AFF3CT: A Fast Forward Error Correction Toolbox!

Adrien Cassagne, Olivier Hartmann, Mathieu Leonardon, Kun He, Camille Leroux, Romain Tajan, Olivier Aumage, Denis Barthou, Thibaud Tonnellier, Vincent Pignoly, et al.

SoftwareX 10 (2019) 100345

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original Software Publication

AFF3CT: A Fast Forward Error Correction Toolbox!

Adrien Cassagne *, Olivier Hartmann, Mathieu Léonardon, Kun He, Camille Leroux, Romain Tajan, Olivier Aumage, Denis Barthou, Thibaud Tonnellier, Vincent Pignoly, Bertrand Le Gal, Christophe Jégo

Inria, Bordeaux Institute of Technology, LaBRI/CNRS, Bordeaux, France
CNRS IMS Laboratory, Bordeaux INP, University of Bordeaux, France

ARTICLE INFO

Article history:
Received 18 February 2019
Received in revised form 5 August 2019
Accepted 15 October 2019

Keywords:
Communication chain
Channel coding
Monte Carlo simulation
Forward error correction library
Digital modulation
Reproducible science
Multi-node
Multi-thread
Vectorization

ABSTRACT

AFF3CT is an open source toolbox dedicated to Forward Error Correction (FEC or channel coding). It supports a broad range of codes: from widespread turbo codes and Low-Density Parity-Check (LDPC) codes to more recent polar codes. The toolbox is written in C++ and can be used either as a simulator to quickly evaluate algorithms characteristics, or as a library in Software Defined Radio (SDR) systems or for other specific needs. Most of the decoding algorithm implementations aim at low latency and high throughput, targeting multiple Gb/s on modern CPUs. This is crucial in both simulation and SDR use cases: Monte Carlo simulations require high performance implementation as they commonly target the estimation of approximately 10^{12} bits. On the other hand, the implementations in real systems have to be very efficient to be competitive against dedicated hardware ones. Finally, AFF3CT emphasizes the reproducibility of state-of-the-art results by providing public references and open, modular source code.

Code metadata

Current code version	v2.3.5
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2019_43
Legal Code License	MIT
Code versioning system used	Git
Software code languages, tools, and services used	C++, Python, Bash, CMake
Compilation requirements, operating environments & dependencies	C++11 compatible compiler
If available Link to developer documentation/manual	https://aff3ct.readthedocs.io/en/v2.3.5/
Support email for questions	https://github.com/aff3ct/aff3ct/issues

1. Motivation and significance

It is now commonplace to state that Humanity has entered the era of communication. Moreover, all kinds of objects will increasingly also use communication technology, to exchange information in the *Internet of Things* (IoT). Despite their variety, all communication systems are based on a common abstract model proposed by Claude Shannon. In his seminal paper [1], he proposed to model a communication system with five components: an information source, a transmitter, a channel, a receiver and a destination. This model was later refined as shown in Fig. 1. The source produces a digital message to be transmitted. The channel encoder transforms it to make it less prone to errors.

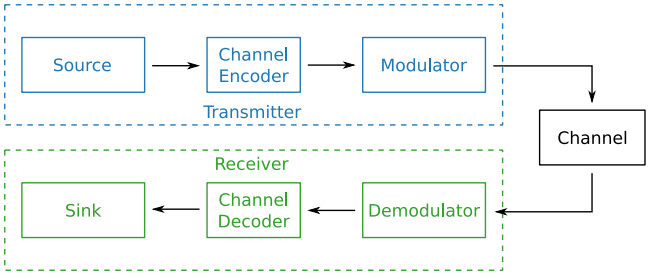


Fig. 1. Digital communication chain.

The modulator translates digital data into a physical signal. The channel alters the signal with some noise and distortions. On the receiver side, the components perform the inverse operations to retrieve the message produced by the source.

* Corresponding author.
E-mail address: adrien.cassagne@u-bordeaux.fr (A. Cassagne).

https://doi.org/10.1016/j.softx.2019.100345

Table 1
C/C++ open source channel coding simulators/libraries.

Name	Contributors	Lines	Commit		License	Polar	LDPC	Turbo	Turbo P.	BCH	RS	Conv.	RA	Rep.	Erasure
			First	Last											
AFF3CT	11	76k	2016	2019	MIT	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
eccpage	20	–	1989	2006	–	✗	✓	✓	✗	✓	✓	✓	✗	✗	✓
EZPWD	2	6k	2014	2018	GPLv3	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
FastECC	2	1k	2015	2017	Apache 2.0	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
FEC-AL	1	3k	2018	2018	BSD 3-Clause	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
FECpp	2	2k	2009	2019	–	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
GNURadio	205	270k	2006	2019	GPLv3	✓	✓	✗	✗	✗	✗	✓	✗	✓	✗
IT++	20	109k	2005	2018	GPLv3	✗	✓	✓	✗	✓	✓	✓	✗	✗	✗
LDPC-codes	1	5k	2012	2012	Copyright	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
LeGal	1	83k	2015	2019	–	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Leopard	4	5k	2017	2018	BSD 3-Clause	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
libcorrect	6	5k	2016	2018	BSD 3-Clause	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗
OpenAir	148	740k	2013	2019	OAI Public	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
OpenFEC	8	55k	2009	2014	CeCCIL-C	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗
Schifra	1	7k	2010	2018	GPLv3	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗
Siamese	1	11k	2018	2019	BSD 3-Clause	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Tavildar-Polar	1	2k	2016	2017	–	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Tavildar-LDPC	1	1k	2016	2016	–	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
the-art-of-ecc	1	–	2006	2009	Copyright	✗	✓	✓	✓	✓	✓	✓	✗	✗	✗
TurboFEC	3	4k	2015	2018	GPLv3	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗

The performance of this chain is measured by estimating the residual error rate at the sink. This rate is directly driven by the choice of the channel encoder and decoder. After Shannon, researchers have designed new coding/decoding schemes to approach Shannon's theoretical limit ever closer. Indeed, recent progresses managed to design practical codings performing very close to that limit, and already integrated in everyday communication systems.

On the eve of the 5G mobile communication generation, the challenge is now to design communication systems able to transmit huge amounts of data in a short time, at a small energy cost, in a wide variety of environments. Researchers work at refining existing coding schemes further, to get low residual error rates with fast, flexible, low complexity decoders.

The validation of a coding scheme requires estimating its error rate performance. Usually, no simple mathematical model exists to predict such performance. The only practical solution is to perform a Monte Carlo simulation of the whole chain: some data are randomly generated, encoded, modulated, noised, decoded, and the performance is then estimated by measuring the Bit Error Rate (BER) and the Frame Error Rate (FER) at the sink. This process leads to three main problems:

1. **Simulation time:** 100 erroneous frames must be simulated to accurately estimate the FER/BER. Thus, measuring a FER of 10^{-7} requires simulating the transmission of $\sim 100 \times 10^7 = 10^9$ frames. Assuming a frame of 1000 bits, the simulator must then emulate the transmission of 10^{12} bits. Keeping in mind that the decoding algorithm complexity may be significant, several weeks or months may be required to accurately estimate the FER/BER of a coding scheme.
2. **Algorithmic heterogeneity:** A large number of channel codes have been designed over time. For each kind of code, several decoding algorithms are available. While it is straightforward to describe a unique coding scheme, it is more challenging to have a unified software description that supports all the coding schemes and their associated algorithms. This difficulty comes from the heterogeneity of the data structure necessary to describe a channel code and the associated decoder: turbo codes use trellises, LDPC codes are well-defined on factor graphs and polar codes are efficiently decoded using binary trees.

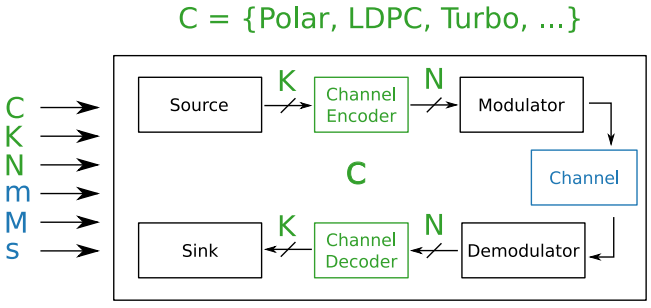


Fig. 2. Main simulation parameters.

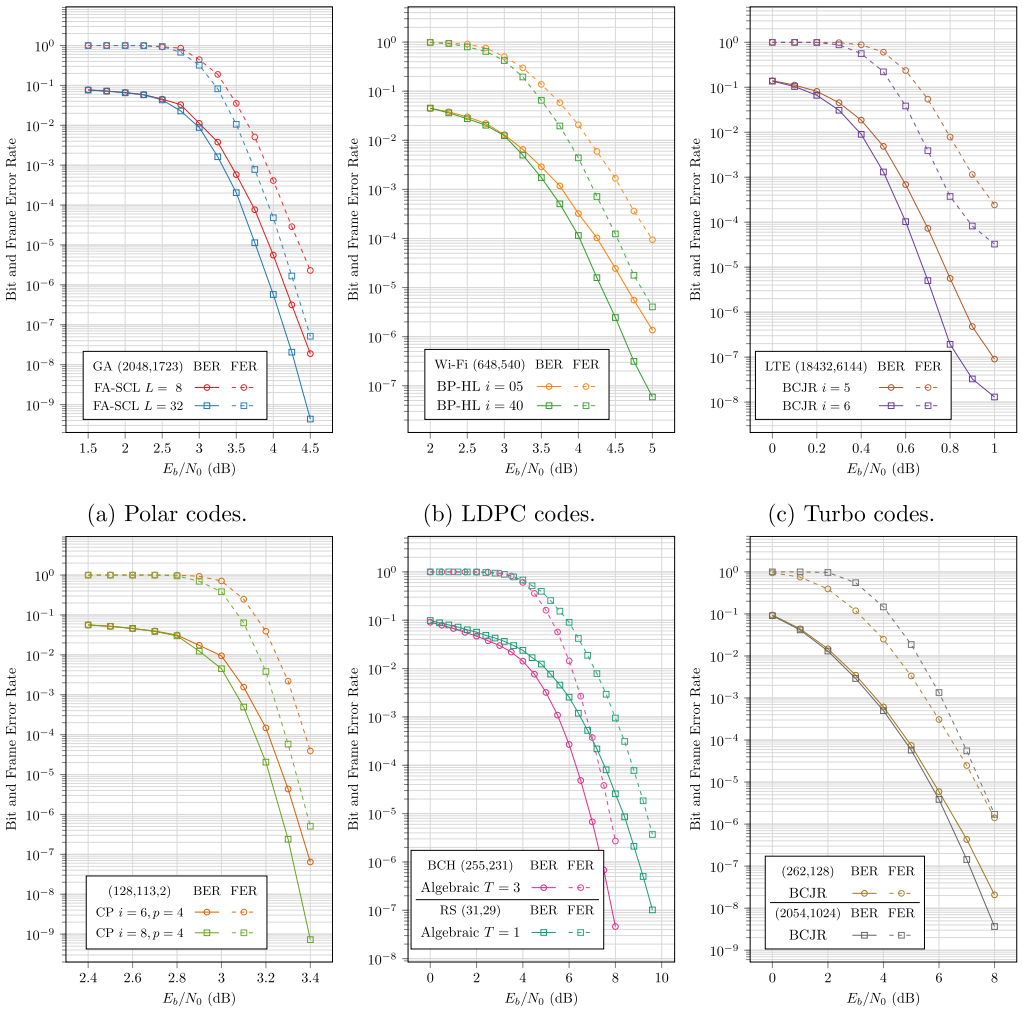
3. **Reproducibility:** It is usually tedious to reproduce results from the literature. This can be explained by the large amount of empirical parameters necessary to define one communication system, and the fact that not all of them are always reported in publications. Moreover, the simulator source codes are rarely publicly available. Consequently, a large amount of time is spent "reinventing the wheel" just to be able to compare to the state-of-the-art results.

The long simulation times make it desirable to have **high throughput implementations**. The algorithmic heterogeneity requires **flexible, modular software**. The reproducibility issue pushes towards a **portable** and **open-source software**. These are the purposes of AFF3CT.

2. Related works

In the digital communications community, many scientists implement their own simulation chain to validate their works. Table 1 presents, to the best of our knowledge, a list of currently available C/C++ open source channel coding simulators/libraries. Note that this table is frequently updated on the AFF3CT website[1]. We choose to compare with projects compiled as binaries, since they aim at high throughput and low latency, as AFF3CT. Many open source projects in Python or in MATLAB exist as well, but these tools are usually slower than compiled binaries, and rather aim at prototyping.

[1] C/C++ Open Source FEC Libraries: https://aff3ct.github.io/fec_libraries.html.



(a) Polar codes. (b) LDPC codes. (c) Turbo codes.

(d) Turbo product codes. (e) BCH & RS codes. (f) Convolutional codes.

Fig. 3. AFF3CT simulation of various code families.

Table 1 shows that, generally, the C/C++ FEC libraries target a single family or a small subset of channel codes. As a consequence, a large effort is spent to re-develop similar features, since all those libraries and tools share many characteristics (except the channel code itself). AFF3CT attempts to lower this redundancy by releasing a full simulator/library that consistently supports a wide range of channel codes to the community. AFF3CT also tries to homogenize usage (command line, C++ interfaces, etc.) for all code families. Table 1 does not aim at comparing channel code implementation performances. Instead, on the AFF3CT website, an overview of software channel decoders state-of-the-art is provided for turbo, LDPC and polar codes[2].

3. Software description

AFF3CT is a Forward Error Correction (FEC) toolbox. The most important tools it includes are the *simulator* and the *library*. Additionally, the toolbox comes with a bank of *simulated references*, a *GUI software* to browse those references, and a set of predefined *configuration files* for common communication standards.

3.1. Simulator

As a standalone simulator, AFF3CT proposes to simulate various communication chains with a broad range of codes. The simulator is a command line program; Fig. 2 depicts its main arguments: `-m` and `-M` set the minimum and the maximum Signal Noise Ratio (SNR, E_b/N_0) to simulate, `-s` specifies the iteration step inbetween. Parameter `-C` selects the channel code type, `-K` sets the length in bits of the initial information message and `-N` sets the codeword size, which is the encoder output size.[3] The command line interface is designed to be easily used from scripts. Fig. 3 shows simulations ran with AFF3CT on various code types (BPSK modulation and AWGN channel).

3.2. Library

As a FEC library, AFF3CT can be used programmatically, for instance in *Software Defined Radio* (SDR) contexts or in simulations. AFF3CT blocks can be used in external projects without restriction. Compute intensive blocks are optimized and vectorized to run fast on a single core. The library is thread-safe; however, it is not multi-threaded by itself, in contrast to the simulator. Instead, it is the responsibility of the user to manage multi-threading.

[2] FEC Software Decoders Hall of Fame: http://aff3ct.github.io/hof_turbo.html.

[3] The AFF3CT simulator comes with more options, omitted here for concision.

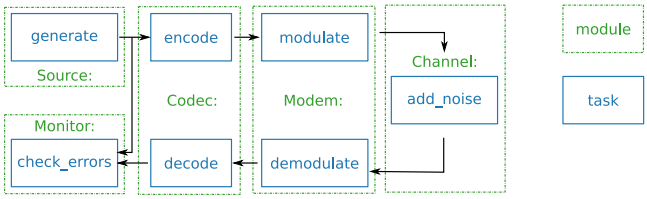


Fig. 4. Modules and tasks.

Table 2
List of supported channel codes (codecs).

Channel code	Standards	Decoders
LDPC	5G (data), WiFi, WiMAX, WRAN, 10 Gigabit Eth., DVB-S2, CCSDS etc.	Scheduling: Flooding and H./V. Layered Sum–Product Algorithm (SPA) Min-Sum (NMS, OMS) Approximate Min-Star (AMS) Bit Flipping: GallagerA/B/E, PPBF, WBF
Polar	5G (control channel)	Successive Cancellation (SC) Successive Cancellation List (SCL) CRC Aided SCL (CA-SCL, Adaptive SCL) Soft Cancellation (SCAN)
Turbo Parallel (single and double binary)	LTE (3G, 4G), DVB-RCS, CCSDS, etc.	Turbo BCJR Turbo BCJR + Early Termination (CRC) Post proc.: Flip aNd Check (FNC)
Turbo Product	WiMAX (opt.)	Turbo Chase-Pyndiah
BCH	CD, DVD, SSD, DVB-S2, Bitcoin, etc.	Berlekamp–Massey + Chien search
Reed–Solomon	CD, DVD, SSD, DVB-T, ADSL, etc.	Berlekamp–Massey + Chien search
Convolutional (single and double binary)	NASA	BCJR - Maximum A Posteriori (MAP) BCJR - Linear Approximation BCJR - Max Approximation

3.3. Software architecture

AFF3CT is developed in C++ in an object-oriented programming style. It provides the fundamental blocks involved in building communication chains (sources, modems, codecs, channels, ...). Those blocks are organized as *modules* and *tasks*.

Module: A set of related tasks sharing some characteristics. For instance, the *modem* module contains the *modulate* and *demodulate* tasks.

Task: An elementary processing performed on some data. For instance, *decode* or *modulate* are tasks. The tasks are characterized by their *sockets*. A socket of a task defines an entry point through which the task will consume and/or produce data. There are three kinds of sockets: *input*, *output* and *input/output*, following a philosophy close to *ports* in component-based development approaches.

As a rule, a task is always a *verb* and a module is always a *noun*. Modules are implemented as C++ classes, and tasks as class methods. AFF3CT defines several abstract modules, for sources, codecs, modems, channels, etc. It readily provides many implementations of those abstract classes; it is also straightforward to add new ones. Fig. 4 presents common modules and tasks typically found in a basic communication chain. It shows that the number of tasks per module can vary depending on the module type.

3.4. Software functionalities

The AFF3CT software functionalities can be decomposed in three main parts: the *codecs*, the *modems* and the *channels*. The codecs are the main part of the toolbox. There is a broad range of supported codes listed in Table 2.

Table 3
List of supported modulations/demodulations (modems).

Modem	Standard	Characteristics
N-PSK	IEEE 802.16 (WiMAX) UMTS (2G, 2G+) EDGE (8-PSK), ...	Phase-Shift Keying
N-QAM	IEEE 802.16 (WiMAX) UMTS (2G, 2G+) 3G, 4G, 5G, ...	Quadrature Amplitude Modulation
N-PAM	IEEE 802.16 (WiMAX) UMTS (2G, 2G+) 3G, 4G, 5G, ...	Pulse Amplitude Modulation
CPM	GMSK, Bluetooth IEEE 802.11 FHSS	Continuous Phase Modulation Coded (convolutional-based) modulation
OOK	IrDA (Infrared) ISM bands	On–Off Keying Used in optical communication systems
SCMA	Considered for 5G	Sparse Code Multiple Access Multi-user modulation
User defined	–	Constellation and order can be defined from an external file

Table 4
List of supported channels.

Channel	Multi-user	Characteristics
AWGN	Yes	Additive White Gaussian Noise
BEC	No	Binary Erasure Channel
BSC	No	Binary Symmetric Channel
Rayleigh	Yes	Flat Rayleigh fading channel
User defined	Not yet	User can import noise samples from an external file

The codecs naturally encompass the encoders and decoders, but they can also include puncturing patterns to shorten frames length according to some communication standards. Most of the codec algorithms come from the literature, while the others have been designed under AFF3CT [2–5]. In channel coding, the decoder is the most time-consuming process, compared to the puncturing and the encoding processes. This is why a specific effort is put on ensuring the high computing performance of the decoders. Most of the decoding algorithms have thus been optimized to satisfy high throughput and low latency constraints [6–9]. Those optimizations generally involve a vectorized implementation, a tailored data quantization and the use of fixed-point arithmetic.

In typical communication chains, it is necessary to adapt the digital signal to the physical support. This operation is performed by the modulator and conversely by the demodulator. AFF3CT comes with a rich set of modems to this purpose. Table 3 lists all the supported modems.

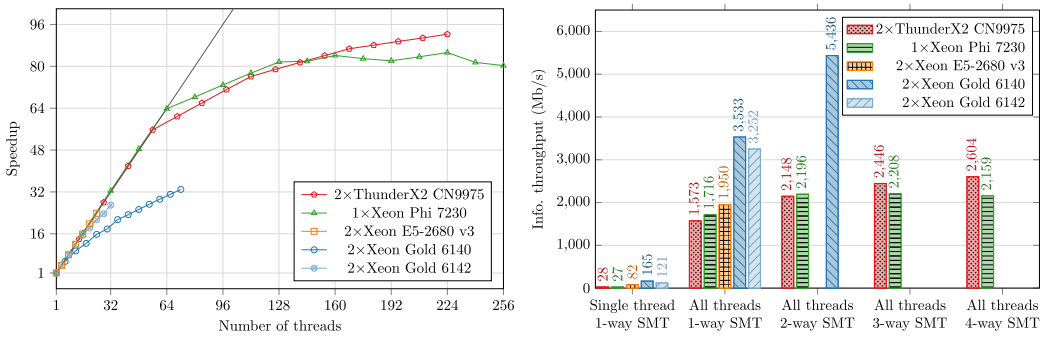
AFF3CT supports several coded modulation/demodulation schemes like the Continuous Phase Modulation (CPM) [10,11] and the Sparse Code Multiple Access (SCMA) modulation [12–14] (many codebooks are supported [15–21]). It allows to easily combine and evaluate the channel codes with several types of modulations. In the case of the CPM, analogical wave shapes are also simulated. The other modulation schemes are at the digital level.

For simulation purposes, it is crucial to emulate the behavior of the physical layer. This is the role of the channel. There are many possible configurations depending on the physics phenomena to simulate. Table 4 reports all the supported channels.

The channels involve complex floating-point computations. It is frequent to use exponential and trigonometric operations. Those types of operations cost a large amount of CPU cycles to be computed. As for the decoders, the channels have been carefully optimized based on branch instructions reduction and massive vectorization.

Table 5
Specifications of the target processors.

CPU		SIMD instr.		# Proc.	# Cores	Freq.	SMT	Turbo
		Name	Size		per Proc.	(GHz)		Boost
ARM	ThunderX2 CN9975	NEON	128-bit	2	28	2.00	4	✗
Intel	Xeon Phi 7230	AVX-512F	512-bit	1	64	1.30	4	✓
Intel	Xeon E5-2680 v3	AVX2	256-bit	2	12	2.50	1	✗
Intel	Xeon Gold 6140	AVX-512BW	512-bit	2	18	2.30	2	✓
Intel	Xeon Gold 6142	AVX-512BW	512-bit	2	16	2.60	1	✗



(a) Simulator speedups.

(b) Simulator throughputs.

Fig. 5. AFF3CT simulation of a (2048,1723) Polar code, FA-SCL decoder $L = 32$, BPSK modulation, AWGN channel, $E_b/N_0 = 4.5$ dB (BER = 4.34e−10, FER = 5.17e−08).

All these features are available from the simulator and in the library. Many additional functional functionalities available are skipped here for concision.

3.5. Software Continuous Integration

AFF3CT's development leverages streamlined Continuous Integration (CI) process. Each new commit to the version control repository triggers a comprehensive sequence of tests to catch potential regressions. Regression tests are based on past simulation results, validated from the state-of-the-art.

The CI process enables us to safely and confidently integrate contributed features and improvements from the community to AFF3CT. It also helps to keep the code review time by the core development team low-enough to swiftly integrate such contributions into the master branch.

4. Illustrative examples

4.1. Using AFF3CT as a simulator

Fig. 5a depicts the speedups achieved on various modern CPU architectures detailed in Table 5, while Fig. 5b exposes the corresponding simulation information throughputs. In Fig. 5a, the speedups on each architecture are computed with respect to the single thread simulation time on the same architecture. Each run assigns at most one AFF3CT thread to each hardware thread, thus, since the architectures have different number of hardware threads, the presented speedups do not all have the same number of measurement points. In Fig. 5, an $N = 2048$ and $K = 1723$ Polar code (FA-SCL decoder, $L = 32$, 32-bit GZip `0x04C11DB7` CRC) is simulated with a BPSK modulation and over an AWGN channel ($E_b/N_0 = 4.5$ dB, last SNR point of the blue curve in Fig. 3a). The frozen bits of the polar code have been generated with the Gaussian Approximation method (GA) [22]. The communication chain is fully vectorized with the `MIPP` wrapper [23] and multi-threaded with `C++11` threads. The vectorization is applied at the tasks level (c.f. Section 3.3) to take advantage of the algorithms intrinsic level of parallelism, when the multi-threaded parallelism is used, to reduce the simulation time by multiplicating the number of concurrent communication chains, thanks to the independence property of Monte Carlo simulations. In order to achieve highest possible throughputs, the receiver part of the simulator is configured to work with 8-bit fixed-point representation for real numbers. It has been shown that this representation does not degrade the decoding performances of the FA-SCL decoder [5]. However AFF3CT algorithms implementations can also be run on other representations like 64/32-bit floating-point and 16-bit fixed-point. For all the CPU targets, the code has been compiled with the `C++` GNU compiler version 8.2.0 on Linux, with the following optimization flags: `-O3 -funroll-loops -march = native`. Note that AFF3CT also works on Windows and macOS at the same level of performance. The simulation scales rather well on the tested architectures. The data remains in the CPU caches because of the moderate frame size ($N = 2048$). Scaling on the Xeon Gold 6142 is not as good as the other targets, because the *Intel Turbo Boost* technology is enabled on this platform: the CPU runs at higher frequencies when the number of active cores is low. AFF3CT effectively leverages the simultaneous multi-threading (SMT) technology. This is especially true for the ThunderX2 CN9975 and Xeon Gold 6140 targets. The SMT technology helps to improve the usage of the available instruction-level parallelism.

Table 6
AFF3CT multi-node speedups (single node: 2×Xeon E5-2680 v3).

Nodes	Cores	Info. T/P (Mb/s)	Speedup
1	24	1 950	1.00
2	48	3 901	1.95
4	96	7 793	4.00
8	192	15 829	8.12
16	384	31 640	16.22
32	768	63 075	32.34

Table 6 shows the multi-node scaling with the OpenMPI library (version 3.1.2). The information throughput (T/P) and the speedup are almost linear with the number of nodes: This is

Listing 1: Sockets binding.

```
// bind the sockets over the tasks
Enc[module::enc::sck::encode       ::U_K ].bind(Src[module::src::sck::generate    ::U_K ]);
Mdm[module::mdm::sck::modulate     ::X_N1].bind(Enc[module::enc::sck::encode      ::X_N ]);
Chn[module::chn::sck::add_noise    ::X_N ].bind(Mdm[module::mdm::sck::modulate    ::X_N2]);
Mdm[module::mdm::sck::demodulate   ::Y_N1].bind(Chn[module::chn::sck::add_noise   ::Y_N ]);
Dec[module::dec::sck::decode_siho  ::Y_N ].bind(Mdm[module::mdm::sck::demodulate  ::Y_N2]);
Mnt[module::mnt::sck::check_errors::U    ].bind(Src[module::src::sck::generate    ::U_K ]);
Mnt[module::mnt::sck::check_errors::V    ].bind(Dec[module::dec::sck::decode_siho::V_K ]);
```

expected because there are very few communications between the various MPI processes. Note that the super-linear scaling is due to the measurements imprecision.

Those aforementioned results demonstrate the high throughput capabilities of AFF3CT. For instance, when using 32 MPI nodes on the given (2048,1723) polar code, it takes about one minute to estimate the $E_b/N_0 = 4.5$ dB SNR point (BER = 4.34e−10, FER = 5.17e−08).

4.2. Using AFF3CT as a library

Listing 2: Modules allocation.

```
#include <aff3ct.hpp>
using namespace aff3ct;

// allocate the module objects
module::Source_random<>           Src(K   );
module::Encoder_repetition_sys<> Enc(K, N);
module::Modem_BPSK<>              Mdm(N   );
module::Channel_AWGN_LLR<>        Chn(N   );
module::Decoder_repetiton_std<>  Dec(K, N);
module::Monitor_BFER<>            Mnt(K, E);
```

In this section the communication chain proposed in Fig. 4 is implemented with the AFF3CT library. The first step is to allocate the modules. In Listing 2 we chose to allocate modules on the stack, but it is also possible to do the same on the heap. K is the number of information bits, N is the frame size and E is the number of erroneous frames to simulate. One can notice that there is a module for the encoder and for the decoder, this differs from Fig. 4 where *encode* and *decode* are tasks of the codec module. In fact the codec module exists in AFF3CT and it is an aggregation of the encoder and decoder modules. For simplicity, we chose not to use the codec module here. In this basic example, a repetition code is selected, it simply repeats the information bits N/K times.

The next step is to bind the sockets of successive tasks together (see Listing 1): The *source* module output socket `module::src::sck::generate::U_K` is connected to the input socket `module::enc::sck::encode::U_K` of the *encoder*, and so on, for all the sockets of the tasks.

Listing 3: Tasks execution.

```
// the simulation loop
while (!monitor.fe_limit_achieved()) {
  Src[module::src::tsk::generate    ].exec();
  Enc[module::enc::tsk::encode      ].exec();
  Mdm[module::mdm::tsk::modulate    ].exec();
  Chn[module::chn::tsk::add_noise   ].exec();
  Mdm[module::mdm::tsk::demodulate  ].exec();
  Dec[module::dec::tsk::decode_siho ].exec();
  Mnt[module::mnt::tsk::check_errors].exec();
}
```

The simulation is then started and each task is executed. In Listing 3, the whole communication chain is executed multiple times, until the E frame error limit is achieved (typically $E = 100$ erroneous frames).

To propose an easy to use interface, sockets and tasks can be selected through the `[]` operator, which takes a C++ strongly typed enumerate. This way it is possible to specialize the code depending on whether it is a socket or a task. Strongly typed enumerates are checked at compile time (contrary to standard enumerates), making it impossible to use wrong values.

5. Impact

AFF3CT is currently used in several industrial contexts for simulation purposes (Turbo concept, Airbus, Thales, Huawei) and for specific developments (CNES, Schlumberger, Airbus, Thales, Orange), as well as in academic projects (NAND French National Agency project, IdEx CPU). The MIT license used in the project is very permissive and gives confidence to industrial and academic partners, who can then invest themselves and reuse parts of AFF3CT in their own projects without any restrictions.

An important aspect of channel coding is the ability to reproduce state-of-the-art results, because there are many possible configurations and it is time-consuming to rediscover those configurations. This is why AFF3CT comes with a large database of pre-simulated performance curves with all the required parameters. Some research projects have been using AFF3CT as a Ref. [24–30]. All pre-computed simulation results are available at a glance on the online comparator,[4] with corresponding command lines to reproduce them. Combined with the possibility to download AFF3CT last builds,[5] testing the reference configurations, replicating the experiments and playing with parameters is straightforward. **AFF3CT aims to achieve results easily reproducible by the scientific community.**

It comes with a comprehensive documentation, to help using, modifying, extending existing coding schemes, to potentially improve them or to adapt to other domains. Moreover, AFF3CT can be used to prototype and evaluate hardware implementations [31].

6. Conclusion and future works

In this paper we presented AFF3CT, a forward error correction toolbox that enables high throughput simulations thanks to multi-node, multi-threaded and vectorization paradigms. AFF3CT makes reproducible and replicable science possible with a large database of reference simulations available, and tools to reproduce them quickly on commonly used systems. Both the AFF3CT library and standalone simulator ship with a wide range of heterogeneous algorithms, and can easily be enriched by the community, for instance with additional families of codes, or to fit new application contexts such as software defined radio.

In the near future, a wrapper is scheduled to directly use the AFF3CT library from MATLAB and Python. It will give the opportunity to non-experts in C++ community to easily take

[4] BER/FER comparator: http://aff3ct.github.io/comparator.html.

[5] AFF3CT last builds: http://aff3ct.github.io/download.html.

advantage of the high speed implementations available in the toolbox. On the other hand, we plan to extend the range of the project with synchronization modules: this will enable full SDR emitter and receiver using the AFF3CT library.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This study has been carried out with financial support from French State, managed by the French National Research Agency (ANR) in Programme IdEx Bordeaux CPU (ANR-10-IDEX-03-02) and in Project NAND, France (ANR-15-CE25-0006-01). Experiments were conducted on the PlaFRIM platform.

References

[1] Shannon CE. A mathematical theory of communication. Bell Syst Tech J 1948;27(4):623–56. http://dx.doi.org/10.1002/j.1538-7305.1948.tb00917.x.
[2] Tonnellier T, Leroux C, Le Gal B, Jégo C, Gadat B, Wambeke NV. Lowering the error floor of double-binary turbo codes: The flip and check algorithm. In: International symposium on turbo codes and iterative information processing (ISTC). IEEE; 2016, p. 156–60. http://dx.doi.org/10.1109/ISTC.2016.7593096.
[3] Tonnellier T, Leroux C, Le Gal B, Gadat B, Jégo C, Wambeke NV. Lowering the error floor of turbo codes with CRC verification. IEEE Wirel Commun Lett (WCL) 2016;5(4):404–7. http://dx.doi.org/10.1109/LWC.2016.2571283.
[4] Tonnellier T. Contribution to the improvement of the decoding performance of turbo codes : Algorithms and architecture (Ph.D. thesis), Université de Bordeaux; 2017, URL https://tel.archives-ouvertes.fr/tel-01580476.
[5] Léonardon M, Cassagne A, Leroux C, Jégo C, Hamelin L-P, Savaria Y. Fast and flexible software polar list decoders. Springer J Signal Process Syst (JSPS) 2019;92:1–16. http://dx.doi.org/10.1007/s11265-018-1430-3.
[6] Le Gal B, Leroux C, Jégo C. Multi-gb/s software decoding of polar codes. IEEE Trans Signal Process (TSP) 2015;63(2):349–59. http://dx.doi.org/10.1109/TSP.2014.2371781.
[7] Cassagne A, Le Gal B, Leroux C, Aumage O, Barthou D. An efficient, portable and generic library for successive cancellation decoding of polar codes. In: International workshop on languages and compilers for parallel computing (LCPC). Springer; 2015, http://dx.doi.org/10.1007/978-3-319-29778-1.
[8] Cassagne A, Tonnellier T, Leroux C, Le Gal B, Aumage O, Barthou D. Beyond gbps turbo decoder on multi-core CPUs. In: International symposium on turbo codes and iterative information processing (ISTC). IEEE; 2016, p. 136–40. http://dx.doi.org/10.1109/ISTC.2016.7593092.
[9] Cassagne A, Aumage O, Leroux C, Barthou D, Le Gal B. Energy consumption analysis of software polar decoders on low power processors. In: European signal processing conference (EUSIPCO). IEEE; 2016, p. 642–6. http://dx.doi.org/10.1109/EUSIPCO.2016.7760327.
[10] Aulin T, Sundberg C. Continuous phase modulation - part i: Full response signaling. IEEE Trans Commun (TCOM) 1981;29(3):196–209. http://dx.doi.org/10.1109/TCOM.1981.1095001.
[11] Aulin T, Rydbeck N, Sundberg C. Continuous phase modulation - part ii: Partial response signaling. IEEE Trans Commun (TCOM) 1981;29(3):210–25. http://dx.doi.org/10.1109/TCOM.1981.1094985.
[12] Nikopour H, Baligh H. Sparse code multiple access. In: International symposium on personal, indoor, and mobile radio communications (PIMRC). IEEE; 2013, p. 332–6. http://dx.doi.org/10.1109/PIMRC.2013.6666156.
[13] Ghaffari A, Léonardon M, Savaria Y, Jégo C, Leroux C. Improving performance of SCMA MPA decoders using estimation of conditional probabilities. In: International conference on new circuits and systems (NEWCAS). IEEE; 2017, p. 21–4. http://dx.doi.org/10.1109/NEWCAS.2017.8010095.
[14] Ghaffari A, Léonardon M, Cassagne A, Leroux C, Savaria Y. Toward high performance implementation of 5G SCMA algorithms. IEEE Access 2019;7:10402–14. http://dx.doi.org/10.1109/ACCESS.2019.2891597.
[15] Program AU. The 1st 5G algorithm innovation competition-SCMA. URL http://www.innovateasia.com/5g/images/pdf/1st5GAlgorithmInnovationCompetition-ENV1.0-SCMA.pdf.
[16] Wu Y, Zhang S, Chen Y. Iterative multiuser receiver in sparse code multiple access systems. In: IEEE international conference on communications (ICC). IEEE; 2015, p. 2918–23. http://dx.doi.org/10.1109/ICC.2015.7248770.
[17] Cheng M, Wu Y, Chen Y. CaPacity analysis for non-orthogonal overloading transmissions under constellation constraints. In: International conference on wireless communications signal processing (WCSP). IEEE; 2015, p. 1–5. http://dx.doi.org/10.1109/WCSP.2015.7341294.
[18] Zhang S, Xiao K, Xiao B, Chen Z, Xia B, Chen D, Ma S. A capacity-based codebook design method for sparse code multiple access systems. In: International conference on wireless communications signal processing (WCSP). IEEE; 2016, p. 1–5. http://dx.doi.org/10.1109/WCSP.2016.7752620.
[19] Klimentyev VP, Sergienko AB. Detection of scma signal with channel estimation error. In: Conference of open innovations association and seminar on information security and protection of information technology (FRUCT-ISPIT). IEEE; 2016, p. 106–12. http://dx.doi.org/10.1109/FRUCT-ISPIT.2016.7561515.
[20] Song G, Wang X, Cheng J. Signature design of sparsely spread code division multiple access based on superposed constellation distance analysis. IEEE Access 2017;5:23809–21. http://dx.doi.org/10.1109/ACCESS.2017.2765346.
[21] Klimentyev VP, Sergienko AB. Scma codebooks optimization based on genetic algorithm. In: European wireless conference. IEEE; 2017, p. 1–6, URL https://ieeexplore.ieee.org/document/8011314.
[22] Trifonov P. Efficient design and decoding of polar codes. IEEE Trans Commun (TCOM) 2012;60(11):3221–7. http://dx.doi.org/10.1109/TCOMM.2012.081512.110872.
[23] Cassagne A, Aumage O, Barthou D, Leroux C, Jégo C. MIPP: a portable c++ SIMD wrapper and its use for error correction coding in 5g standard. In: Workshop on programming models for SIMD/Vector processing (WPMVP). Vösendorf/Wien, Austria: ACM; 2018, http://dx.doi.org/10.1145/3178433.3178435.
[24] Léonardon M, Leroux C, Binet D, Langlois JP, Jégo C, Savaria Y. Custom low power processor for polar decoding. In: International symposium on circuits and systems (ISCAS). IEEE; 2018, p. 1–5. http://dx.doi.org/10.1109/ISCAS.2018.8351739.
[25] Léonardon M, Leroux C, Jääskeläinen P, Jégo C, Savaria Y. Transport triggered polar decoders. In: International symposium on turbo codes and iterative information processing (ISTC). IEEE; 2018, http://dx.doi.org/10.1109/ISTC.2018.8625310.
[26] Florian F. Physim - a physical layer simulation software. In: International conference on consumer electronics (ICCE). IEEE; 2018, p. 1–6. http://dx.doi.org/10.1109/ICCE-Berlin.2018.8576187.
[27] Pignoly V, Le Gal B, Jégo C, Gadat B. High data rate and flexible hardware qc-ldpc decoder for satellite optical communications. In: International symposium on turbo codes and iterative information processing (ISTC). IEEE; 2018, p. 1–5. http://dx.doi.org/10.1109/ISTC.2018.8625274.
[28] Ghanaatian R, Balatsoukas-Stimming A, Müller TC, Meidlinger M, Matz G, Teman A, Burg A. A 588-gb/s LDPC decoder based on finite-alphabet message passing. IEEE Trans Very Large Scale Integr (VLSI) Syst 2018;26(2):329–40. http://dx.doi.org/10.1109/TVLSI.2017.2766925.
[29] Cavatassi A, Tonnellier T, Gross WJ. Asymmetric construction of low-latency and length-flexible polar codes. In: International conference on communications (ICC). IEEE; 2019, p. 1–6. http://dx.doi.org/10.1109/ICC.2019.8761129.
[30] Cavatassi A, Tonnellier T, Gross WJ. Fast decoding of multi-kernel polar codes. In: Wireless communications and networking conference (WCNC). IEEE; 2019, (in press) arXiv:1902.01922.
[31] Cassagne A, Hartmann O, Léonardon M, Tonnellier T, Delbergue G, Leroux C, Tajan R, Le Gal B, Jégo C, Aumage O, Barthou D. Fast simulation and prototyping with aff3ct. In: International workshop on signal processing systems (SiPS). IEEE; 2017, URL https://hal.archives-ouvertes.fr/hal-01965633.