Vol.2021-HPC-182 No.16
2021/12/7

MPIにおける小規模実行時の通信トレース解析による大規模実行時の通信タイミング予測の評価

岡田 悠希[1,a)]　三輪 忍[1]　八巻 隼人[1]　本多 弘樹[1]

概要：
高性能計算分野においてMPI並列アプリケーションの通信トレースは幅広く利用されている．現在一般的に利用されている通信トレース取得手法では，通信トレースの取得に必要なコスト（時間と計算資源）がアプリケーションの実行規模に依存するため，大規模並列環境における通信トレースの取得は困難である．上記の問題に対して，小規模実行時の通信トレースの解析結果に基づいて大規模実行時の通信タイミングを予測するScalaExtrapが提案されている．しかし，ScalaExtrapは特定の通信パターンを有するアプリケーションにしか対応しておらず，問題サイズを変更した際の通信トレース予測は行えないという問題がある．現在，我々は通信トレース予測技術としてPredComを開発しており，PredComに通信タイミング予測を行う機能を追加することで，PredComのユースケースを更に拡大することを目指している．本稿では，PredComに上記の機能を実装するにあたって，ScalaExtrapで行われている通信タイミング予測手法の評価と分析を行った．この結果から，ScalaExtrapに実装されている4種類のモデルでは，任意のプログラムに対してデルタ時間の和の最大値を正確に予測するのは困難であり，特に並列数が増加するにつれてデルタ時間の和のばらつきが大きくなるようなケースにおいて，正確な予測が困難であることがわかった．

キーワード：MPI，性能解析，通信トレース

1. はじめに

高性能計算分野においてMPI並列アプリケーションの通信トレースはデバッグと異常検出 [14]，性能分析とチューニング [3], [5], [9], [10]，ネットワークの消費電力予測 [7] などに幅広く利用されている．通信トレースはアプリケーションによるMPI通信関数の実行履歴を記録したものであり，通常は計測用コードが挿入されたアプリケーションプログラムを実行することによって通信トレースを取得する．しかし，上記の方法では通信トレースの取得に必要なコスト（時間と計算資源）がアプリケーションの実行規模（問題サイズと並列数）に依存するため，1,000ノードを超えるような大規模並列環境における通信トレースの取得は困難である．

上記の問題に対してScalaExtrapが提案されている [13]．ScalaExtrapはMPIアプリケーションを複数の少ない並列数で実行した時の通信トレースからアプリケーション内の各MPI通信関数の通信量，通信先，実行時刻などに関するスケーラビリティモデルを生成し，生成したモデルを用いて多い並列数で実行した時の通信トレースを予測する．モデルを用いた予測は少ない計算資源で短時間に行うことができるため，大規模実行時の通信トレースを低コストで取得できる．ただし，ScalaExtrapは1) ステンシル計算のような特定の通信パターンを有するアプリケーションにしか対応しておらず，2) 問題サイズを大きくした際の通信トレース予測を行いたいケースでは利用できない．

一方，我々はより多くのユースケースに対応可能な通信トレース予測技術としてPredComを開発している [6]．PredComは複数の小規模（問題サイズと並列数）実行時の通信トレースからアプリケーション内の各MPI通信関数の通信量や通信先などに関するスケーラビリティモデルを生成し，生成したモデルを用いて大規模（問題サイズと並列数）実行時の通信トレースを予測する．ScalaExtrapとは異なりステンシル計算以外の通信パターンを有するアプリケーションや問題サイズを変更した際の通信トレース予測にも対応しているものの，現在のPredComはMPI通信関数の実行タイミング予測には対応していない．

そこで本研究では，現在のPredComを拡張し，MPI通信関数の実行タイミング（通信タイミング）予測を行う機

[1] 電気通信大学
1-5-1, Chofugaoka, Chofu, Tokyo 182–8585, Japan
[a)] okada@hpc.is.uec.ac.jp

能を追加することを考える．同機能の搭載により，各 MPI 通信関数の通信先や通信量だけでなく実行タイミングもトレースとして出力できるようにし，PredCom のユースケースをさらに拡大することが本研究の狙いである．

上記の機能の実装にあたって本稿では，ScalaExtrap で行われれている通信タイミング予測手法の評価と分析を行う．そして分析結果をもとに，PredCom における通信タイミング予測手法を検討する．

本稿の構成は以下のようになっている．まず次章では関連研究として通信トレース取得時間を短縮する既存手法についてまとめる．続く 3 章では，ScalaExtrap における通信タイミング予測手法を詳しく述べる．4 章で評価方法について述べ，5 章で評価結果を示し，最後 6 章で本稿をまとめる．

2. 関連研究

通信トレースの取得に要する時間を短縮するため，いくつかの手法が提案されている．以下ではこれらの手法について述べる．

2.1 通信トレース取得時のオーバーヘッド軽減

大規模実行時の通信トレース取得においてはトレースファイルに対するアクセス時間が性能オーバーヘッドとなることから，トレースファイルに記載する情報を圧縮することで上記のオーバーヘッドを軽減する手法が提案されている [1], [8]．また，通信トレースの取得時間を短縮するために，すべての通信イベントの実行履歴を取得するのではなく，実行履歴の取得を行う通信イベントを一部に限定する手法が広く用いられている [12]．これらの手法は大規模実行時の通信トレース取得時間をある程度短縮する効果はあるものの，解析対象のアプリケーションプログラムを解析対象の規模で実際に実行する必要がある点において通常のトレース取得手法と変わりはない．

2.2 高速な通信トレース生成

通信トレースのいくつかの応用例では，各 MPI 通信関数の通信タイミングよりも通信先と通信量の方が重要な意味を持つ．また，解析対象のプログラム内で行われる計算のうち，各 MPI 通信関数の通信先と通信量の決定に関与する計算は一部である．

そこで先行研究 [15] では，専用コンパイラを用いて解析対象プログラムから通信トレース生成に関与する命令だけを抽出した通信トレース生成用プログラムを作成し，専用ライブラリとともに上記プログラムを実行することで高速に通信トレースを生成する手法を提案している．ただし，この方法では C や C++のようなグローバル変数を用いて記述されるプログラムにおいては通信トレース生成に真に関与する命令のみを抽出することが難しく，高速な通信トレース生成用プログラムを作成できない．また，上記の手法で生成される通信トレースは通信タイミングに関する情報が失われているという問題もある．

2.3 通信トレース予測

Wu らは小規模実行時（少ない並列数）の通信トレースから大規模実行時（多い並列数）の通信トレースを予測するツールとして ScalaExtrap を提案している [13]．ScalaExtrap では通信トレース予測を行う対象のプログラムをステンシル計算などの特定の通信パターンを有するプログラムに限定し，各 MPI 通信関数の通信先のランク番号，通信メッセージの量，通信タイミング（実行時刻）の予測を行っている．ScalaExtrap 内で行われる通信タイミング予測については次章で詳しく述べる．ScalaExtrap の通信タイミング予測に関しては，NAS Parallel Benchmarks の BT，EP，FT，CG，IS を用いた評価において，プログラム全体の実行時間を概ね 90%以上，最大 98%以上の精度を示すことが報告されている．ただし，プログラムの実行中に通信パターンが変化する IS に関しては，常に低い予測精度を示すことが確認されている．また，ScalaExtrap は異なる問題サイズの通信トレース予測には対応しておらず，問題サイズを変更した場合は予測に使用する（少ない並列数の）通信トレースを取得し直す必要がある．

我々は，より多くのユースケースに対応可能な通信トレース予測技術として PredCom を開発している [6]．PredCom は，複数の小規模（問題サイズと並列数）実行時の通信トレースからアプリケーション内の各 MPI 通信関数の通信量や通信先などに関するスケーラビリティモデルを生成する．そして LLVM ベースのコード変換器が，上述のようにして生成したモデルを用いて解析対象アプリケーションのソースコードを通信トレース予測を行うコードへと自動変換する．上記の変換の過程で元のプログラムに含まれていた数値計算などの通信トレース生成と無関係な処理はすべて削除されるため，通信トレース予測プログラムは非常に少ないコスト（例えば元のアプリケーションプログラムの 1,000 倍以上の速さ）で実行できる．

ScalaExtrap とは異なり，PredCom ではステンシル計算以外の通信パターンを有するアプリケーションや問題サイズを変更した際の通信トレース予測も可能である．ただし，PredCom は現時点では MPI 通信関数の通信先や通信量などの空間的な情報の予測のみに対応しており，同関数の実行タイミングといった時間的な情報の予測はできない．

3. ScalaExtrap における通信タイミング予測

本章では ScalaExtrap における通信タイミング予測の方法を説明する．ScalaExtrap の通信タイミング予測には，一般的な通信トレースに記録されている各 MPI 通信関数

の実行時刻を表すタイムスタンプの代わりに，各 MPI 通信関数の終了時刻から次の MPI 通信関数の開始時刻までの時間を表すデルタ時間が用いられる．

まず，プログラム内のループに含まれる MPI 通信関数の実行区間（MPI 通信関数の終了から次の MPI 通信関数の開始まで）ごとに，小規模実行を行うことで得た各並列数の通信トレースからその並列数におけるデルタ時間のヒストグラムを作成する．ヒストグラムにはその実行区間におけるデルタ時間の分布がビンとして記録されるだけでなく，各ビンに含まれるデルタ時間の平均値，および，各ビンの下限/上限値も記録される．また，ヒストグラム全体に含まれるデルタ時間の平均/最小/最大値も併せて記録される．

次に，複数の並列数のヒストグラムに記録された値を種別毎に集めてサンプルデータとし，サンプルデータに対してフィッティングを行うことで，並列数をスケールさせた時の値を予測するモデルを作成する．フィッティングは Constant, Linear, Inverse Proportional, Inverse Proportional + Constant の 4 つのモデルに対して行い、最も適合度の高いモデルがその実行区間のデルタ時間を予測するモデルとして最終的に選択される．各モデルの式を以下に示す．

$$t = f(n) = c \tag{1}$$

$$t = f(n) = an + b \tag{2}$$

$$t = f(n) = \frac{k}{n} \tag{3}$$

$$t = f(n) = \frac{k}{n} + c \tag{4}$$

上記の式において n は並列数，t はデルタ時間を表す．以下，各モデルの詳細を述べる．

Constant　並列数によらず一定の値を示すモデルである．誤差の影響を軽減するために偏差の絶対値が最大となるサンプルデータ c_i を外れ値として除外し，残りのサンプルデータから求めた平均値 $average$ を c とする．フィッティング後のモデルの評価には，外れ値を除いたサンプルデータの標準偏差を sd として $d_1 = sd/average$ を用いる．

Linear　並列数に比例するモデルである．フィッティングには最小二乗法が用いられる．フィッティング後のモデルの評価には，残差 $residual$，サンプルデータの各並列数 n_i に対するこのモデルの予測値 t_i の平均値を $average$ として，$d_2 = \sqrt{residual}/average$ を用いる．

Inverse Proportional　並列数に反比例するモデルである．サンプルデータ内の並列数 n_i に関するデータを t_i とし，$k_i = t_i \times n_i$ を満たす k_i の平均値を k とする．ただし，k_i の平均値を求める際は k_i の外れ値を除外する．フィッティング後のモデルの評価には，k_i の標準偏差を sd，$d_3 = sd/k$ を用いる．

Inverse Proportional+Constant　並列数に反比例するモデルである．定数項 c を有する点が Inverse Proportional と異なる．フィッティングの際は式 (4) に対して直接フィッティングを行うのではなく，同式を $t' = tn = cn + k$ と変形した上で線形回帰によるフィッティングを行い，k と c を求める．フィッティング後のモデルの評価には，残差 $residual$，サンプルデータの各並列数 n_i に対するこの線形回帰モデルの予測値 t_i' の平均値を $average$ として，$d_4 = \sqrt{residual}/average$ を用いる．

モデルの選択においては，4 つのモデルの評価値 d の中から最も評価値の小さいモデルを最適なモデルとして採用する．

また，各実行区間がプログラム実行時に何回実行されたかを表す，各実行区間の実行回数については，各実行区間毎に，実行並列数やプログラムの問題サイズを入力，実行回数を出力とする連立方程式を解くことで，予測したい並列数で実行した際の各実行区間の実行回数の予測を行う．

4. 評価

4.1 実験環境

実験は東京工業大学学術国際情報センターが運用するスーパーコンピュータ TSUBAME3.0[16] を用いて行った．TSUBAME3.0 のハードウェア構成を**表 1** に示す．TSUBAME3.0 は 540 台の計算ノードから構成され，計算ノード 1 台あたり 14 コアの CPU を 2 基搭載している．予測に用いる通信トレースの取得をするにあたって，通信トレースファイル書き込み時の I/O 負荷による実行時間増加を軽減するため，通信トレースファイルを共有ストレージへ直接書き込みを行う代わりに，TSUBAME3.0 の各計算ノードに搭載されているローカルストレージを用いたスクラッチ領域へ一度全て書き込みを行った後共有ストレージへコピーを行うようにした．

通信トレース予測の対象とする並列プログラムとして，2D ステンシルプログラム POISSON_MPI[11]，NAS Parallel Benchmarks[2] の IS (NPB-IS)，流体力学計算プログラム LULESH[4] の 3 つを用いた．問題サイズは，POISSON_MPI は，Interior vertices in one dimension を 2048，NPB-IS は，CLASS E，LULESH は Cube mesh length を 32 とした．POISSON_MPI と NPB-IS は，通信関数 1 回あたりの通信時間や，プログラム内の通信関数の呼び出し回数が並列数のみに依存するため，ScalarExtrap による予測が容易と予想される．一方，LULESH は，通信関数の呼び出し回数が並列数以外にも依存するため，ScalarExtrap による予測が困難と予想される．

ScalarExtrap のソースコードは公開されているものの十分なメンテナンスが行われておらず，今回の実験で使用したプログラムの通信トレース予測はできなかった． そのた

表 1: TSUBAME3.0 のハードウェア構成

計算ノード		540 台
共有ストレージ		DDN SFA14KXE 及び EXAScaler
ノード構成（1 台あたり）	CPU	Intel Xeon E5-2680 V4 2.4 GHz × 2 CPU
	コア数 / スレッド数	14 コア / 28 スレッド × 2 CPU
	GPU	Tesla P100 for NVLink-Optimized Servers × 4
	RAM	256 GiB
	ローカルストレージ	Intel DC P3500 2 TB
	インターコネクト	Intel Omni-Path 100 Gb/s × 4

め本実験では，前章で述べた手法を再現した自作の通信トレース予測プログラムを使用した．フィッティング等で使用する通信トレースは，文献 [6] と同様，計測用コードを挿入する Pass を実装した LLVM を用いて各プログラムのコンパイルを行い，コンパイル後のプログラムを所望の並列数で実行することによって取得した．

4.2 評価項目

本稿では以下の 2 つの評価を行う．

デルタ時間の和の予測精度 ランク毎にデルタ時間の和を求め，まずはその最大値を実測値と予測値で比較する．最大値を比較する理由は，デルタ時間の和が最大となるランクが最も実行時間が長く，またデルタ時間の和の最大値が通信時間や通信の待ち合わせ時間の影響を無視したプログラム全体の実行時間とほぼ等しいと考えられるためである．さらに，上記のランク以外のランクにおけるデルタ時間の和の予測精度を評価するため，ランク毎のデルタ時間の和を用いてヒストグラムを作成し，実測値と予測値でその形状を比較する．

実行区間毎のデルタ時間の予測精度 プログラム中の実行区間毎にデルタ時間のヒストグラムを作成して実測値と予測値の比較を行うことで，実行区間毎のデルタ時間の予測精度を評価する．

4.3 評価手法

まずは評価対象の並列プログラムを 4 通りの並列数で実行することにより通信トレースを取得し，取得したそれぞれのトレースに対してプログラム中のデルタ時間のヒストグラムを作成する．次にこうして作成した 4 つのヒストグラムを用いて，より多い並列数におけるデルタ時間のヒストグラムを予測する．最後に予測対象の並列数でプログラムを実際に実行することにより通信トレースを取得し，取得したトレースから生成したヒストグラムと予測したヒストグラムを比較する．

具体的には，POISSON_MPI と NPB-IS に関しては，並列数 64, 128, 256, 512 の通信トレースを用いて，並列数 1,024 におけるデルタ時間予測を行った．また LULESH に関しては，並列数 216, 343, 512, 729 の通信トレースを用いて，並列数 1,000 におけるデルタ時間予測を行った．

本実験で使用した通信トレースには，MPI 通信関数毎にソースコード上での呼び出し位置，実行開始時刻，実行終了時刻が記録されている．この通信トレースに記録された，ある MPI 通信関数の終了時刻から次の MPI 通信関数の開始時刻までの差分をデルタ時間とする．本実験では，各プログラムにおいて `MPI_Init()` が実行されてから `MPI_Finalize()` が実行されるまでの間のデルタ時間を評価した．

5. 評価結果

5.1 デルタ時間の和の予測精度

予測精度は式 (5) を用いて評価した．ただし，以下の式において $\hat{t}$ は予測値，t は実測値である．

$$Accuracy\,[\%] = \left(1 - \frac{\left|\hat{t} - t\right|}{t}\right) \times 100 \qquad (5)$$

各プログラムに対するデルタ時間の和の最大値の予測精度を**表 2** に示す．また参考までに，予測に使用した 4 つの通信トレースの並列数とデルタ時間の和の最大値を**表 3** に示す．

表 2 より，ScalaExtrap の通信タイミング予測手法は LULESH では 95%以上の予測精度を示した一方，POISSON_MPI と NPB-IS では 65%以下の予測精度にとどまった．POISSON_MPI と NPB-IS の予測精度が低かった理由として，実行区間毎のデルタ時間と並列数の関係を予測に反映できていないことが挙げられる．この評価ではプログラム中の全てのデルタ時間の和に対して 3 章で述べた 4 つのモデルを用いて予測を行ったため，並列数とデルタ時間の関係が実行区間によって異なるプログラムはうまく予測できない．実際，プログラム中の各実行区間におけるデルタ時間と並列数の関係を分析したところ，デルタ時間が並列数に対して，1) 反比例，2) 比例，3) ほぼ一定，の組み合わせで構成されており，これらのデルタ時間の和は 4 つのモデルのいずれにもフィットしなかった．

デルタ時間の和のヒストグラムを**図 1**, **図 3**, **図 5** に示す．赤いヒストグラムが予測値，青いヒストグラムが実測値である．また参考までに，各プログラムにおいて予測に

表 2: デルタ時間の和の最大値の予測精度

プログラム	並列数	実測値 [µs]	予測値 [µs]	精度
POISSON_MPI	1,024	1131380.0	1606645.3	58.0%
NPB-IS	1,024	138804152.0	88104753.1	63.5%
LULESH	1,000	627985822.0	658495132.9	95.1%

表 3: 予測に用いたデルタ時間の和の最大値

プログラム	並列数	デルタ時間の合計の最大値 [µs]
POISSON_MPI	64	7938954.0
	128	4676880.0
	256	2818114.0
	512	2040480.0
NPB-IS	64	644250437.0
	128	388778762.0
	256	224377159.0
	512	125488511.0
LULESH	216	366978974.0
	343	428496307.0
	512	492032255.0
	729	553775812.0

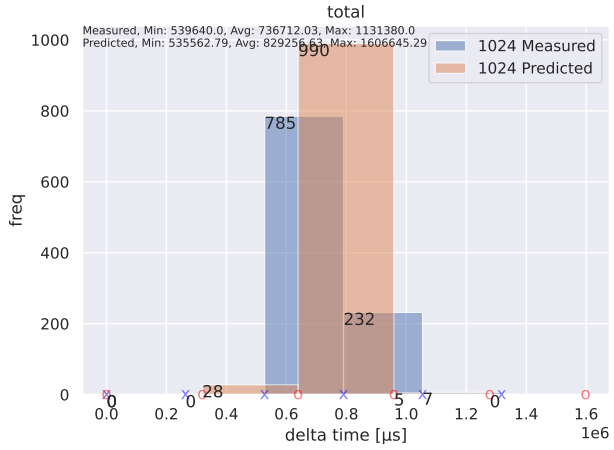

図 1: POISSON_MPI, 並列数 1,024 におけるデルタ時間の和の分布（赤：予測値，青：実測値）

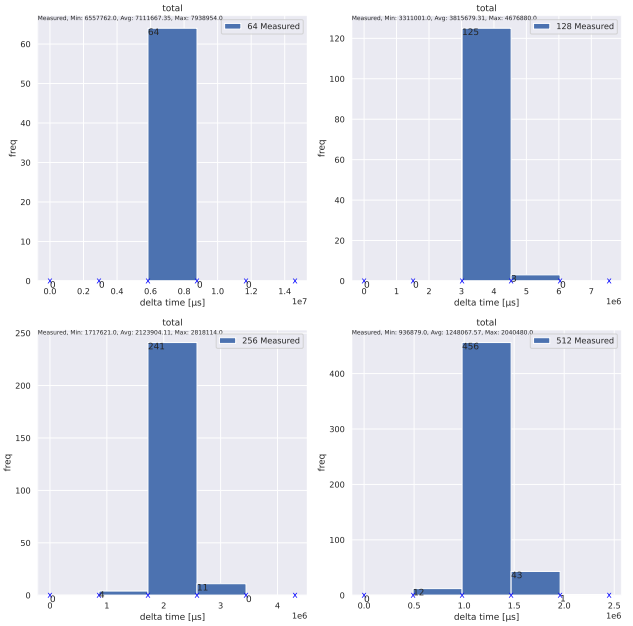

図 2: POISSON_MPI, 並列数 64, 128, 256, 512 におけるデルタ時間の和の分布

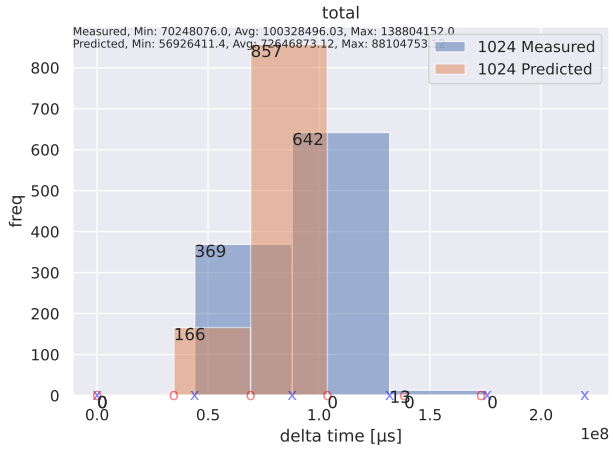

図 3: NPB-IS, 並列数 1,024 におけるデルタ時間の和の分布（赤：予測値，青：実測値）

使用した 4 つのヒストグラムを**図 2**, **図 4**, **図 6** に示す．

図 5 より，LULESH では ScalaExtrap の通信タイミング予測手法によって実測値のヒストグラムをほぼ正確に再現することができた．これは，図 6 に示すように，いずれの並列数においてもランク毎のデルタ時間の和のばらつきが小さかったためと考えられる．

一方，図 1 と図 3 より，POISSON_MPI と NPB-IS ではビンの下限/上限値のズレが大きく，また各ビンの頻度もあまり正確に予測できなかった．これは，図 2 と図 4 に示すように，並列数が増加するにつれてランク毎のデルタ時間の和のばらつきが大きくなったことが原因と考えられる．特に予測対象の並列数である 1,024 ではピークのビン以外のビンの割合が急激に増加しており，ScalarExtrap の通信タイミング予測手法はこの傾向をうまく捉えることができなかった．

5.2 実行区間毎のデルタ時間の予測精度

実行区間毎にデルタ時間の予測を行い，ヒストグラムを作成することでその予測精度を評価した．NPB-IS の中で予測精度が特に高かった区間と低かった区間の 2 つの結果を**図 7**~**10** に示す．

図 7~8 より，予測精度の高かった区間では全ての並列数においてほぼ同じ形のヒストグラムが得られた．デルタ時間は並列数によらずほぼ同じであり，1 ランクだけがおよそ 60 µs，それ以外のランクが 0–25 µs の範囲のビンに含まれていた．このように，並列数によらずほぼ同じ形のヒ

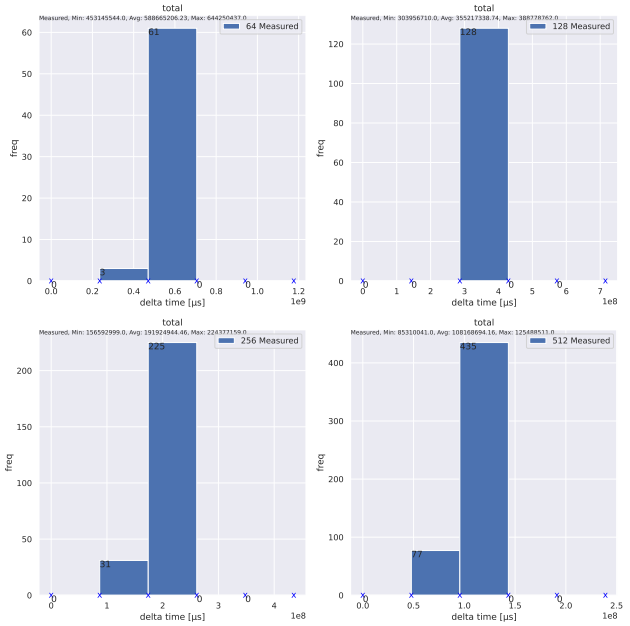

図 4: NPB-IS, 並列数 64, 128, 256, 512 におけるデルタ時間の和の分布

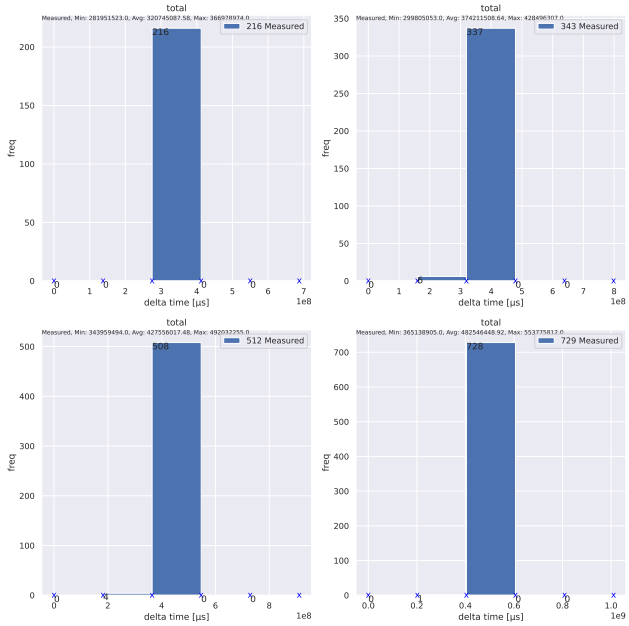

図 6: LULESH, 並列数 216, 343, 512, 729 におけるデルタ時間の和の分布

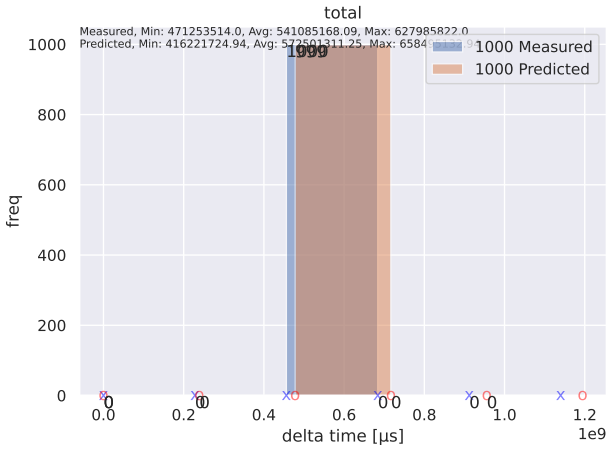

図 5: LULESH, 並列数 1,000 におけるデルタ時間の和の分布（赤：予測値，青：実測値）

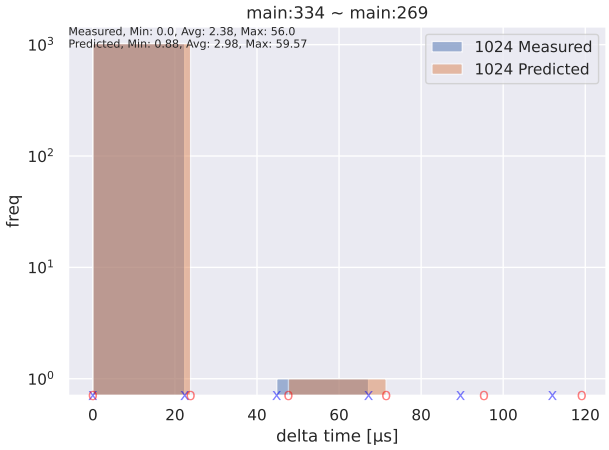

図 7: NPB-IS の main:334-main:269, 並列数 1,024 におけるデルタ時間の分布

ストグラムだったため，ScalaExtrap の通信タイミング予測手法は並列数 1,024 におけるデルタ時間のヒストグラムを正確に予測できたと考えられる．

一方，図 9~10 より，予測精度の低かった区間ではヒストグラムの形が並列数によって大きく異なる．また，デルタ時間は並列数に対して反比例関係を示す区間であったと考えられるが，並列数が増加するにつれてヒストグラムの分布が広がっており，単純な反比例関係ではないことがわかる．これは並列数の増加に対してデルタ時間のばらつきが大きくなっていることを意味しているが，ScalaExtrap の通信タイミング予測手法はこのような傾向をうまく捉えることができていない．予測したヒストグラムでは，実測値をもとに作成したヒストグラムに対して，局所的にビンが存在するような形となった．

6. おわりに

6.1 まとめ

本稿では，小規模実行時の通信トレースの解析結果に基づいて大規模実行時の通信タイミングを予測する既存手法について評価を行った．プログラム中のデルタ時間の和の最大値の予測精度を評価した結果，1 種類のプログラムでは予測精度が 95%以上であったが，他 2 種類のプログラムでは 65%以下となった．この結果から，ScalaExtrap に実装されている 4 種類のモデルでは，任意のプログラムに対してデルタ時間の和の最大値を正確に予測するのは困難であることが確認できた．

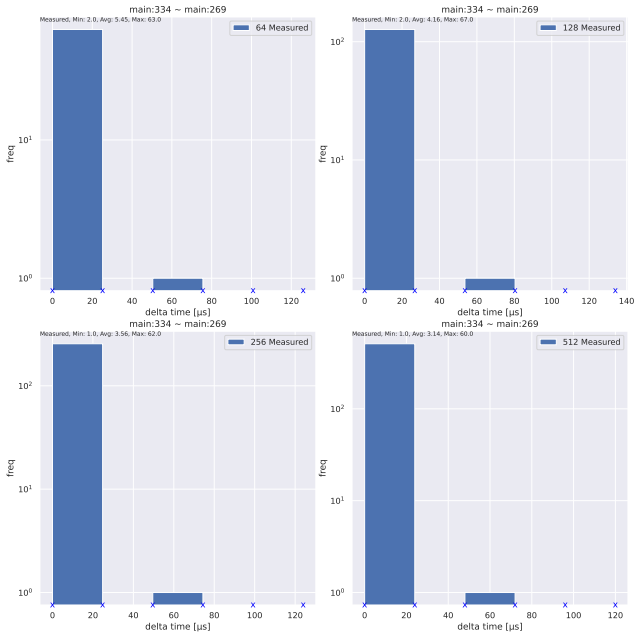

図 8: NPB-IS の main:334-main:269, 並列数 64, 128, 256, 512 におけるデルタ時間の分布

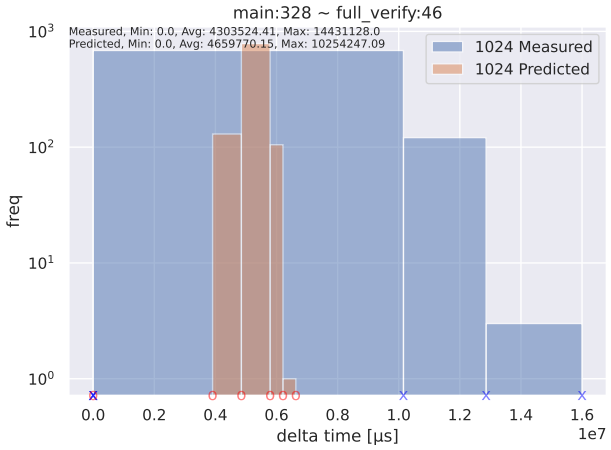

図 9: NPB-IS の main:328-full_verify:46, 並列数 1,024 におけるデルタ時間の分布

また，各ランクのデルタ時間の和からヒストグラムを作成し，予測したヒストグラムと実測値をもとに作成したヒストグラムを比較した．その結果，並列数が増加するにつれてデルタ時間の和のばらつきが大きくなるようなケースにおいて，既存手法は正確なヒストグラムの予測が困難であることがわかった．

最後に，実行区間毎のデルタ時間のヒストグラムを作成し，予測したヒストグラムと実測値をもとに作成したヒストグラムを比較した．その結果，非常に高い精度で予測ができる区間と，予測精度が低い区間がプログラム中に混在することがわかった．

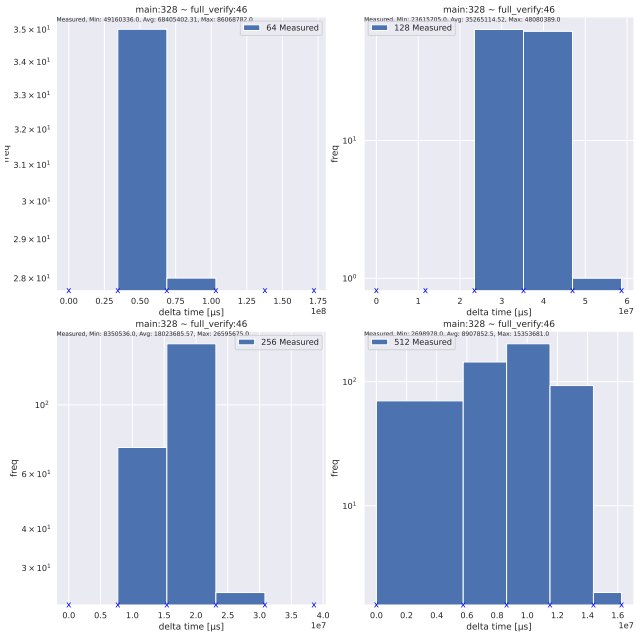

図 10: NPB-IS の main:328-full_verify:46, 並列数 64, 128, 256, 512 におけるデルタ時間の分布

6.2 今後の展望

今後は静的/動的解析による MPI プログラムの通信タイミング予測手法を開発する予定である．新たに開発する手法では，小規模実行時の通信トレースの解析（動的解析）結果だけでなく，ソースコード解析（静的解析）結果も通信タイミング予測に利用することで，既存手法では予測困難だったアプリケーションも含めて様々な並列プリケーションに対して通信トレース予測を可能にする予定である．また，今回の評価結果を踏まえて，上記の手法で使用するデルタ時間の新しい予測モデルについても検討する予定である．

謝辞 本研究の一部は JSPS 科研費 JP20H04193 の助成を受けたものである．本研究は東京工業大学のスパコン TSUBAME3.0 を用いて行った．

参考文献

[1] Bahmani, A. and Mueller, F.: Scalable Performance Analysis of Exascale MPI Programs Through Signature-based Clustering Algorithms, *Proceedings of the 28th ACM International Conference on Supercomputing (ICS'14)*, pp. 155–164 (2014).

[2] Barszcz, E., Barton, J., Dagum, L., Frederickson, P., Lasinski, T., Schreiber, R., Venkatakrishnan, V., Weeratunga, S., Bailey, D., Bailey, D., Browning, D., Browning, D., Carter, R., Carter, R., Fineberg, S., Fineberg, S., Simon, H. and Simon, H.: The nas parallel benchmarks, Technical report, The International Journal of Supercomputer Applications (1991).

[3] Chen, H., Chen, W., Huang, J., Robert, B. and Kuhn, H.: MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters, *Proceedings of the 20th Annual International Conference on Supercomputing*

(ICS'06), pp. 353–360 (2006).
[4] Karlin, I., Keasler, J. and Neely, R.: LULESH 2.0 Updates and Changes, Technical Report LLNL-TR-641973 (2013).
[5] McPherson, A. J., Nagarajan, V. and Cintra, M.: Static Approximation of MPI Communication Graphs for Optimized Process Placement, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Vol. 8967, pp. 268–283 (2014).
[6] Miwa, S., Laguna, I. and Schulz, M.: PredCom: A Predictive Approach to Collecting Approximated Communication Traces, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 32, No. 1, pp. 45–58 (2021).
[7] Miwa, S. and Nakamura, H.: Profile-Based Power Shifting in Interconnection Networks with On/Off Links, *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15)*, pp. 37:1–37:11 (2015).
[8] Noeth, M., Ratn, P., Mueller, F., Schulz, M. and de Supinski, B. R.: ScalaTrace: Scalable Compression and Replay of Communication Traces for High-performance Computing, *Journal of Parallel Distributed Computing*, Vol. 69, No. 8, pp. 696–710 (2009).
[9] Preissl, R., Schulz, M., Kranzlmüller, D., de Supinski, B. R. and Quinlan, D. J.: Using MPI Communication Patterns to Guide Source Code Transformations, *Proceedings of the International Conference on Computational Science 2008*, pp. 253–260 (2008).
[10] Preissl, R., Schulz, M., Kranzlmüller, D., de Supinski, B. R. and Quinlan, D. J.: Transforming MPI Source Code Based on Communication Patterns, *Future Generation Computer Systems*, Vol. 26, No. 1, pp. 147–154 (2010).
[11] Quinn, M.: *Parallel codeming in C with MPI and OpenMP*, McGraw-Hill (2004).
[12] Shende, S. S. and Malony, A. D.: The TAU Parallel Performance System, *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 287–311 (2006).
[13] Wu, X. and Mueller, F.: ScalaExtrap: Trace-Based Communication Extrapolation for Spmd Programs, New York, NY, USA, Association for Computing Machinery (2011).
[14] Xue, R., Liu, X., Wu, M., Guo, Z., Chen, W., Zheng, W., Zhang, Z. and Voelker, G.: MPIWiz: Subgroup Reproducible Replay of Mpi Applications, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*, pp. 251–260 (2009).
[15] Zhai, J., Sheng, T., He, J., Chen, W. and Zheng, W.: FACT: Fast Communication Trace Collection for Parallel Applications through Program Slicing (2009).
[16] 東京工業大学学術国際情報センター：TSUBAME とは，Tokyo Institute of Technology（オンライン），入手先 ⟨https://www.gsic.titech.ac.jp/tsubame⟩（参照 2021-10-31）.