LULESHを対象とした関数コール回数予測

有馬 海人[1,a)] 長谷川 健人[1] 三輪 忍[1] 八巻 隼人[1] 本多 弘樹[1]

概要：近年、スーパーコンピュータのシステム規模やその上で実行されるアプリケーションの規模は大幅に増加しており、超大規模な並列アプリケーションのプロファイリングを現実的なコスト（計算資源と所要時間）で行うのは難しくなっている。この問題に対して、アプリケーションを小規模実行した際の結果から大規模実行した際の結果を予測するツールである Extra-P が開発されている。Extra-P は、一般にアプリケーションや関数の総実行時間のスケーラビリティ予測に用いられるが、原理的には総実行時間以外のメトリクスに対するスケーラビリティ予測にも利用できる。そこで、CORAL-2 ベンチマークプログラムの LULESH を対象に Extra-P を関数コール回数のスケーラビリティ予測に応用したところ、Extra-P は非常に高い予測精度を示すことが確認できた。また、関数の総実行時間を Extra-P で直接予測するのではなく、関数コール回数とコールあたりの実行時間に分離した上でそれぞれを Extra-P で予測する方がスケーラビリティ予測の精度が高くなることも確認できた。本稿では以上の結果を報告する。

キーワード：MPI, プロファイル, 関数コール回数, モデル

1. 序論

並列アプリケーションのプロファイリングは、性能解析や性能チューニングなどを目的として、スーパーコンピュータのユーザ間で広く行われている。プロファイリングは、通常、解析対象のアプリケーションを解析対象のシステム上で実行することによって行われる。ところが、超大規模な並列アプリケーションの場合は実行そのものに膨大な時間と計算資源を必要とするため、そのようなアプリケーションのプロファイリングは一般に困難である。

上記の問題に対し、並列アプリケーションの性能解析ツール Extra-P が開発されている [4]。Extra-P はアプリケーションを小規模実行した際の結果から各メトリクスのスケーラビリティモデルを構築し、構築したモデルを用いて同アプリケーションを大規模実行した際の当該メトリクスの値を予測する。この方法により、大規模実行に必要な計算資源と時間を節約しつつ、アプリケーションを大規模実行した際にプロファイラが出力する種々のメトリクス値を見積もる。Extra-P は原理的には任意のメトリクスに対するスケーラビリティ予測が可能であるが、これまでに有効性が確認されているのはアプリケーションや関数の実行時間、消費エネルギーや MPI 関数の通信量などの一部のメトリクスにとどまっている [4]。

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

そこで我々は、CORAL-2 ベンチマークプログラムの LULESH を対象に、関数コール回数のスケーラビリティ予測における Extra-P の有効性を確認する。また、関数コール回数予測の応用例として、関数の実行時間を Extra-P で直接予測するのではなく、関数コール回数とコールあたりの実行時間に分離した上でそれぞれを Extra-P で個別に予測する手法を提案する。評価の結果、Extra-P は十分高い精度で関数コール回数を予測可能なこと、および、提案手法は従来手法よりも高い精度で関数の実行時間を予測可能なことがわかった。以下本稿ではこれらの結果を中心に報告する。

2. 研究背景

2.1 プロファイリング

並列アプリケーションのプロファイリングは性能解析やチューニングを目的として高性能計算分野で広く行われている。プロファイリングでは、一般に、実行された関数の名称、関数のコール回数、 関数の実行時間などのアプリケーションの動的な情報を取得する。さらには、CPU のキャッシュメモリのアクセス数やミス数などのハードウェアカウンタ値を追加で取得することも可能である [3]。

プロファイルの取得は、一般に、まずコンパイラあるいは動的ライブラリを利用して解析対象のアプリケーションに対して計測用のコードを挿入し、その後、解析対象のシステム上で当該アプリケーションを実行することによって

ⓒ 2022 Information Processing Society of Japan

行われる。計測用のコードを挿入するためのツールとしてTAU や Score-P などのツールがある。

後述する Extra-P とは異なり、プロファイリングによって取得された情報は実測値である。その一方で、プロファイルを取得するにあたっては実際に解析対象のアプリケーションを実行する必要があるため、プロファイルの取得に必要なコスト（時間と計算資源）は相対的に大きい。情報を取得する関数を限定するなどのプロファイリングコストを削減する技術も開発されているが [2][3]、解析対象のアプリケーションを解析対象の規模で実行する必要がある点に変わりはない。以上まとめると、プロファイリングの利点は詳細で正確なプロファイルを取得できることであるが、欠点は取得のためのコストが大きいことである。

2.2 Extra-P

並列アプリケーションの性能解析ツールである Extra-P[4] は、アプリケーションを小規模実行した際の実行結果から実行プロセス数や問題サイズをパラメータとするスケーラビリティモデルをメトリクスごとに構築し、それらを用いて同アプリケーションを大規模実行した際のメトリクス値を予測する。

m 個のアプリケーションパラメータを入力変数とする予測において Extra-P が用いるモデル式の一般形を式 (1) に示す。

$$f(x_1, ..., x_m) = \sum_{k=1}^{n} c_k \cdot \prod_{l=1}^{m} x_l^{i_{kl}} \log_2^{j_{kl}}(x_l) \qquad (1)$$

式 (1) は n 個の項の重み付き和となっており、n 個の項のそれぞれが m 個のパラメータのべき乗と対数のべき乗の積で表される。c_k は回帰分析によって決定する係数である。指数 i_{kl} と j_{kl} は回帰分析に使用するモデル式（Extra-P では hypothesis と呼ばれる）を決めるパラメータであり、i_{kl} と j_{kl} それぞれの集合 I, J はユーザが指定できる。

Extra-P はアプリケーションを大規模実行した際の実行時間の見積もりを低コストで行うことができるが、プロファイリングとは異なり、出力する実行時間は実測値ではなく、モデルによって予測された近似的な値である。

3. Extra-P による関数コール回数予測

3.1 関数コール回数と性能チューニング

関数コールには性能オーバーヘッドが存在する。そのため、多くの回数コールされる関数が性能上のボトルネックとなっている場合には、関数のコール回数を減らすことでアプリケーションの性能を改善できる可能性がある。関数のコール回数はアプリケーションのソースコードの解析によって求めることができる場合もあるが、アプリケーションを実際に実行してみなければわからない場合もある。例えば、関数が IF 文や収束計算などの不定回数ループの中でコールされる場合には、実際に入力データを与えて実行しなければ、その関数のコール回数はわからない。

コール回数の多い関数に対する性能チューニング技術の 1 つであるインライン展開では、関数内で実行される命令列を caller 側に展開することによって関数コールそのものをプログラム中から削除する。関数のインライン展開によってコードサイズは増加するが、関数コールによる性能オーバヘッドは削減されるため、結果としてアプリケーション性能が向上する可能性がある。そのため、コール回数の多い関数を特定することはアプリケーションの性能チューニングにおいて重要である。

表 1: TSUBAME3.0 のシステム構成

総理論演算性能	12.15PFlops（倍精度）
総ノード数	540
総主記憶容量	138,240GB
ネットワークトポロジ	フルバイセクション・ファットツリー

3.2 評価方法

3.2.1 評価内容

Extra-P を関数コール回数予測に応用した場合の予測精度、ならびに、関数コール回数の取得に必要なコストを評価する。具体的には、TAU を用いて複数の小さな規模でアプリケーションを実行することにより各規模における関数コール回数を取得し、取得した関数コール回数と取得時の実行規模を入力として Extra-P によりアプリケーション内の各関数のコール回数のスケーラビリティモデルを作成する。そして、上述の方法で作成したスケーラビリティモデルを用いてモデル作成時よりも大きな規模でアプリケーションを実行した際の関数コール回数予測を行い、同規模で TAU を用いてアプリケーションを実行することにより関数コール回数を取得する場合と比較する。

Extra-P によるモデル構築時には、modeler オプションに multi-parameter を指定した。 多項式の指数には-1,0,1,2,3 を、対数の指数には 0 および 1 を、force_combination_exponents オプションおよび allow_negative_exponents オプションには True を指定した。

3.2.2 評価環境

以下では、実験に使用するシステム、ツール、ベンチマークプログラムについて述べる。

東京工業大学学術国際情報センターのスーパーコンピュータ TSUBAME3.0 を本実験に使用する。TSUBAME3.0 は 540 台の計算ノードにより構成され、1 ノード当たり CPU(Intel Xeon E5-2680 V4) が 2 基搭載されている。TSUBAME3.0 のシステム構成を表 1、ノード構成を表 2 に示す。

アプリケーションの関数コール回数の実測には TAU を

表 2: TSUBAME3.0 のノード構成

CPU	プロセッサ名	Intel Xeon E5-2680 V4
	物理（論理コア数）	14(28)
	動作周波数	2.40GHz
Memory	容量	256GB
	メモリ帯域幅	153.6GB/s
GPU	プロセッサ名	NVIDIA Tesla P100

表 3: モデル構築および予測対象の実行規模

	モデル構築時	予測時
プロセス数	8, 27, 64, 125, 216, 343	512, 729, 1000
イテレーション数	8, 16, 32, 64, 128	256, 512, 1024
メッシュの大きさ	16, 24, 32, 48	64, 96, 128

使用する。TAU は、C 言語や Python などで記述された並列プログラムの性能分析を行うために開発された、包括的なプロファイリングツールキットである [3]。

評価対象のアプリケーションとして、CORAL-2 ベンチマークプログラムの LULESH [1] を利用する。LULESH は解析的な答えを持つセドフブラスト問題を解くプログラムである。LULESH には CPU 向けの実装と GPU 向けの実装があるが、今回の実験では MPI を用いた CPU 向けの実装を利用した。

LULESH の実行時の制限として、実行時プロセス数が立法数（n の 3 乗）である必要がある。また、問題サイズを表すパラメータとして、計算ステップ（イテレーション）数とメッシュの大きさを今回使用した。

モデル構築時と予測時の実行規模を表 3 にまとめる。モデル構築時はプロセス数が 6 通り、イテレーション数が 5 通り、メッシュの大きさが 4 通りの計 120 通りの組み合わせで実行し、得られた計 120 通りのコール回数を Extra-P に入力して各関数のコール回数モデルを作成した。また予測は、モデル構築時よりも大きなプロセス数が 3 通り、イテレーション数が 3 通り、メッシュの大きさが 3 通りの計 27 通りの組み合わせに対して行った。

3.2.3 評価項目

評価は以下の 2 つの項目に対して行う。

モデルの適合度　モデル構築に使用したデータに対するモデルの適合度を評価する。評価には平均絶対パーセント誤差 (MAPE) を用いる。平均絶対パーセント誤差はモデルが出力する値 (F_t) と観測データ値 (A_t) の絶対誤差率の平均であり、式 (2) で表すことができる。

$$MAPE = \frac{100}{N} \sum_{t=1}^{N} \left| \frac{A_t - F_t}{A_t} \right| \quad (2)$$

モデルの予測精度　予測対象の実行規模において TAU で取得した関数コール回数に対するモデルの予測精度を評価する。各関数に対する予測精度の評価には以下の式で表される絶対誤差率（APE）を使用する。

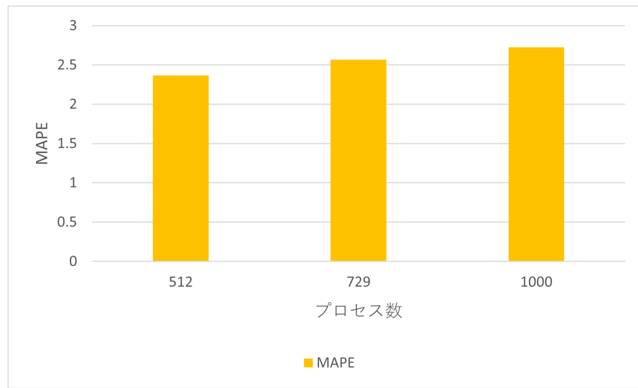
表 4: 関数コール回数のモデル適合度

関数名	MAPE
.TAU_application	1.11E-13
MPI_Allreduce()	4.06E-14
MPI_Barrier()	1.11E-13
MPI_Comm_rank()	6.45E-14
MPI_Comm_size()	1.11E-13
MPI_Finalize()	1.11E-13
MPI_Init()	1.11E-13
MPI_Irecv()	28.14179732
MPI_Isend()	28.14179732
MPI_Reduce()	1.11E-13
MPI_Wait()	28.14179732
MPI_Waitall()	3.22E-14
Real_t_CalcElemVolume()	0.000492255
StrToInt	1.11E-13
int_main()	1.11E-13
void_CalcKinematicsForElems()	0
void_CommMonoQ()	0
void_CommRecv()	3.22E-14
void_CommSBN()	1.70E-14
void_CommSend()	3.22E-14
void_CommSyncPosVel()	0
void_Domain::BuildMesh()	1.11E-13
void_Domain::CreateRegionIndexSets()	1.11E-13
void_Domain::Domain()	1.11E-13
void_Domain::SetupBoundaryConditions()	1.11E-13
void_Domain::SetupCommBuffers()	1.11E-13
void_Domain::SetupElementConnectivities()	1.11E-13
void_Domain::SetupSymmetryPlanes()	1.11E-13
void_Domain::D̃omain()	1.11E-13
void_InitMeshDecomp()	1.11E-13
void_ParseCommandLineOptions()	1.11E-13
void_VerifyAndWriteFinalOutput()	8.13E-05
平均	2.63

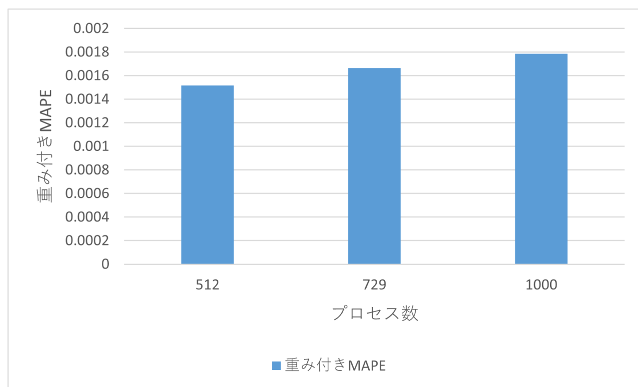
$$APE = 100 \times \left| \frac{A_t - F_t}{A_t} \right| \quad (3)$$

アプリケーション内では多数の関数が実行されるため、ある規模における関数コール回数予測の評価指標として全関数の平均絶対パーセント誤差 (MAPE) および重み付き平均絶対パーセント誤差 (重み付き MAPE) を用いる。重み付き平均絶対パーセント誤差は絶対誤差率の関数コール回数による加重平均であり、式 (4) で表すことができる。式 (4) において c_t , c_{sum} はそれぞれ TAU を用いて取得した関数コール回数と関数コール回数の総和（どちらも実測値）を表している。

$$重み付き\ MAPE = \frac{100}{c_{sum}} \sum_{t=1}^{N} c_t \times \left| \frac{A_t - F_t}{A_t} \right| \quad (4)$$

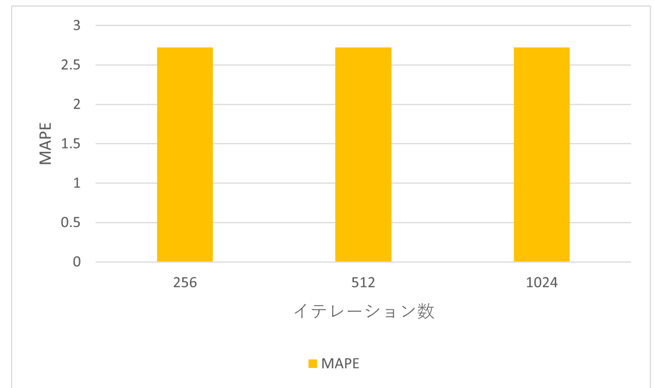


(a) MAPE

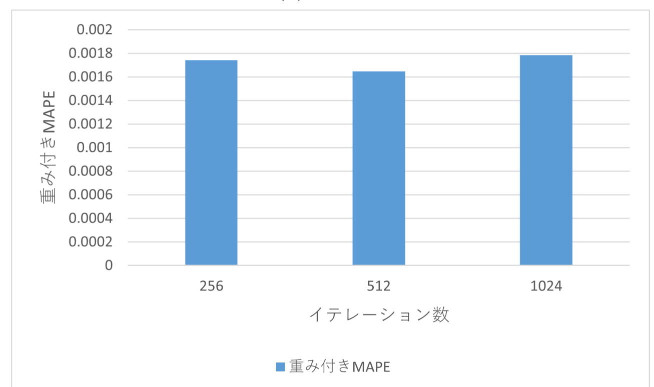


(b) 重み付き MAPE

図 1: 異なるプロセス数に対する予測精度



(a) MAPE



(b) 重み付き MAPE

図 2: 異なるイテレーション数に対する予測精度

3.3 評価結果

3.3.1 モデルの適合度

表 4 に関数毎のモデルの適合度をまとめる。MPI_Irecv() などの一部の MPI 関数を除き、ほとんどの関数で高い適合度を得ることができた。一部の MPI 関数の適合度が低い理由は、これらの MPI 関数はプログラム中の複数の場所から呼び出されており、プロセス数や問題サイズとコール回数との関係が複雑なことから、2.2 節で述べたモデルで表現できなかったことが原因と考えている。

3.3.2 モデルの予測精度

図 1 は異なるプロセス数に対して各関数のコール回数を予測した場合の予測精度である。グラフの横軸はプロセス数を表しており、図 1 (a) の縦軸は MAPE、図 1 (b) の縦軸は重み付き MAPE を表している。なお、予測対象のイテレーション数は 1,024、メッシュサイズは 128 とした。

図より、MAPE および重み付き MAPE のいずれの評価指標においても Extra-P は非常に高い予測精度を示している。MAPE と重み付き MAPE の値は、最良（プロセス数 512）の場合はそれぞれ 2.36 と 0.00151、最悪（プロセス数 1,000）の場合でもそれぞれ 2.72 と 0.00178 であった。

図 2 は異なるイテレーション数に対して各関数のコール回数を予測した場合の予測精度である。予測対象のプロセス数は 1,000、メッシュサイズは 128 としている。図より、いずれのイテレーション数に対しても Extra-P は高い精度で関数コール回数を予測することが可能である。最悪の場合でも、MAPE と重み付き MAPE の値はそれぞれわずか 2.72 と 0.00178 であった。

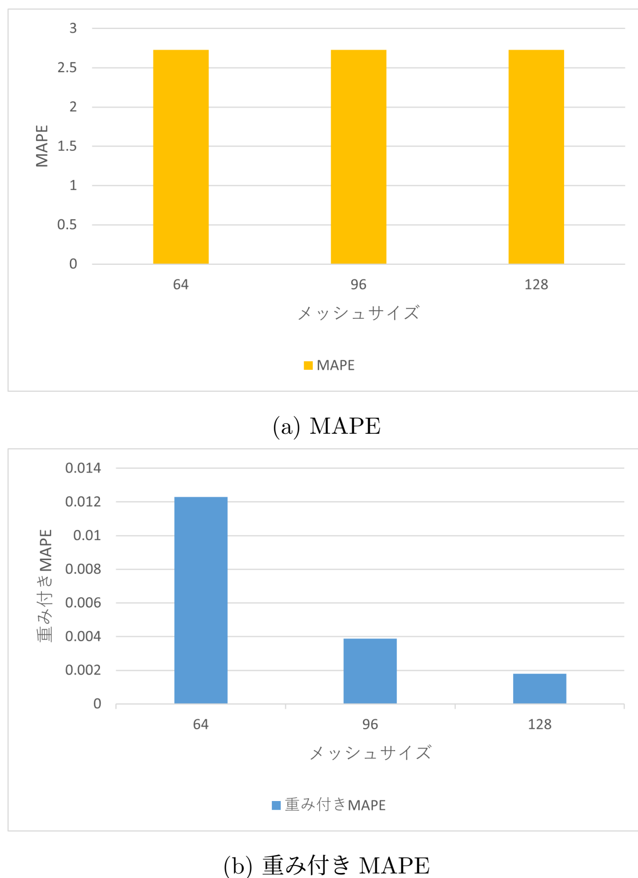
図 3 は異なるメッシュサイズに対して各関数のコール回数を予測した場合の予測精度である。予測対象のプロセス数は 1,000、イテレーション数は 1,024 とした。グラフより、いずれのメッシュサイズに対しても Extra-P による関数コール回数予測は高い精度（MAPE と重み付き MAPE の最悪値はそれぞれ 2.72 と 0.0123）を示すことが確認できた。

4. 実行時間予測への応用

前章で述べたように、Extra-P は非常に高い精度の関数コール回数予測モデルを生成できる。本章では、予測により得られた関数コール回数の近似値の応用例として、Extra-P による関数の総実行時間（関数の呼び出しごとの実行時間の総和）のスケーラビリティ予測を行う。

4.1 関数の総実行時間のスケーラビリティ予測

関数の総実行時間のスケーラビリティ予測では、アプリケーション内に含まれるスケーラビリティバグの発見などを目的として、各関数を小規模実行した場合の総実行時間



(a) MAPE

(b) 重み付き MAPE

図 3: 異なるメッシュサイズに対する予測精度

から大規模実行した場合の総実行時間を予測する。本章では上記の予測に Extra-P を利用する。

Extra-P を用いた関数の総実行時間予測は、一般的には以下のようにして行う。まず最初に、TAU などを用いて複数の小さな規模でアプリケーションを実行することにより各規模における各関数の総実行時間を取得する。続いて、取得した関数毎の総実行時間と実行規模の組み合わせを Extra-P に入力し、各関数の総実行時間モデルを得る。最後に、作成したモデルを用いてモデル作成時よりも大きな規模で実行した場合の同関数の総実行時間を予測する。次節で述べる関数コール回数予測と組み合わせた手法と区別するために、以下では上記の手法を直接方式と呼ぶことにする。

4.2 関数コール回数予測との組み合わせ

関数の総実行時間は、理論的にはコール回数とコールあたりの平均実行時間の積で表すことができる。プロセス数や問題サイズなどの実行規模は、関数のコール回数とコールあたりの実行時間それぞれに対して異なる影響を与えると考えられる。そのため、関数の総実行時間を直接予測するのではなく、関数の総実行時間をコール回数とコールあたりの実行時間に分離した上でそれぞれのメトリクスを個別に予測し、それぞれの予測結果を掛け合わせることで予測精度が向上する可能性がある。以下ではこの方法を分離方式と呼ぶことにし、分離方式の有用性を示すことで関数コール回数予測の有用性を示す。

4.3 評価方法

4.3.1 評価内容

プロファイラが出力する各関数の総実行時間には inclusive と exclusive がある。本稿では、inclusive と exclusive それぞれの時間に対して直接方式と分離方式による予測を行い、その予測結果を比較する。

4.3.2 評価環境

本実験には、3.2.2 項で述べたシステム、ツール、ベンチマークプログラムと同じものを使用する。モデル構築時および予測時に実行時規模も表 3 と同様である。

4.3.3 評価項目

3.2.3 項で述べたモデルの適合度と予測精度に加え、本実験ではメトリクス（関数の総実行時間）の取得コストも評価する。

本稿では、メトリクスの取得に必要なコストを「(アプリの実行に使用するコア数) × (コアの使用時間)」と定義する。これは多くのスーパーコンピューティングサービスはポイント制を採用しており、エンドユーザは使用コア数とコアの使用時間の積に応じてポイントを消費するためである。関数の総実行時間予測を行う場合、Extra-P の実行に必要な時間と計算資源、および、構築したモデルを用いて予測を行うための時間と計算資源が追加で必要となる。ただし、これらのコストは前述のコストに比べて極めて小さいため、今回の実験では無視している。

4.4 評価結果および考察

提案手法の評価結果と結果に対する考察を述べる。なお、以下ではすべての規模で実行される関数のみを評価対象とする。

4.4.1 モデルの適合度

モデルの関数毎の適合度を表 5 に示す。表内の各要素の値は MAPE である。表より、分離方式は関数の exclusive 時間を予測する場合において高い予測精度を示す。特に直接方式と比較した場合、分離方式は exclusive 時間の予測において誤差が 30%減少している。これは関数の総実行時間をコール回数とコールあたりの実行時間に分離したことにより、Extra-P による各メトリクスの予測精度が向上したためである。

一方、inclusive 時間の予測においては直接方式と分離方式との間に予測精度の差がほとんどない。これは関数の inclusive な総実行時間は、当該関数のコール回数だけでなく、関数内部でどのような関数が何回実行されるかにも依存するからである。そのため、単純に当該関数のコール回数とコールあたりの実行時間に分離しただけでは、Extra-P

表 5: モデル適合度

関数名	Exclusive		Inclusive	
	直接方式	分離方式	直接方式	分離方式
.TAU_application	4.5	4.5	18.5	47.9
MPI_Allreduce()	64.2	61.1	70.0	54.3
MPI_Barrier()	375.8	413.8	396.7	409.6
MPI_Comm_rank()	13.9	21.1	13.9	21.1
MPI_Comm_size()	14.6	14.6	14.6	14.6
MPI_Finalize()	58.9	76.7	57.0	72.4
MPI_Init()	22.8	23.0	19.2	18.8
MPI_Irecv()	29.9	34.6	30.8	33.9
MPI_Isend()	194.1	173.8	196.5	168.8
MPI_Reduce()	680.7	609.3	711.2	608.4
MPI_Wait()	637.5	676.4	670.2	593.4
MPI_Waitall()	238.4	229.1	235.5	227.9
Real_t_CalcElemVolume()	16.1	15.7	26.7	9.7
StrToInt	4.8	4.6	4.8	4.6
int_main()	7.0	13.1	18.5	47.9
void_CalcKinematicsForElems(	63.5	16.2	20.1	8.7
void_CommMonoQ(	55.8	37.6	337.4	587.2
void_CommRecv(	189.2	22.3	28.3	27.5
void_CommSBN(	411.4	39.8	653.8	647.3
void_CommSend(	847.5	54.3	175.7	174.2
void_CommSyncPosVel(	147.5	83.5	187.4	142.9
void_Domain::BuildMesh()	25.1	29.9	26.8	30.1
void_Domain::CreateRegionIndexSets()	9.2	9.6	8.0	7.4
void_Domain::Domain()	29.7	24.0	16.6	12.9
void_Domain::SetupBoundaryConditions()	61.2	56.7	61.2	56.7
void_Domain::SetupCommBuffers()	25.1	25.1	22.8	22.8
void_Domain::SetupElementConnectivities()	5.0	5.7	5.6	5.1
void_Domain::SetupSymmetryPlanes()	6.6	6.5	6.5	6.3
void_Domain::D̃omain()	587.9	591.6	587.9	591.6
void_InitMeshDecomp()	5.0	5.0	5.0	5.0
void_ParseCommandLineOptions()	25.4	24.5	12.5	12.6
void_VerifyAndWriteFinalOutput()	6.0	6.4	9.1	5.7
平均	152.0	106.6	145.3	146.2

によるコールあたりの実行時間の予測精度が改善しなかったと考えられる。

以下では、exclusive な時間の予測の結果のみを示す。

4.4.2 予測精度

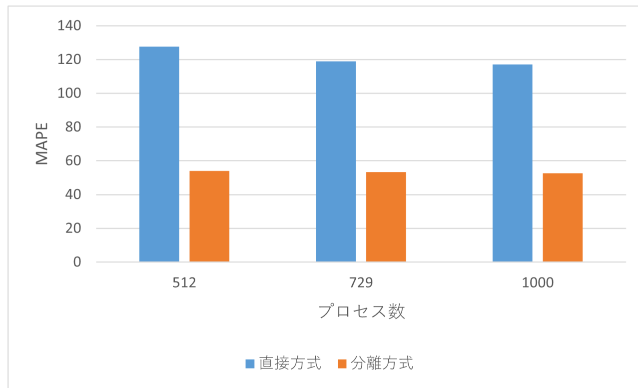
図 4 は異なるプロセス数に対して各関数の exclusive な総実行時間を予測した場合の予測精度である。グラフの横軸はプロセス数を表しており、図 4 (a) の縦軸は MAPE、図 4 (b) の縦軸は重み付き MAPE を表している。なお、予測対象のイテレーション数は 1,024、メッシュサイズは 128 とした。

図より、MAPE と重み付き MAPE いずれの評価指標においても、分離方式は直接方式よりも高い予測精度を示すことがわかる。特にプロセス数が 1000 の場合、分離方式は直接方式に対して MAPE と重み付き MAPE がそれぞれ 57.7%、47.0%改善した。

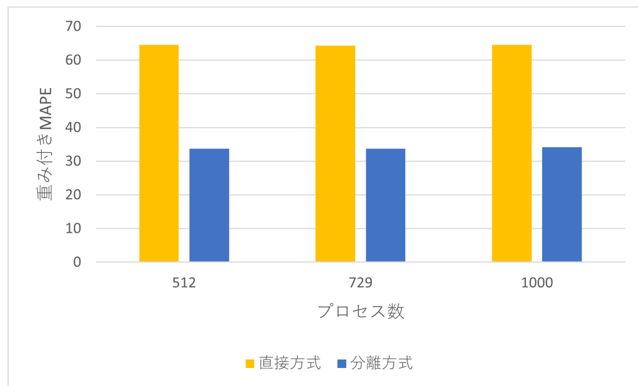
異なるイテレーション数に対して exclusive な時間を予測した場合の予測精度を図 5 に示す。予測対象のプロセス数は 1,000、メッシュサイズは 128 とした。

グラフより、直接方式はイテレーション数が増加するほど予測精度が悪化するのに対し、分離方式はイテレーション数によらずほぼ一定の予測精度を保っていることが確認できる。これは後述する Extra-P の実装上の制約により、総実行時間がイテレーション数に依存する一部の関数のモデル化が直接方式では正しく行えなかったからと考えている。

異なるメッシュサイズに対して exclusive な時間を予測した場合の予測精度を図 6 に示す。予測対象のプロセス数は 1,000、イテレーション数は 1,024 とした。

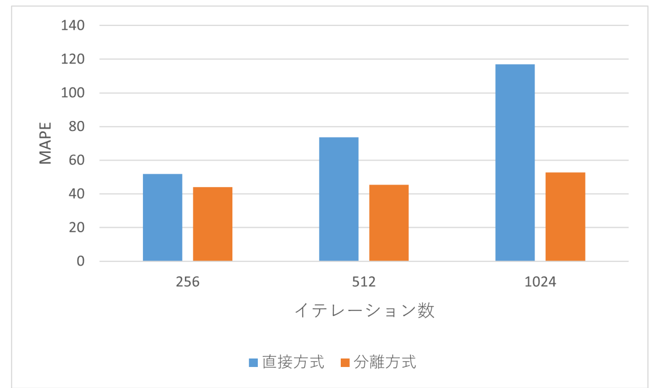


(a) MAPE

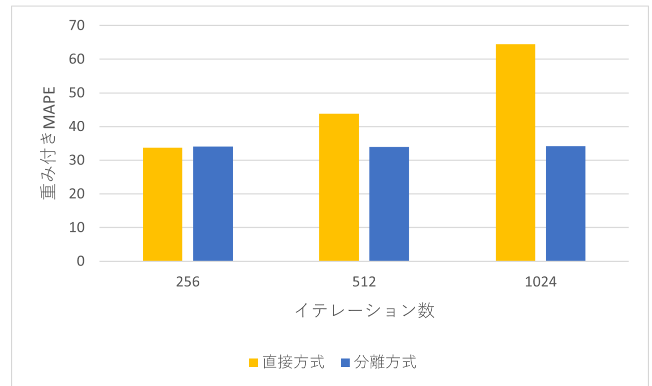


(b) 重み付き MAPE

図 4: 異なるプロセス数に対する予測精度



(a) MAPE



(b) 重み付き MAPE

図 5: 異なるイテレーション数に対する予測精度

グラフより、MAPE と重み付き MAPE いずれの評価指標においても、分離方式は直接方式よりも予測精度が高い。特にメッシュサイズが 128 の場合、分離方式は直接方式に対して MAPE と重み付き MAPE がそれぞれ 54.9%、47.0%改善した。

分離方式が直接方式よりも精度が高くなる原因として、Extra-P の実装上の制約があると考えられる。Extra-P は 2.2 節で述べたモデルに対して回帰分析を行うことでモデルを生成するが、2022 年 10 月現在で公開されているバージョンではその際の項数の上限が 3 となっている。

具体例として直接方式および分離方式で求められた関数 MPI_Comm_rank() のモデル式を挙げる。式 (5) と (6) はそれぞれ、直接方式及び分離方式から求められたモデル式である。

$$1.263516949700165 \times 10^{-7} \times iteration \times size + 6.434085012210024 \times 10^{-8} \times size + 6.03650793650789 \times 10^{-6} \tag{5}$$

$$(8.210259548958283 \times 10^{-7} - \frac{9.167435109616658 \cdot 10^{-6}}{size}) * (9.000000000000002 \times iteration + 5.000000000000079) \tag{6}$$

上記の式の $iteration$ はイテレーション数、$size$ はメッシュサイズを表す。また、式 (6) は $*$ の前後で 2 つのモデルに分かれており、前半部分がコールあたりの実行時間のモデル、後半部分が関数コール回数のモデルとなっている。なお、プロセス数 1,000、イテレーション数 1,024、メッシュサイズ 128 に対する APE は、式 (5) が 97.7、式 (6) が 27.2 であった。

式 (5) の項数は 3 である一方で、式 (6) の展開後の項数は 4 である。そのため、直接方式では、式 (6) と等価な式を推定できなかったと考えられる。これらに代表される項数の制限が Extra-P による直接予測の精度を下げている可能性がある。

4.4.3 コスト

図 7 は関数の総実行時間の取得コストを表したグラフである。グラフの横軸は取得対象の実行規模の数を表しており、縦軸はコストを表している。縦軸は対数軸である点に注意されたい。青線は関数の総実行時間を実際に計測する場合のコストを表し、赤線は直接方式あるいは分離方式を用いて関数の総実行時間予測を行う場合のコストを表している。4.3.3 項で述べたように、今回の評価では Extra-P を用いたモデルの生成/予測に必要なコストを無視しているため、関数の総実行時間予測を行う場合のコストは取得対象の実行規模の数によらず一定である。それに対し、関

ⓒ 2022 Information Processing Society of Japan

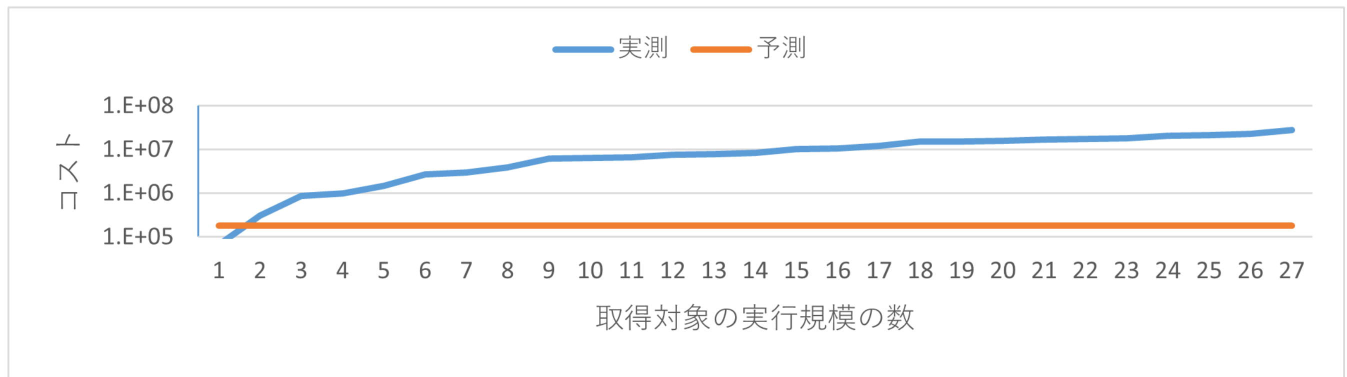
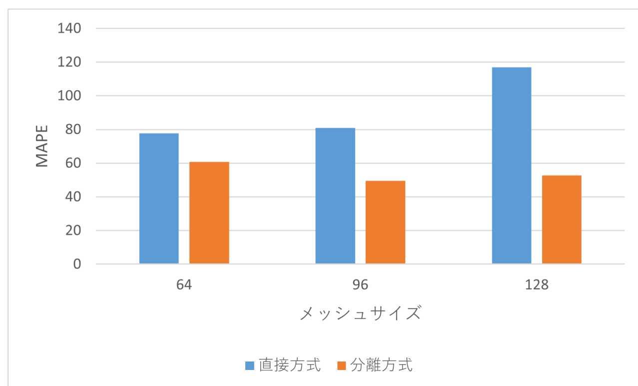
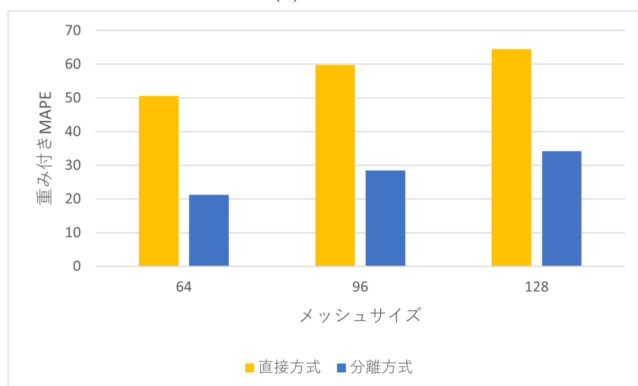


図 7: 取得対象の実行規模の数とコストの関係



(a) MAPE



(b) 重み付き MAPE

図 6: 異なるメッシュサイズに対する予測精度

数の総実行時間を実測する場合には、取得対象の実行規模の数の増加に応じてアプリケーションの実行回数が増えるため、取得コストが増加する。特に取得対象の実行規模が 27 種類の場合、関数の総実行時間予測を行う場合のコストは実測する場合のコストの 0.65%で済む。このように、関数の総実行時間予測は複数の実行規模に対する情報を取得したい場合において有効である。

5. 結論

5.1 まとめ

本稿では、LULESH を対象に、小規模実行した際の結果から大規模実行した際の結果を予測するツールである Extra-P を用いて関数コール回数予測を行った。評価の結果、Extra-P は関数コール回数に対しても、非常に精度の高いスケーラビリティモデルを作成可能なことが確認できた。また、関数コール回数予測の応用例として、関数の総実行時間をコール回数とコールあたりの実行時間に分離した上でそれぞれを個別に予測する方法を評価したところ、関数の総実行時間予測を直接予測する場合よりも高い精度を示すことが確認できた。

5.2 今後の展望

本稿では関数のコール回数予測の応用先として関数の総実行時間予測を対象としたが、メトリクスを関数のコール回数とコールあたりのメトリクスに分離して予測する方法は他のメトリクスの予測にも利用できる可能性がある。そこで、今後は関数コール回数予測の他のメトリクス予測への応用を評価する予定である。また、本稿では LULESH のみを評価対象としていたが、他のベンチマークプログラムでの評価を行う予定である。

謝辞 本研究は JSPS 科研費 JP20H04193 の助成を受けたものです.

参考文献

[1] Karlin, I., Keasler, J. and Neely, R.: LULESH 2.0 Updates and Changes, Technical Report LLNL-TR-641973 (2013).
[2] Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A. D., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B. and Wolf, F.: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir (2011).
[3] Performance Research Lab: TAU, https://www.cs.uoregon.edu/research/tau/home.php.
[4] Technical University of Darmstadt: Extra-P, https://www.scalasca.org/scalasca/software/extra-p/download.html.

ⓒ 2022 Information Processing Society of Japan