Vol.2021-HPC-182 No.5
2021/12/6

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

松田 一仁[1,a)]　山崎 雅文[1]　川上 健太郎[1]　栗原 康志[1]　山田 芙夕楓[1]　田原 司睦[1]　横山 乾[1]

概要：統計的因果探索は，因果関係が未知である観測変数群に対してこれを推定するアプローチである．統計的因果探索手法の一つである LiNGAM (Linear Non-Gaussian Acyclic Model) アルゴリズムにより，各観測変数の独立成分が非ガウス分布であることが仮定できる場合に因果関係を一意に特定できることが知られている．一方で LiNGAM アルゴリズムは変数の数 d_x に対し $O(d_x^3)$ の計算量を必要とするため，適用領域が限られていた．我々は LiNGAM の OSS 実装を高速化し，1 万変数以上の因果探索を現実的な時間で解くことで適用領域の拡大を目指している．本稿では，LiNGAM における行列に対する算術演算の SIMD 化および MPI を用いた並列化のアプローチを報告する．評価は，スーパーコンピュータ富岳を用いて行った．富岳 96 ノードを用いた場合には，2 万変数で既存 OSS 実装と比較して最大 17,531 倍の高速化を実現し，17.7 時間で実行可能なことを示す．

Accelerating LiNGAM causal discovery with massively parallel execution on Fugaku

Abstract: Statical causal discovery is an approach to infer the causal relationship between observed variables whose causalities are not revealed. LiNGAM (Linear Non-Gaussian Acyclic Model), which is an algorithm for causal discovery, can calculate the causal relationship uniquely if the independent components of variables are assumed to be non-Gaussian. However, use-cases of LiNGAM are limited because of its $O(d_x^3)$ computational complexity, where d_x is the number of variables. In this report, we show two approaches, that is, SIMD utilization for LiNGAM's mathematical matrixes operations, and MPI parallelization. We evaluate the implementation on the supercomputer Fugaku. In the case using 96 nodes of Fugaku, our improved version can achieve 17,531 times faster (completed in 17.7 hours) than the original OSS implementation.

1. はじめに

近年，医療現場において各々の患者固有の発がんに影響を与える特徴的な遺伝子の特定や，マーケティングにおける個々の顧客固有の購買行動につながる特徴的な要因の特定が求められている．このようなユースケースでは，観測した個々の事象間の相関関係だけでなく，因果関係を明らかにすることで，観測事象同士の主従関係を踏まえた考察や介入による結果の操作が可能になる．観測事象群全体の因果関係を求める手法として，統計的因果探索が有効であり，前述したユースケースを含め幅広い分野での応用が考えられている．

統計的因果探索では，多次元ベクトルとして観測された確率変数のサンプルデータから，各変数間の因果関係を推定する．各変数の独立成分が非ガウス分布であり，変数間の因果関係が線形非巡回であることが仮定できる場合，LiNGAM (Linear Non-Gaussian Acyclic Model) [1] を適用することで，因果関係を一意に推定できることが知られている．いくつかの LiNGAM を用いたアルゴリズムは OSS として Python による実装が公開されている [2].

現在我々は LiNGAM を適用可能なユースケースを拡大するため，1 万変数以上の多変数を対象とする因果探索を実行可能にすることを目指している．例を挙げると，人間の遺伝子は約 2 万あるので，これを直接対象変数とするようなユースケースが扱えるようになる．しかし OSS として公開されている現状の LiNGAM 実装は 1 台の計算機で実行することを前提とされており，実用的な時間で実行可

[1] 富士通株式会社
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa 211–8588, Japan
[a)] m.kazuhito@fujitsu.com

表 1: ベクトルデータの例

被験者 ID	年齢	睡眠時間	血圧
1	18	7.5	125
2	35	6.0	110
3	60	9.0	130
...	...	...	...

能な対象は数百変数に留まっている．

そこで我々は，2 つのアプローチで LiNGAM による因果探索の高速化を行った．1 つ目は，SIMD (Single Instruction, Multiple Data) 命令を用いた算術演算の CPU コア内での並列処理である．OSS の SIMD ライブラリである SLEEF (SIMD Library for Evaluation Elementary Function) [3], [4] を用いて実装を行った． 2 つ目は，MPI (Message Passing Interface) を用いた計算機ノードをまたぐマルチプロセスによる大規模並列実行である．既存の LiNGAM アルゴリズムはそのまま並列実行することは不可能であるため，MPI 通信による適切な同期によってこれを可能にした．

性能評価として，スーパーコンピュータ富岳 [5], [6] の大規模並列実行環境で，1 万変数以上の多変数を対象とした LiNGAM による因果探索が，実用的な時間で実行可能であることを示す．2 万変数を対象にした富岳 96 ノードでの実行では，既存 OSS の LiNGAM 実装の 17,531 倍の高速化を実現し，17.7 時間で計算を完了した．

2. 背景

2.1 統計的因果探索

統計的因果探索とは，統計的因果推論において，因果グラフを未知としてどのような条件で因果グラフが推測可能なのかを研究するものである．統計的因果探索において入力となるのは，あるインスタンス (対象が人であれば個々人が相当) に対し，観測した複数の事象を数値化したものである．対して出力は，事象間の因果関係，すなわちある事象の発現が，他の事象を引き起こすという関係にあるか否かであり，一般に重み付きの非巡回有向グラフ (DAG) によって表現される．

例えば，疫学調査において年齢，睡眠時間と血圧の調査を行った結果から，その因果関係を推定したいとする．この時，統計的因果探索の入力となるのは，年齢，睡眠時間，及び血圧を個人ごとに並べた多次元ベクトルデータとなる (表 1)．統計的因果探索はこのデータに対し，なんらかのアルゴリズムを用いて，これら観測変数間に因果関係があるのか否か，その影響はどの程度の大きさなのかを重み付き DAG で出力する (図 1)．図 1 では，年齢が高いほど睡眠時間が低くなる一方血圧は高く，睡眠時間が長いほど血圧が低くなる，という因果関係を示している．

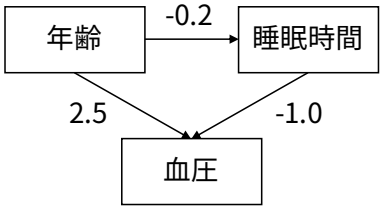

図 1: 因果探索の出力例

2.2 LiNGAM

LiNGAM (Linear Non-Gaussian Acyclic Model) は，観測した確率変数 (以降，単に観測変数) のサンプルの独立成分が非ガウス分布に従うこと，及び変数間の因果関係が線形非巡回であることが仮定できる場合に，因果グラフを一意に推定可能なモデルである．d_x 個の観測変数 $x_0, x_1, ..., x_{d_x}$ が LiNGAM に従って生成される場合，x_i は次のように記述できる．

$$x_i = \sum_{j \neq i} b_{ij} \times x_j + e_i (i = 1, ..., d_x) \tag{1}$$

すなわち，それぞれの観測変数 x_i は，自身以外の観測変数 $x_j (j = 1, ..., d_x)$ とその誤差変数 (独立成分) e_i の線形和である．b_{ij} は変数 x_j から x_i への因果関係の強さを表しており，この値が 0 である場合には直接的な因果関係がないことを示す．x_i, b_{ij}, 及び e_i をベクトル $\boldsymbol{x}$, $\boldsymbol{B}$, $\boldsymbol{e}$ と表現すると，式 (1) は以下のように記述できる．

$$\boldsymbol{x} = \boldsymbol{B}x + \boldsymbol{e} \tag{2}$$

このとき，$\boldsymbol{B}$ は隣接行列と呼ばれる．隣接行列を求めることは，図 1 に示したような重み付き DAG を求めることに等しい．

2.3 LiNGAM に基づく因果探索アルゴリズム

LiNGAM において隣接行列 $\boldsymbol{B}$ を推定するアプローチには大きく分けて独立成分分析の手法を用いるアプローチ (ICA-LinGAM [1]) と回帰分析と独立性評価を繰り返すアプローチ (Direct-LiNGAM [7], [8]) の 2 つが存在する．どちらも Python を用いた OSS 実装が公開されている [2] が，我々はより新しい手法である Direct-LiNGAM を対象とし，1 万変数規模の因果探索を目標として高速化を行った．

Direct-LiNGAM アルゴリズムの概要は以下の通りである．

(1) 各変数に対して，他変数への相互情報量と逆方向の相互情報量の差を計算し，これの和をとる．
(2) 全ての変数のうち，(1) で計算した値が最も小さいものを因果の先頭にあると推定する．
(3) 先頭と推定した変数から他変数への寄与を取り除き，残った変数で手順 (1) から繰り返す．これをすべての変数の因果順が確定するまで繰り返す．

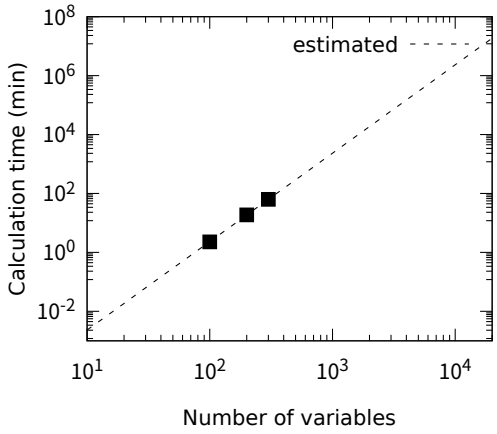

図 2: d_x に対する既存実装の計算時間

(4) 確定した因果順の先頭から，自身より上位の因果順を持つ変数を用いて回帰分析を行い，隣接行列 $\boldsymbol{B}$ を計算する．

GitHub.com 上に OSS として公開されている実装 "`lingam/direct_lingam.py`" (コミット ID 022a3a2) では，手順 (1) が `_search_causal_order` (L195-215) 関数に相当し，手順 (2) が `fit` 関数のメインループ処理 (L86-98) に相当する．手順 (3) は，`fit` 関数の `return` 文で実行される `_estimate_adjacency_matrix` 関数が相当する．これらの関数が高速化の対象となる．

図 2 は，既存 OSS 実装を Intel Xeon Gold 5218 2.30GHz を用いて実行した場合，及びそこから予測される計算時間をプロットしたものである．実装からもわかるように計算時間は $O(d_x^3)$ (縦軸が分の時で，係数は 2.327×10^{-6} 程度) に従っており，仮に 2 万変数を対象とした場合に既存実装では約 35 年かかることになり，実用的な計算時間ではないことがわかる．

3. Direct-LiNGAM の高速化

高速化にむけて，我々はまず既存 OSS 実装のプロファイリングを取得し，相互情報量の計算がボトルネックとなっていることを明らかにした．このボトルネックに対して，個々の相互情報量演算を高速化するアプローチとして SIMD による算術演算の並列計算を実装した．加えて，大規模並列実行を可能にするために，MPI 通信を用いた計算処理の分割，同期機構を実装した．

3.1 SIMD による算術演算の並列計算

SIMD (Single Instruction, Multiple Data) は，1 つの命令を複数の入力データを対象に適用する並列化である．例えば長さ n, *double* (64 bit) 型の一次元配列 $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ に対して，$\boldsymbol{C} = \boldsymbol{A} + \boldsymbol{B}$ のような計算を行うとする．この時，SIMD 命令を用いない場合は配列の長さと同じ回数のループを回すため，n 回の加算命令が実行される．一方

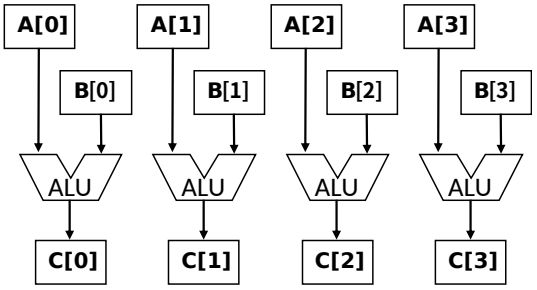

図 3: SIMD による並列計算

で SIMD 命令を用いる場合，対応するベクトル長を k bit とすると $\frac{k}{64}$ 個のデータを並列計算可能なため，$\frac{n}{(k/64)}$ 回の加算命令で計算できる (図 3)．例えば富岳で採用されている CPU A64FX (AArch64 アーキテクチャ) であれば，SVE (Scalable Vector Extensions) 拡張命令セットを持っており，$k = 512$ となる．

SIMD を利用するに，我々は C 言語標準の算術演算関数の入出力をベクトル化した OSS ライブラリである SLEEF [3], [4] を用いて実装を行った．2.3 節の手順 (1) である相互情報量の差の計算には，ある変数 x_i に対して，$x_j (j \neq i)$ との回帰分析の残差を双方向に計算し，それらのエントロピーを求める必要がある．既存 OSS 実装の ソースコード 1 に示す．

ソースコード 1: 相互情報量の差を計算するコード

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

M_list = []
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)]
```

ここで，入力となる配列 U, X をベクトルとして扱い，コンパイラによる最適化，及び SLEEF で実装されている算

術演算関数を用いることで SIMD 化を行う．具体的には，我々は ソースコード 1 の L7-18 を代替する関数を C++ で実装し，Cython [9] を介して Python コードとバインドした．入力としてベクトル U, X, 及び X から事前に計算した標準偏差，分散，エントロピーをそれぞれ X_std, X_std_var, X_h として受け取り，全ての変数において，他の変数に対する相互情報量の差の和をリスト化したもの (ソースコード 1 における M_list) を戻り値とする．

実装した C++ 関数の内部では，例えば残差計算において，ソースコード 2 のように，与えられたベクトルの各要素に対する演算を `for` 文内の 1 ブロック内で完結する形で書くことで，コンパイラによる最適化によって SIMD 命令を利用できるよう記述している．加えてエントロピー演算では，SLEEF で実装されているベクトル入力を受け付ける log 関数，exp 関数，及び cosh 関数を用いた．

ソースコード 2: SIMD 命令にコンパイルされるコード

```
1 double cov_0_1 = calc_cov_0_1(arr0, arr1,
      num0, num1);
2 double coef = cov_0_1 / xj_var;
3
4 for (size_t i = 0; i < num_dst; i++) {
5   dst_arr[i] = arr0[i] - coef * arr1[i];
6 }
```

SIMD 命令を使わない場合と比べて，SIMD レジスタ長が 512, 各変数の bit 長を 64 とした場合で 8 並列での動作が可能であるため，算術演算部分に関しては理論上 8 倍の高速化が可能になる．ただし，実際には `for` ループ回数や関数呼び出しの回数の増減，メモリアクセス方法の差なども影響する．

3.2 MPI による大規模並列計算

2.3 節で述べたアルゴリズムは，手順 (2) において，全ての変数に関して手順 (1) の計算結果を比較し，値が最も小さいものを選択する必要がある．よって，処理全体を単純に分割してプロセス及びノード並列で実行することはできない．一方で，各変数に対して手順 (1) を計算する箇所，及び手順 (4) で隣接行列を計算する箇所については，変数単位で分割できるため，ここに大規模並列実行の可能性がある．

まず手順 (1) の箇所に関して，我々は対象とする変数群 U を，並列実行するプロセス群に均等に分配することとした．各変数に対する計算量に差がないため，このようなシンプルな分割でも実行時間の差は理論上存在しない．具体的には，MPI の $COMM\_WORLD$ コミュニケータを用いて，そのサイズで U を分割したものを配列とし，各プロセスの ID であるランクで処理すべき変数群 $U\_$ を決定する．各プロセスは $U\_$ に関して，手順 (1) のような計算を行う (ソースコード 3)．M_value が手順 (1) の計算結果リスト，M_arg が変数の識別子のリストである．

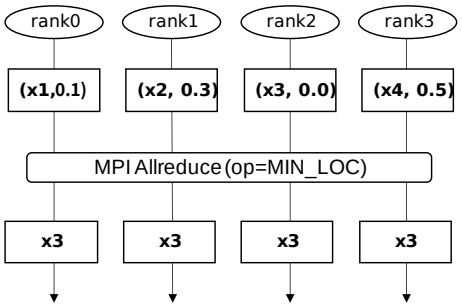

図 4: MPI Allreduce を用いた因果順の先頭の共有

手順 (2) では各プロセスが計算した値の大小を比較する必要があるが，ここは MPI の Allreduce 集合通信を用いることができる．手順 (2) のために必要となるのは，”手順 (1) で計算した値に関し，最も小さい値を持つ変数の識別子” であるので，Allreduce において MIN_LOC 演算を用い，自身が計算した結果中の最小の値と変数識別子のタプルを与えることで，得たい結果である ”次の因果順の先頭となる変数の識別子” をプロセス間で同期することができる (図 4)．

ソースコード 3: MPI Allreduce による並列実行

```
1 comm = MPI.COMM_WORLD
2 _mpi_rank = comm.Get_rank()
3 _mpi_size = comm.Get_size()
4
5 U_ = np.array_split(U, _mpi_size)[_mpi_rank]
6
7 M_value, M_arg = self.
      calculate_diff_mutual_simd(U_, X)
8
9 M_min_local = (np.inf, -1)
10 if len(Uc) > 0:
11   M_min_local = (M_value[0], M_arg[0])
12 return comm.allreduce(M_min_local, MPI.MINLOC
      )[1]
```

手順 (3) に関しては，推定した因果順の配列を手順 (2) と同様に $COMM\_WORLD$ コミュニケータのサイズで分割し，各プロセスで割り当てられた変数の回帰分析を行う．各プロセスが計算した隣接行列は全体の隣接行列の部分行列に相当する．隣接行列は変数の数の 2 乗の 32 bit 浮動小数点を含むため，仮に 2 万変数を対象とした因果探索結果であれば，データサイズは $(2 \times 10^4)^2 \times 4 = 1.6$ GB になる．よって，各プロセスは各々の担当する部分行列のみを保持することとし，ランク 0 のプロセスに Gatherv 通信によって集約することで不要な通信およびメモリ消費を回避した (図 5)．

MPI による高速化では，対象となる変数の数 d_x が最大

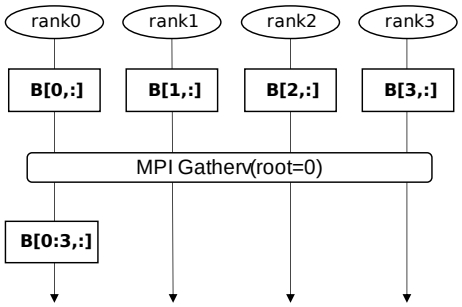

図 5: MPI Gatherv を隣接行列の集約

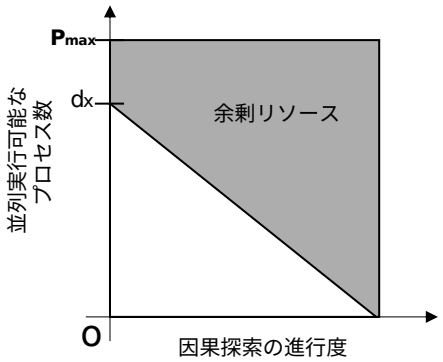

図 6: 因果探索の進行に伴う並列数の減少

並列数となる．特に手順 (2) に対する並列化は処理が進行するに従って因果順が確定していくため，対象となる変数が減少していく．その結果とりうる並列数も減少していくため，例えば変数 2 万に対し並列数 2 万を実行できる環境を用いたとしても，線形に高速化効果が得られるわけではない (図 6，P_{max} は個々の環境において実行可能な最大プロセス数)．実行環境の並列数を決定する際に留意が必要な点であるといえる．

4. 性能評価

我々はスーパーコンピュータ富岳を用いて，3 章の高速化を施した Direct-LiNGAM の性能評価を行った．各環境の主な諸元は表 2 の通りである．詳細な仕様は web ページにて確認できる [10].

4.1 評価結果

図 7 は，サンプル数 $n = 200$ のとき，横軸に変数の数 d_x，縦軸に計算時間をプロットしたものである．横軸縦軸共に対数軸としている．富岳は 96 ノードを使用した．富岳は 1 ノードあたり計算用に 48 CPU コアを持つため，最大 $4,608$ プロセスを並列実行できる．計算時間は d_x に対して対数で線形に計算時間は増加しているが，これは先述したとおり Direct-LiNGAM の計算時間が $O(d_x^3)$ で決まるためである．$d_x = 20,000$ で $1,062$ 分，すなわち 17.7 時間で計算できており，これは既存実装と比較して約 $17,531$ 倍の高速化である．既存実装を評価した環境とは CPU 性能が異なるため高速化実装のみの効果ではないが，現実的な時間で処理できていることがわかる．

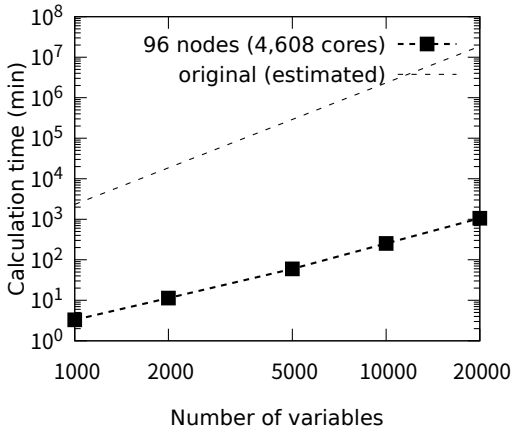

図 7: d_x に対する計算時間 (n=200)

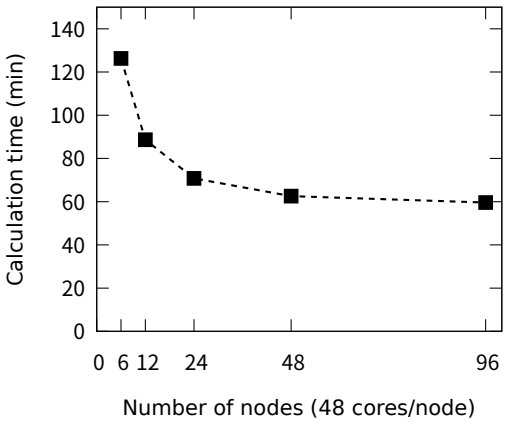

図 8: ノード数に対する計算時間 ($n = 200$, $d_x = 5000$)

図 8 及び図 9 は，$n = 200$ でそれぞれ $d_x = 5,000$, $d_x = 20,000$ のとき，横軸に富岳のノード数，縦軸に計算時間をプロットしたものである．96, 192, 384 ノードではそれぞれ $4,608$, $9,216$, $18,432$ プロセスを並列実行可能である．いずれの設定においても計算を開始する段階では d_x が並列実行可能プロセス数を上回っているため並列化による高速化を受けられるが，3.2 節で述べた通り d_x は探索が進行するに従って減少するため，とりうる並列数も減少する．富岳において利用可能なノード数は現時点で $158,976$ ノードであるが，今回得られた評価結果から，d_x の半分程度のプロセス数でも高速化は上限に漸近することがわかる．

5. おわりに

我々は，因果探索において 1 万以上の多変数を扱うことを目標に，因果探索手法である Direct-LiNGAM アルゴリズムに対し，既存 OSS 実装をベースに SIMD，MPI を利用した高速化を実装した．これにより，スーパーコンピュータ富岳を用いた大規模並列実行評価において，既存

表 2: 評価環境の主な諸元

名称	CPU	ノードあたりコア数	メモリ	インターコネクト
富岳	A64FX (2.2 GHz)	48	HBM2 32 GiB, 1024 GB/s	Tofu Interconnect D (28 Gbps x 2 lane x 10 port)

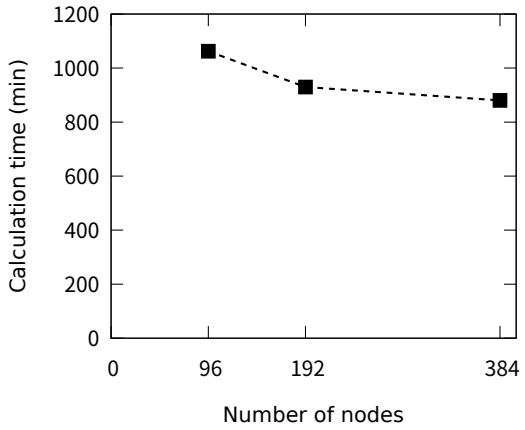

図 9: ノード数に対する計算時間 ($n = 200$, $d_x = 20000$)

OSS 実装の 17,531 倍の高速化を実現し，2 万変数を対象とした因果探索を 17.7 時間で完了できることを示した．

一方で，今回の高速化では並列数の上限が変数の数で決まること，及び因果探索の進行に伴って並列数が減少していくことにより，ノード数，プロセス数に対しスケーラビリティを獲得したとは言えない．ノード数に対するスケーラビリティ獲得についても今後検討を進めていく予定である．

また我々は，富岳の A64FX CPU と相性のよい JIT (Just-in-Time) コード生成による高速化検討をすでに行っており [11]，3 倍程度の高速化を実現できる想定である．

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．

参考文献

[1] Shimizu, S. et al.: A Linear Non-Gaussian Acyclic Model for Causal Discovery, *Journal of Machine Learning Research*, Vol. 7, No. 72, pp. 2003–2030 (2006).
[2] cdt15: LiNGAM - Discovery of non-gaussian linear causal models. available at `https://github.com/cdt15/lingam`.
[3] Shibata, N. and Petrogalli, F.: SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 31, No. 6, pp. 1316–1327 (2020).
[4] shibatch: SLEEF Vectorized Math Library. available at `https://github.com/shibatch/sleef`.
[5] Fujitsu: Fujitsu and RIKEN Complete Joint Development of Japan ' s Fugaku, the World ' s Fastest Supercomputer. available at `https://www.fujitsu.com/global/about/resources/news/press-releases/2021/0309-02.html`.
[6] 理化学研究所 計算科学研究センター：「富岳」について. available at `https://www.r-ccs.riken.jp/fugaku`.
[7] Shimizu, S. et al.: DirectLiNGAM: A Direct Method for Learning a Linear Non-Gaussian Structural Equation Model, Vol. 12, p. 1225–1248 (2011).
[8] Hyvärinen, A. and Smith, S. M.: Pairwise Likelihood Ratios for Estimation of Non-Gaussian Structural Equation Models, Vol. 14, No. 1, p. 111–152 (2013).
[9] Cython: C-Extensions for Python. available at `https://cython.org/`.
[10] Fujitsu: スーパーコンピュータ「富岳」: 仕様. available at `https://www.fujitsu.com/jp/about/businesspolicy/tech/fugaku/specifications/`.
[11] 栗原康志ほか：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，技術報告 26 (2021).