Implementation and parallelization of batch fully homomorphic encoding with compressed public key

Jess Woods[1], Ada Sedova[2], and Oscar Hernandez[1]

[1] *Computer Science Research Group, Oak Ridge National Laboratory*
[2] *Biophysics Group, Oak Ridge National Laboratory*

July 24, 2020

Abstract

Fully homomorphic encoding allows the evaluation of arbitrary functions on private data by untrusted parties. From safe health care transactions to secure, outsourced machine learning, the real-world applications of efficient and adaptable data encoding are significant and plentiful. However, real-world implementations remain impractical due to time-consuming operations and large integers used for security. To overcome this limitation, we create the first parallel implementation of Dijk *et al.*'s fully homomorphic encoding scheme over the integers with two theoretical optimizations, public key compression and batching, as well as parallelism. We test our implementation with several different levels of security, and describe its extension to higher-level operations.

1 INTRODUCTION

An encoding scheme is *homomorphic* if it supports operations on data done exclusively by manipulating encoded data (Figure 3). A *fully homomorphic* scheme, first described by Gentry in 2009 [1] and later implemented in [2], supports an arbitrary number of both addition and multiplication operations. Gentry's fully homomorphic scheme is achieved by first constructing a *somewhat homomorphic* scheme, which supports only a limited number of multiplications. Each encoding contains a certain amount of "noise", which grows after each multiplication. Eventually, this noise prevents correct decoding. To overcome this, the decode circuit is "squashed" to a limited number of steps, so it can be homomorphically evaluated without producing too much noise. This is called "bootstrapping"; instead of evaluating the new decode circuit with the private key, the decode circuit is evaluated with an *encoding* of the private key. This recode operation produces a different encoding for the same underlying data, but with (possibly) reduced noise. This is shown in Figure 1.

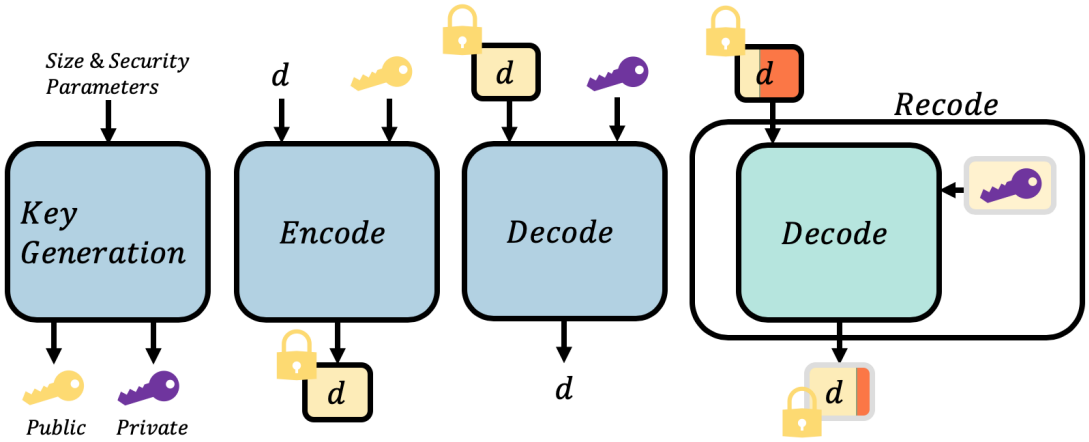

Figure 1: Illustration of operations for fully homomorphic encoding scheme, d representing encoded data, in our case, the vector $m_0, m_1, ..., m_{\ell-1}$

.

Fully homomorphic encoding allows a user to outsource computations to an untrusted party, like a remote server or supercomputer, without compromising privacy (Figure 2). Today, the main barriers to feasible, real-world implementation of fully homomorphic schemes are computational memory and computational speed. Implementations remain slow due to large integers used for security and high-latency encode and recode operations.

Contributions We implement the first parallel version of Dijk, Gentry, Halevi, and Vaikuntanathans fully homomorphic integer scheme (DGHV) [3] to overcome these obstacles. We incorporate two theoretical improvements, public key compression [4] and a batching technique, first constructed in [5], and proven suitable and secure for DGHV in [6]. We explain our implementations, parallelization, and test cases, and extend our construction to higher-level operations in a library.

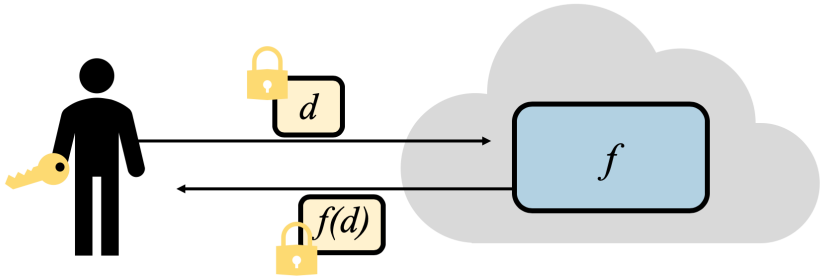

Figure 2: The desired model; The cloud can perform the computation f for the user without having access (the key) to the user's data d.

Notations For a real number x, we denote the nearest integer of x as $\lfloor x \rceil$. For integers z, p, we denote the reduction of z modulo p by $[z]_p \in (-p/2, p/2]$ and by $(z \bmod p) \in [0, p)$. A vector, $\boldsymbol{v}$, is written in bold, with its members denoted $v_0, v_1, ...v_i, ...$

2 BACKGROUND

A The DGHV scheme over the integers

In 2010, Dijk *et al.* constructed a fully homomorphic scheme [3] using Gentry's method. The DGHV scheme is over the integers and endeavors to be conceptually simpler to understand and implement than previous fully homomorphic schemes. Essentially, the large, odd integer p is chosen as the private key, and encodings are of the form $c = pq + 2r + m$, with random integers r and q, and data bit m. We review the semantically secure somewhat homomorphic version (without recoding) briefly, to give the reader a basic understanding:

KeyGeneration():
Generate an η-bit random prime inteter p.
Let $D_{\gamma,\rho}(p) = \{\text{Choose } q \leftarrow_R \mathbb{Z} \cap [0, 2^\gamma/p), r \leftarrow_R \mathbb{Z} \cap (-2^\rho, 2^\rho) : \text{Output } x \leftarrow q \cdot p + r\}$
$x_i \leftarrow D_{\gamma,\rho}(p)$ for $i \in [0, \tau]$
Assure x_0 is the largest of the x_i's, x_0 is odd, and $[x_0]_p$ is even.
Output $\boldsymbol{pk} \leftarrow (x_0, x_1, ..., x_\tau)$
Output $\boldsymbol{sk} \leftarrow p$

Encode($\boldsymbol{pk}, m$): Choose a random subset $S \subsetneq \{1, 2, ..., \tau\}$ and a random integer $r \in (-2^{\rho'}, 2^{\rho'})$. Output:

$$c \leftarrow \left[m + 2r + 2 \sum_{i \in S} x_i \right]_{x_0}$$

Decode($\boldsymbol{sk}, c$): Output $m \leftarrow [c]_p \bmod 2$.

B Homomorphic operations

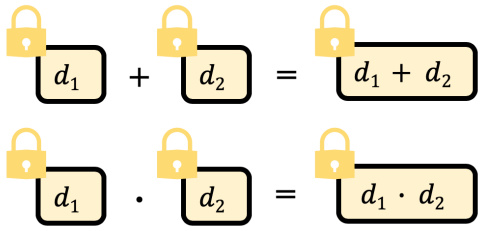

Figure 3: Example of desired homomorphic operations; d represents our data

It is easy to see the (limited) homomorphic capabilities of the scheme. Given two encodings of the form $c = q \cdot p + 2r + m$ and $c' = q' \cdot p + 2r' + m'$, the sum $c + c'$ is an encoding of $m + m'$ mod 2 under a $(\rho' + 1)$-bit noise and the product $c \cdot c'$ is an encoding of $m \cdot m'$ with noise bit-length $2\rho'$. The noise must remain below p to maintain correct decoding, so the scheme allows η/ρ' homomorphic multiplications.

C Compression of the public key

The public key consists of several lists of integers with large random terms embedded in them. This provides security against lattice attacks, but comes with the price of having to send and store large keys. Compressing the public key reduces the size of the DGHV scheme by several orders of magnitude. To compress public key integers, we use a pseudo-random number generator f with a public random seed se to generate a set of integers, $\boldsymbol{\chi}$, of the same size (γ bits) as $\boldsymbol{x}$. We compute small corrections, $\boldsymbol{\delta}$, to $\boldsymbol{\chi}$ such that $x_i = \chi_i - \delta_i$. Only the small $\boldsymbol{\delta}$ needs to be stored in the public key. Knowing $\boldsymbol{\delta}$, f, and se is enough to recover the $\boldsymbol{x}$.

Our public key elements $\boldsymbol{\chi}, \boldsymbol{\chi'}, \boldsymbol{\chi^{\Pi}}$ and the encoding of the private key bits $\boldsymbol{\sigma}$ are all compressed, using the deltas $\boldsymbol{\Delta}, \boldsymbol{\Delta'}, \boldsymbol{\Delta^{\Pi}}$, and $\boldsymbol{\Delta^{\sigma}}$, respectively. This method allows an encoding to be compressed from $\gamma = \tilde{\mathcal{O}}(\lambda^5)$ bits down to $\ell \cdot \eta + \lambda = \ell \cdot \tilde{\mathcal{O}}(\lambda^2)$ bits.

D Batched encodings

"Batching" denotes the packing of multiple bits, $m_0, ..., m_\ell$, into one encoding, rather than a single bit. This is made possible by "assigning" one of ℓ co-prime integers $p_0, ...p_\ell$ to each m_i and using the Chinese Remainder Theorem. Encoding changes from this form:

$$c \leftarrow q \cdot p + 2r + m$$

To this form:

$$c \leftarrow q \cdot \prod_{i=0}^{\ell-1} p_i + \text{CRT}_{p_0,...p_{l-1}}(2r_0 + m_0, ..., 2r_{\ell-1} + m_{\ell-1})$$

Note that there are some extra security terms, excluded for clarity here, detailed in Section 3. Decoding is now over each bit:

$$m_i \leftarrow [c]_{p_i} \bmod 2$$

Each bit of c modulo each p_i then behaves as in the original, non-batched scheme. Homomorphic addition and multiplication and the recode operation work in parallel and componentwise over the m_i's.

Batching reduces the asymptotic overhead per gate, while maintaining essentially the same complexity cost and same security. This construction also allows for the possibility of moving bits between "slots" with a permutation operation [6], allowing for a complete set of operations on encoded sets. This is a consideration for future work.

E Bootstrapping

To achieve bootstrapping, which will allow homomrphic evaluation for circuits of any depth, the decode circuit must first be "squashed". The private key is represented as a matrix $\boldsymbol{s}$ of 0s and 1s. Together with the "hint" $\boldsymbol{u}$ in the public key, this forms the original private key, $\boldsymbol{p}$. Each $\boldsymbol{s_i}$ has θ boxes of $B = \Theta/\theta$ bits each, with a single 1 per box. This enables a grade-school addition algorithm, requiring $\mathcal{O}(\theta^2)$ multiplications instead of $\mathcal{O}(\Theta \cdot \theta)$. Using $\boldsymbol{u}$, the encoding c is expanded to the vector $\boldsymbol{z}$. Details are provided in Section 3.

Decoding changes as follows:

$$m_i \leftarrow [c]_{p_i} \bmod 2$$

$$m_i \leftarrow (c - p_i \cdot \lfloor c/p_i \rceil) \bmod 2$$

$$m_j \leftarrow (c \bmod 2) - \left(\left\lfloor \sum_{i=0}^{\Theta-1} s_{j,i} \cdot z_i \right\rceil \bmod 2 \right)$$

$$m_j \leftarrow \left\lfloor \sum_{i=0}^{\Theta-1} s_{j,i} \cdot z_i \right\rceil \bmod 2 \oplus (c \bmod 2)$$

See [3] for a proof of correctness and proof that the decode circuit is now "shallow" enough, omitted here due to space constraints.

Recoding is achieved by providing an encoding of every set of private key bits $s_{0,i}, ..., s_{\ell-1,i}$ as σ_i. The decode circuit is homomorphiclly evaluated as shown in Figure 4. Each z_i is treated as a binary vector of $n + 1$ bits (since z_i is calculated to n bits of precision). This works well since 0 and 1 are valid (and noise-free!) encodings of themselves.

3 COMPLETE FULLY HOMOMORPHIC SCHEME

Here we provide a complete description of our DGHV variant with the compressed public key and batching improvements.

KeyGeneration():

Generate ℓ primes $\boldsymbol{p} = (p_0, ..., p_{\ell-1})$ of η bits each and denote their product π.
$x_0 \leftarrow q_0 \cdot \pi$, where $q_0 \leftarrow_R \mathbb{Z} \cap [0, 2^\gamma/\pi)$ and q_0 is a 2^{λ^2}-rough integer.[1] Note x_0 is error-free.

Generate at random ℓ vectors $\boldsymbol{s}_j$ of length Θ split into θ boxes of size Θ/θ, such that the Hamming Weight of each box is 1 and $s_{j,j} = 1$ for $j \in [0, \ell - 1]$

Initialize pseudo-random generator f_1 with a random seed se_1
Use $f_1(se_1)$ to generate sets of integers:
$\chi_i \leftarrow_R [0, x_0)$ for $i \in [1, \tau]$
$\chi'_i \leftarrow_R [0, x_0)$ for $i \in [1, \ell - 1]$
$\chi_i^\Pi \leftarrow_R [0, x_0)$ for $i \in [1, \ell - 1]$

Initialize pseudo-random generator f_2 with a random seed se_2
Use $f_2(se_2)$ to generate a set of integers: $u_i \leftarrow_R [0, 2^{\kappa+1})$ for $i \in \mathbb{Z} \cap [0, \Theta - 1]$

Then set $\boldsymbol{u} = (u_0, ..., u_{\ell-1})$ such that

$$x_{p_j} \leftarrow \sum_{i=0}^{\Theta-1} s_{j,i} \cdot u_i \bmod 2^{\kappa+1}$$

where $x_{p_j} = \lfloor 2^\kappa / p_j \rceil$ for $j \in [0, \ell - 1]$

Initialize pseudo-random generator f_3 with a random seed se_3
Use $f_3(se_3)$ to generate a set of integers: $\chi_i^\sigma \leftarrow_R [0, x_0)$ for $i \in [0, \Theta - 1]$

Define γ-bit integers as follows:

1. $x_i \leftarrow \chi_i - \Delta_i$ for $i \in [1, \tau]$ with[2]

$$\Delta_i \leftarrow [\chi_i]_\pi + \xi_i \cdot \pi - \mathrm{CRT}_{p_0,...,p_{\ell-1}}(2r_{i,0}, ..., 2r_{i,\ell-1})$$

where $r_{i,j} \leftarrow_R \mathbb{Z} \cap (-2^{\rho'-1}, 2^{\rho'-1})$ and $\xi_i \leftarrow_R \mathbb{Z} \cap [0, 2^{\lambda + log_2(\ell) + \ell \cdot \eta}/\pi)$

2. $x'_i \leftarrow \chi'_i - \Delta'_i$ for $i \in [1, \ell - 1]$ with[3]

$$\Delta'_i \leftarrow [\chi'_i]_\pi + \xi'_i \cdot \pi - \mathrm{CRT}_{p_0,...,p_{\ell-1}}(2r'_{i,0} + \delta_{i,0}, ..., 2r'_{i,\ell-1} + \delta_{i,\ell-1})$$

where $r'_{i,j} \leftarrow_R \mathbb{Z} \cap (-2^\rho, 2^\rho)$ and $\xi'_i \leftarrow_R \mathbb{Z} \cap [0, 2^{\lambda + log_2(\ell) + \ell \cdot \eta}/\pi)$

3. $\Pi_i \leftarrow \chi_i^\Pi - \Delta_i^\Pi$ for $i \in [1, \ell - 1]$ with

$$\Delta_i^\Pi \leftarrow [\chi'_i]_\pi + \xi_i^\Pi \cdot \pi - \mathrm{CRT}_{p_0,...,p_{\ell-1}}(2r_{i,0}^\Pi + \delta_{i,0} 2^{\rho'+1}, ..., 2r_{i,\ell-1}^\Pi + \delta_{i,\ell-1} 2^{\rho'+1})$$

where $r_{i,j}^\Pi \leftarrow_R \mathbb{Z} \cap (-2^\rho, 2^\rho)$ and $\xi_i^\Pi \leftarrow_R \mathbb{Z} \cap [0, 2^{\lambda + log_2(\ell) + \ell \cdot \eta}/\pi)$

[1] An integer is said to be a-rough when it does not contain prime factors smaller than a. If $a > 2$ the integer must be odd. $q0$ can be generated as a product of 2^{λ^2}-bit primes

[2] CRT denotes Chinese Remainder Theorem.

[3] $\delta_{i,j}$ denotes the Kronecker delta: $\delta_{i,j} = 1$ if $i = j$ and 0 otherwise.

4. $\sigma_i \leftarrow \chi_i^\sigma - \Delta_i^\sigma$ for $i \in [0, \Theta - 1]$ with

$$\Delta_i^\sigma \leftarrow [\chi_i^\sigma]_\pi + \xi_i^\sigma \cdot \pi - \mathrm{CRT}_{p_0,...,p_{\ell-1}}(2r_{i,0}^\sigma + s_{0,i}, ..., 2r_{i,\ell-1}^\sigma + s_{\ell-1,i})$$

where $r_{i,j}^\sigma \leftarrow_R \mathbb{Z} \cap (-2^\rho, 2^\rho)$ and $\xi_i^\sigma \leftarrow_R \mathbb{Z} \cap [0, 2^{\lambda + log_2(\ell) + \ell \cdot \eta}/\pi)$
Note that each σ_i is just an encoding of a column of $\boldsymbol{s}$. This is what enables **Recode** to work in parallel.

Output the private key $\boldsymbol{sk} \leftarrow \boldsymbol{s}$, a two dimensional vector of 0s and 1s.
Output the public key:

$$\boldsymbol{pk} \leftarrow \left\langle x_0, se_1, \boldsymbol{\Delta}, \boldsymbol{\Delta'}, \boldsymbol{\Delta^\Pi}, se_2, \boldsymbol{u}, se_3, \boldsymbol{\Delta^\sigma} \right\rangle$$

Encode($\boldsymbol{pk}, m \in \{0,1\}^\ell$):
Use $f_1(se_1)$ to recover integer vectors $\boldsymbol{\chi}, \boldsymbol{\chi'}, \boldsymbol{\chi^\Pi}$
$\boldsymbol{x} \leftarrow \boldsymbol{\chi} - \boldsymbol{\Delta}$
$\boldsymbol{x'} \leftarrow \boldsymbol{\chi'} - \boldsymbol{\Delta'}$
$\boldsymbol{\Pi} \leftarrow \boldsymbol{\chi^\Pi} - \boldsymbol{\Delta^\Pi}$
Generate a random vector $b_i \leftarrow_R (-2^\alpha, 2^\alpha)$ for $i \in [1, \tau]$
Generate a random vector $b'_i \leftarrow_R (-2^{\alpha'}, 2^{\alpha'})$ for $i \in [0, \ell - 1]$
Output the encoding:

$$c \leftarrow \left[\sum_{i=0}^{\ell-1} m_i \cdot x'_i + \sum_{i=1}^{\tau} b_i \cdot x_i + \sum_{i=0}^{\ell-1} b'_i \cdot \Pi_i \right]_{x_0}$$

Add($\boldsymbol{pk}, c_1, c_2$): Output $c_1 + c_2 \bmod x_0$

Multiply($\boldsymbol{pk}, c_1, c_2$): Output $c_1 \cdot c_2 \bmod x_0$

Expand($\boldsymbol{pk}, c$):
Use $f_1(se_1)$ to recover $\boldsymbol{u}$
Compute $z_i \leftarrow (\lfloor c \cdot (u_i/2^\kappa) \rceil \bmod 2)$ for $i \in [0, \Theta - 1]$ to n bits of precision after the binary point
Output $\boldsymbol{z}$

Decode($\boldsymbol{sk}, c, \boldsymbol{z}$): Output $\boldsymbol{m} = (m_0, ..., m_{\ell-1})$ with

$$m_j \leftarrow \left\lfloor \sum_{i=0}^{\Theta-1} s_{j,i} \cdot z_i \right\rceil \bmod 2 \oplus (c \bmod 2)$$

Recode($\boldsymbol{pk}, c, \boldsymbol{z}$): Apply decode circuit to expanded encoding $\boldsymbol{z}$, and encoded private key bits σ_i (in place of $s_{i,j}$). Note that each bit of each z_i is treated individually, as shown in Figure 4. There are a finite number of bits for each z_i, since each is only calculated to n bits of precision after the binary point. Output new encoding.

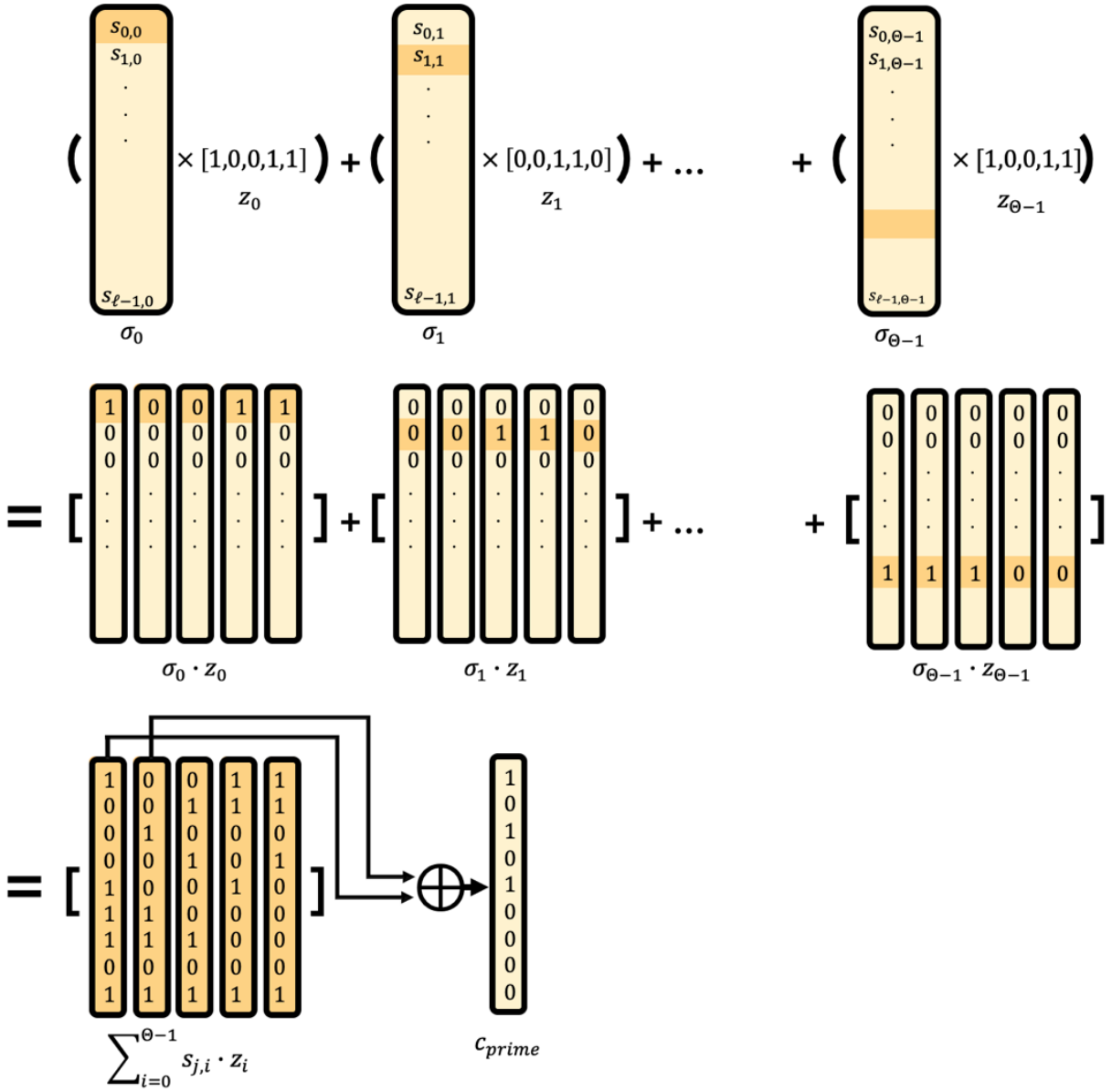

Figure 4: Evaluation of $\boldsymbol{z}$ and $\boldsymbol{\sigma}$ in the recode circuit. Each encoding is shown as a yellow box, with its innards obscured to parties without the private key. The orange slots illustrate where the "1" bits are in each encoded column of $\boldsymbol{s}$. There is one bit per each ℓ-length column of $\boldsymbol{s}$, and one bit for every θ-length "box" in the Θ-length rows of $\boldsymbol{s}$. Each z_i vector denotes one binary number between 0 and 2, with n bits of precision. For example, z_0 is 1.0011, or 1.1875 in decimal. When the z_i vectors are added (second row), the addition is done with appropriate carries, naturally in parallel for each "slot". The last line, where the most significant bits of the sum are XORed together, is the final rounding of the binary towards one or zero. c_{prime} is then XORed with (c mod 2) (not shown), completing the squashed decode circuit. Method inspired by [7].

A Semantic security and correctness

This scheme is semantically secure under the *Error-Free ℓ-Decisional Approximate-GCD* assumption, in the random oracle model. We direct the reader to [4][6] for a proof of security and a proof of correctness, omitted here due to space constraints.

4 IMPLEMENTATION

We implemented our scheme with both theoretical improvements in Python [8] first, using "toy" paramaters, as a proof of concept. The code is straightforward, unoptimized, and easy to reference: `https://github.com/jkwoods/PythonFHE`. It is similar to [9], but without external dependencies and with the batching improvement. Using Dask, a flexible library for parallel computing in Python [10], we ran this code in parallel as a proof of efficiency.

Instance	λ	ℓ	ρ	η	$\gamma \cdot 10^{-6}$	τ	Θ	pk size
toy	42	10	26	988	0.29	188	150	647KB
small	52	37	41	1558	1.6	661	555	13.3MB
medium	62	138	56	2128	8.5	2410	2070	304MB
large	72	531	71	2698	39	8713	7965	5.6GB

Table 1: Important Parameters; See security/correctness reasoning for parameters in [6][9]. Our library has a function for asserting correctness if a user wishes to specify their own parameters.

We then ported the code to C++ [11] to better control memory: `https://github.com/jkwoods/FHE-DGHV`. GMP, the GNU Multiple Precision Arithmetic Library [12], was used to handle large integers. It must be separately installed, but is useful due to built-in optimization and C++ bindings that compile to optimal C code without runtime overhead. OpenMP, an API for high-level parallel programming [13], was used to add another level of parallelism the scheme. There are embarrassingly parallel parts of the scheme, like the decode operation, that can be done for each m_i at the same time, as well as less obvious loops, like random generation during the encode operation. Four different parameter settings, for different levels of security (λ) and size of data (ℓ) shown in Table 1, were built and tested on a personal laptop (Intel i7-3635QM) and the Summit supercomputer (two IBM POWER9 processors and six NVIDIA Volta V100 accelerators per node).

A Higher-level abstraction

It may at first seem restrictive to only have access to addition and multiplication operations. But most classical integer manipulation can be performed using XOR and AND gates, available respectively as addition mod 2 and multiplication mod 2 within our scheme. Negation is done by XORing all encoded bits with an encoding of 1. From here, more complex operations can be built [14]. We can encode two's complement binary numbers, and perform n-bit addition and multiplication with textbook recipes, as we did with the z sets of binary numbers during the recode operation. n-bit subtraction is achieved by negation, followed by

adding an encoding of 1 with carrys. More specialized operators include exponentiation, or the selection operator:

$$\text{select}(c, a, b) = \begin{cases} a & \text{if } c = 1 \\ b & \text{if } c = 0 \end{cases}$$

$$\text{select}(c, a, b) = ca \oplus (\neg c)b$$

This makes vector assignment possible. With this "homomorphic assembly langauge" at our disposal, we construct several high level algorithms, from bubble sort [14], to AES evaluation [**?**], to key-value stores [**?**], to matrix multiplication. It is worth noting that any embarrassingly parallel underlying data remains embarrassingly parallel during any of these homomorphic high-level operations, due to the implicitly parallel nature of the batching implementation and structure of the scheme. We can extend our C++ implementation to a library. This library abstracts away complex operations, the need to consistently recode noisy encodings, and parallel and offloaded code.

5 CONCLUSION

Our C++ implementation is the first parallel implementation of the DGHV scheme, as well as the first implementation to incorporate both the compressed public key improvement and the batching improvement. Currently, this is the fastest implementation of the DGHV operations, most notably, the time-consuming recode operation. Our library enables applications like secure machine learning or healthcare transactions and makes fully homomorphic encoding more feasible than ever for practical use by non-specialists. Our experience with the programmability and performance of different languages/models will be used to help test and choose future supercomputer software and architectures.

Future work includes using special hardware, like general-purpose graphics processing units (GPUs) or field programmable gate arrays (FPGAs), to further mitigate the data-transformation cost. Operations like random number generation and modular multiplication (with Strassen's FFT based integer multiplication algorithm) and reduction (with Barrett's Modular Reduction algorithm) [15][16] can be offloaded to GPUs using CUDA [17]. We also hope to implement our scheme in Julia, an up and coming language for high performance computing [18], for the sake of comparison and ease of development.

6 ACKNOWLEDGEMENTS

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internship program.

References

[1] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *Stoc*, volume 9, pages 169–178, 2009.

[2] Craig Gentry and Shai Halevi. Implementing gentrys fully-homomorphic encryption scheme. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 129–148. Springer, 2011.

[3] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.

[4] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 446–464. Springer, 2012.

[5] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[6] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrede Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 315–335. Springer, 2013.

[7] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Annual Cryptology Conference*, pages 487–504. Springer, 2011.

[8] Python Core Team. *Python: A dynamic, open source programming language.* Python Software Foundation, 2015.

[9] Implementation of the dghv fully homomorphic encryption scheme. https://github.com/coron/fhe, 2012.

[10] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[11] Bjarne Stroustrup. *The C++ programming language.* Pearson Education, 2013.

[12] Torbjrn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012.

[13] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[14] Simon Fau, Renaud Sirdey, Caroline Fontaine, Carlos Aguilar-Melchor, and Guy Gogniat. Towards practical program execution over fully homomorphic encryption schemes. In *2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 284–290. IEEE, 2013.

[15] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Accelerating fully homomorphic encryption using gpu. In *2012 IEEE conference on high performance extreme computing*, pages 1–5. IEEE, 2012.

[16] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Exploring the feasibility of fully homomorphic encryption. *IEEE Transactions on Computers*, 64(3):698–706, 2013.

[17] J. Nickolls. Scalable parallel programming with cuda introduction. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–9, Aug 2008.

[18] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

Research and Technical Experience Deliverable

Higher Education Research Experiences (HERE) at ORNL

Mentored by Dr. Oscar Hernandez

Computer Science and Mathematics Division

Jess Woods

January 6, 2020 - July 31, 2020

1 Personal Background

I am a recent graduate with a B.S. in computer science and studio art from the University of North Carolina at Chapel Hill. I spent the last year at Oak Ridge National Laboratory under various internship programs, working with the Center for Molecular Biophysics and the Computer Science Research Group, with Ada Sedova and Oscar Hernandez. I was lucky to receive wonderful mentorship and guidance in applying to graduate school. I accepted at the University of Pennsylvania and will be staring a computer science Ph.D. program in the fall of 2020. My time at ORNL has made me confident I wish to pursue a career in computer science research.

2 Project Introduction

As high-performance computing (HPC) becomes more heterogeneous, questions are raised about the best way to program supercomputers. Successful parallel programming models aim to obtain the highest possible performance for breakthrough science applications without compromising user efficiency. During my HERE appointment, I worked with Oscar Hernandez to develop a scientific application across three different parallel programming models. We implement a program in C++, in conjunction with OpenMP and MPI. C++ is a classic fine-tuned parallel programming language. We also implement the same program in Python, which is not a traditional HPC programming language, but is now rapidly entering the HPC arena due to its use in advanced AI workflows. Third, we use Julia, a new language which aims to improve the programmability of parallel computers by bridging the gap between efficient low-level systems programming (like C) and attractively simple high-level languages (like Python). We investigate the effect of these different models on performance, portability, and programmability.

For this comparison program, I chose to implement a library for abstract algebra operations (additions, matrix multiplications, etc.) that support big integers. A more complete mathematical explanation of the abstract algebra is included in the Appendix. These operations are a good candidate for parallelization, since they can be made into data parallel work. The bottleneck to many modern abstract algebra applications is computation time on large integers; that is, integers bigger than the typical 64-bit storage size. Big numbers like these are used in applications like in cosmology, hash tables, probability simulations, and math sequence exploration. HPC presents a interesting way to solve this problem.

3 Design and Implementations

C++ and OpenMP Our C++ implementation optimizes performance and provides a benchmark. GMP, the GNU Multiple Precision Arithmetic Library [1], handles large integers. It must be separately installed, but is useful due to built-in optimizations and C++ bindings that compile to optimal C code without run time overhead. OpenMP [2], an API for high-level parallel programming, parallelizes the scheme. Both explicit threading and the tasking constructs, with accompanying atomic and barrier

constructs, were tested. OpenMPI [3], an open source "Message Passing Interface" implementation, distributes the C++ code across multiple nodes.

Python and Dask Originally, we prototyped our application in Python [4], due to its programmability. Dask [5], a flexible library of data structures and operations for parallel computing in Python, was used to parallelize this code. The SymPy Python library [6] handles our random numbers. SymPy is built on top of NumPy, for symbolic computation. The Python library gmpy2 [7], a Python wrapper for the GMP library, handles our big integers. Python has a big integer class, but in practice, we found it causes a slow down of several orders of magnitude.

Julia Julia [8] is a new programming language designed with HPC in mind, circumventing many of Python's original performance roadblocks. It has high-level syntax, dynamic typing, just-in-time compilation, and supports interactive use. It also has built in support for concurrent, parallel, and distributed computing, as well as direct C and Fortran code calling. This proved useful in our Julia implementation, since we did not require outside libraries to for parallelization or big integer handling. Running Julia on Summit required building Julia from source. The Julia project does not maintain consistent support for Power9 processors, so building required some extra work. This experience will be helpful to other Julia users on Summit and future supercomputers. We provide a script for replicating our Summit build, and hope to formalize an official Julia module on Summit.

4 Results

We did most development and testing for correctness on a small personal machine, as a sequential program first. But all implementations had to be adapted to Summit for parallelization, testing, and performance benchmarks. Summit is a supercomputer with two 22-core IBM POWER9 processors (with 4 hardware threads per core) and six NVIDIA Volta V100 GPUs per node. It is sometimes difficult to get things running on Summit directly "out of the box", given the security firewalls and complex batch job submission system. This is most true for Dask and Julia, which run in parallel by spawning their own worker processes. Summit will shut this down. So preliminary building and testing for the Python and Julia implementations was done on Raptor, which is an isolated system resembling a single Summit node. It does not have the permissions and security firewall issues Summit does, but it tops out at 8 cores (24 hardware threads).

Performance In Figures 1, 2 and 3, we show the scalablility of a matrix multiplication, as a sample. The multiplication is divided into three parts, and each is tested for a "small"(er) and "large" integer size. These two sets contain about 13MB and 300MB of total data, respectively. The nature of C++ means our implementation has little to no overhead, performs excellently, and scales well. Explicit OpenMP threads do a little better than OpenMP tasks.

Dask can provide a good speed-up from vanilla Python. But we found this varies a lot based on the scheduler, data structures, and data "chunk" size used. For example, the Dask distributed

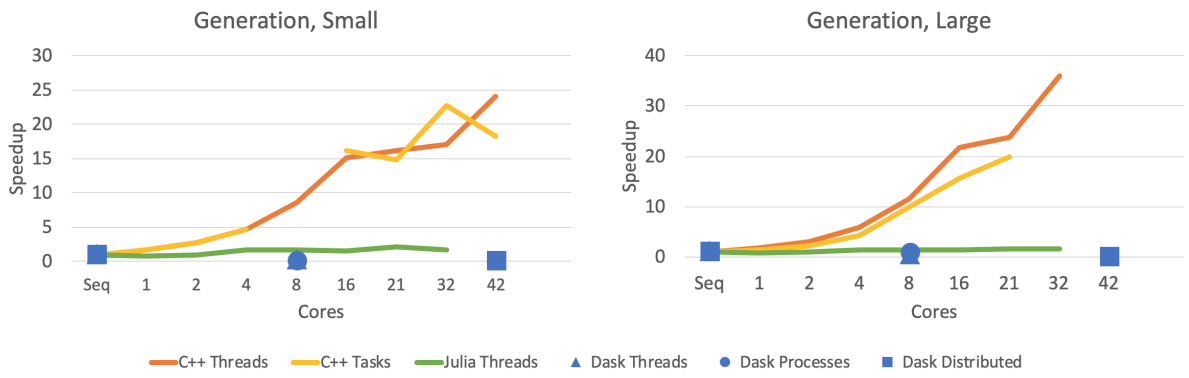

Figure 1: The matrix multiplication starts with the generation of many preliminary data structures; this only has to be done once, at the beginning of the work session. Note Dask does not allow for running a specific thread number like C++ and Julia, so we do not have the same type of data. Additionally, some Dask schedulers could only be run on the Raptor system, which tops out at 8 cores. Julia tops out at 32 cores on both machines.

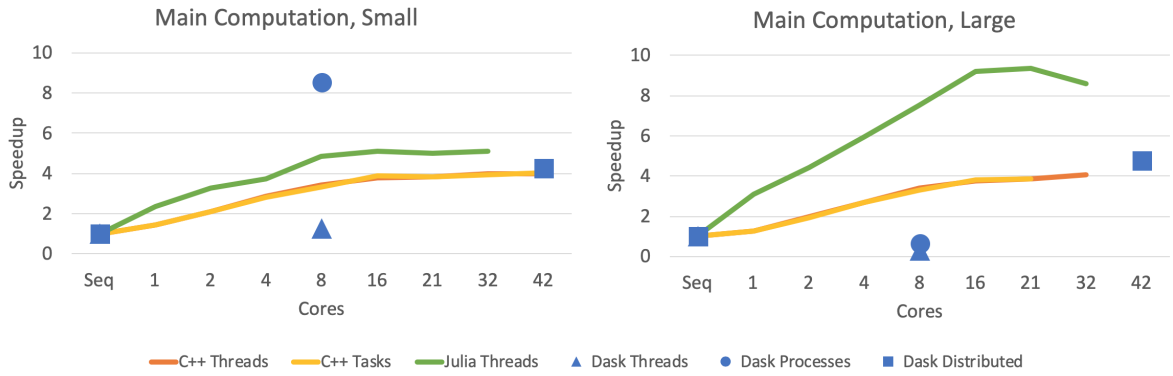

Figure 2: The second operation is done the most often and involves the bulk of the large number processing. Our parallelization work focused here. Distributed Dask/Python scales well and Julia scales excellently.

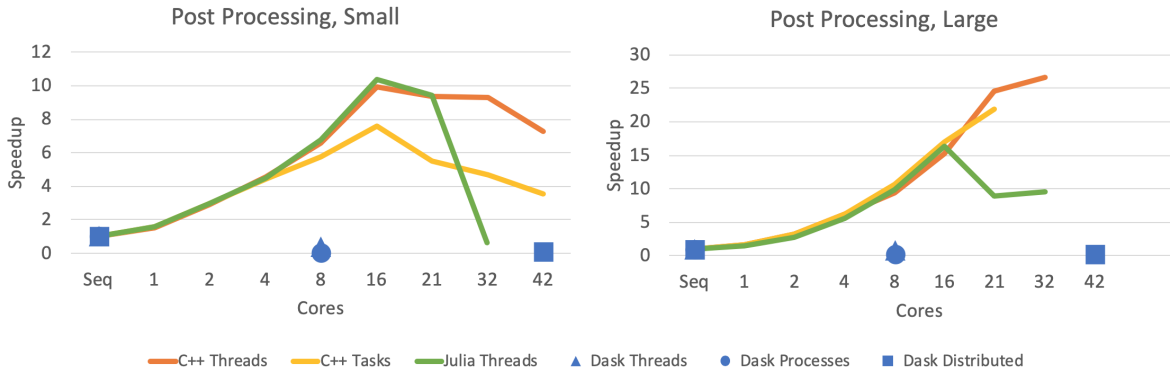

Figure 3: The third is a quick post-processing. Because no computationally heavy work is done, the overhead quickly overwhelms the run time, especially in Python.

scheduler preformed best with our code, made of mostly Dask arrays and Dask "delayed" constructs. Also, different operations obviously do better than others. The first and third operations are heavier on pure Python objects (lists, external library objects, custom reductions, etc.), and scale less well.

Our Julia implementation performs and scales surprisingly well, considering it's Python-like appearance. It is comparable to (and in some areas, better than) the performance of the statically typed, compiled C++ implementation. However, there were a few outlying longer run times, due to Julia's unpredictable garbage collector. We mention this here since it is unclear in the charted data; each data point is an average of dozen of runs.

Portability C++ is ultra portable, given the correct compiler for your machine. In our C++ code, OpenMP provides a mechanism for turning on and off parallelization in a single line. MPI is required for distribution and OpenMP code changes are required for GPU use, but the conversion between single-machine and distributed-machine parallelism is otherwise simple.

Dask enabled our Python code to be very easily portable. A conversion between sequential code, single-machine parallelism, and distributed-machine parallelism could be done by simply changing the scheduler used, often in a single line of code. In fact, different schedulers can be used for different code sections. However, Summit presented some obstacles. Dask's single-machine schedulers simply would not run. The distributed scheduler worked well for me, given the previous set-up work by Benjamin Hernandez and other Dask/Summit users. But even then, certain functionalities (like the Dask job queue) were unavailable. The distributed scheduler often produced hard-to-trace errors like internal package dependency issues, unreplicable network disconnects, or worker spawning errors. Portability to another supercomputer would require Dask installation and setup, which might come with its own system-specific difficulties and surprises.

Julia has the potential is be easily portable as well. Parallelism is built in, in many forms, from threads and atomic operations, to tasks, to message passing. You can write code with "shared" or "distributed" arrays accessible from single-core or distributed machines. However, the Summit system required code changes and separate MPI and CUDAnative packages for use on the distributed machine and GPUs. Similarly to Dask, getting Julia installed and running on a different supercomputer might require system-specific adjustments.

Programmability Python is a widely used high-productivity language, and a favorite of many domain experts, both inside and outside computer science. As is evident, Python can be made to work well for high-performance computing, but it requires familiarity with outside libraries. C++ and OpenMP require a more extensive knowledge of coding and computer memory. Users outside the computer science field may find it harder to learn without reading a textbook. We ran into strange memory errors and invisible race conditions. Of particular interest were the stack-size errors caused by large integer handling; running the program in a new environment required checking and changing the system stack size. A C++ program is also much more time-consuming to write. While Julia is a new language, it is very easy to learn. It has all the features that made Python popular: dynamic typing, simple programming syntax, and code readability. But it is much faster and has many built-in

features Python requires outside libraries for. The Julia community, though currently small, is fast growing.

5 Conclusion

My implementations are the first parallel implementations of this theoretical scheme, as well as the first implementations to incorporate both a data-compression improvement and a parallel data-batching improvement. All implementations, in Python, C++, and Julia, with sequential, parallel, and distributed versions, including GPU work, are available on Github [9, 10, 11]. Such an application can be useful for users interested abstract algebra encoding, secure machine learning, and more. My code can also serve as a useful reference for people interested in the programming aspects. For example, earlier this year, I spoke with a Dask working group at ORNL about my Dask experiences and strategies.

I believe Julia is an overlooked HPC language. In many cases, it can serve as a better alternative to Python for non-expert users. Some operations are faster than C++ and it has built in parallel and distributed programming. Python has come a long way for HPC efficiency. But often, good performance requires outside libraries and new syntax anyway. Lastly, for computer science experts focused on performance, there is nothing quite as good as classic C++.

6 Experience

Publications and Presentations I was accepted to give a talk on this work at the P3HPC 2020 conference, which will be held virtually in September 2020. We also are planning on submitting a paper on this work to a Supercomputing 2020 workshop. I may attend the Supercomputing conference in November. Additionally, I assisted with writing a white paper on HPC Python, where I used my work as an example use case, with some more senior staff at ORNL and Georgia Tech.

Additional Participation During my appointment, I attended the Computer Science Research group's biweekly meetings. This gave me an exposure to the management/administrative side of research; Research must function within an organization, not a vacuum. This requires a certain amount of cooperation and social interaction I was not taught to anticipate in school. I was also given the opportunity to attend (in person and virtually) many interesting research talks, both inside and outside of computer science.

Concluding Thoughts and Future Plans My experience was, overall, very positive. I enjoyed the national lab's social and academic environment. The people I met and the resources I had access to were fantastic. The last few months have obviously been less than ideal, due to COVID-19, but I am thankful I was here when it hit. I thought ORNL handled the situation better than most schools and workplaces did. As mentioned above, I plan on attending graduate school in the fall. I have tentative membership in a lab focused on security and distributed systems. My experience at Oak Ridge using the Summit supercomputer and exploring parallel and HPC programming models will definitely give me an advantageous and useful base for my graduate research.

References

[1] Torbjrn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012.

[2] OpenMP Architecture Review Board. OpenMP application program interface version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[3] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[4] Python Core Team. *Python: A dynamic, open source programming language.* Python Software Foundation, 2015.

[5] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[6] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[7] gmpy2 documentation. `https://gmpy2.readthedocs.io/en/latest/`, 2017.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[9] Jess Woods. Pythonfhe: Python implementation of dghv scheme. `https://github.com/jkwoods/PythonFHE`, 2020.

[10] Jess Woods. Fhe-dghv: C++ implementation of dghv scheme. `https://github.com/jkwoods/FHE-DGHV`, 2020.

[11] Jess Woods. Juliafhe: Julia implementation of dghv scheme. `https://github.com/jkwoods/JuliaFHE`, 2020.

[12] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.

[13] Implementation of the dghv fully homomorphic encryption scheme. `https://github.com/coron/fhe`, 2012.

[14] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *Stoc*, pages 169–178, 2009.

[15] Craig Gentry and Shai Halevi. Implementing gentrys fully-homomorphic encryption scheme. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 129–148. Springer, 2011.

[16] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 446–464. Springer, 2012.

[17] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[18] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrede Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 315–335. Springer, 2013.

[19] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Annual Cryptology Conference*, pages 487–504. Springer, 2011.

A Appendix: Complete Fully Homomorphic Scheme

Here we provide a complete description of our DGHV variant [12, 13] (based on earlier work in [14, 15]) with the compressed public key [16] and batching improvements [17, 18].

KeyGeneration():

Generate ℓ primes $\boldsymbol{p} = (p_0, ..., p_{\ell-1})$ of η bits each and denote their product π.
$x_0 \leftarrow q_0 \cdot \pi$, where $q_0 \leftarrow_R \mathbb{Z} \cap [0, 2^\gamma/\pi)$ and q_0 is a 2^{λ^2}-rough integer.[1] Note x_0 is error-free.

Generate at random ℓ vectors $\boldsymbol{s}_j$ of length Θ split into θ boxes of size Θ/θ,
such that the Hamming Weight of each box is 1 and $s_{j,j} = 1$ for $j \in [0, \ell - 1]$

Initialize pseudo-random generator f_1 with a random seed se_1
Use $f_1(se_1)$ to generate sets of integers:
$\chi_i \leftarrow_R [0, x_0)$ for $i \in [1, \tau]$
$\chi'_i \leftarrow_R [0, x_0)$ for $i \in [1, \ell - 1]$

[1] An integer is said to be a-rough when it does not contain prime factors smaller than a. If $a > 2$ the integer must be odd. $q0$ can be generated as a product of 2^{λ^2}-bit primes

$\chi_i^{\Pi} \leftarrow_R [0, x_0)$ for $i \in [1, \ell - 1]$

Initialize pseudo-random generator f_2 with a random seed se_2
Use $f_2(se_2)$ to generate a set of integers: $u_i \leftarrow_R [0, 2^{\kappa+1})$ for $i \in \mathbb{Z} \cap [0, \Theta - 1]$

Then set $\boldsymbol{u} = (u_0, ..., u_{\ell-1})$ such that

$$x_{p_j} \leftarrow \sum_{i=0}^{\Theta-1} s_{j,i} \cdot u_i \bmod 2^{\kappa+1}$$

where $x_{p_j} = \lfloor 2^{\kappa}/p_j \rceil$ for $j \in [0, \ell - 1]$

Initialize pseudo-random generator f_3 with a random seed se_3
Use $f_3(se_3)$ to generate a set of integers: $\chi_i^{\sigma} \leftarrow_R [0, x_0)$ for $i \in [0, \Theta - 1]$

Define γ-bit integers as follows:

1. $x_i \leftarrow \chi_i - \Delta_i$ for $i \in [1, \tau]$ with[2]

$$\Delta_i \leftarrow [\chi_i]_{\pi} + \xi_i \cdot \pi - \mathrm{CRT}_{p_0,...,p_{\ell-1}}(2r_{i,0}, ..., 2r_{i,\ell-1})$$

   where $r_{i,j} \leftarrow_R \mathbb{Z} \cap (-2^{\rho'-1}, 2^{\rho'-1})$ and $\xi_i \leftarrow_R \mathbb{Z} \cap [0, 2^{\lambda+log_2(\ell)+\ell\cdot\eta}/\pi)$

2. $x'_i \leftarrow \chi'_i - \Delta'_i$ for $i \in [1, \ell - 1]$ with[3]

$$\Delta'_i \leftarrow \left[\chi'_i\right]_{\pi} + \xi'_i \cdot \pi - \mathrm{CRT}_{p_0,...,p_{\ell-1}}(2r'_{i,0} + \delta_{i,0}, ..., 2r'_{i,\ell-1} + \delta_{i,\ell-1})$$

   where $r'_{i,j} \leftarrow_R \mathbb{Z} \cap (-2^{\rho}, 2^{\rho})$ and $\xi'_i \leftarrow_R \mathbb{Z} \cap [0, 2^{\lambda+log_2(\ell)+\ell\cdot\eta}/\pi)$

3. $\Pi_i \leftarrow \chi_i^{\Pi} - \Delta_i^{\Pi}$ for $i \in [1, \ell - 1]$ with

$$\Delta_i^{\Pi} \leftarrow \left[\chi'_i\right]_{\pi} + \xi_i^{\Pi} \cdot \pi - \mathrm{CRT}_{p_0,...,p_{\ell-1}}(2r_{i,0}^{\Pi} + \delta_{i,0}2^{\rho'+1}, ..., 2r_{i,\ell-1}^{\Pi} + \delta_{i,\ell-1}2^{\rho'+1})$$

   where $r_{i,j}^{\Pi} \leftarrow_R \mathbb{Z} \cap (-2^{\rho}, 2^{\rho})$ and $\xi_i^{\Pi} \leftarrow_R \mathbb{Z} \cap [0, 2^{\lambda+log_2(\ell)+\ell\cdot\eta}/\pi)$

4. $\sigma_i \leftarrow \chi_i^{\sigma} - \Delta_i^{\sigma}$ for $i \in [0, \Theta - 1]$ with

$$\Delta_i^{\sigma} \leftarrow [\chi_i^{\sigma}]_{\pi} + \xi_i^{\sigma} \cdot \pi - \mathrm{CRT}_{p_0,...,p_{\ell-1}}(2r_{i,0}^{\sigma} + s_{0,i}, ..., 2r_{i,\ell-1}^{\sigma} + s_{\ell-1,i})$$

   where $r_{i,j}^{\sigma} \leftarrow_R \mathbb{Z} \cap (-2^{\rho}, 2^{\rho})$ and $\xi_i^{\sigma} \leftarrow_R \mathbb{Z} \cap [0, 2^{\lambda+log_2(\ell)+\ell\cdot\eta}/\pi)$
   Note that each σ_i is just an encoding of a column of $\boldsymbol{s}$. This is what enables **Recode** to work in parallel.

[2] CRT denotes Chinese Remainder Theorem.
[3] $\delta_{i,j}$ denotes the Kronecker delta: $\delta_{i,j} = 1$ if $i = j$ and 0 otherwise.

Output the private key $\boldsymbol{sk} \leftarrow \boldsymbol{s}$, a two dimensional vector of 0s and 1s.
Output the public key:

$$\boldsymbol{pk} \leftarrow \left\langle x_0, se_1, \boldsymbol{\Delta}, \boldsymbol{\Delta'}, \boldsymbol{\Delta^{\Pi}}, se_2, \boldsymbol{u}, se_3, \boldsymbol{\Delta^{\sigma}} \right\rangle$$

Encode($\boldsymbol{pk}, m \in \{0,1\}^{\ell}$):

Use $f_1(se_1)$ to recover integer vectors $\boldsymbol{\chi}, \boldsymbol{\chi'}, \boldsymbol{\chi^{\Pi}}$
$\boldsymbol{x} \leftarrow \boldsymbol{\chi} - \boldsymbol{\Delta}$
$\boldsymbol{x'} \leftarrow \boldsymbol{\chi'} - \boldsymbol{\Delta'}$
$\boldsymbol{\Pi} \leftarrow \boldsymbol{\chi^{\Pi}} - \boldsymbol{\Delta^{\Pi}}$
Generate a random vector $b_i \leftarrow_R (-2^{\alpha}, 2^{\alpha})$ for $i \in [1, \tau]$
Generate a random vector $b'_i \leftarrow_R (-2^{\alpha'}, 2^{\alpha'})$ for $i \in [0, \ell - 1]$
Output the encoding:

$$c \leftarrow \left[\sum_{i=0}^{\ell-1} m_i \cdot x'_i + \sum_{i=1}^{\tau} b_i \cdot x_i + \sum_{i=0}^{\ell-1} b'_i \cdot \Pi_i \right]_{x_0}$$

Add($\boldsymbol{pk}, c_1, c_2$): Output $c_1 + c_2 \bmod x_0$

Multiply($\boldsymbol{pk}, c_1, c_2$): Output $c_1 \cdot c_2 \bmod x_0$

Expand($\boldsymbol{pk}, c$):

Use $f_1(se_1)$ to recover $\boldsymbol{u}$
Compute $z_i \leftarrow (\lfloor c \cdot (u_i / 2^{\kappa}) \rceil \bmod 2)$ for $i \in [0, \Theta - 1]$ to n bits of precision after the binary point
Output $\boldsymbol{z}$

Decode($\boldsymbol{sk}, c, \boldsymbol{z}$): Output $\boldsymbol{m} = (m_0, ..., m_{\ell-1})$ with

$$m_j \leftarrow \left\lfloor \sum_{i=0}^{\Theta-1} s_{j,i} \cdot z_i \right\rceil \bmod 2 \oplus (c \bmod 2)$$

Recode($\boldsymbol{pk}, c, \boldsymbol{z}$): Apply decode circuit to expanded encoding $\boldsymbol{z}$, and encoded private key bits σ_i (in place of $s_{i,j}$). Note that each bit of each z_i is treated individually, as shown in Figure 4. There are a finite number of bits for each z_i, since each is only calculated to n bits of precision after the binary point. Output new encoding.

A.1 Semantic security and correctness

This scheme is semantically secure under the *Error-Free ℓ-Decisional Approximate-GCD* assumption, in the random oracle model. We direct the reader to [16][18] for a proof of security and a proof of correctness, omitted here due to space constraints.

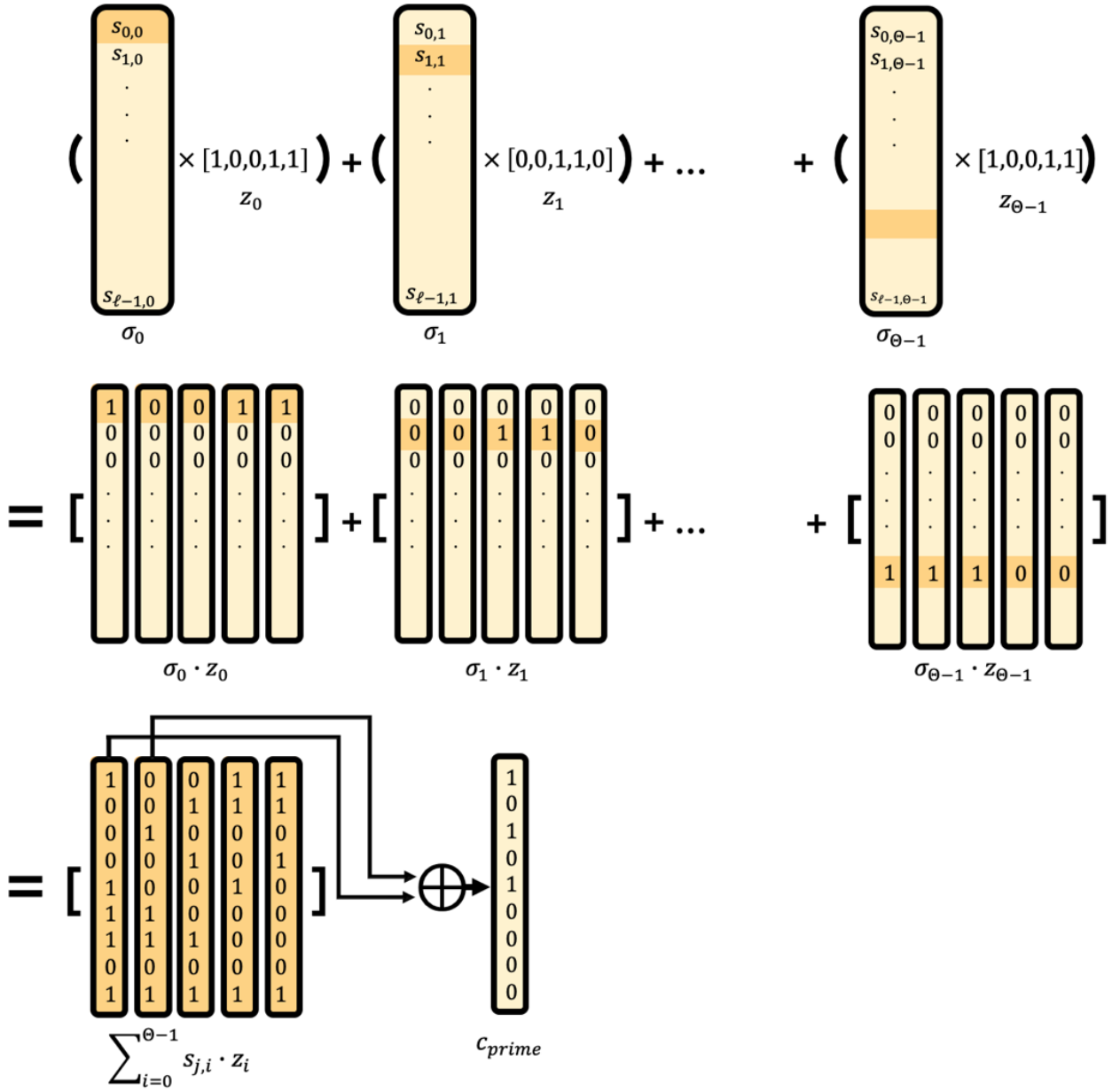

Figure 4: Evaluation of $\boldsymbol{z}$ and $\boldsymbol{\sigma}$ in the recode circuit. Each encoding is shown as a yellow box, with its innards obscured to parties without the private key. The orange slots illustrate where the "1" bits are in each encoded column of $\boldsymbol{s}$. There is one bit per each ℓ-length column of $\boldsymbol{s}$, and one bit for every θ-length "box" in the Θ-length rows of $\boldsymbol{s}$. Each z_i vector denotes one binary number between 0 and 2, with n bits of precision. For example, z_0 is 1.0011, or 1.1875 in decimal. When the z_i vectors are added (second row), the addition is done with appropriate carries, naturally in parallel for each "slot". The last line, where the most significant bits of the sum are XORed together, is the final rounding of the binary towards one or zero. c_{prime} is then XORed with (c mod 2) (not shown), completing the squashed decode circuit. Method inspired by [19].