スーパーコンピュータ「富岳」と「発見するAI」で、がんの薬剤耐性に関わる未知の因果メカニズムを高速に発見する新技術を開発

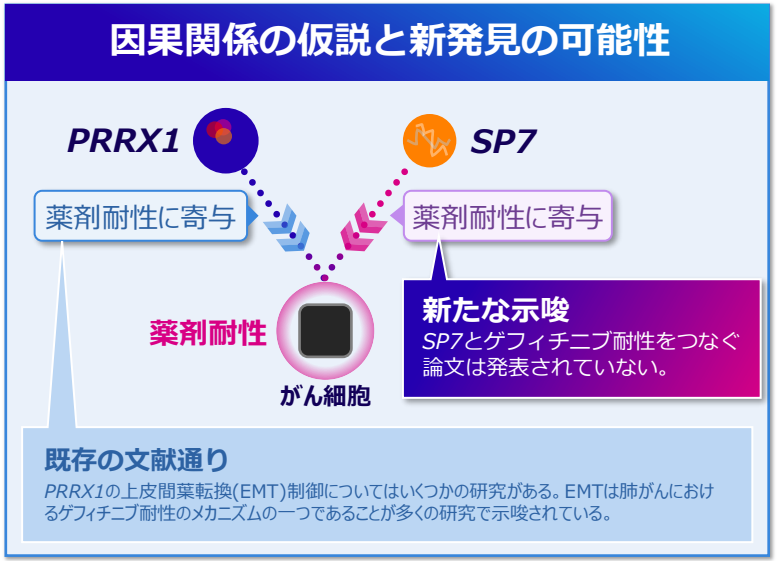
富士通株式会社　富士通研究所

コンピューティング研究所　CWB CPJ

栗原 康志

本発表の概要

- 背景、課題
  - 「発見するAI」を薬剤耐性の原因究明に適用したい
  - 大規模データを扱う必要があるため、実用時間内で処理が完了しない
- やったこと
  - 「発見するAI」の処理を並列化
  - 「富岳」の利用
- 成果
  - 実用時間内に処理が完了
  - 耐性獲得メカニズムに関して、専門家が認める新仮説を抽出



目次

- 「発見するAI」の概要
- 「発見するAI」の課題
- 「発見するAI」の大規模変数対応
  - 条件探索と因果探索の高速化
- 「発見するAI」のゲノム医療への応用結果
- まとめと参考情報

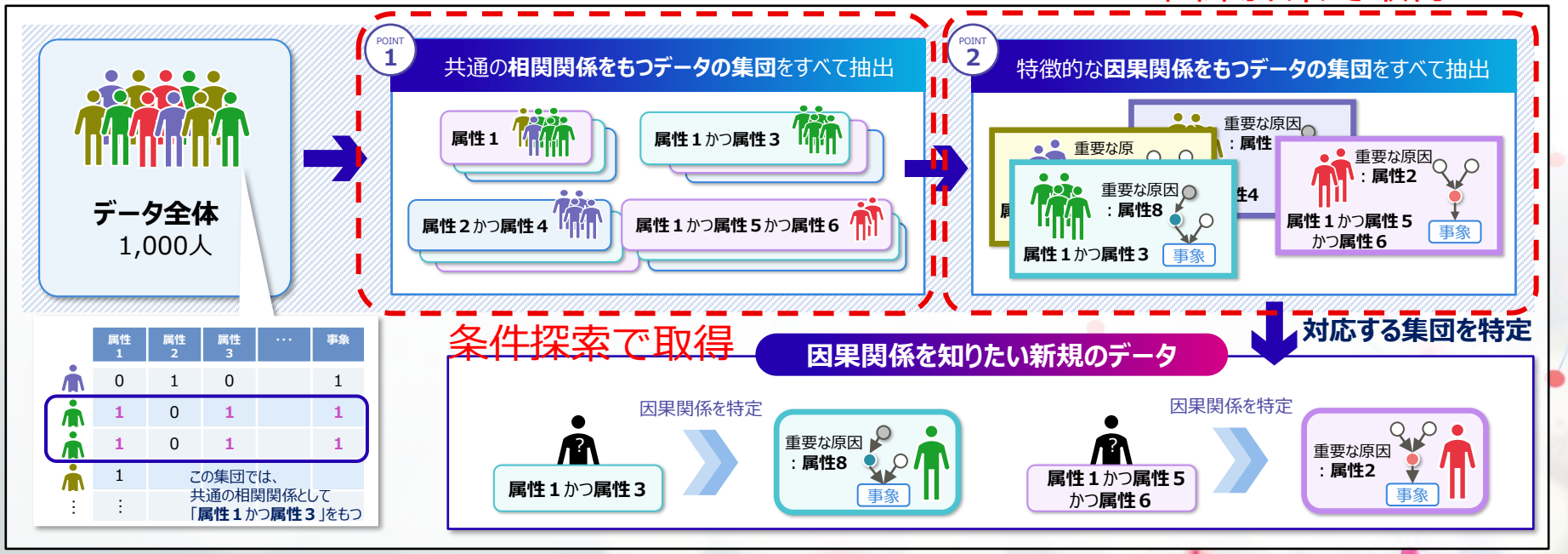
「発見するAI」の概要

富士通の「発見するAI」

FUJITSU

データ全体に共通する因果関係だけでなく、

個々のデータに特徴的な因果関係を網羅的に抽出可能に。



※2020年12月17日プレスリリース 「ヒトやモノなどのデータの一つひとつが持つ特徴的な因果関係を発見する技術を開発」
https://pr.fujitsu.com/jp/news/2020/12/17.html

Phase1:条件探索

	属性					事象
	遺伝子1発現	遺伝子2発現	…		遺伝子50発現	薬剤耐性発現
	1				1	1
	1				1	1
	1				1	1
		1				1

薬剤耐性発源と相関する属性の抽出結果

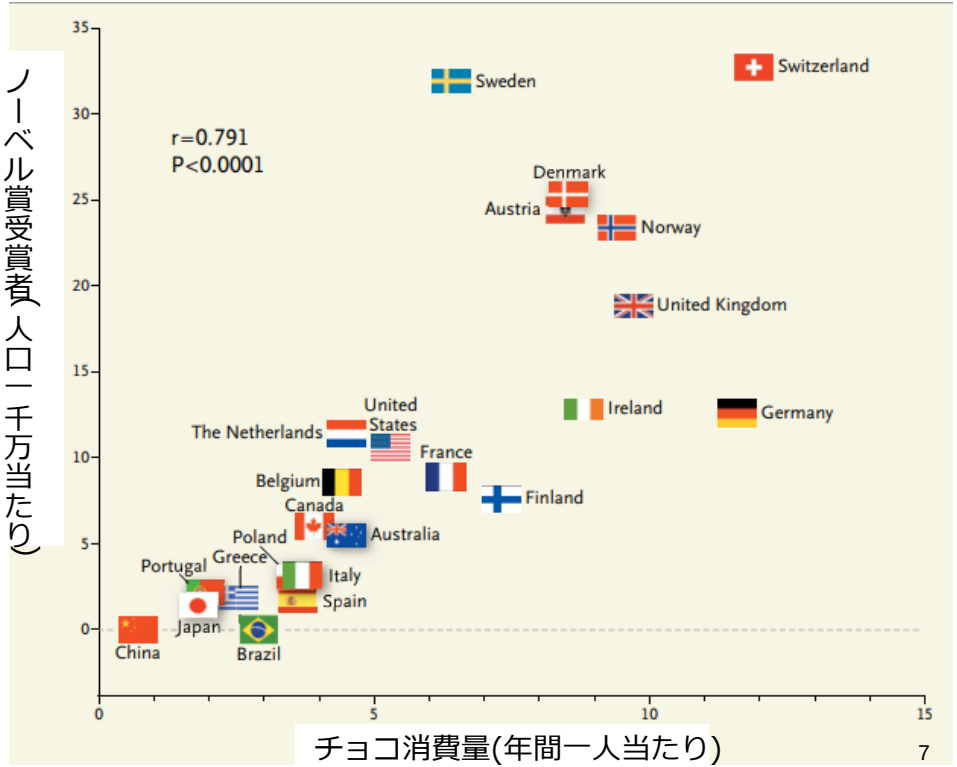
遺伝子1発現&
遺伝子50発現

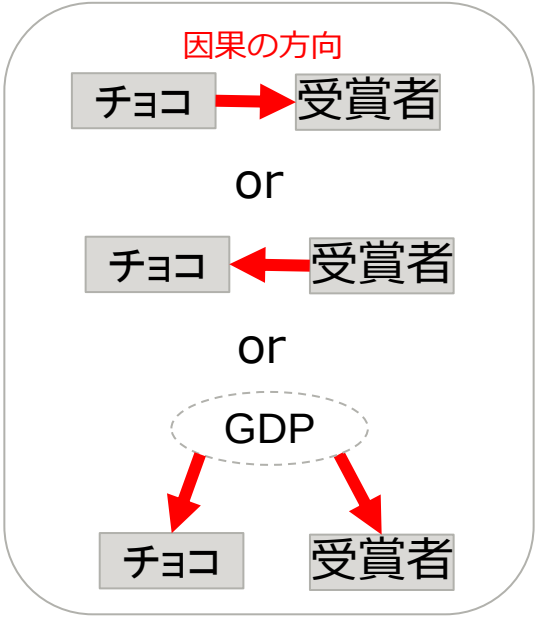
遺伝子2発現

- 薬剤耐性を発現し、属性間に共通の相関関係を持つデータ集団を抽出
  - 条件を満たすデータの集団を抽出
    - 例）遺伝子1発現＆遺伝子50発現、遺伝子2発現
  - 抽出したデータ集団のサンプル数が一定数以上ある場合、全属性間の相関関係を調査し、相関関係があれば条件を出力

Phase2:因果探索

- 因果関係(因果グラフ)を見つけ出す





参考: 清水昌平，「機械学習プロフェッショナルシリーズ 統計的因果探索」，講談社
図引用: Chocolate Consumption, Cognitive Function, and Nobel Laureates, Franz H. Messerli, M.D., New England Journal of Machine (367), 2012

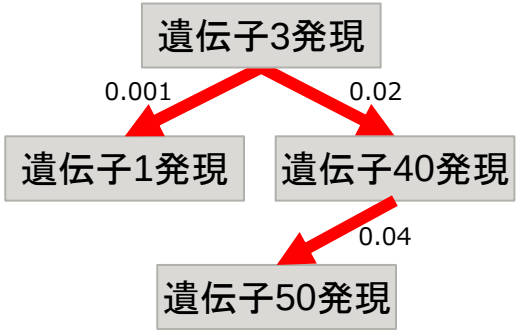
Phase2:因果探索

- 観測データがLiNGAMモデル(※)に基づくと仮定すると、変数間の因果関係は数値解析で計算可能
- Phase2の因果探索ではLiNGAMモデルを仮定して因果グラフを求める

遺伝子1発現&遺伝子50発現のデータ集団

	属性					事象
	遺伝子1発現	遺伝子2発現	…		遺伝子50発現	薬剤耐性発現
	1				1	1
	1				1	1
	1				1	1
⋮	1				1	1

因果グラフ



※：LiNGAM: Linear Non-Gaussian Acyclic Model
以下の過程が成り立つときに，因果グラフを一意に推定可能なモデル
観測した確率変数サンプルの独立成分が非ガウス分布に従うこと
変数間の因果関係が線形・非巡回であること

FUJITSU

「発見するAI」の課題と大規模変数対応

「発見するAI」のゲノム医療への応用時の課題

FUJITSU

- データの属性数やサンプル数が多くなると、条件探索や因果探索の処理が実用時間内で完了しない
  - 属性数をd_x 、条件数をx　とすると
  - 条件探索の計算量のオーダーは$O(xd_x^2)$
  - 因果探索の計算量のオーダーは$O(d_x^3)$
  - 今回のゲノム医療への応用では2万の属性数(d_x)を持ったデータ
  - 1台のIntel計算機で4,000年以上かかる試算

- 並列化に向くように実装を変更
  - もともとはpythonで逐次処理実装だった
- 処理チューニング&HPCを使った並列化で解決

利用した高速化手法のまとめ

FUJITSU

高速化手法	条件探索	因果探索
SIMD化	✓	✓
命令レベル並列化	✓	
マルチスレッド化	✓	
マルチプロセス化	✓	✓
CythonによるC++実装	✓	✓

- SIMD化、命令レベル並列化
  - Intrinsic関数によるアセンブラレベルで実装
- マルチスレッド化、マルチプロセス化
  - 依存関係がない処理を並列化
- CythonによるC++実装
  - 上記最適化を実施するため

条件探索の処理

FUJITSU

入力

調査対象のデータ

サンプルid	年齢	体重	身長	・・・
1	21	69	170	
2	13	48	155	
3	52	52	169	
4	30	64	174	
・・・				

調べる条件(データの抽出条件)
- 身長170cm以上
- 年齢20歳未満
- 体重50kg以上
- 身長160cm以下
- 体重60以下

、、、

条件探索処理

1. 条件に合致するデータを抽出し、一定数あるか調査
2. 一定数ある場合、全属性間の相関関係を調査し、相関関係があれば条件を出力
3. 全条件について1～2を実施

出力

属性間に相関関係を持つデータの集団の抽出する条件

身長170cm以上

サンプルid	年齢	体重	身長	・・・
1	21	69	170	
4	30	64	174	
・・・				

身長160cm以下かつ体重60以下
(条件2つの組み合わせ)

サンプルid	年齢	体重	身長	・・・
2	13	48	155	
3	52	52	169	
・・・				

・
・
・

並列化後の条件探索の疑似コード

FUJITSU

例）組み合わせる条件の数が2の場合

```
for(k1=mpi_rank; k1<d_x; k1+=mpi_size) {
#pragma omp parallel for schedule(dynamic)
  for(k2=k1+1; k2<d_x; k2+=stream数) {
    count = bit_and_count結果(SIMD処理);
    if(count >= s_min) 相関係数計算(SIMD処理);
  }
}
```

青字が利用した高速化

1. 条件に合致するデータを抽出し、一定数あるか調査
2. 一定数ある場合、全属性間の相関関係を調査し、相関関係があればデータを出力
3. 全条件について1～2を実施

- SIMD命令を使用し、複数サンプルを同時に処理できるように実装
- MPI並列とスレッド並列、さらにストリーム処理を使用して複数の条件を並列処理

処理のSIMD化：SUM計算の例

FUJITSU

- sum計算(float)の従来技術の疑似コードとSIMD処理化後の疑似コードは以下のようになる

例）属性ペアの相関係数算出処理のsum計算(float)

疑似コード(従来)

```
for (size_t m = 0; m < dy; m++) {
  for (size_t e = 0; e < n; e++) {
    if (th_cand[e] == 1) {
      y_sum += y1[e];
      count ++;
    }
  }
}
```

疑似コード(例：SIMD幅＝512におけるACLE(※)によるSIMD化)

```
for (size_t m = 0; m < dy; m++) {
  for (size_t e = 0; e < n; e=e+16) {
    svuint32_t mask0 = svld1ub_vnum_u32(all, th_cand, 0);
    const svbool_t pred0 = svcmpgt_wide_n_u32(all, mask0, 0);
    const svfloat32_t y_val = svld1_vnum_f32(all, &y[e], 0);
    y_sum += svaddv_f32(pred0, y_val);
  }
}
```

処理対象のデータがfloat型、SIMD幅が512bitの場合、演算回数を1/16に削減

※：Arm C Language Extensions

ストリーム処理：SUM計算の例

FUJITSU

- ストリーム処理なし

```
for (size_t m = 0; m < dy; m++) {
  for (size_t e = 0; e < n; e=e+16) {
    svuint32_t mask0 = svld1ub_vnum_u32(all, th_cand+e, 0);
    const svbool_t pred0 = svcmpgt_wide_n_u32(all, mask0, 0);
    const svfloat32_t y_val = svld1_vnum_f32(all, &y[e], 0);
    y_sum += svaddv_f32 (pred0, y_val);
  }
}
```

1条件分のプレディケートレジスタの設定

プレディケートレジスタを利用して要素加算

- ストリーム処理あり

(例：ストリーム数＝4)

```
for (size_t m = 0; m < dy; m++) {
  for (size_t e = 0; e < n; e=e+16) {
    svuint32_t mask0 = svld1ub_vnum_u32(all, th_cand0+e, 0);
    svuint32_t mask1 = svld1ub_vnum_u32(all, th_cand1+e, 0);
    svuint32_t mask2 = svld1ub_vnum_u32(all, th_cand2+e, 0);
    svuint32_t mask3 = svld1ub_vnum_u32(all, th_cand3+e, 0);
    const svbool_t pred0 = svcmpgt_wide_n_u32(all, mask0, 0);
    const svbool_t pred1 = svcmpgt_wide_n_u32(all, mask1, 0);
    const svbool_t pred2 = svcmpgt_wide_n_u32(all, mask2, 0);
    const svbool_t pred3 = svcmpgt_wide_n_u32(all, mask3, 0);
    const svfloat32_t y_val = svld1_vnum_f32(all, &y[e], 0);
    y_sum0 += svaddv_f32 (pred0, y_val);
    y_sum1 += svaddv_f32 (pred1, y_val);
    y_sum2 += svaddv_f32 (pred2, y_val);
    y_sum3 += svaddv_f32 (pred3, y_val);
  }
}
```

4条件分のプレディケートレジスタの設定

ロードした1つのデータに対して、4つのプレディケートレジスタを使用して、同時に4条件の場合の値を計算

ストリーム処理：SUM計算の例

FUJITSU

- ストリーム処理なし

```
for (size_t m = 0; m < dy; m++) {
  for (size_t e = 0; e < n; e=e+16) {
    svuint32_t mask0 = svld1ub_vnum_u32(all, th_cand+e, 0);
    const svbool_t pred0 = svcmpgt_wide_n_u32(all, mask0, 0);
    const svfloat32_t y_val = svld1_vnum_f32(all, &y[e], 0);
    y_sum += svaddv_f32 (pred0, y_val);
  }
}
```

1条件分のプレディケートレジスタの設定

プレディケートレジスタを利用して要素加算

- 青字がメモリアクセス
- 1回のメモリアクセスでストリーム数個(ここでは4)のsum計算を実行

メモリアクセス回数が1/ストリーム数に削減

- ストリーム処理あり
  (ストリーム数＝4)

```
for (size_t m = 0; m < dy; m++) {
  for (size_t e = 0; e < n; e=e+16) {
    svuint32_t mask0 = svld1ub_vnum_u32(all, th_cand0+e, 0);
    svuint32_t mask1 = svld1ub_vnum_u32(all, th_cand1+e, 0);
    svuint32_t mask2 = svld1ub_vnum_u32(all, th_cand2+e, 0);
    svuint32_t mask3 = svld1ub_vnum_u32(all, th_cand3+e, 0);
    const svfloat32_t y_val = svld1_vnum_f32(all, &y[e], 0);
    const svbool_t pred0 = svcmpgt_wide_n_u32(all, mask0, 0);
    const svbool_t pred1 = svcmpgt_wide_n_u32(all, mask1, 0);
    const svbool_t pred2 = svcmpgt_wide_n_u32(all, mask2, 0);
    const svbool_t pred3 = svcmpgt_wide_n_u32(all, mask3, 0);

    y_sum0 += svaddv_f32 (pred0, y_val);
    y_sum1 += svaddv_f32 (pred1, y_val);
    y_sum2 += svaddv_f32 (pred2, y_val);
    y_sum3 += svaddv_f32 (pred3, y_val);
  }
}
```

4条件分のプレディケートレジスタの設定

ロードした1つのデータに対して、4つのプレディケートレジスタを使用して、同時に4条件の場合の値を計算

評価環境・入力データ

FUJITSU

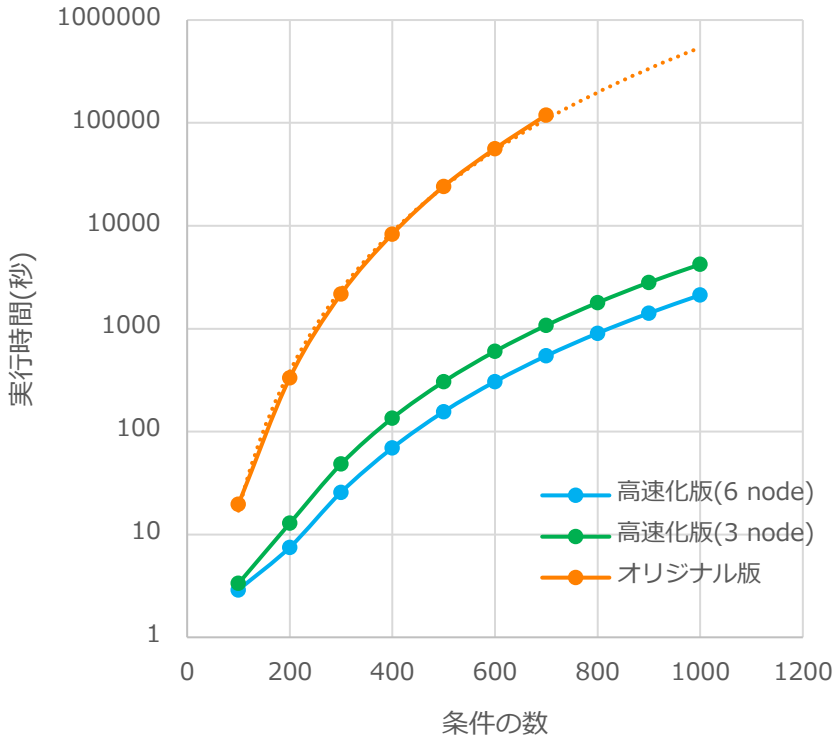
	オリジナル版	高速化版
システム	Xeonサーバー	FX700クラスタ
CPU	Intel(R) Xeon(R) Gold 6148 CPU	A64FX
動作周波数[GHz]	2.4（ターボブースト時3.7)	2.0
SIMDレジスタサイズ[bits]	512	512
物理コア数/ノード	40	48
使用ノード数	1	3 or 6
プロセス数	1	3 or 6
スレッド数/プロセス	1	48

- 入力データ
  - 入力データのサンプル数：800　→　調査対象のデータのサンプル数に対応
  - 条件を組み合わせる数(k)：3　→　調べたい条件(データの集団)の数に影響するパラメータ
- 条件の数と属性の数を変化させて実行時間を測定
  - 条件の数と属性の数は同じとした

評価結果: 対象変数の数 vs 計算時間

FUJITSU

- オリジナル版と高速化版の計算時間をプロット
  - 横軸は条件の数
    - 変数の数も同じ
  - プロセス数はノード数と同じとした
    - 1プロセス48スレッド
- 既存実装の Python と比べると200倍以上の高速化
- ノード数に応じて性能向上
  - 6ノード使用時の実行時間は3ノード使用時の約半分



因果探索の概要

FUJITSU

入力(条件探索の出力)

属性間に相関関係を
持つデータの集団

身長170cm以上

サンプルid	年齢	体重	身長	. . .
1	21	69	170	
4	30	64	174	
. . .				

身長160cm以下かつ体重60以下

サンプルid	年齢	体重	身長	. . .
2	13	48	155	
3	52	52	169	
. . .				

・
・
・

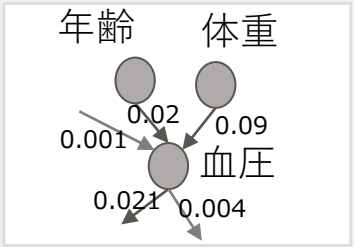
因果探索
処理

1. 因果順の決定
   1-1 相互情報量を基に因果の先頭を決定
   1-2 先頭の寄与を取り除き、新たな入力を作成
   1-3 全ての変数の因果順が決まるまで
   1-1, 1-2を繰り返す
2. 推定した因果順の先頭から回帰分析により、
   因果の強さを決定

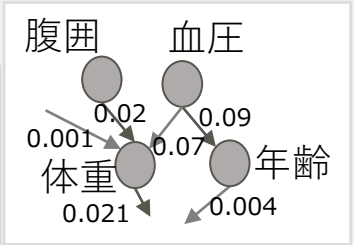
出力

各データの集団の
因果グラフ

身長170cm以上



身長160cm以下かつ体重60以下



・
・
・

並列化後の因果探索の疑似コード

FUJITSU

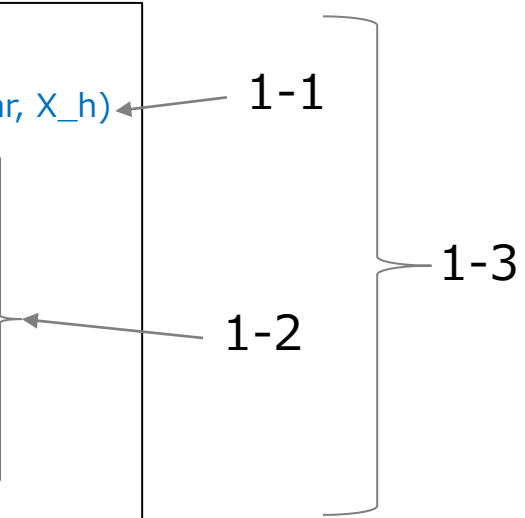
1. 因果順の決定
   1-1 相互情報量を基に因果の先頭を決定
   1-2 先頭の寄与を取り除き、新たな入力を作成
   1-3 全ての変数の因果順が決まるまで
   1-1, 1-2を繰り返す

因果順の決定処理

```
for _ in _range:
    初期計算
    m = self._search_causal_order_fast_flat_mpi(U, X_std, X_std_var, X_h)

    X_m = np.ascontiguousarray(X_[:, m]
    for i in U:
        X_i = np.ascontiguousarray(X_[:, i])
        if i != m:
            X_[:, i] = self._residual_fast(X_i, X_m, np.var(X_[:, m]))

    K.append(m)
    U = U[U != m]

self._causal_order = K
```

C++実装(SIMD化)

C++実装(SIMD化)



SIMD化

FUJITSU

- 例)_residual関数でcallされる_entropy関数をSIMD処理化

もともとの_entropy関数

```
def _entropy(self,u)
# uはfp32の1次元配列
 return (1 + np.log(2 * np.pi)) / 2 - ¥
          k1 * (np.mean(np.log(np.cosh(u))) - gamma)**2 - ¥
          k2 * (np.mean(u * np.exp((-u**2) / 2)))**2
```

単純なC実装(疑似コード)

```
double entropy(double* array, int N) { // Nは配列の要素数
   double retVal;
   double temp1 = 0.0;
   double temp2 = 0.0;
   for (int i = 0; i < N; i++) {
     temp1 += log(cosh(array[i]));
     temp2 += array[i] * exp(-1 * pow(array[i], (double)2) / 2.0);
   }

    retVal = (1.0 + log(2.0 * M_PI)) / 2.0 - k1 * pow(((temp1 / size) - gamma), 2.0) - k2 * pow((temp2 / size), 2.0);

    return retVal;
}
```

SIMD関数を使用して実装
⇒ループの回数を削減
(SIMD幅が512ビットの場合)

並列化後の因果探索の疑似コード

FUJITSU

- 1-1の処理をMPIで並列処理

もともとの_search_causal_order関数(1-1を処理する関数)

```
def _search_causal_order(self, X, U):
    """Search the causal ordering."""
    Uc, Vj = self._search_candidate(U)
    if len(Uc) == 1:
        return Uc[0]

    M_list = []                                   並列処理の対象
    for i in Uc:
        M = 0
        for j in U:
            if i != j:
                xi_std = (X[:, i] - np.mean(X[:, i])) / np.std(X[:, i])
                xj_std = (X[:, j] - np.mean(X[:, j])) / np.std(X[:, j])
                ri_j = xi_std if i in Vj and j in Uc else self._residual(
                    xi_std, xj_std)
                rj_i = xj_std if j in Vj and i in Uc else self._residual(
                    xj_std, xi_std)
                M += np.min([0, self._diff_mutual_info(xi_std,
                                                       xj_std, ri_j, rj_i)])**2
        M_list.append(-1.0 * M)
    return Uc[np.argmax(M_list)]
```

対応

対応

並列化後の_search_causal_order関数

```
def _search_causal_order_fast_flat_mpi(self, U, X_std, X_std_var, X_h):
    """Search the causal ordering by flat mpi."""
    Uc, Vj = self._search_candidate(U)
    if len(Uc) == 1:
        return Uc[0]

    # For MPI
    if self._mpi:
        from mpi4py import MPI
        comm = MPI.COMM_WORLD
        _mpi_rank = comm.Get_rank()
        _mpi_size = comm.Get_size()
        Uc = np.array_split(Uc, _mpi_size)[_mpi_rank]

    if len(Vj) == 0: # Change Python list to Numpy array
        Vj = np.empty(0, np.int64)

    M_value = np.full(1, -1 * np.inf)
    M_arg = np.full(1, -1)
    self._jit.thread_in_c(Vj, U, X_std, X_std_var, X_h, Uc, M_value, M_arg)

    if self._mpi:
        M_max_local = (-1 * np.inf, -1)
        if len(Uc) > 0:
            M_max_local = (M_value[0], M_arg[0])
        return comm.allreduce(M_max_local, MPI.MAXLOC)[1]
    else:
        return Uc[np.argmax(M_list)]
```

Ucをプロセス数で分割

ループ内の処理を行うC++関数
(各MPIプロセスで実行. SIMD化済)

各ノードのCPUコアにMPIプロセスを
割り当てることによる並列処理

評価環境

FUJITSU

	オリジナル版	高速化版
システム	Xeonサーバー	富岳
CPU	Intel Xeon Gold 5218 2.30GHz	A64FX
動作周波数[GHz]	2.3（ターボブースト時3.9)	2.2
SIMDレジスタサイズ[bits]	512	512
物理コア数/ノード	32	48
ノード数	1	98
プロセス数	1	4608 (98 x 48)
スレッド数/プロセス	1	1

- 変数の数を変化させて実行時間を測定
  - サンプル数は200に固定

評価結果: 対象変数の数 vs 計算時間

FUJITSU

- 富岳 96 ノード利用時の計算時間をプロット
  - 横軸は変数の数 d_x
- 2 万変数を対象とした場合で **17.7 時間**で計算を完了できている
  - 既存実装の Python と比べると約 17,000 倍以上の高速化

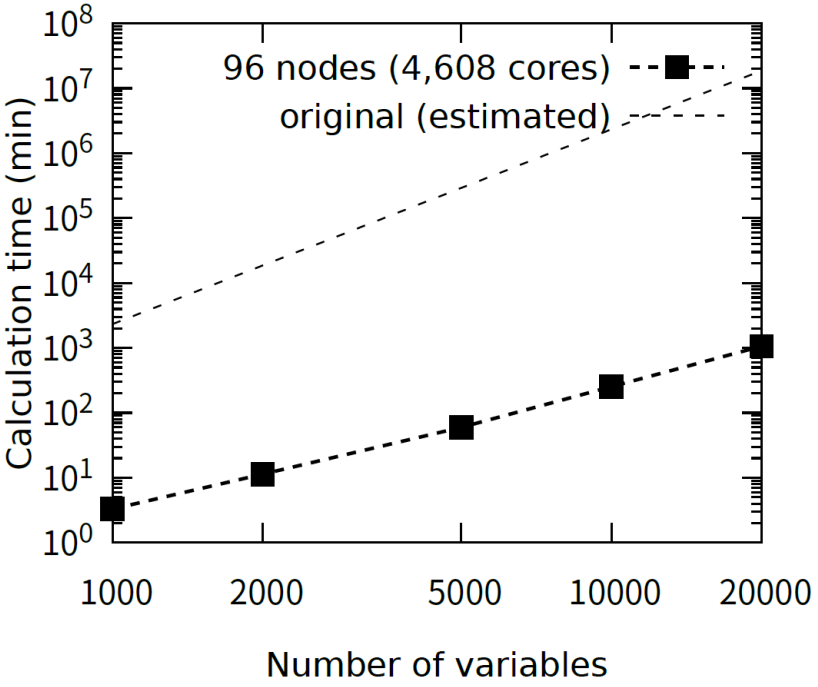


図 7: d_x に対する計算時間 (n=200)

FUJITSU

「発見するAI」のゲノム医療への応用結果

ゲノム医療への応用時のパラメータ

FUJITSU

- 入力データサイズ
  - サンプル数：276
  - 変数の数：19145
- 条件探索
  - 探索条件数：2,775,963,449
  - 変数の数：19,145
- 因果探索
  - データ集団数：112,909
- 実行環境
  - 富岳

評価結果：「発見するAI」のゲノム医療への応用

FUJITSU

事例

ゲノム医療への応用

薬剤耐性の原因究明

がんへの薬剤応答データから、
薬剤耐性メカニズムに関わる遺伝子関係を抽出。

- **ヒトの全遺伝子（20000種類）**に対して
  重要な因果関係を網羅的に抽出

  既存実装では4000年かかっていた ➔ **１日で完了**

- 肺がんの治療薬として知られる「ゲフィチニブ」の
  耐性獲得メカニズムに関して、
  専門家が認める新仮説の抽出に成功

共同研究：東京医科歯科大）宮野先生

スーパーコンピュータ「富岳」成果創出加速プログラム
「大規模データ解析と人工知能技術によるがんの起源と多様性の解明」　研究代表

因果関係の仮説と新発見の可能性

既存の文献通り

*PRRX1*の上皮間葉転換(EMT)制御についてはいくつかの研究がある。EMTは肺がんにおけるゲフィチニブ耐性のメカニズムの一つであることが多くの研究で示唆されている。

因果メカニズムの発見による業務改善

FUJITSU

- 薬剤開発以外の現場においても、様々な状況によって異なる因果メカニズムを発見し、業務改善につなげることが可能

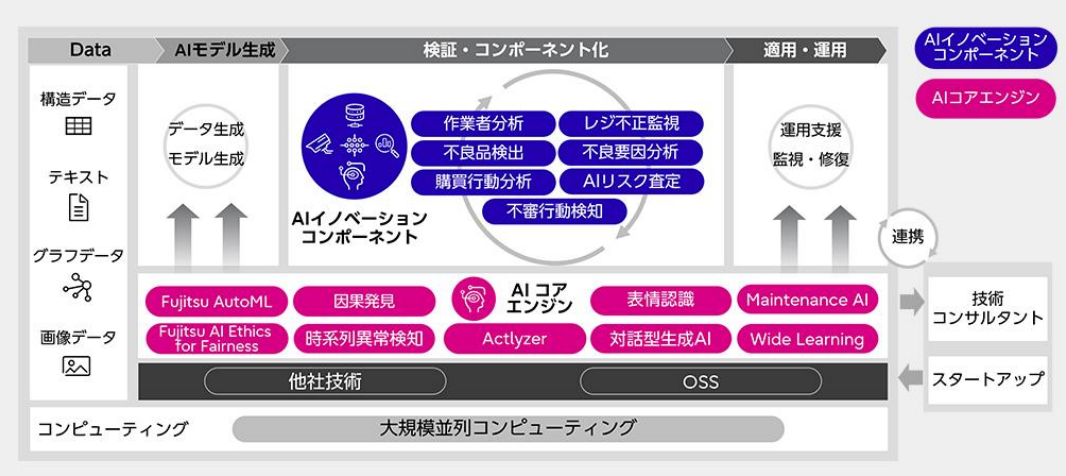
分野	発見したい因果メカニズム	期待される業務改善
マーケティング	顧客の特性ごとのサービス解約に至ったルートと根本原因	サービス離れしようとしている顧客を特定し、解約の根本原因を明らかにすることで解約防止にむけた効果的な施策を策定
システム運用	複雑なシステムにおける障害発生箇所とその原因	障害箇所と原因をリアルタイムで提示し、早期復旧を支援
生産現場	稼働環境に応じて不良品発生を引き起こす制御パラメータ	現在の稼働環境に適したパラメータの自動設定により、不良品の発生率を減らす

まとめ

- 「発見するAI」について紹介
- 「発見するAI」の大規模変数対応として条件探索と因果探索の高速化を実施
  - 条件探索で約数百倍、因果探索で約17000倍の性能向上を確認
- 「発見するAI」のゲノム医療への応用に成功
  - 既存の実装では4000年かかっていた処理を1日で完了
  - ゲフィチニブの耐性獲得メカニズムに関して、専門家が認める新仮説の抽出に成功

参考情報

- Fujitsu Kozuchi(AIプラットフォーム)を公開しています
  - https://www.fujitsu.com/jp/about/research/technology/ai/fujitsu-ai-platform/#anc-05
  - 富士通が研究開発した先端AI技術を迅速に試すことができるプラットフォーム
  - Kozuchi上で公開されているAIコアエンジンを無料で試せます



  - 今回紹介した因果探索も含まれているのでぜひご利用ください！！

参考情報

FUJITSU

- 富岳(AArch64)向け高速化情報
  - SVE対応版のNumpy
    - For A64FX (SVE512)
      https://github.com/kawakami-k/numpy/tree/sve_enablement
    - For Graviton3(SVE256)
      https://github.com/kawakami-k/numpy/tree/sve_enablement256
  - Xbyak_aarch64; just-in-time assembler for Armv8-a/9-a
    - Source code: https://github.com/fujitsu/xbyak_aarch64
    - LVC21 presentation:
      https://resources.linaro.org/en/resource/4V4esS3cY9BjeRExw7dwXt

参考文献

- LiNGAM(original)
  - https://github.com/cdt15/lingam
- Kazuhito MATSUDA, Kouji KURIHARA, Kentaro KAWAKAMI, Masafumi YAMAZAKI, Fuyuka YAMADA, Tsuguchika TABARU, Ken YOKOYAMA: Accelerating LiNGAM Causal Discovery with Massive Parallel Execution on Supercomputer Fugaku, IEICE Transactions on Information and Systems, (2022).
- Kazuhito MATSUDA, Kouji KURIHARA, Kentaro KAWAKAMI, Masafumi YAMAZAKI, Fuyuka YAMADA, Tsuguchika TABARU, Ken YOKOYAMA: Accelerating LiNGAM Causal Discovery with Massive Parallel Execution on Supercomputer Fugaku, IEICE Transactions on Information and Systems, (2022).
- Kawakami, S. Moriyuki, K. Kurihara and N. Fukumoto: Xbyak aarch64;JIT Assembler for Next Generation Supercomputer, Proc. CoolCHIPS23, (online), (2020). [20]
- K. Kawakami: Xbyak aarch64; Just-In-Time Assembler for Armv8-A and Scalable Vector Extention, https://connect.linaro.org/resources/lvc21/lvc21- 203/ (accessed 2021-06-15), Linaro Virtual Connect 2021, (online), (2021).
- 松田 一仁, 山崎 雅文, 川上 健太郎, 栗原 康志, 山田 芙夕楓, 田原 司 睦, 横山 乾:富岳を用いた大規模並列実行による因果探索手法 LiNGAM の高速化,第182回ハイパフォーマンスコンピューティング研究発表会
- 栗原 康志, 川上 健太郎, 山崎 雅文, 松田 一仁, 山田 芙夕楓, 田原 司睦, 横山 乾: LINGAMを用いた大量変数の因果探索処理に向けた計算カーネルの高速化の検討, 第180回ハイパフォーマンスコンピューティング研究発表会（SWoPP2021）

謝辞

- 本研究成果の一部は，理化学研究所のスーパーコンピュータ 「富岳」を利用して得られたものです.

Thank you