Computer Security Symposium 2017
23 - 25 October 2017

超高速秘密計算ソートの設計と実装：秘密計算がスクリプト言語に並ぶ日

五十嵐 大[1,a)]　濱田 浩気[1]　菊池 亮[1]　千田 浩司[1]

概要：本稿では３パーティの秘密分散ベース秘密計算におけるソート処理の高速化を行う. ソート処理は秘密計算において, 垂直結合, 最大/最小/中央値, 集約演算 (または統計における数量表/集計表、SQL における group-by 演算), 一括表参照など, 統計およびデータベースにおける非常に重要な演算の要素処理である. 本稿ではプロトコルおよび実装の最適化により, スクリプト言語上の平文のソートと比較しうる性能 (1,000 万件 6.0 秒) を実現できることを示す.

キーワード：マルチパーティ計算, 基数ソート, 秘密分散

A Design and an Implementation of Super-high-speed Multi-party Sorting: The Day When Multi-party Computation Reaches Scripting Languages

Dai IKARASHI[1,a)]　Koki HAMADA[1]　Ryo KIKUCHI[1]　Koji CHIDA[1]

1. はじめに

秘密計算は暗号分野において大きく注目されている分野のひとつである. 特に秘密分散ベース秘密計算においては, 既に sharemind [1] などの高速なソフトウェアが利用可能であり, 実用に向けた機運も高まりつつある.

秘密計算の実用に関しては, ソート処理の課題がある. ソートは業務システムで多く利用されるデータベースシステムでは, 最も基本的かつボトルネックであり, 最重要の処理である. また統計処理を行う際にもソートは最大値・最小値・中央値等の統計処理を導くほか, これまでに筆者らはソートを利用した高速な表結合 [2], 並列アドレス参照 [3], また統計分野で集計表・数量表, データベース分野でグループ化と呼ばれる処理を提案した [4]. このように, それ自体重要な演算であるだけでなく多くの重要な演算がソートから構成可能であり, ますます秘密計算におけるソート処理の重要性は増したといえる.

一方, 10G-Ether が安価になりサーバ用途でかなり普及してきており, ネットワークを用いる秘密分散ベース秘密計算においても 10G ネットワーク環境を視野に入れた性能向上が行われるようになってきている. 従来は通信量が最大のボトルネックだったが, 10G ネットワーク環境では通信が高速なため, ローカル計算のコストも重要なボトルネックとなる.

筆者らは秘密計算の広い実用化のため, 導入障壁としての秘密計算の性能の低さを解消を狙って, 平文と可能な限り遜色のない性能を目指している. 本稿では, 実用的な応用において最も重要と考えられるソートに関して, 筆者らが提案した秘密計算基数ソート [5] [6] のプロトコル改良および実装の高速化により, スクリプト言語において平文をソートする場合と比較しうるほどの性能を実現する. スクリプト言語は C などのコンパイラ言語と比較すれば低速であるがそれでも広く実用されている処理のフレームワークであり, このレベルの性能に達することは実際の広汎な利用シーンで性能面の障害とならないことを意味する.

1.1 既存研究

秘密計算によるソートについては以下のようなものがある.

Wang らは 2 パーティの秘密計算ソフトウェア Fairplay [7] 上で幾つかの既存のソート手法を実装し, 256 データで最良約 3,000 秒であった [8].

Jónsson らは秘密分散ベースの 3 パーティ秘密計算ソフトウェア sharemind 上でやはり既存のソート手法を実装し, 16,384 件で最良約 200 秒であった [9].

筆者らは文献 [5] において秘密計算による基数ソートを提案し, 約 360 件/秒の性能を示した. またランダム置換を用いたクイックソートの提案を行い [10], 最大約 1,070 件/秒の性能を示した. さらに文献 [6] においては基数ソートプロトコルを通信量・ラウンド数とも改良し, 通信量がボトルネックとなる 10G LAN 上 1,000 万データにおいて約 30 秒, ラウンド数がボトルネックとなるインターネット環境 1,000 データにおいて約 1 秒の応答時間を実現した.

1　NTT セキュアプラットフォーム研究所, NTT Secure Platform Laboratories
a)　ikarashi.dai@lab.ntt.co.jp

© 2017 Information Processing Society of Japan

1.2 本稿の改善の構成

本稿では秘密計算基数ソート [5] [6] をもとに, 以下の改良を行う.

1. ソートプロトコルの理論的改善
2. passive モデル向けのプロトコル効率化
3. プロトコルの通信路負荷の最適化
4. ローカル演算の高速化

なお, 本稿の提案手法は全て 3 パーティの passive(semi-honest) プロトコルである. 完全秘匿の passive プロトコルにおいては, 安全性の証明は煩雑なだけで難しくないことから省略する.

2. 準備

2.1 共通的な記法

2.1.0.1 キーとバリュー

ソートには**キー**と**バリュー**がある. キーはソート後の順序を決めるタグであり, バリューはソートされるべきデータ本体である. キーとバリューが同一のこともある.

2.1.0.2 秘密分散値

秘密分散された全パーティのシェアを仮想的に集めた組を, 秘密分散値と呼ぶ.

2.1.0.3 置換の適用

置換 π をベクトル $\vec{x}$ に適用した結果を乗法的に表記し, $\pi\vec{x}$ と書く. また整数ベクトルは置換としても扱い, ベクトルの乗法的記法 $\vec{y}\vec{x}$ は, $\vec{x}$ の $\vec{y}$ による置換である.

2.1.0.4 記号類

- m: ソートおよび置換対象のベクトルの要素数.
- L: キーのビット長.
- p, q: 素数.
- $|p|, |q|$: 素数のビット長.
- $[\![x]\!]$: 準同形性を持つ秘密分散値.
- $\{x\}$: 複製秘密分散値であることを明示する記法.
- $[x]$: $(2, 2)$ 加法的秘密分散値.
- $\langle x\rangle$: 準公開値. すなわち, k パーティで共有する平文.
- $[\![X]\!]$ など: X 上の秘密分散値の集合.
- $[x]^{X,\mathcal{P}}$: 上記分散値等で, 肩の上の 1 個目の添え字は群/環/体, 2 個目はシェアを持つパーティ集合を表す.
- $\vec{x}, [\![\vec{x}]\!]^X$: 長さ m のベクトルおよびその秘密分散値.
- $\mathbb{P}$: パーティ全体の集合.
- 01, 12, 20: 添え字に使ったときそれぞれ, パーティ 0 と 1, パーティ 1 と 2, パーティ 2 と 0 のパーティ集合を表す.
- $[\![x]\!]_P$: パーティ P のシェア.
- $\{\pi\}$: 置換 π の複製秘密分散値.
- $\{\pi\}_{\mathcal{P}}$: $\mathcal{P} \subseteq \mathbb{P}$ が共有する $\{\pi\}$ の断片 (サブシェア).
- $\{\pi\}^{01,12,20}$ のように添え字付きの置換の複製秘密分散値: $\pi = \{\pi\}_{20}\{\pi\}_{12}\{\pi\}_{01}$ となる複製秘密分散値. すなわち, 置換の適用順まで気にした場合の記法.
- $\{\pi\}^{01,12}$ のように 3 個未満の添え字付きの置換の複製秘密分散値: $\pi = \{\pi\}_{12}\{\pi\}_{01}$ となる複製秘密分散値.

2.2 (2,3)-複製秘密分散

複製秘密分散とは, 複数のパーティ集合ごとに, パーティ集合内で同じ値を共有するような秘密分散である.

例えば $(2, 3)$-複製秘密分散は, $a = a_{01} + a_{12} + a_{20}$ とし, 各パーティのシェアは $(a_{20}, a_{01}), (a_{01}, a_{12}), (a_{12}, a_{20})$ となる. 各 a_{01}, a_{12}, a_{20} を**サブシェア**と呼ぶ.

本稿においては, 置換が群であることを利用し, 置換を秘匿する秘密分散としても利用する.

3. 従来手法: 旧 IHKC ソート

旧 IKHC ソートは筆者らが遅延の大きいインターネット環境で 1,000 データにおいて 1 秒以内のレスポンスを目指して提案した基数ソートプロトコルである. この目標も非常に厳しい条件であり, IKHC ソートは下記のように既に非常に最適化されたプロトコルになっている.

1. 処理の段階に応じて最適なシェアに変換しながら処理を行う
2. プロトコルの並列化がされている
3. 基数の解析がされ最適な基数が選択されている
4. 下位プロトコルも効率化されている

そのため, ここからの改善は大変困難であると考えられるが, ソート処理の実用的重要性を考えるとそれでもチャレンジすべき課題である.

Scheme 1, Scheme 2 に [6] の基数ソート (以降 IHKC ソートと呼ぶ) を記した. Scheme 3, Scheme 4, Scheme 5, Scheme 6 は下位プロトコルである. Scheme 1 がキーを処理してソートを表現する置換を得る部分, Scheme 2 が並び替え対象のバリューに上記置換を用いてソートを行う部分である. 負荷のほとんどはキー処理の方にある.

3.1 基数ソート概要

基数ソートとはキーを 0 桁目に着目してソートし, 次に 1 桁目に着目してソートし, ..., という繰り返しによりソートするアルゴリズムである. データが固定長である場合は線形計算量となりソートアルゴリズム中最速となる. 秘密計算においては秘匿性のためデータは普通固定長であり, 基数ソートが適している.

3.2 入出力の概要

Scheme 1 はビット列入力であり, 平文の置換と秘匿された置換の組を出力する.

ビット列は複製秘密分散での分散が想定される. (Shamir 秘密分散は mod 2 の分散が効率的にできない)

出力に関しては, 平文の置換の方で, ソートを表す置換であり秘匿対象である σ は, 一様ランダム置換 π の逆写像が乗ぜられることで秘匿されていることに注意. Scheme 2 はこの出力を用いて計算された並び順通りにソートする.

3.3 下位プロトコル概要

Scheme 3, Scheme 4 は 2 つ併せて ℓ ビットソート, すなわち基数 2^ℓ としたときの基数ソートにおける 1 回のソートである. 旧 IHKC ソートでは通信量, ラウンド数を解析した結果 $\ell = 3$ が最良であった.

Scheme 5 はビット形式すなわち mod 2 の入力を mod q に変換する処理である. 本稿では [6] よりも具体的なプロトコルを記述した. 詳細は割愛するが XOR 計算時, 入力に定数 0 が含まれるため 2 回の XOR が 1 回の乗算と等しい通信量で済むことに注意されたい.

Scheme 6 は [6] で効率化されたランダム置換プロトコルである. 入出力は $(2, 2)$-加法的秘密分散であり, Shamir/複製秘密分散を入出力とする際は変換を伴う. 他に, 公開値入力分散値出力/分散値入力公開値出力のランダム置換があり, Scheme 1, Scheme 2 における shuffle は入出力の形式に応じて 3 つのランダム置換を使い分ける.

© 2017 Information Processing Society of Japan

Scheme 1 [旧 IHKC ソート: キー処理]
入力: $[\![\vec{k}_0]\!]^{\mathbb{Z}_2^\ell}, \cdots, [\![\vec{k}_{N-1}]\!]^{\mathbb{Z}_2^\ell}$, ただし $N\ell = L$
出力: $\sigma\pi^{-1}, \{\pi\}$, ただし σ はソートを表す置換関数

1: **for each** $i < N$ **do** in paralel
2: $\{\pi_i\}, \{\pi'_i\}$ を生成する. ただし $\{\pi_0\}$ は恒等置換とする.
3: ランダム ID 列 $[\![\vec{h}_i]\!]^{\mathbb{Z}_q} := \mathrm{shuffle}(I, \{h_i\})$ を作成する. ただし I は恒等置換すなわちベクトル $0, 1, \cdots, m-1$ である.
4: $\langle\pi_N\rangle$ を生成する.
5: **for each** $1 \le i < N$ **do** in parallel
6: $(\pi_i\vec{h}_{i-1}, [\![\pi_i\vec{k}_i]\!]^{\mathbb{Z}_2^\ell}, [\![\pi_i\vec{h}_i]\!]^{\mathbb{Z}_q})$ $:= \mathrm{shuffle}([\![\vec{h}_{i-1}]\!]^{\mathbb{Z}_q}, [\![\vec{k}_i]\!]^{\mathbb{Z}_2^\ell}, [\![\vec{h}_i]\!]^{\mathbb{Z}_q}, \{\pi_i\})$
7: $\pi_N\vec{h}_{N-1} := \mathrm{shuffle}([\![\vec{h}_{N-1}]\!]^{\mathbb{Z}_q}, \{\pi_N\})$
8: **for each** $i < N$ **do** in parallel
9: $[\![\pi_i\vec{k}_i]\!]^{\mathbb{Z}_2^\ell}$ の各要素に対しフラグ作成 (Scheme 3) を行い, $[\![\pi_i\vec{f}_i]\!]^{\mathbb{Z}_p^{2^\ell}}$ を得る.
10: σ_0 を恒等置換とおく.
11: **for each** $i < N$ **do** in series
12: $[\![\sigma_i\vec{f}_i]\!]^{\mathbb{Z}_p^{2^\ell}}$ の順位表作成を行い, $[\![\vec{s}]\!]^{\mathbb{Z}_p}$ を作成する.
13: $(\pi'_i\vec{s}, \pi'_i\sigma_i\vec{h}_i) := \mathrm{shuffle}([\![\vec{s}]\!]^{\mathbb{Z}_q}, [\![\sigma_i\vec{h}_i]\!]^{\mathbb{Z}_2^\ell}, \{\pi'_i\})$
14: $\sigma_{i+1}\vec{h}_i := (\pi'_i\vec{s})^{-1}\pi'_i\sigma_i\vec{h}_i$ where $\sigma_{i+1} := \vec{s}^{-1}\sigma_i$
15: **if** $i < N-1$ **then**
16: $([\![\sigma_{i+1}\vec{f}_{i+1}]\!]^{\mathbb{Z}_2^\ell}, [\![\sigma_{i+1}\vec{h}_{i+1}]\!]^{\mathbb{Z}_2^\ell}) :=$ $\sigma_{i+1}\vec{h}_i(\pi_{i+1}\vec{h}_i)^{-1}([\![\pi_{i+1}\vec{f}_{i+1}]\!]^{\mathbb{Z}_2^\ell}, [\![\pi_{i+1}\vec{h}_{i+1}]\!]^{\mathbb{Z}_2^\ell})$
17: **output** $\sigma_N\pi_N^{-1} := \sigma_N\vec{h}_{N-1}(\pi_N\vec{h}_{N-1})^{-1}, \{\pi_N\}$

Scheme 2 [旧 IHKC ソート: バリュー処理]
入力: $\sigma\pi^{-1}, \{\pi\}, [\![\vec{v}]\!]$
出力: $[\![\sigma\vec{v}]\!]$, ただし σ はソートを表す置換

1: $[\![\pi\vec{v}]\!] := \mathrm{shuffle}([\![\vec{v}]\!], \{\pi\})$
2: $[\![\sigma\vec{v}]\!] := (\sigma\pi^{-1})[\![\pi\vec{v}]\!]$

Scheme 3 [フラグ作成]
入力: $[\![k]\!]^{\mathbb{Z}_2^\ell}$
出力: $[\![f]\!]^{\mathbb{Z}_q^\ell}$, $f_k = 1, i \neq k$ で $f_i = 0$

1: **for each** $j < \ell$ **do** in parallel
2: mod 2 → mod q 変換 (Scheme 5) により, $[\![k_j]\!]^{\mathbb{Z}_2}$ を $[\![k_j]\!]^{\mathbb{Z}_q}$ に変換する. ただし $q > m$ とする.
3: **for each** $\eta < \lceil\log\ell\rceil$ **do** in series
4: **for each** $D \subseteq \mathbb{Z}_\ell$ such that $2^\eta + 1 \le |D| \le \min(2^{\eta+1}, \ell)$ **do** in parallel
5: キーの積 $[\![\prod_{j\in D} k_j]\!]^{\mathbb{Z}_q}$ を計算する. $(|D|+1)/2$ 個までの積が前段までに計算してあるため, 乗算 1 回である.
6: **for each** $j < 2^\ell$ **do** in parallel
7: キーの積の線形結合で, フラグ $[\![f_j]\!]^{\mathbb{Z}_q}$ を求める.

Scheme 4 [順位表作成 (オフライン処理)]
入力: $[\![\vec{f}]\!]^{\mathbb{Z}_q^{2^\ell}}$
出力: $[\![\vec{s}]\!]^{\mathbb{Z}_q}$, ただし s は $\vec{f}$ の変換前である $\vec{k}$ の各要素の大小比較における, 0 スタート昇順の順位を表す

1: $[\![S]\!]^{\mathbb{Z}_q} := [\![0]\!]^{\mathbb{Z}_q}$
2: **for each** $i < m$ **do** in series
3: **for each** $j < 2^\ell$ **do** in series
4: $[\![S]\!]^{\mathbb{Z}_q} := [\![S]\!]^{\mathbb{Z}_q} + [\![(f_i)_j]\!]^{\mathbb{Z}_q}$
5: $[\![s'_{i,j}]\!]^{\mathbb{Z}_q} := [\![S]\!]^{\mathbb{Z}_q}$
6: **for each** $i < m$ **do** in parallel
7: $[\![s_i]\!] = \sum_{j<l}[\![s'_{i,j}]\!]^{\mathbb{Z}_q}[\![(f_i)_j]\!]^{\mathbb{Z}_q} - 1$

Scheme 5 [プロトコル] mod 2 → mod q 変換
入力: $\{a\}^{\mathbb{Z}_2}$
出力: $[\![a]\!]^{\mathbb{Z}_q}$

1: 乱数 $\{r\}^{\mathbb{Z}_2}$ を生成する.
2: $r_{01} := \{r\}_{01}^{\mathbb{Z}_2}, r_{12} := \{r\}_{12}^{\mathbb{Z}_2}, r_{20} := \{r\}_{20}^{\mathbb{Z}_2}$ とおき,
それぞれを　mod q の複製秘密分散値と見なして $\{r_{01}\}^{\mathbb{Z}_q}, \{r_{12}\}^{\mathbb{Z}_q}, \{r_{20}\}^{\mathbb{Z}_q}$ とおく.
たとえば r_{01} は $(0, r_{01}), (r_{01}, 0), (0, 0)$ と見なせばよい.
3: $\{r\}^{\mathbb{Z}_q} := \{r_{01}\}^{\mathbb{Z}_q} \oplus \{r_{12}\}^{\mathbb{Z}_q} \oplus \{r_{20}\}^{\mathbb{Z}_q}$
ただし $\oplus$ は XOR であり, 等式 $a \oplus b = a + b - 2ab$ が利用できる.
4: 複製秘密分散 →Shamir 変換により $[\![r]\!]^{\mathbb{Z}_q}$ を計算する.
5: $a' := \mathrm{reveal}(\{a+r\}^{\mathbb{Z}_2})$
6: $[\![a]\!]^{\mathbb{Z}_q} := a' \oplus [\![r]\!]^{\mathbb{Z}_q}$. つまり,
7: $a \oplus r = 0$ なら $[\![a]\!]^{\mathbb{Z}_q} := [\![r]\!]^{\mathbb{Z}_q}$
8: $a \oplus r = 1$ なら $[\![a]\!]^{\mathbb{Z}_q} := 1 - [\![r]\!]^{\mathbb{Z}_q}$

Scheme 6 [(2, 2)-加法的ランダム置換]
入力のシェア所持パーティ: $\mathcal{P}_{\mathrm{in}} = \{P_0, P_1\}$
出力のシェア所持パーティ: $\mathcal{P}_{\mathrm{out}} = \{P_1, P_2\}$
入力: $[\vec{a}]^{\mathcal{P}_{\mathrm{in}}} \in [X]^{01}$
出力: $[\pi\vec{a}]^{12} \in [X]^{12}$

1: ラウンド 0 (事前処理可能部分)
2: 置換 $\{\pi\}$ を生成する.
3: P_0 と P_1 は $\vec{r}_{01}$ を, P_1 と P_2 は $\vec{r}_{12}$ を生成, 共有する.
4: ラウンド 1
5: P_0 は P_2 に $\vec{a_{P_0}} := \{\pi\}_{01}\langle\!\langle\vec{a}\rangle\!\rangle_{P_0}^{\mathcal{P}_{\mathrm{in}}} - \vec{r}_{01}$ を送信
6: P_1 は P_0 に $\vec{a_{P_1}} := \{\pi\}_{12}(\{\pi\}_{01}\langle\!\langle\vec{a}\rangle\!\rangle_{P_1}^{\mathcal{P}_{\mathrm{in}}} - \vec{r}_{12})$ を送信
7: P_0 は $\{\pi\}_{20}\vec{a_{P_1}}$ を出力
8: P_2 は $\{\pi\}_{20}(\{\pi\}_{12}\vec{a_{P_0}} + \vec{r}_{12})$ を出力

以下に割愛したものも含め最下位プロトコルの通信量 (各パーティの平均送信ビット数) をまとめておく.

1. 乗算/積和: $|q|$([4])
2. mod 2 to mod q 変換 (Scheme 5): $1 + |q|$
3. (2, 2)-加法的ランダム置換 (Scheme 6): $2/3|X|$
4. 公開値入力 (2, 2) 加法的ランダム置換: $1/3|X|$([6])
5. 公開値出力 (2, 2) 加法的ランダム置換: $4/3|X|$([6])
6. Shamir/複製秘密分散 → (2, 2)-加法的秘密分散 変換: 0([6])
7. (2, 2)-加法的秘密分散 →Shamir/複製秘密分散 変換: $2/3|X|$([6])

以下に参考までに最下位でないプロトコルの通信量を記す.

1. ℓ ビットソート (Scheme 3+Scheme 4): $\ell = 1$ で $1 + 2|q|$, $\ell = 2$ で $2 + 4|q|$, $\ell = 3$ で $3 + 8|q|$
2. Shamir/複製秘密分散上ランダム置換: $4/3|X|$
3. Shamir/複製秘密分散上公開値入力ランダム置換: $|X|$
4. Shamir/複製秘密分散上公開値出力ランダム置換: $4/3|X|$

3.3.0.1 通信量

通信路あたり平均通信量はデータ 1 件あたり, データ長 1 ビットあたり, 約 $4.8|q|$[bits] である.

3.3.0.2 ラウンド数

ラウンド数に関しては $3N + 11$ であり, 基数長 $\ell = 3$ とすれば $N = L/\ell \le 7$ であり, 32 ラウンドとなる.

3.4 旧 IHKC ソートの課題

旧 IKCH ソートの課題は, プロトコルを見て分かる通り, 非常に複雑な点である. これは改善前の HICT ソートも同様であった. プロトコルが複雑だということは, 理解が難しい

© 2017 Information Processing Society of Japan

だけでなく, 最適化のための解析が難しいということを意味する.

4. 提案 1: 理論的レベルの改善

ソートは置換の一種である. 本節は, 秘密計算における置換の理論を構成し, それに基づいて効率的でシンプルな基数ソートを構成する.

4.1 秘密計算における置換の理論

秘密計算において置換を体系的に扱われたことはなかった. 本稿ではより置換を自由に扱えるようにするため, 置換に関して体系的に整理する. まず置換自体や, 既存の秘密計算における置換をおさらいし, その上で置換の理論を展開する.

4.1.1 おさらい

置換とは

置換とは並び替えを表現する数学的構造である. たとえば置換 $(0,2,1)$ はベクトル $(10,5,2)$ を $(10,2,5)$ に並び替える. 置換は非可換群を為すことが知られている. すなわち, 単位元が存在し (恒等置換, 並びを変化させない置換), 逆元が存在し (置換した後に逆元で置換すると元に戻る), 結合律が成り立つ.

秘密計算における置換の基本

秘密計算におけるランダム置換は筆者らおよび Laur によって独立に提案され [11], [12], さらに筆者らが [6] で改良した (Scheme 6).

複製秘密分散は群で構成できるため, 置換の複製秘密分散を考えることができ, 例えば $(2,3)$ 複製秘密分散では, $\pi = \pi_{20}\pi_{12}\pi_{01}$ とするとき, 各分散値は (π_{20},π_{01}), (π_{01},π_{12}), (π_{12},π_{20}) となる. 上記ランダム置換はいずれもこの置換を適用するプロトコルと見なすことができる.

逆置換

ランダム置換プロトコルで, 置換 $\{\pi\}$ の各サブシェア $\{\pi\}^{01}, \{\pi\}^{12}, \{\pi\}^{20}$ の逆元 $(\{\pi\}^{01})^{-1}, (\{\pi\}^{12})^{-1}, (\{\pi\}^{20})^{-1}$ を逆順に適用するように実行すると, $\{\pi^{-1}\}$ で置換したのと同じ効果が得られる [13]. これは, 一度ランダム置換して, 元の並びに戻すなどの処理に使うことができる.

4.1.2 本論

まず, 本稿では置換の複製秘密分散を**ネイティブ置換**と呼ぶ. ネイティブ置換だけでは, ランダム置換しか表現できず, より豊富な置換を扱うことはできない. そのため, 本稿では**インデックス置換**, **ハイブリッド置換**の 2 つを導入する. 実はどちらも Scheme 1, Scheme 2 に既に出現している. しかし秘密計算プロトコルの構成者がこれらを自由に扱えるようになるためには, これらを明確に体系づける必要がある.

インデックス置換

長さ m の整数分散値のベクトルであって, $0\sim m-1$ の互いに異なる値を要素とするものをインデックス置換と呼ぶことにする. インデックス置換を置換として直接適用することはできないが, ネイティブ置換を適用することで, ネイティブ置換との置換の合成を行うことができる. たとえば $[\![\pi]\!]$ に $\{\rho\}$ を適用すると $[\![\rho\pi]\!]$ となり, 合成されたインデックス置換となる. 恒等置換 I のインデックス置換 $[\![I]\!]$ に合成を適用すれば, ネイティブ置換 $\{\pi\}$ をインデックス置換 $[\![\pi]\!]$ に変換することもできる.

ハイブリッド置換

ハイブリッド置換は Scheme 1, Scheme 2 で用いたような, ネイティブ置換と平文の置換の組である. $(\{\rho\},\rho^{-1}\pi)$ を $\{\{\pi\}\}$ と書き, ハイブリッド置換と呼ぶことにする. ハイブリッド置換はネイティブ置換と平文の置換の組であるから, 順番に適用することで適用および逆適用が可能である. また, 適用が可能なので, インデックス置換と置換の合成および, インデックス置換への変換を行うことができる.

ハイブリッド置換ではさらに, インデックス置換からの変換が可能である. $[\![\pi]\!]$ に $\{\rho^{-1}\}$ を適用すると $[\![\rho^{-1}\pi]\!]$ となり, 公開により $\rho^{-1}\pi$ を得る. $(\{\rho\},\rho^{-1}\pi$ はハイブリッド置換になっている.

インデックス置換がランダム置換以外も扱えるため, ハイブリッド置換もランダムでない置換を秘匿することができる. ランダムでない置換を秘匿性をもって適用するためにはハイブリッド置換が必要になる.

ハイブリッド置換はネイティブ置換への変換も可能である. $\{\rho\}$ を $\{\rho\}^{01,12,20}$ と書くとき, $\{\rho\}^{01,12,20}_{20}$ に $\rho^{-1}\pi$ を合成すればよい. $\rho^{-1}\pi$ が公開値なのでこの処理はオフラインである.

なおハイブリッド置換からネイティブ置換への変換がオフライン処理なので, 抽象的な粒度のプロトコルではハイブリッド置換をネイティブ置換と同一視して $\{\pi\}$ のように書くことにする.

4.1.2.1 置換操作の整理

以下に上記および簡単に構成可能な置換操作をまとめた. ネイティブ置換だけでなくインデックス置換とハイブリッド置換を経由することで, 置換, 逆置換, 変換, 合成の 4 操作を自由に行うことができる. 合成がインデックス置換, 合成後の置換/逆置換がハイブリッド置換を必要とすることに注意.

置換適用

1. ネイティブ置換: $\{\pi\}[\![x]\!] = [\![\pi x]\!]$ (ランダム置換プロトコル (Scheme 6 など))
2. インデックス置換: 直接は不可. ハイブリッド置換を経由
3. ハイブリッド置換: $(\{\rho\}\rho^{-1}\pi[\![x]\!])$

逆置換適用

1. ネイティブ置換: $\{\pi\}^{-1}[\![x]\!]$ (4.1.1 節)
2. インデックス置換: 直接は不可. ハイブリッド置換を経由
3. ハイブリッド置換: $(\rho^{-1}\pi)^{-1}\{\rho\}^{-1}[\![x]\!]$

変換

1. ネイティブ置換 → インデックス置換: $\{\pi\}[\![I]\!] = [\![\pi]\!]$
2. インデックス置換 → ネイティブ置換: ハイブリッド置換を経由
3. ネイティブ置換 → ハイブリッド置換: $(\{\pi\}, I)$
4. ハイブリッド置換 → ネイティブ置換: $\{\rho\}^{01,12,20}$ の $\{\rho\}^{01,12,20}_{20}$ に $\rho^{-1}\pi)$ を合成する
5. インデックス置換 → ハイブリッド置換: $(\{\rho\},\rho^{-1}\pi) = \{\{\pi\}\}$ (公開値出力ランダム置換プロトコル (Scheme **??**) など)
6. ハイブリッド置換 → インデックス置換: $\{\{\pi\}\}[\![I]\!] = [\![\pi]\!]$

合成

1. インデックス置換とネイティブ置換: $\{\rho\}[\![\pi]\!] = [\![\rho\pi]\!]$
2. インデックス置換とハイブリッド置換: $\{\{\rho\}\}[\![\pi]\!]$
3. その他の組み合わせはインデックス置換を経由

4.2 置換ベースの基数ソート

前節で構成した理論により, 置換の操作を自在に扱えるようになった. これを用いて, Scheme 3 と Scheme 4 の組み合わせのような ℓ ビットソートと置換のみからシンプルな基数ソートを構成することができる.

まず, Scheme 4 の出力の i 番目の値は, ℓ ビットソートとしての入力である Scheme 3 の i 番目の値を何番目に移動すればソートされるということを表現している. このような表現

© 2017 Information Processing Society of Japan

は, ソートを表す置換の逆置換に他ならない. (置換は i 番目の出力を入力の何番目“から”とればよいかを表すので, 何番目“へ”移動するかという表現とは逆である.)

これを踏まえて新基数ソートのキー処理を構成する. 基数ソートにおいて, ある $i-1$ 桁までのソートが済んでいるとき, 次の桁に注目してソートを行う. この処理に対応して, $i-1$ 桁までのソートを表す置換 σ_{i-1} をキーの i 桁目に適用し, その結果のソート σ'_i を得る. このソート σ'_i は, 先に σ_{i-1} を適用済みの状態からのソートなので, i 桁目“まで”のソート σ_i についてはその合成である $\sigma'_i\sigma_{i-1}$ となる. これをプロトコルとして書くと Scheme 7 となる. バリューの処理は Scheme 8 である.

Scheme 7 [プロトコル] 新基数ソート: キー処理
入力: 属性値 $\llbracket k_0 \rrbracket^{\mathbb{Z}_2^\ell}, ..., \llbracket k_{N-1} \rrbracket^{\mathbb{Z}_2^\ell} \in \llbracket \mathbb{Z}_2^\ell \rrbracket$, ただし $N\ell = L$
出力: ソートの逆置換を表すハイブリッド置換 $\{\sigma^{-1}\}$

1: $\{k_0\}$ の ℓ ビット安定ソートの逆置換 $\llbracket \sigma_0^{-1} \rrbracket$ を得る.
2: **for each** $1 \le i \le N$ **do**
3: $\llbracket \sigma_{i-1}^{-1} \rrbracket$ をハイブリッド置換に変換し $\{\sigma_{i-1}^{-1}\}$ を得る.
4: $\{\vec{k_i}\}$ を $\{\sigma_{i-1}^{-1}\}$ で逆置換し, $\{\sigma_{i-1}\vec{k_i}\}$ を得る.
5: $\llbracket \sigma_{i-1}\vec{k_i} \rrbracket$ の ℓ ビット安定ソートの逆置換 $\llbracket \sigma_i'^{-1} \rrbracket$ を得る.
6: $\llbracket \sigma_i'^{-1} \rrbracket$ に $\{\sigma_{i-1}^{-1}\}$ を合成し $\llbracket \sigma_i^{-1} \rrbracket := \llbracket \sigma_{i-1}^{-1}\sigma_i'^{-1} \rrbracket$ を得る.
7: $\llbracket \sigma_{N-1}^{-1} \rrbracket$ をハイブリッド置換に変換し $\{\sigma_{N-1}^{-1}\}$ を得て出力する.

Scheme 8 [新基数ソート: バリュー処理]
入力: $\{\sigma^{-1}\}, \llbracket \vec{v} \rrbracket$, ただし σ はソートを表す置換
出力: $\llbracket \sigma\vec{v} \rrbracket$

1: $\llbracket \vec{v} \rrbracket$ を $\{\sigma^{-1}\}$ で逆置換し, $\llbracket \sigma\vec{v} \rrbracket$ を出力する.

Scheme 1 と Scheme 7 を比較すると, アルゴリズムの行数だけ見てもかなりシンプルになっていることが分かる. また, 置換を理解している前提で言えば, 非常に直感的な表現になっており理解しやすいアルゴリズムになっている.

Scheme 2 と Scheme 8 の比較でも, ハイブリッド置換を定式化したことで直感的になっていることが分かる.

4.2.1 通信量

キー処理 (Scheme 7) に関して, 各操作の回数は以下である.

1. インデックス置換 → ハイブリッド置換変換 (公開値出力ランダム置換) N 回
2. ビットの分散値に対するハイブリッド置換による逆置換 (ランダム置換) $N-1$ 回
3. ℓ ビット安定ソート N 回
4. インデックス置換に対するハイブリッド置換の合成 $N-1$ 回 (ランダム置換)

3.3 節を参照しながら合計すると, 最良な基数となる $\ell = 2$ で, データ 1 件あたりキー処理のパーティあたり平均データ送信量は下記となる.

$$\frac{20N|q| - 4|q| + 10N - 4}{3} = \frac{2(5N-1)(2|q|-1) - 2}{3} [\text{bits}]$$

$\ell = 2$ だと $N = L/2$ なので, $N|q|$ の項のみ考慮すれば, キー 1 ビットとデータ 1 件あたり, 約 3.3$|q|$[bits] となる. 旧 IHKC ソートでは 4.8$|q|$ だったので, 約 30%の削減がされていることが分かる.

バリュー処理の通信量は Scheme 2 と Scheme **??**で等しい.

4.2.2 ラウンド数

ラウンド数は $8N-2$ であり, 例えば $L = 20$ では $N = L/2$ なので 78 ラウンドである. 旧 IHKC ソートの方が良いが, 大規模データや LAN 環境ではボトルネックにならない. 例えばネットワーク遅延 0.05ms 程度で 78 ラウンドだと, 通信遅延分のロスは 3.9ms にすぎない.

5. 提案 2: passive モデル向けのプロトコル最適化

前節では理論的かつ汎用的な改善を行った. 次に本節では, より実用フェーズに近いと考えられる passive モデルに特化した改善を行う. 例えば active モデルのプロトコルでは改ざんを検知するため分散値に常に冗長性を持たさなければならず, 冗長性の無い (2, 2)-秘密分散がそのままでは利用できないなどの難しさを持つ. passive モデルではそのような心配なく通信量に関して最適なプロトコルを構成することができる.

Scheme 9 に具体的なプロトコルを示す. Scheme 10 はその初段, Scheme 13 は二段目以降の処理である. Scheme 11, Scheme 12 は下位プロトコルである. Scheme 3, Scheme 4 に相当する処理は中に織り込まれている. ポイントは乗算および積和を行う時以外全て (2, 2)-加法的秘密分散で構成していることであり, そのおかげで特に下記 3 点の恩恵を受けている.

1. 積和が (2, 2) 出力積和 (Shamir 秘密分散の積和を (2, 2)-加法的秘密分散で出力するプロトコル) となり, 通信量が計 3 要素送信から, 計 1 要素送信に削減
2. ランダム置換時に複製秘密分散/Shamir 秘密分散と (2, 2)-加法的秘密分散との変換が不要となり, 通信量が計 2 要素送信 ×2 箇所削減
3. ハイブリッド置換における平文の置換が, 公開値ではなく準公開値 (k パーティで共有する平文) となり, インデックス置換からハイブリッド置換への変換が, (2, 2)-加法的秘密分散入力準公開値出力となり, 通信量が計 4 要素送信から 3 要素送信に削減

Scheme 9 [プロトコル] 新 passive 最適基数ソート: キー処理全体
入力: 属性値 $\{k_0\}^{\mathbb{Z}_2^\ell}, ..., \{k_{N-1}\}^{\mathbb{Z}_2^\ell} \in \{\mathbb{Z}_2^\ell\}$, ただし $N\ell = L$
出力: ソートの逆置換を表すハイブリッド置換 $\{\sigma^{-1}\}$

1: $\{k_0\}^{\mathbb{Z}_2^\ell}$ を入力として初段の処理 (Scheme 10) を行う.
2: 前段の出力および $\{k_i\}^{\mathbb{Z}_2^\ell}$ を入力として 2 段目以降の処理を N 段目まで行う.
3: 最終段の出力を出力する.

Scheme 10 [プロトコル] 新 passive 最適基数ソート: 初段
入力: $\{\vec{k}_0\}^{\mathbb{Z}_2}, \{\vec{k}_1\}^{\mathbb{Z}_2} \in \{\mathbb{Z}_2\}$
出力: ソートの逆置換を表すハイブリッド置換 $\{\sigma^{-1}\} = (\langle \pi\sigma^{-1} \rangle^{01}, \{\pi\}^{01,12})$

1: $\bmod 2 \rightarrow \bmod q$ 変換により $\{\vec{k}_0\}^{\mathbb{Z}_2}, \{\vec{k}_0\}^{\mathbb{Z}_2}$ を $\llbracket \vec{k}_0 \rrbracket^{\mathbb{Z}_q}, \llbracket \vec{k}_1 \rrbracket^{\mathbb{Z}_q}$ に変換する.
2: $\llbracket \vec{f}_3 \rrbracket^{\mathbb{Z}_q} := \llbracket \vec{k}_0 \rrbracket^{\mathbb{Z}_q} \llbracket \vec{k}_1 \rrbracket^{\mathbb{Z}_q}$ を計算する.
3: $\llbracket \vec{f}_2 \rrbracket^{\mathbb{Z}_q} := \llbracket \vec{k}_1 \rrbracket^{\mathbb{Z}_q} - \llbracket \vec{f}_3 \rrbracket^{\mathbb{Z}_q}$,
$\llbracket \vec{f}_1 \rrbracket^{\mathbb{Z}_q} := \llbracket \vec{k}_0 \rrbracket^{\mathbb{Z}_q} - \llbracket \vec{f}_3 \rrbracket^{\mathbb{Z}_q}$
$\llbracket \vec{f}_0 \rrbracket^{\mathbb{Z}_q} := 1 - \llbracket \vec{k}_0 \rrbracket^{\mathbb{Z}_q} - \llbracket \vec{k}_1 \rrbracket^{\mathbb{Z}_q} + \llbracket \vec{f}_3 \rrbracket^{\mathbb{Z}_q}$ を計算する.
4: 各 f_j に関して, 以下の prefix-sum を計算する.
$\llbracket (\vec{f'}_j)_u \rrbracket^{\mathbb{Z}_q} := \sum_{0 \le t < u} \llbracket (\vec{f}_j)_t \rrbracket^{\mathbb{Z}_q} + \llbracket s_j \rrbracket^{\mathbb{Z}_q}$,
ただし $s_0 := 0$ であり, $j > 0$ では $s_j := \sum_{0 \le u < m} (\vec{f}_{j-1})_u + s_{j-1}$
すなわち f_0, f_1, f_2, f_3 の順に, 自分以前の全要素の和をその要素の値とする.
5: (2, 2) 出力積和により, $[\sigma^{-1}]^{\mathbb{Z}_q,01} := \sum_{0 \le j < 4} [\vec{f'}_j \vec{f}_j]^{\mathbb{Z}_q,01}$ を計算する.
6: $\langle \pi\sigma^{-1} \rangle^{20} := \{\pi\}^{01,12} [\sigma^{-1}]^{\mathbb{Z}_q,01}$ を計算して出力する.

© 2017 Information Processing Society of Japan

Scheme 11 [プロトコル] $(2,2)$ 出力積和

入力: $[\![a_0]\!], \cdots, [\![a_{u-1}]\!], [\![[\![b_0]\!], \cdots, [\![b_{u-1}]\!]]\!]$

出力: $\sum_{0 \le i < u} [a_i b_i]^{01}$

1: ラウンド 0
2: パーティ P_0 と P_2 は乱数 r を共有しておく.
3: ラウンド 1
4: 各パーティ P は $c_P := \sum_{0 \le i < u} [\![a_i]\!]_P [\![b_i]\!]_P$ を計算する.
5: パーティ P_2 は $\lambda_2 c_2$ を $(2,2)$-加法的秘密分散で P_0 と P_1 に分散する. 具体的には P_1 に $c'_2 := \lambda_2 c_2 - r$ を送信する. 事前に共有した r が P_0 のシェアである.
ただし $\lambda_0, \lambda_1, \lambda_2$ は, P_0, P_1, P_2 に分散された (積和により次数の上がった)$(3,3)$-Shamir 秘密分散を復元するときの係数である.
6: パーティ P_0 は $\lambda_0 c_0 + r$, パーティ P_1 は $\lambda_1 c_1 + c'_2$ を出力する.

Scheme 12 [$(2,2)$-加法的準公開値出力ランダム置換]

入力のシェア所持パーティ: $\mathcal{P}_{\mathrm{in}} = \{P_0, P_1\}$

出力のシェア所持パーティ: $\mathcal{P}_{\mathrm{out}} = \{P_2, P_0\}$

入力: $[\vec{a}]^{01} \in [X]^{01}$

出力: $\langle \pi \vec{a} \rangle^{20}$

1: ラウンド 0 (事前処理可能部分)
2: 置換 $\{\pi\}^{01,12}$ を生成する.
3: P_0 と P_1 は $\vec{r}_{01}$ を共有する.
4: ラウンド 1
5: P_0 は P_2 に $a\vec{_{P_0}} := \{\pi\}_{01}[\vec{a}]^{01}_0 - \vec{r}_{01}$ を送信
6: P_1 は P_2 に $a\vec{_{P_1}} := \{\pi\}_{01}[\vec{a}]^{01}_1 + \vec{r}_{01}$ を送信
7: ラウンド 2
8: P_2 は $\pi\vec{a} = \{\pi\}_{20}\{\pi\}_{12}(a\vec{_{P_0}} + a\vec{_{P_1}})$ を計算し, P_0 に送信
9: P_0 と P_2 は $\pi\vec{a}$ を出力

Scheme 13 [プロトコル] 新 passive 最適基数ソート: 2 段目以降

入力: $\{\vec{k}_0\}^{\mathbb{Z}_2}, \{\vec{k}_1\}^{\mathbb{Z}_2} \in \{\mathbb{Z}_2\}$, 前段の出力である $\{\sigma_0^{-1}\} = (\langle \pi_0 \sigma_0^{-1} \rangle^{01}, \{\pi_0\}^{01,12})$

出力: ソートの逆置換を表すハイブリッド置換 $\{\sigma^{-1}\} = (\langle \pi \sigma^{-1} \rangle^{01}, \{\pi\}^{01,12})$

1: $\{\vec{k}_0\}^{\mathbb{Z}_2}, \{\vec{k}_1\}^{\mathbb{Z}_2}$ を $(2,2)$ 加法的秘密分散値 $[\vec{k}_0]^{\mathbb{Z}_2,01}, [\vec{k}_1]^{\mathbb{Z}_2,01}$ に変換する. (オフライン)
2: $[\vec{k}_0]^{\mathbb{Z}_2,01}, [\vec{k}_1]^{\mathbb{Z}_2,01}$ に $\{\sigma_0^{-1}\}$ を逆適用し, $[\vec{b}_0]^{\mathbb{Z}_2,20} := [\sigma_0 \vec{k}_0]^{\mathbb{Z}_2,20}, [\vec{b}_1]^{\mathbb{Z}_2,20} := [\sigma_0 \vec{k}_1]^{\mathbb{Z}_2,20}$ を得る. 出力が入力とパーティ集合の異なる $(2,2)$-加法的秘密分散となることに注意.
3: mod 2 → mod q 変換により $[\vec{b}_0]^{\mathbb{Z}_2,20}, [\vec{b}_1]^{\mathbb{Z}_2,20}$ を $[\![\vec{b}_0]\!]^{\mathbb{Z}_2}, [\![\vec{b}_1]\!]^{\mathbb{Z}_2}$ に変換する.
4: $[\![\vec{f}_3]\!]^{\mathbb{Z}_q} := [\![\vec{b}_0]\!]^{\mathbb{Z}_q} [\![\vec{b}_1]\!]^{\mathbb{Z}_q}$ を計算する.
5: $[\![\vec{f}_2]\!]^{\mathbb{Z}_q} := [\![\vec{k}_1]\!]^{\mathbb{Z}_q} - [\![\vec{f}_3]\!]^{\mathbb{Z}_q}$,
$[\![\vec{f}_1]\!]^{\mathbb{Z}_q} := [\![\vec{k}_0]\!]^{\mathbb{Z}_q} - [\![\vec{f}_3]\!]^{\mathbb{Z}_q}$
$[\![\vec{f}_0]\!]^{\mathbb{Z}_q} := 1 - [\![\vec{k}_0]\!]^{\mathbb{Z}_q} - [\![\vec{k}_1]\!]^{\mathbb{Z}_q} + [\![\vec{f}_3]\!]^{\mathbb{Z}_q}$ を計算する.
6: 各 f_j に関して, 以下の prefix-sum を計算する.
$[\![(\vec{f'}_j)_u]\!]^{\mathbb{Z}_q} := \sum_{0 \le t < u} [\![(\vec{f}_j)_t]\!]^{\mathbb{Z}_q} + [\![s_j]\!]^{\mathbb{Z}_q}$,
ただし $s_0 := 0$ であり, $j > 0$ では $s_j := \sum_{0 \le u < m} (\vec{f}_{j-1})_u + s_{j-1}$
すなわち f_0, f_1, f_2, f_3 の順に, 自分以前の全要素の和をその要素の値とする.
7: $(2,2)$ 出力積和により, $[\sigma'^{-1}]^{\mathbb{Z}_q,20} := \sum_{0 \le j < 4} [\vec{f'}_j \vec{f}_j]^{\mathbb{Z}_q,20}$ を計算する.
8: $[\sigma^{-1}]^{\mathbb{Z}_q,20}$ に $\{\sigma_0^{-1}\}$ を適用し, $[\sigma^{-1}]^{\mathbb{Z}_q,01} := [\sigma_0^{-1} \sigma'^{-1}]^{\mathbb{Z}_q,01}$ を得る.
9: $\langle \pi \sigma^{-1} \rangle^{20} := \{\pi\}^{01,12} [\sigma^{-1}]^{\mathbb{Z}_q,01}$ を計算して出力する.

5.0.1 通信量

本節による最適化で, 恩恵として上記した通信量を計算すると, 1 段あたり計 7 要素の $|q|$bit 要素の送信が削減されている. 1 段が $\ell = 2$ ビットなので, データ長 1 ビットあたりでパーティあたり平均では, $7/2/3 = 7/6|q|$[bits] の送信が削減されている. 前節の状態では $3.3|q|$ だったので, $2.16|q|$ となり, さらに 35%の削減である. 旧 IHKC ソートからでは, 55%の削減と, 半減以下に達する.

5.0.2 ラウンド数

ラウンド数は $5N - 2 = 2.5L - 2$ である. この程度のラウンド数がボトルネックにならないことは前節で述べた通りである.

6. プロトコル中の通信の最適化

前節で, passive モデル向けに基数ソートプロトコルを最適化した. 本節ではさらに, 主要なボトルネックである通信に関して, 通信路利用の最適化を行う.

前節までで, パーティあたり平均送信ビット数で通信量を評価してきたが, これは平均なので, これだけでは最大送信量のパーティがボトルネックとなる可能性を排除できない. 特に本稿で扱うような最適化されたプロトコルでは, 通信がパーティ間で非対称のプロトコルがほとんどである. そのため何も考えない構成では最大の性能が発揮されない.

Scheme 7 には通信のタイミングが固定の下位プロトコル (すなわちクリティカルパスに位置する処理) と, 比較的通信のタイミングに自由度のあるプロトコルがある. 固定タイミングの下位プロトコルの処理中に, 空いている通信路を用いることができれば, 処理が並列化されてパーティごとの通信負荷は平均化され, 結果として高速化される. 本節ではこのような, 通信のタスクスケジューリングを行う.

まず, mod 2 要素の通信は mod q 要素に比べて非常に小さいので最適化対象から省く. また, 初段も 1 回のみで影響が小さいため同様である. すると, 考慮すべきは以下の 5 演算となる.

1. step 3 の mod 2 to mod q
2. step 4 の乗算
3. step 7 の $(2,2)$ 出力積和
4. step 8 の $(2,2)$-加法的ランダム置換
5. step 9 の $(2,2)$-加法的準公開値出力ランダム置換

ここでのポイントは 2 点である.

1 点目は, タイミングである. 実行されるプロトコルのうち, mod 2 to mod q の mod q に係る通信は, 入力が乱数のため任意のタイミングで開始可能である.

2 点目は, 通信の方向の自由度である. あるパーティ P だけが持つ値 a を分散するとき, 複製秘密分散/加法的秘密分散では r と $a - r$ を, Shamir 秘密分散では r と $\lambda_0 a + \lambda_1 r$ を送信する. (ただし λ_0 は送信先パーティに対応する座標の関数値を, 座標 0 と送信先でないパーティ (3 パーティなので残りのパーティは一意である) に対応する座標の関数値から補完するときの係数である.) いずれも片方は乱数であり, 疑似乱数を許す場合通信は不要であり, かつ, どちらに乱数を送ってどちらに非乱数を送っても問題ない. よって, 通信の方向に自由度がある. このように自由度があるのは, mod 2 to mod q, 乗算, $(2,2)$ 出力積和の 3 つである.

結果として, 各ラウンドの通信方向は例えば表 1 のように最適化される. 1 ラウンド, 2 ラウンドは 2〜5 ラウンド目に注目すると, 固定部分と $(2,2)$ 出力積和を除くとちょうど 6 通信路, つまり乗算 (= mod 2 to mod q)2 回分空きがある. ここに 2 回分の mod 2 to mod q がぴったり当てはまる.

© 2017 Information Processing Society of Japan

表 1　各ラウンドの通信方向 (全二重向け最適化前)

ラウンド	P_0　P_1	P_1　P_2	P_2　P_0
1	⇒(1)	⇒(1)	⇒(1)
2	⇒(4)	⇒(2)	⇒(3)
3	⇒(0)	⇒(4)	⇒(0)
4	⇒(3)	⇒(0)	⇐(0)
5	⇐(3)	⇐(4)	⇒(0)

(0): 固定部分 (1): 乗算 (2): $(2, 2)$ 出力積和
(3): mod 2 to mod q ($\vec{b}_0$) (4): mod 2 to mod q ($\vec{b}_1$)

さらに, 一般的である全二重通信路においては, 送信と受信でそれぞれ別の帯域を持っている. すなわち, 10G 通信路であれば, 送信 10G + 受信 10G = 計 20G の帯域を持ち, 最大性能は送受信両方を同時に行ったときに発揮される. これに向けてさらに最適化すると, ラウンド 1 と 2 は全て方向に自由度があるプロトコルのため, データ数 $m/2$ ずつで別々の方向に送信することで最大帯域を利用できる (表 2).

表 2　各ラウンドの通信方向 (全二重向け最適化後)

ラウンド	P_0　P_1	P_1　P_2	P_2　P_0
1-0	⇒(1)	⇒(1)	⇒(1)
1-1	⇐(1)	⇐(1)	⇐(1)
2-0	⇒(4)	⇒(2)	⇒(3)
2-1	⇐(2)	⇐(3)	⇐(4)
3	⇒(0)	⇒(4)	⇒(0)
4	⇒(3)	⇒(0)	⇐(0)
5	⇐(3)	⇐(4)	⇒(0)

(0): 固定部分 (1): 乗算 (2): $(2, 2)$ 出力積和
(3): mod 2 to mod q ($\vec{b}_0$) (4): mod 2 to mod q ($\vec{b}_1$)

7. ローカル演算の高速化

10G ネットワーク環境では, ローカル演算がボトルネックとなる. 特に 1,000 万件の 32 ビット整数のランダム置換は下記のような特に無駄の無い実装でも 3Gbps 程度しか出ず, 10G ネットワークではボトルネックとなってしまう.

```
for(UINTA i = 0; i < length; ++i){
    dst[i] = src[perm[i]];
}
```

しかも厄介なことにこの演算はランダムアクセスなため SSE や AVX などの SIMD 演算による高速化も適用できない.

7.0.1　ランダム置換の高速化

そこで, ランダムアクセスを回避するアルゴリズムを構成する (Scheme 14).

まず d 個 (具体的には 16 個) のポインタ (つまりは仮想バッファ) を用意し, データを 16 箇所のバッファに, 各バッファの中ではシーケンシャルに振り分ける. 16 個程度の固定数のメモリ位置であればキャッシュに収まりきるため, ランダムアクセスにはならない. 再度同じ操作を行うと 256 箇所のバッファに振り分けられる.

各バッファのサイズがキャッシュサイズを下回るところで通常の置換を行えば, キャッシュはランダムアクセスでも高速なため, 低速なメインメモリのランダムアクセスを防ぐことができる.

注意点は, ランダムに振り分けるため各バッファのサイズは不定である点であり, 何回でキャッシュサイズを下回るかは厳密には確定しない. しかし, データ数が多いときはほぼ確実に期待値に非常に近いサイズとなるため, 実用的に問題にはならない.

Scheme 14 [アルゴリズム] 新ランダム置換 (ローカル)
入力: $\vec{a}$
出力: $\pi\vec{a}$

- d 未満の乱数を m 個生成し $\vec{r}$ とする.
- $\vec{r}$ のうち, 各 $i < d$ について, i が何個現れたか集計し c_i とする.
- $p_i := \sum_{j<i} c_j$ とする. ただし $p_i := 0$ とする.
- **for** $i = 0$ **to** $m - 1$
  - $b_{p_{r_i}} := a_i$ とする.
  - $p_{r_i} := p_{r_i} + 1$ とする.
- ここまでで, c_i 個ずつの d 箇所のバッファにランダムに振り分けられたので, 各 d 箇所のバッファに対して再帰的にランダム置換を行う. 本アルゴリズムを再帰的に繰り返し, 各バッファのサイズがキャッシュ以下になるところで通常のランダム置換を行う.

7.1　非ランダムな置換

Scheme 14 は任意に生成するランダム置換で置換する場合のアルゴリズムであり, 生成するランダム置換にやや特殊な形式を用いた. しかし, 旧 IHKC ソートやハイブリッド置換においては, 与えられた置換で置換する処理が含まれている. この場合, Scheme 14 のランダム置換と同じ形式に変換する必要がある. そのアルゴリズムを Scheme 15 に示す. やはり d 箇所に荒く振り分け, その後で各仮想バッファの中で通常の置換を行うような置換に変換している. このアルゴリズムでは k 番目のバッファのデータ個数は $q + (k < r?1 : 0)$ となる. Scheme 15 を再帰的に繰り返し, 各バッファサイズがキャッシュサイズ以下になるようにする. アルゴリズムは単純であるが正しく振り分けられることの証明は簡単ではない. 約半ページを要するため, 残念ながら割愛する.

Scheme 15 [アルゴリズム] 通常の逆置換から新ランダム置換で生成するランダム置換と同じ形式に変換
入力: $\vec{a}$
出力: 振り分け先を表す d 未満の値の列 $\vec{b}$, 振り分け先の中での置換先の列 $\vec{x}$

- $q := m/d$
- $r := m \mod d$
- q による割り算を乗算で実現するための, 擬逆数 R を計算する.
- **for** $i = 0$ **to** $m - 1$
  - a の q による商を R を用いて乗算で計算し, j' とおき, 余りを s とおく.
  - $b_i := k' - (s < \min(r, k')?1 : 0)$ とする.
  - $x_i := a_i - b_i q + \min(r, b_i)$ とする.

7.1.1　その他の演算の高速化

置換以外の演算は, 以下の方針で高速化する.

1. SSE, AVX による高速化
2. メモリアクセスを減らすため, 一度データを読み込んだら同じデータを使う演算を可能な限り一度に行う
3. 疑似乱数生成のスループットがメモリアクセスより高速なため, 疑似乱数をメインメモリに書き出さず利用時に生成する

8. 実験結果

本節では提案手法 (Scheme 9) の性能を検証する. 残念ながら Scheme 9 全体の実装が完了していないため, 実装が完了している下位処理の性能を元に, 並列化がされていないクリティカルパスにある各処理の処理時間を計測し和を計算することで全体性能を推定する. 下記の理由により, この方法で十分精密に推定が可能である.

© 2017 Information Processing Society of Japan

1. 表 2 から分かる通り, Scheme 9 の下位プロトコル間の並列性は高々 3 と, 現代の CPU コア数に比べれば低い
2. 通信の並列性は表 2 で解析済みのため予測可能

0 ソート全体の実測値は発表時に掲載する.
　秘密計算の測定環境は以下である.

1. CPU: Core^{TM} i7 6900K (3.2GHz (自動オーバークロックで最大 4.0GHz) x 8 core)
2. Memory: 32GB
3. Network: Intel® X550T 10Gbps x 2 port リング構成 (実測帯域 9,094 Mbps, 実測 ping 0.097 ms)
4. OS: CentOS 7.2.1511
5. コンパイラ: gcc 4.8.5

スクリプト言語の環境は以下である. これは筆者の開発環境であり, 実際的な環境の代表として選定した.

1. CPU: Core^{TM} i7 3635QM (2.4GHz x 4 core)
2. Memory: 16GB
3. OS: Windows 7 Professional
4. 言語: ruby 2.1.2

各下位処理の処理時間は表 3 の通りであった.

表 3　各下位処理の改善前後の処理時間 [ms]

	1 億件		1,000 万件	
処理	改善後	改善前	改善後	改善前
step 3	1,907	7,157	192	748
step 4	564	1,312	60	137
step 5〜7	749	2,322	88	225
置換変換	181	-	18	-
置換	190	757	17	43
逆置換	196	757	18	43

　全ての演算において 2 倍超の改善がされていることが分かる. また, 7.1 節の非ランダム置換に関して, 変換を含めても改善前の置換として向上していることも分かる. 特に 1 億件では倍速程度であり, キャッシュヒット率が低下する通常の実装と比較して安定した速度であることが分かる.

　この結果から, クリティカルパスとなる処理の和をとると, step 4〜7 + 置換変換 + 置換 3 回 + 逆置換 2 回 + データ送信 3 回である. (step 4, 5〜7 の測定値は通信を含んだ測定値である. step 3 は 6 節で論じた通り空いたリソースで処理するためクリティカルパスには入らない) そこから改善後の Scheme 9 の処理時間を推定し, 不確定要素を見込んで 1.1 を乗じた数値を表 4 に記載した. 比較として, 理論レベルでの改善のみ盛り込んだ実装 (Scheme 7), [6] の測定値, ruby 言語による測定値を記載した.

表 4　ソート処理の処理時間 [ms]

処理	全改善後 (推定)	Scheme 7	[6]	ruby
1,000 万件/10bits	3.03	5.94	-	1.70
1,000 万件/20bits	6.07	11.4	30.2	3.46
1,000 万件/30bits	9.10	16.7	-	3.82
1 億件/10bits	29.9	65.6	-	16.6
1 億件/20bits	59.7	125	-	30.6
1 億件/30bits	89.6	183	-	39.7

　届きはしなかったものの, データ長 20bit 以下で ruby の半分以上の速度に達していることが分かる.

9. おわりに

　本稿では [5] で提案され, [6] で改善された秘密計算上の基数ソートを, 実際に広く使われる環境であるスクリプト言語程度の速度を目指してさらに改良し高速化した. 例えば 1,000 万件/20bit データにおいて, 理論面からの改良で 11.4 秒, 最適化まで含めて 6.07 秒 (予測値) という, 届きはしなかったものの ruby 言語の 3.46 秒と比較しうる性能を示した. 発表時には最適化実装を全て完了させ, 実測値を発表する.

参考文献

[1] : sharemind News blog. `http://sharemind.cyber.ee/`.
[2] 濱田浩気，　菊池亮，五十嵐大，千田浩司：秘匿計算上の一括写像アルゴリズム，第 26 回人工知能学会全国大会 (2012).
[3] 濱田浩気，五十嵐大，千田浩司：秘匿計算上の一括写像アルゴリズム (2013).
[4] 五十嵐大，千田浩司，濱田浩気，高橋克巳：軽量検証可能 3 パーティ秘匿関数計算の効率化及びこれを用いたセキュアなデータベース処理，*SCIS2011* (2011).
[5] 濱田浩気，五十嵐大，千田浩司，高橋克巳：秘匿関数計算上の線形時間ソート，*SCIS2011* (2011).
[6] 五十嵐大，濱田浩気，　菊池亮，千田浩司：インターネット環境レスポンス 1 秒の統計処理を目指した, 秘密計算基数ソートの改良，*SCIS2014* (2014).
[7] Malkhi, D., Nisan, N., Pinkas, B. and Sella, Y.: Fairplay - Secure Two-Party Computation System, *USENIX Security Symposium*, USENIX, pp. 287–302 (2004).
[8] Wang, G., Luo, T., Goodrich, M. T., Du, W. and Zhu, Z.: Bureaucratic protocols for secure two-party sorting, selection, and permuting, *ASIACCS* (Feng, D., Basin, D. A. and Liu, P., eds.), ACM, pp. 226–237 (2010).
[9] Jónsson, K. V., Kreitz, G. and Uddin, M.: Secure Multi-Party Sorting and Applications, *IACR Cryptology ePrint Archive*, Vol. 2011, p. 122 (2011).
[10] Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K. and Takahashi, K.: Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms, *ICISC* (Kwon, T., Lee, M.-K. and Kwon, D., eds.), Lecture Notes in Computer Science, Vol. 7839, Springer, pp. 202–216 (2012).
[11] 濱田浩気，五十嵐大，千田浩司，高橋克巳：3 パーティ秘匿関数計算のランダム置換プロトコル，*CSS2010* (2010).
[12] Laur, S., Willemson, J. and Zhang, B.: Round-Efficient Oblivious Database Manipulation, *ISC* (Lai, X., Zhou, J. and Li, H., eds.), Lecture Notes in Computer Science, Vol. 7001, Springer, pp. 262–277 (2011).
[13] 桐淵直人，五十嵐大，諸橋玄武，濱田浩気：属性情報と履歴情報の秘匿統合分析に向けた秘密計算による高速な等結合アルゴリズムとその実装，*CSS2016* (2016).
[14] 五十嵐大，　菊池亮，濱田浩気，千田浩司：複数体上 Active モデルで秘匿性・正当性を保証する秘密分散ベース秘密計算と高速秘密計算ソートへの応用，*SCIS2015* (2015).
[15] 濱田浩気，五十嵐大，　菊池亮，千田浩司，諸橋玄武，冨士仁，高橋克巳：実用的な速度で統計分析が可能な秘密計算システム MEVAL，*CSS2013* (2013).
[16] 五十嵐大，濱田浩気，　菊池亮，千田浩司：少パーティの秘密分散ベース秘密計算のための $O(\ell)$ ビット通信ビット分解および $O(p')$ ビット通信 Modulus 変換法，*CSS2013* (2013).
[17] Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J. B. and Toft, T.: Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation, *TCC* (Halevi, S. and Rabin, T., eds.), Lecture Notes in Computer Science, Vol. 3876, Springer, pp. 285–304 (2006).
[18] 五十嵐大，　菊池亮，濱田浩気，千田浩司：少パーティ数の秘密分散ベース秘密計算における効率的な malicious モデル上 SIMD 計算の構成法，*CSS2013* (2013).

© 2017 Information Processing Society of Japan