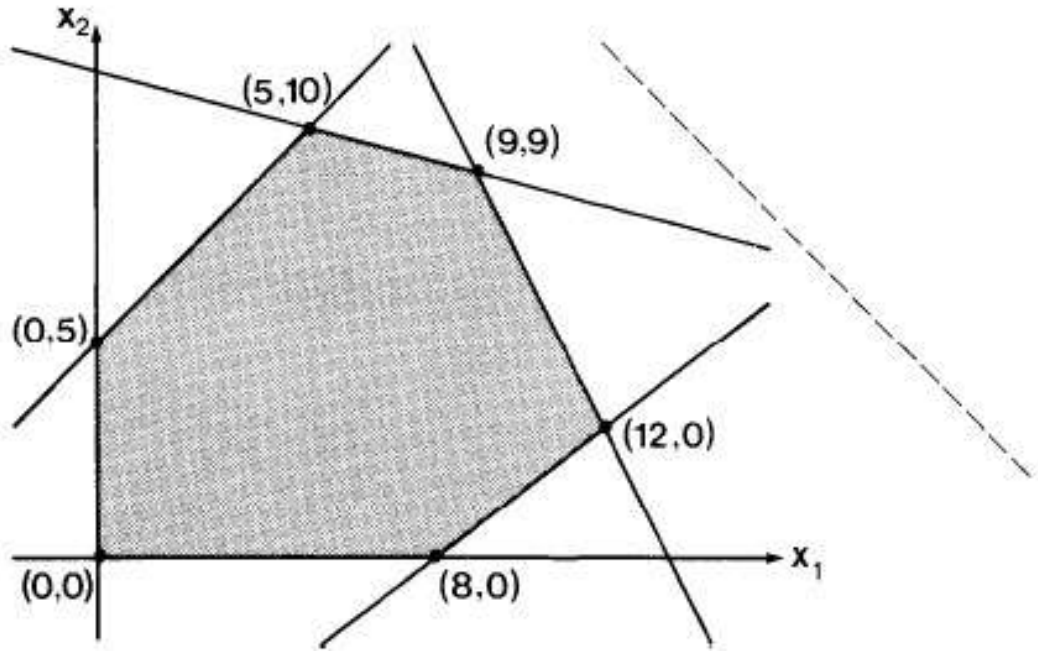
Related works on LDPC decoding

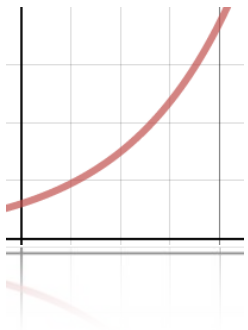
- During the last decade, lots of works focused on LDPC codes. For instance :
  - Find an « efficient » SPA approximation ,
    - SPA algorithm is efficient but complex to implement,
    - MS, OMS, NMS, 2NMS, lambda-min, ANMS, etc.
  - Reduce computation complexity through different computation schedules,
    - 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.





Increase mainly according to N, N-K and deg(Ci) parameters

[1] J. Feldman, Decoding Error-Correcting Codes via Linear Programming. **PhD thesis, Massachussets Institute of Technology**, 2003.

LP decoding of LDPC codes

- 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.**

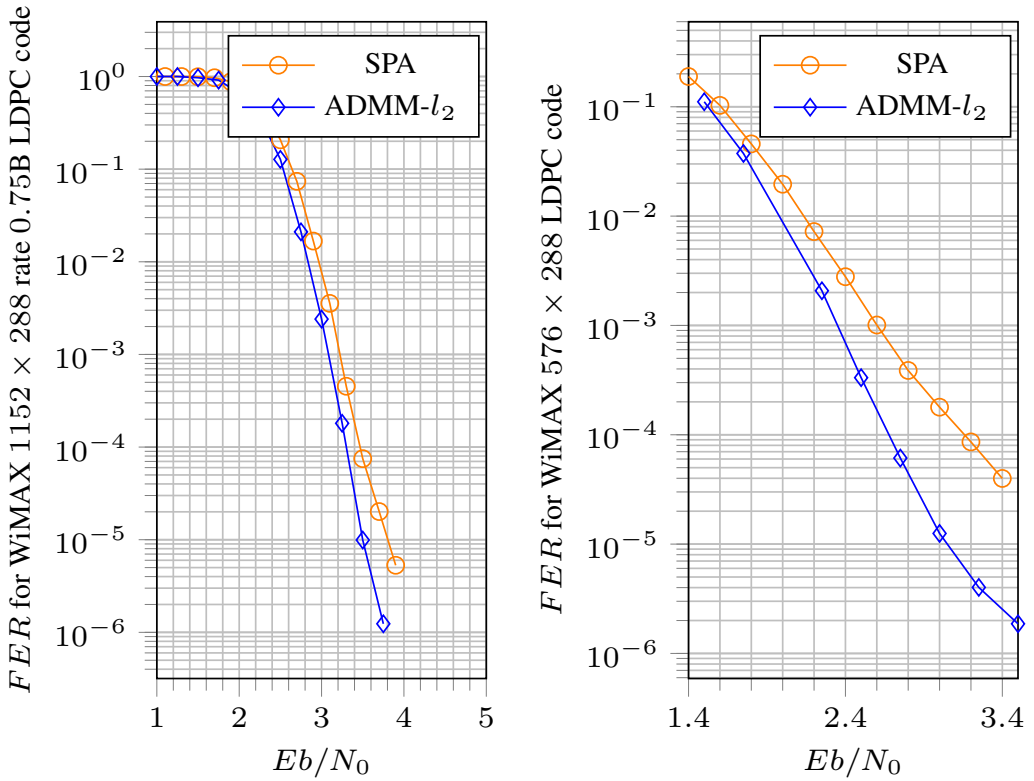


Fig. 1. FER comparison of ADMM-l_2 penalized decoders with SPA decoders on AWGN channel.

[2] Xiaojie Zhang and Paul H.Siegel, "Efficient iterative LP decoding of LDPC codes with alternating direction method of multipliers," **IEEE International Symposium on Information Theory (ISIT)**, 2013.

[3] X. Jiao, H. Wei, J. Mu, and C. Chen, "Improved ADMM penalized decoder for irregular low-density parity-check codes," **IEEE Communications Letters**, June 2015.

[4] H. Wei, X. Jiao, and J. Mu, "Reduced-complexity linear programming decoding based on ADMM for LDPC codes," **IEEE Communications Letters**, June 2015.

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 -l_2 Algorithm.

Kernel 1: Initialization

$\forall j \in \mathcal{J}, i \in N_c(j) : z_{j\to i}^{(0)} = 0.5,\ \lambda_{j\to i}^{(0)} = 0$

$\forall i \in \mathcal{I} : n_i = \frac{\gamma_i}{\mu}$

for all $k = 1 \to q$ **when** stop criterion = false **do**

  Kernel 2: For all variable nodes in the code

  for all $i \in \mathcal{I}, j \in N_v(i)$ **do**

    $t_i^{(k)} = \sum_{j \in N_v(i)} (z_{j\to i}^{(k-1)} - \lambda_{j\to i}^{(k-1)})$

    $L_{i\to j}^{(k)} = \Pi_{[0,1]}(\frac{1}{d_{v_i} - 2\frac{\alpha}{\mu}}(t_i^{(k)} - n_i - \frac{\alpha}{\mu}))$

  end for

  Kernel 3: For all check nodes in the code

  for all $j \in \mathcal{J}, i \in N_c(j)$ **do**

    $z_{j\to i}^{(k)} = \Pi_{P_{d_{c_j}}}[\rho L_{i\to j}^{(k)} + (1-\rho) z_{j\to i}^{(k-1)} + \lambda_{j\to i}^{(k-1)}]$

    $\lambda_{j\to i}^{(k)} = \lambda_{j\to i}^{(k-1)} + \rho L_{i\to j}^{(k)} + (1-\rho) z_{j\to i}^{(k-1)} - z_{j\to i}^{(k)}$

  end for

end for

Kernel 4: Hard decisions from soft-values

$\forall i \in \mathcal{I} : \hat{c}_i = \left(\sum_{j \in N_v(i)} L_{i\to j} \right) > 0.5$

[2] Xiaojie Zhang and Paul H.Siegel, "Efficient iterative LP decoding of LDPC codes with alternating direction method of multipliers," **IEEE International Symposium on Information Theory (ISIT)**, 2013.

[3] X. Jiao, H. Wei, J. Mu, and C. Chen, "Improved ADMM penalized decoder for irregular low-density parity-check codes," **IEEE Communications Letters**, June 2015.

Formulation of the ADMM decoding algorithm

- 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 VNs;
  - Kernel 3, processes all CNs;
  - Kernel 4, takes hard decision;

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

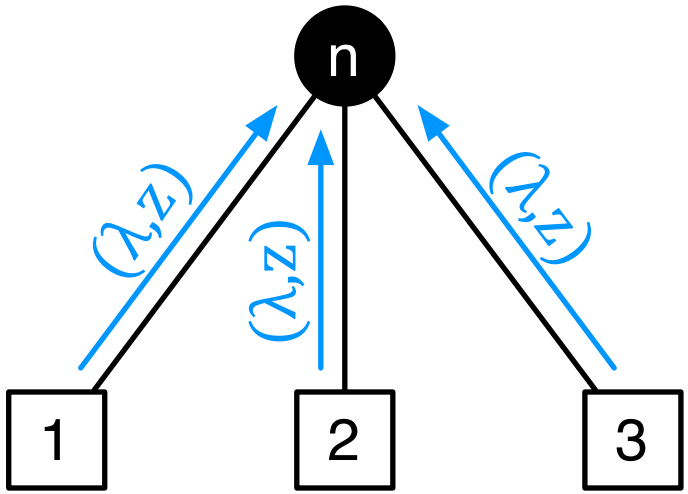
Algorithm 1 Flooding based ADMM -l_2 Algorithm.

Kernel 1: Initialization
$\forall j \in \mathcal{J}, i \in N_c(j) : z_{j\to i}^{(0)} = 0.5,\ \lambda_{j\to i}^{(0)} = 0$
$\forall i \in \mathcal{I} : n_i = \frac{\gamma_i}{\mu}$
for all $k = 1 \to q$ **when** stop criterion = false **do**
  Kernel 2: For all variable nodes in the code
  for all $i \in \mathcal{I}, j \in N_v(i)$ **do**
    $t_i^{(k)} = \sum_{j \in N_v(i)} (z_{j\to i}^{(k-1)} - \lambda_{j\to i}^{(k-1)})$
    $L_{i\to j}^{(k)} = \Pi_{[0,1]}(\frac{1}{d_{v_i} - 2\frac{\alpha}{\mu}}(t_i^{(k)} - n_i - \frac{\alpha}{\mu}))$
  end for
  Kernel 3: For all check nodes in the code
  for all $j \in \mathcal{J}, i \in N_c(j)$ **do**
    $z_{j\to i}^{(k)} = \Pi_{P_{d_{c_j}}}[\rho L_{i\to j}^{(k)} + (1-\rho) z_{j\to i}^{(k-1)} + \lambda_{j\to i}^{(k-1)}]$
    $\lambda_{j\to i}^{(k)} = \lambda_{j\to i}^{(k-1)} + \rho L_{i\to j}^{(k)} + (1-\rho) z_{j\to i}^{(k-1)} - z_{j\to i}^{(k)}$
  end for
end for
Kernel 4: Hard decisions from soft-values
$\forall i \in \mathcal{I} : \hat{c}_i = \left(\sum_{j \in N_v(i)} L_{i\to j} \right) > 0.5$

[2] Xiaojie Zhang and Paul H.Siegel, "Efficient iterative LP decoding of LDPC codes with alternating direction method of multipliers," **IEEE International Symposium on Information Theory (ISIT)**, 2013.

[3] X. Jiao, H. Wei, J. Mu, and C. Chen, "Improved ADMM penalized decoder for irregular low-density parity-check codes," **IEEE Communications Letters**, June 2015.

The VN and CN computation kernels



One broadcasted message

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

Two « messages » per VN

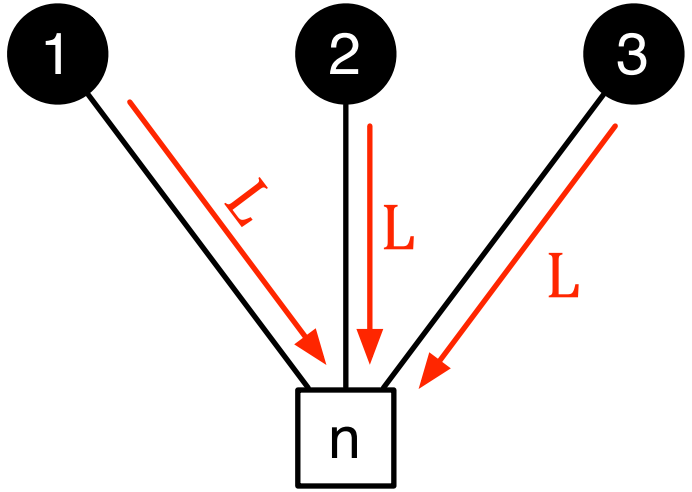
$$\omega_i = \rho \times L^k_{i\to j} + (1-\rho) z_j^{(k-1)} + \lambda_j^{(k-1)}$$

$$z = \Pi_{P_{dc_j}}(\omega)$$

$$\lambda^k_{j\to i} = \omega_i - z_i$$

$$L^{(k)}_{j\to i} = (z_j^{(k)})_i - (\lambda_j^{(k)})_i$$

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

$$\omega_i = \rho \times L^k_{i\to j} + (1-\rho) z^{(k-1)}_j + \lambda^{(k-1)}_j$$

$$z = \Pi_{P_{dc_j}}(\omega)$$

$$\lambda^k_{j\to i} = \omega_i - z_i$$

$$L^{(k)}_{j\to i} = (z^{(k)}_j)_i - (\lambda^{(k)}_j)_i$$

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,
  - ➡ { *sorted values, initial positions* } = *SORT(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** PROJECTION(x_j : float values)
2: **if** $\forall j \in [0, d_c[, x_j \leq 0$ **then**
3: **return** $\{0, 0, \ldots, 0\}$
4: **else if** $\forall j \in [0, d_c[, x_j \geq 1$ **then**
5: **return** $\{1, 1, \ldots, 1\}$
6: **end if**
7: $\{x^r, p^r\}$ = 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: **if** $sc \leq r$ **then**
13: **return** reorder$(\{x^{rc}, p^r\})$
14: **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: $\{y^r, p^r\}$ = Sort in Ascending Order and Store Positions (y)
17: Set $\beta_{max} = \frac{1}{2}(y_{f+1}^r - y_{f+2}^r)$
18: Construct a set of breakpoints $\mathcal{B} = \{y_i^r \mid 0 \leq i \leq d_{c-1}; 0 \leq y_i^r \leq \beta_{max}\}$
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: March through the breakpoints to find $i \mid \sum_{j=0}^{d_c-1} y_j^r(\beta) \leq r$
21: Find $\beta_{opt} \in [\beta_{i-1}, \beta_i]$ by solving Equation (4.28) in [39]
22: **return** reorder$(y^r(\beta_{opt})\ ,\ p^r)$
23: **end function**

Comparison with traditional LDPC decoding algorithms

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

	MSA		SPA		ADMM [30]		ADMM-l_2 [37]	
	VN	CN	VN	CN	VN	CN	VN	CN
add & sub	$2d_v - 1$		$2d_v - 1$		$2d_v$	$4d_c$	$2d_v + 2$	$4d_c$
multiply & div				$4d_c$	1	$2d_c$	2	$2d_c$
$\arctan^{1/-1}$				$2d_c$				
min, max, abs, xor, cmp		$9d_c$		$6d_c$	2		2	
projection*						1		1
Memory access	$2d_v + 1$	$2d_c$	$2d_v + 1$	$2d_c$	$2d_v + 2$	$5d_c$	$2d_v + 2$	$5d_c$
Memory reads	–	–	–	–	$2d_v + 1$	$3d_c$	$2d_v + 1$	$3d_c$
Memory writes	–	–	–	–	1	$2d_c$	1	$2d_c$

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)

Code	SNR=1.5dB				SNR=2dB			
	VN	CN	Proj.	Sort	VN	CN	Proj.	Sort
576 × 288	15	85	53	38.5	16	84	50	41
1152 × 288	14	86	60	45	15	85	59	44
2304 × 1152	15	86	54	36	16	84	49	38.5
2640 × 1320	15	85	52	38	17	83	47.5	41
4000 × 2000	15	85	51	38	18	82	46	41.5

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

	MSA		SPA		ADMM [30]		ADMM-l_2 [37]	
	VN	CN	VN	CN	VN	CN	VN	CN
add & sub	$2d_v - 1$		$2d_v - 1$		$2d_v$	$4d_c$	$2d_v + 2$	$4d_c$
multiply & div				$4d_c$	1	$2d_c$	2	$2d_c$
$\arctan^{1/-1}$				$2d_c$				
min, max, abs, xor, cmp		$9d_c$		$6d_c$	2		2	
projection*						1		1
Memory access	$2d_v + 1$	$2d_c$	$2d_v + 1$	$2d_c$	$2d_v + 2$	$5d_c$	$2d_v + 2$	$5d_c$
Memory reads	–	–	–	–	$2d_v + 1$	$3d_c$	$2d_v + 1$	$3d_c$
Memory writes	–	–	–	–	1	$2d_c$	1	$2d_c$

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)

Code	SNR=1.5dB				SNR=2dB			
	VN	CN	Proj.	Sort	VN	CN	Proj.	Sort
576 × 288	15	85	53	38.5	16	84	50	41
1152 × 288	14	86	60	45	15	85	59	44
2304 × 1152	15	86	54	36	16	84	49	38.5
2640 × 1320	15	85	52	38	17	83	47.5	41
4000 × 2000	15	85	51	38	18	82	46	41.5

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

	MSA		SPA		ADMM [30]		ADMM-l_2 [37]	
	VN	CN	VN	CN	VN	CN	VN	CN
add & sub	$2d_v - 1$		$2d_v - 1$		$2d_v$	$4d_c$	$2d_v + 2$	$4d_c$
multiply & div				$4d_c$	1	$2d_c$	2	$2d_c$
$\arctan^{1/-1}$				$2d_c$				
min, max, abs, xor, cmp		$9d_c$		$6d_c$	2		2	
projection*						1		1
Memory access	$2d_v + 1$	$2d_c$	$2d_v + 1$	$2d_c$	$2d_v + 2$	$5d_c$	$2d_v + 2$	$5d_c$
Memory reads	–	–	–	–	$2d_v + 1$	$3d_c$	$2d_v + 1$	$3d_c$
Memory writes	–	–	–	–	1	$2d_c$	1	$2d_c$

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

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

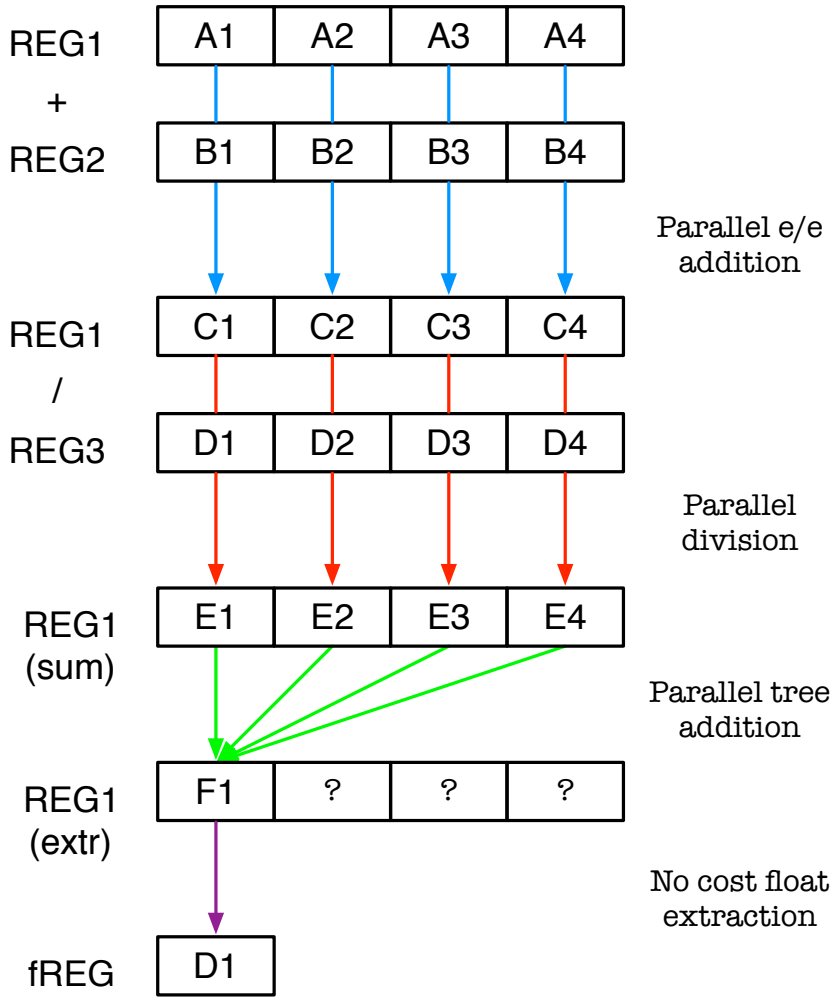
Code	SNR=1.5dB				SNR=2dB			
	VN	CN	Proj.	Sort	VN	CN	Proj.	Sort
576 × 288	15	85	53	38.5	16	84	50	41
1152 × 288	14	86	60	45	15	85	59	44
2304 × 1152	15	86	54	36	16	84	49	38.5
2640 × 1320	15	85	52	38	17	83	47.5	41
4000 × 2000	15	85	51	38	18	82	46	41.5

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)

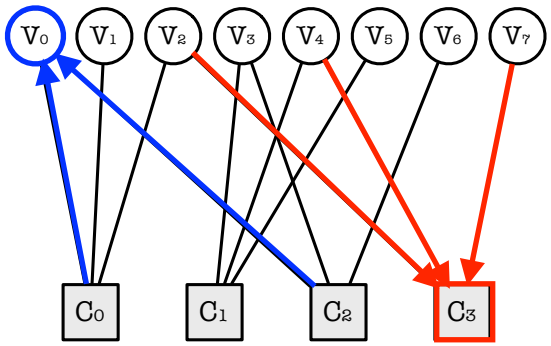
- Work focuses on multicore (Intel x86),
  - Efficient as (or more than) GPUs for ECCs [5, 6],
- Two parallel programming features,
  - SIMD programming model (Single Instruction, Multiple Data),
  - SPMT/MPMT programming model (Single Program, Multiple Threads),
- Targeted INTEL Core-i7 device:
  - 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.



[5] B. Le Gal, C. Leroux and C. Jego. Multi-Gb/s software decoding of Polar Codes. **IEEE Transactions on Signal Processing**, pages 349 – 359, January 2015.

[6] B. Le Gal and C. Jego. High-throughput multi-core LDPC decoders based on x86 processor. **IEEE Transactions on Parallel and Distributed Systems**, May 2015.

The parallelism levels available for SIMD parallelization

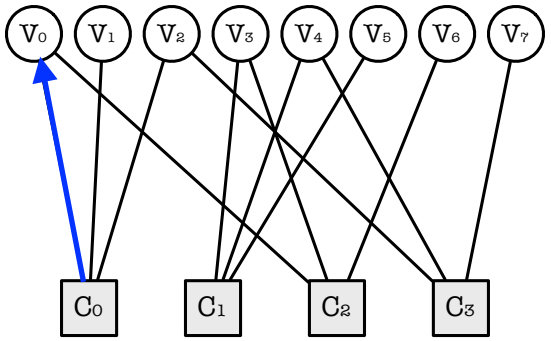


An « easy » parallelization is possible inside CN and VN elements. For instance, compute all in/out messages in parallel using SIMD feature.

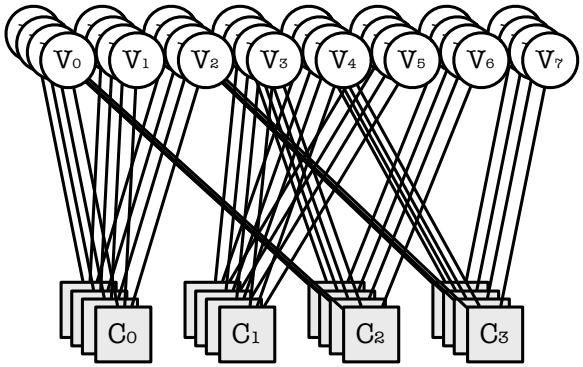
However, efficiency depends on CN/VN degree.

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.



Tanner graph representation.

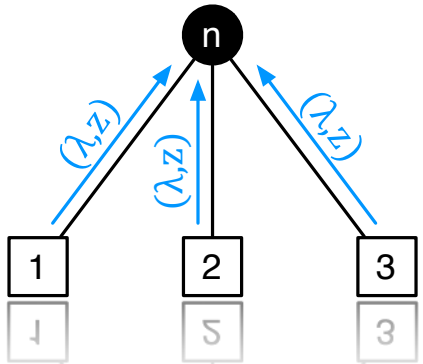


An another « quite easy » parallelization way consists in decoding multiple frames in parallel with SIMD feature.

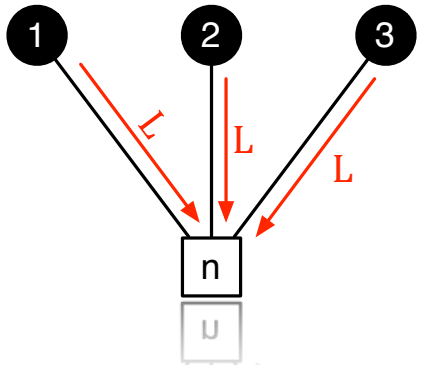
However, complex conditional statements in Euclidian projection discard this approach for SIMD parallelization.

The first (naive) decoder implementation

- In 1st implementation parallelization was performed inside CNs/VNs,
- For VN elements,
  - Semi-// sum of message input messages,
  - Seq. message generations,
- For CN elements,
  - Semi-// ωi computations from messages,
  - Semi-parallel Euclidian projection,
  - Semi-// 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 (eg. sorting) generate or process scalar results and cannot be parallelized.



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



$$\omega_i = \rho \times L^k_{i\to j} + (1-\rho) z_j^{(k-1)} + \lambda_j^{(k-1)}$$

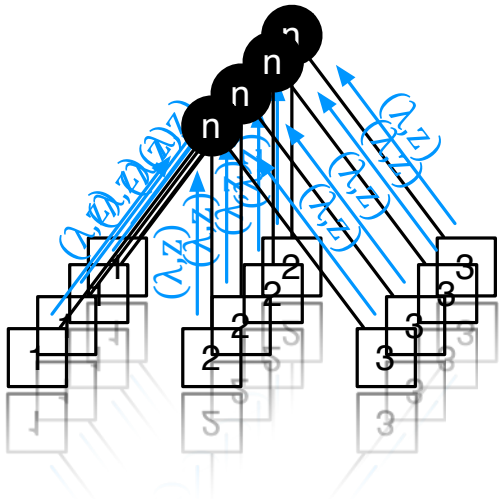
$$z = \Pi_{P_{d_{c_j}}}(\omega)$$

$$\lambda^k_{j\to i} = \omega_i - z_i$$

$$L^{(k)}_{j\to i} = (z_j^{(k)})_i - (\lambda_j^{(k)})_i$$

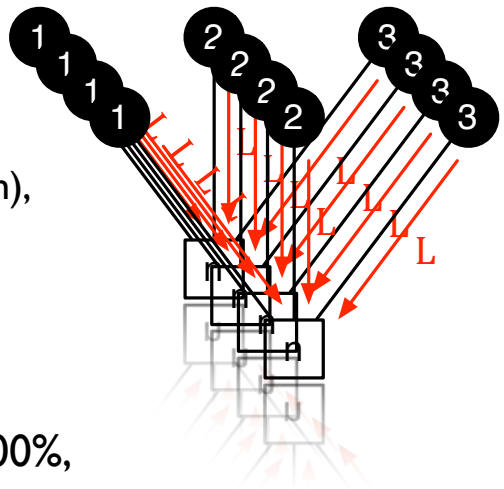
The second (improved) decoder implementation

- In 2nd implementation parallelization inside and **across** CNs/VNs,
- For VN elements,
  - Fully-// sum of message input messages,
  - Fully-// message generations,



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

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



$$\omega_i = \rho \times L^k_{i\to j} + (1-\rho) z_j^{(k-1)} + \lambda_j^{(k-1)}$$
$$z = \Pi_{P_{d_{c_j}}}(\omega)$$
$$\lambda^k_{j\to i} = \omega_i - z_i$$
$$L^{(k)}_{j\to i} = (z_j^{(k)})_i - (\lambda_j^{(k)})_i$$

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

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).

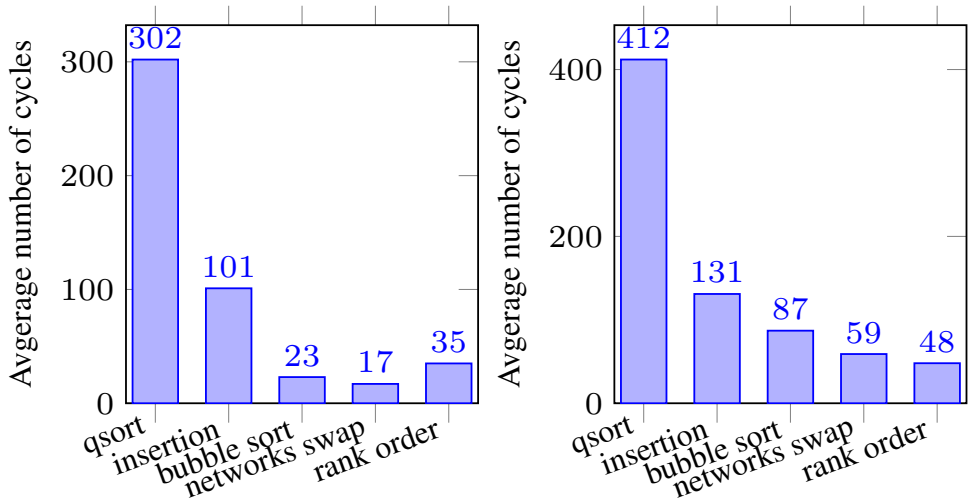


Fig. 2. Average number of cycles of (a) Reference sorting functions of 6 floats (b) Sorting functions of 6 floats keeping input positions.

Algorithm 2 Projection to the convex polytope.

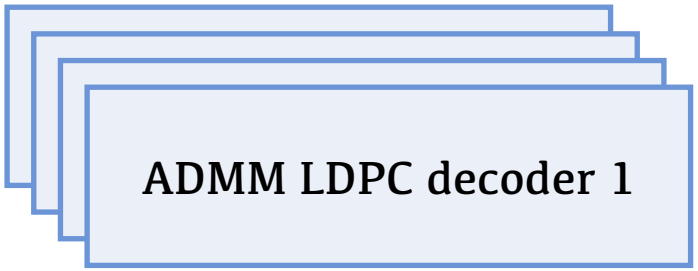
function PROJECTION(x_j : float values)
  if $\forall j \in [0, d_c[, x_j \leq 0$ **then**
    return $\{0, 0, \ldots, 0\}$
  else if $\forall j \in [0, d_c[, x_j \geq 1$ **then**
    return $\{1, 1, \ldots, 1\}$
  end if
  $\{x^r, p^r\}$ = Sort in Ascending Order and Store Positions (x)
  $x^{rc} = \text{clamp}(\, x^r, [0, 1])$
  $cp = \sum_{i=0}^{d_c-1} x_i^{rc}$
  $f = \lfloor cp \rfloor - \lfloor cp \rfloor \mod 2$
  $sc = \sum_{i=0}^{f} x_i^{rc} - \sum_{i=f+1}^{d_c-1} x_i^{rc}$
  if $sc \leq r$ **then**
    return reorder($\{x^{rc}, p^r\}$)
  end if
  $\forall j \in [0, d_c[, y_j = \begin{cases} (x_j^{rc} - 1) & \text{if } j \leq f \\ -x_j^{rc} & \text{otherwise} \end{cases}$
  $\{y^r, p^r\}$ = Sort in Ascending Order and Store Positions (y)
  Set $\beta_{max} = \frac{1}{2}(y_{f+1}^r - y_{f+2}^r)$
  Construct a set of breakpoints $\mathcal{B} = \{y_i^r \mid 0 \leq i \leq d_{c-1}; 0 \leq y_i^r \leq \beta_{max}\}$
  $\forall j \in [0, d_c[, y_j^r(\beta) = \begin{cases} \text{clamp}(y_j^r - \beta, [0, 1]) & \text{if } j \leq f \\ \text{clamp}(y_j^r + \beta, [0, 1]) & \text{otherwise} \end{cases}$
  March through the breakpoints to find $i \mid \sum_{j=0}^{d_c-1} y_j^r(\beta) \leq r$
  Find $\beta_{opt} \in [\beta_{i-1}, \beta_i]$ by solving Equation (4.28) in [39]
  return reorder($y^r(\beta_{opt})$, p^r)
end function

Euclidian projection was implemented and accelerated thanks to SIMD feature, however:

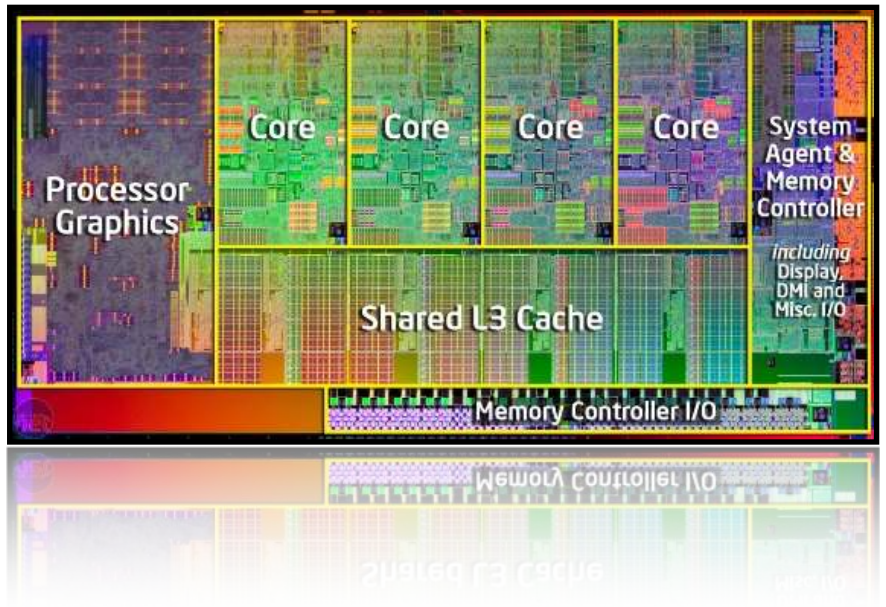
- Reach only a partial SIMD usage (degc is often < SIMD width);
- Requiers horizontal computations that are slow in SIMD mode.
- Parts cannot be parallelized using SIMD (scalar or sequential processing).

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 sciable to other multicore targets,
  - ✓ Increase memory footprint (cache misses),



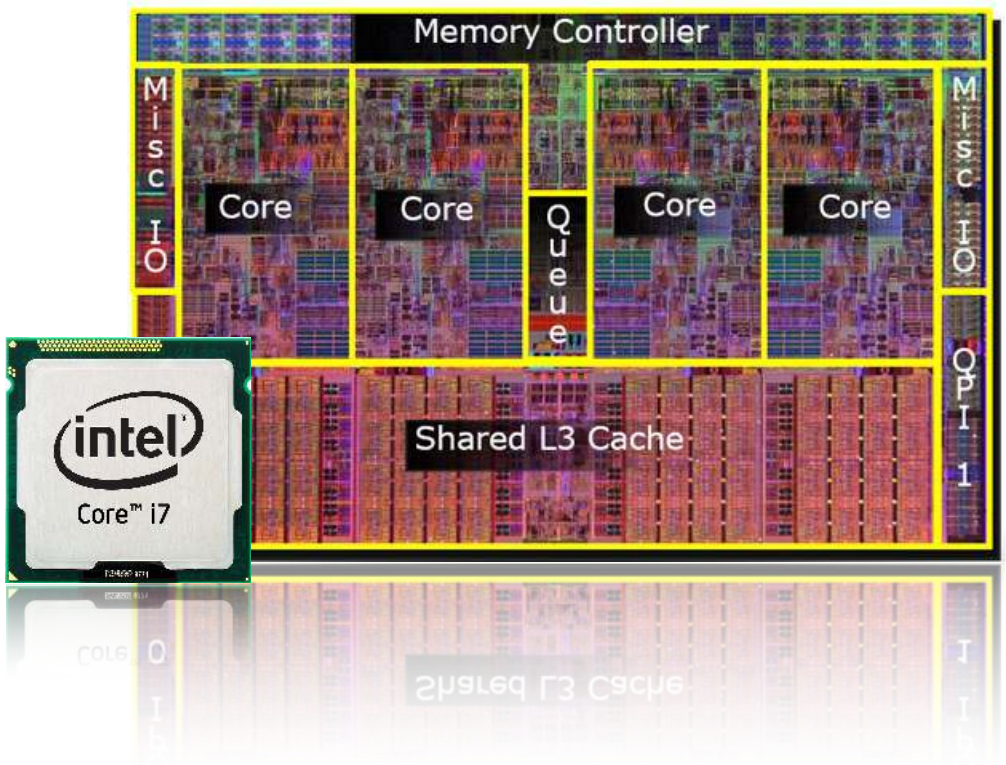
One decoder per physical core



Experiments

The targeted platform for experiments (a laptop computer)

- Evaluation plateform,
  - ✓ 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

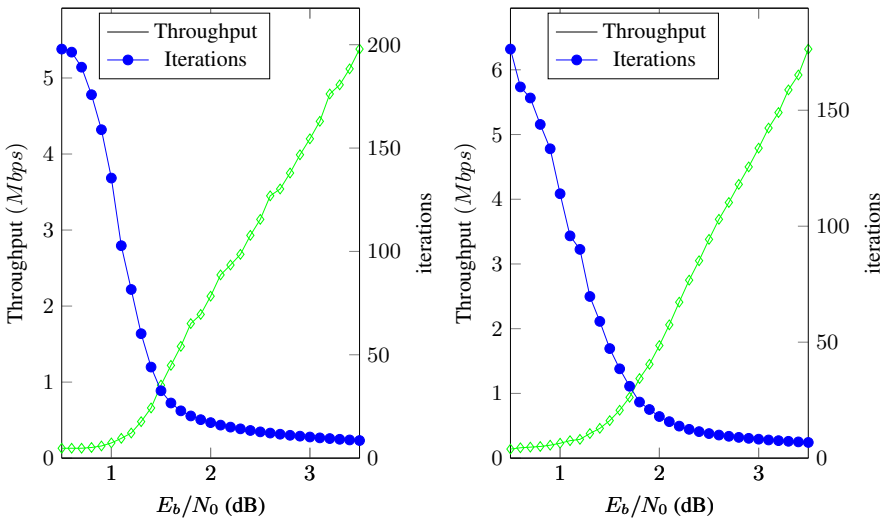


Fig. 4. Average number of iterations Vs throughput evolution (a) 2304×1152 LDPC code (b) 576×288 LDPC code.

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. 3. ADMM-l_2 optimized decoder measured throughputs *wrt* the number of threads (a) 2304×1152 code (b) 576×288 code.

Measure of the ADMM-l2 decoder throughputs

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

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 200 decoding 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

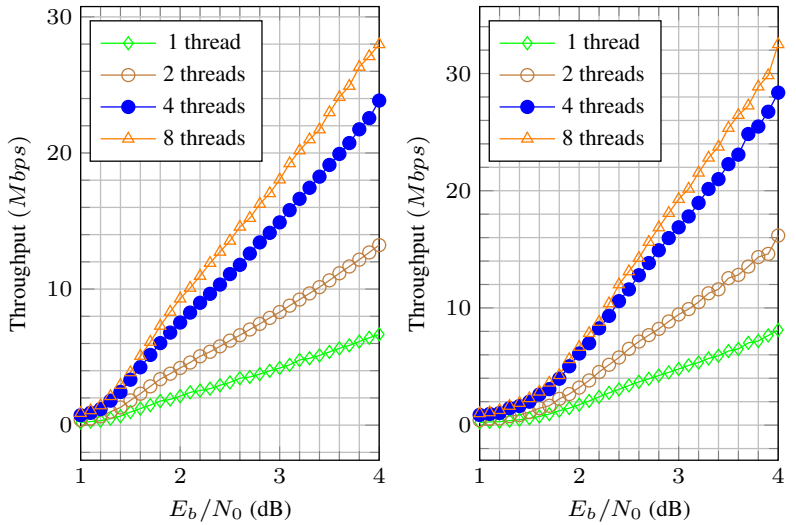


Fig. 3. ADMM-l_2 optimized decoder measured throughputs *wrt* the number of threads (a) 2304 × 1152 code (b) 576 × 288 code.

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

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…

[7] I. Debbabi, B. Le Gal, N. Khouja, F. Tlili and C. Jego. Fast Converging ADMM Penalized Algorithm for LDPC Decoding. **IEEE Communication Letters**, February 2016.

[8] I. Debbabi, B. Le Gal, N. Khouja, F. Tlili and C. Jego, Comparison of different schedulings for the ADMM based LDPC decoding, **Submitted** to the International Symposium on Turbo Codes & Iterative Information Processing, Brest France, September 2016.