Vol.2017-HPC-161 No.14
2017/9/20

疑似MPIトレースプロファイルを用いた通信性能推定手法SCAMPのための疑似トレースファイル作成手法の検討

辻 美和子[1,a)]　李 珍泌[1]　朴 泰祐[2]　佐藤 三久[1]

概要：HPC システムとアプリケーションのコデザインにおいては，将来システムの構成によってアプリケーションの性能がどのように影響されるか調査することが重要である．トレースを用いるネットワークシミュレーションは，その平易さから広く用いられている一方，想定する将来システムのサイズと同じ数のトレースファイルが必要とされる．ゆえに，将来システムが現在のシステムよりも大規模な場合，トレースファイルを直接使用することができない．このようなトレースを用いるネットワークシミュレーションにおけるスケーリング問題を解決するために，われわれは SCAlable Mpi Profiler (SCAMP) と呼ばれる手法を提案している．SCAMP 法では，アプリケーションを実システムで実行して MPI イベントトレースファイルを取得し，得られた実ファイルを複製・加工して疑似 MPI イベントトレースファイルを生成し，これらの疑似ファイルを入力としてネットワークシミュレータを実行して，通信性能を推定する．本稿では，疑似ファイルの生成手法に焦点をあて，ソースコードの中間表現である LLVM-IR の解析に基づき，自動的に疑似ファイルを生成するジェネレータを開発する．

1. はじめに

実アプリケーション実行時に最大限に性能を発揮するような大規模並列システムを設計するために，システムとアプリケーションのコデザインは不可欠である．コデザインは，システムとアプリケーションの双方向的なアプローチであり，システムがアプリケーションの要求を満足するように設計される一方，アプリケーションはシステムに向けて最適化される [2]．コデザインの過程で，アプリケーション性能を最大化するようなシステムを探索する，あるいは複数の候補システムのなかからよりよいシステムを選択するためには，想定される将来システムの上でのアプリケーション性能を推定する必要がある．

本研究では，とくに通信性能の推定に着目し，MPI イベントトレースに基づく通信性能推定を，既存のシステムでは十分な数のトレースファイルが取得できないような大規模将来システムに適用するための方法論として SCAlable Mpi Profiler (SCAMP) を提案している [8]．SCAMP は，既存システムと将来システムのノード数の差を埋めるために，既存のシステムから得られた実トレースファイルを加工して，擬似 MPI イベントトレースファイルを生成し，これを用いてネットワークシミュレータを実行し，通信性能推定を行う方法論である．

先行研究 [8] では，手動で実ファイルの加工を行い擬似ファイルを生成していたが，複数のアプリケーションに対して手動で擬似ファイルを生成することは現実的ではない．そこで，本稿では，SCAMP 法で用いられる疑似ファイル作成手法の自動化について検討し，ソースコードの中間表現である LLVM-IR の解析による疑似 MPI イベントトレースファイルジェネレータのプロトタイプを実装した．

実験を行い，SCAMP 法および疑似 MPI イベントトレースファイルジェネレータを検証するとともに，疑似ファイルの生成コスト，シミュレーションにかかる時間について検討した．実験の結果，疑似 MPI イベントトレースファイルに基づくネットワークシミュレーションはアプリケーション実行時の通信性能について一定の知見を与えるが，実実行と比較して楽観的な見積りとなることが多いことがわかった．比較的単純なソースコードに対して，われわれの疑似ファイルジェネレータは，合理的な時間内に適切な疑似 MPI イベントトレースファイルを生成することができた．

本稿の構成は以下である：まず，2 章で SCAMP 法について概説するとともに，疑似ファイルジェネレータについて述べる．3 章で実験を行い，4 章で関連研究について述

1 理化学研究所計算科学研究機構
RIKEN Advanced Institute for Computational Science
2 筑波大学
Univeristy of Tsukuba
a) miwako.tsuji@riken.jp

ⓒ 2017 Information Processing Society of Japan

べる．5 章で結論を述べる．

2. SCAlable Mpi Profiler (SCAMP) および 疑似 MPI イベントトレースファイル自動生成

本章では，疑似 MPI イベントトレースファイルに基づく大規模ネットワークシミュレーションを可能にする SCAMP 法について概説し，疑似 MPI イベントトレースファイルの自動生成について述べる．また本稿で用いるネットワークシミュレータである SST/macro についても述べる．

2.1 SCAlable Mpi Profiler (SCAMP)

HPC システムとアプリケーションのコデザインにおいては，将来システムにおけるアプリケーションの性能を見積もるために，演算時間および通信時間を推定する必要がある．本研究では，通信性能に焦点を当てている．通信性能推定のために最も広く用いられている手法のひとつは，ソースコードを解析し，MPI 関数呼び出し回数やデータ量を数え上げることによる，マニュアルの推定である．しかしながら，マニュアル推定は非常に多くの労力を要する．もうひとつの一般的に用いられる手法としては，トレースを用いるネットワーク推定がある．これは，(1) 実システムでアプリケーションを実行し，MPI イベントをトレースファイルに記録し (2) 得られたファイルおよび将来システムのパラメータを入力し，ネットワークシミュレーションを行う手法である．

トレースを用いるネットワークシミュレーションの利点は，ほぼ機械的に行うことができ，一度トレースファイルを取得してしまえば，さまざまなパラメータやネットワークトポロジ上での通信性能推定が容易に行えることである．一方で，シミュレーションを行う将来システムのサイズと同数のトレースファイルが必要とされるため，将来システムが現在のシステムよりも大きい場合，トレースファイルを直接用いることができず，シミュレーションが困難である．

われわれの研究では，トレースを用いるネットワークシミュレーションにおけるスケーリング問題を解決するために，SCAlable Mpi Profiler (SCAMP) 法と方法論を提案している [8]．**図 1** に SCAMP 法の概要を示す：

(1) 実システムでアプリケーションを実行し，MPI イベントトレースファイルを得る．1MPI ランクから 1 つのファイルが得られるものとする．
(2) 実ファイルを複製し，将来システムのサイズおよびターゲット MPI ランクにあわせて編集し，疑似 MPI イベントトレースファイルを生成する．これを将来システムの MPI ランク数だけ繰り返す．
(3) 疑似ファイルおよび将来システムのパラメータを入力としてネットワークシミュレータを実行し，通信性能を推定する．

上述の処理によって，将来システムのサイズが現在のものよりも大きい場合であっても，将来システムの通信性能の推定が可能になる．

図 2 に，非周期的境界条件の 2 次元ステンシル計算のハロの交換における MPI イベントトレースファイルの例を示す．この例では，16 ノードを用いて得られた実ファイルから，64 ノードの将来システムに向けて疑似ファイルを生成している．例えば，ランク 13 の実ファイルを加工することでランク 57 やランク 58 のノードに対応する疑似ファイルが得られる．疑似ファイルの生成手法については，2.3 で述べる．

SCAMP 法においては，疑似 MPI イベントトレースファイルを生成するために，送信先プロセスや送信元プロセスのランク，データの個数などの MPI 関数呼出しの引数が，将来システムのノード数の変化に応じてどのように変化するかを考慮する必要がある．そのため，これらの引数が容易に決定できないアプリケーションについては，SCAMP 法を用いることはできない．しかしながら，通信性能に影響を与える引数の多くは，比較的単純なルールによって決定できると考えられる．**図 3** はそのような単純なアプリケーションの例である．このソースコードは，1 次元のステンシルアプリケーションの例であり，通信性能に影響を与える引数は青文字で，これらの変数を計算するために必要な変数は赤文字で，それぞれ示されている．

2.2 SST/macro and DUMPI

本節では，本稿での実験に用いたネットワークシミュレータである SST/macro および DUMPI ライブラリについて述べる．なお，本稿ではこれらのシミュレータおよびライブラリを採用したが，SCAMP の方法論および疑似 MPI イベントトレースファイルの生成手法は，他のトレースを用いるシミュレータに対しても用いることができる．

SST/macro[4], [5] は，サンディア国立研究所で開発されているネットワーク設計のためのツールである．MPI アプリケーションの振る舞いは，SST/macro とともに配布されている DUMPI ライブラリによって記録することができる．DUMPI トレースは SST/macro によってリプレイされ，トレースを取得したものとは異なるシステム上でのシミュレーションが可能である．DUMPI ライブラリで定義されるイベント情報収集と MPI 通信を行う MPI 関数は，ライブラリのリンク時にすウィークシンボルであるもとの MPI 関数と置き換わる．ゆえに，ユーザはソースコードを書き換えることなく，DUMPI ライブラリによる MPI イベントの収集を行うことができる．

図 4 に DUMPI で収集される MPI イベントトレースファイルの例を示す．MPI 関数名，MPI 関数の開始およ

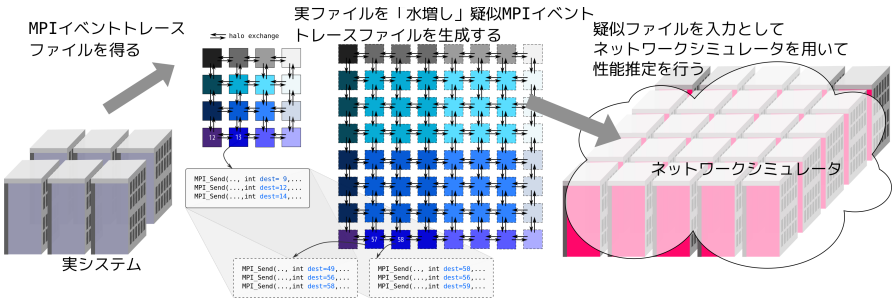


図 1　SCAMP 法の概要

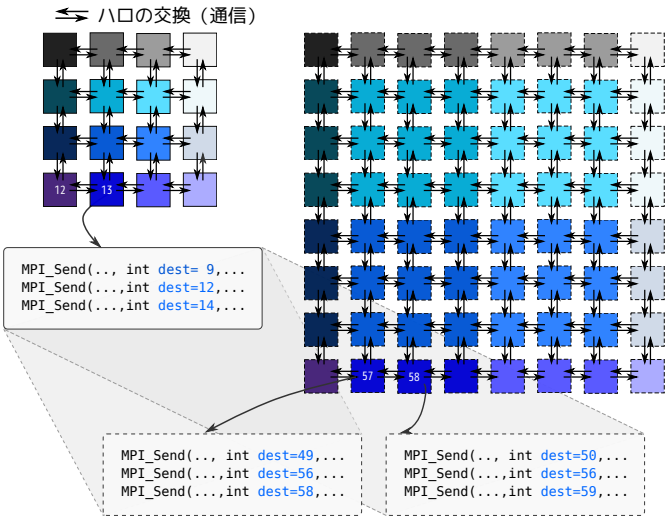


図 2　SCAMP 法の例

```
#define XSIZE 1000

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
xsize = XSIZE/nprocs;
ux = (double *)malloc((xsize+2)*sizeof(double));
uy = (double *)malloc((xsize+2)*sizeof(double));

p1 = (myrank-1+nprocs)%nprocs;
p2 = (myrank+1)%nprocs;

MPI_Send(&(ux[1]),       1, MPI_DOUBLE, p1, ...);
MPI_Recv(&(ux[xsize+1]), 1, MPI_DOUBLE, p2, ...);
MPI_Send(&(ux[xsize]),   1, MPI_DOUBLE, p2, ...);
MPI_Recv(&(ux[0]),       1, MPI_DOUBLE, p1, ...);
for(i=1; i<=xsize; i++){
  uy[i] = (ux[i-1]+ux[i+1]) - 2.0*ux[i];
}
for(i=1; i<=xsize; i++){
  ux[i] = uy[i];
}
```

図 3　ステンシル計算のコード例.

```
MPI_Send entering at walltime 8835524.347071757,
         cputime 0.271717400 seconds in thread 0.
int count=1048576
MPI_Datatype datatype=9 (MPI_INT)
int dest=1
int tag=0
MPI_Comm comm=2 (MPI_COMM_WORLD)
MPI_Send returning at walltime 8835524.347971034,
         cputime 0.272616000 seconds in thread 0.
MPI_Recv entering at walltime 8835524.348989074,
          cputime 0.272629700 seconds in thread 0.
int count=1048576
MPI_Datatype datatype=9 (MPI_INT)
int source=1
int tag=0
MPI_Comm comm=2 (MPI_COMM_WORLD)
MPI_Status status=<IGNORED>
MPI_Recv returning at walltime 8835524.349915164,
         cputime 0.273555000 seconds in thread 0.
```

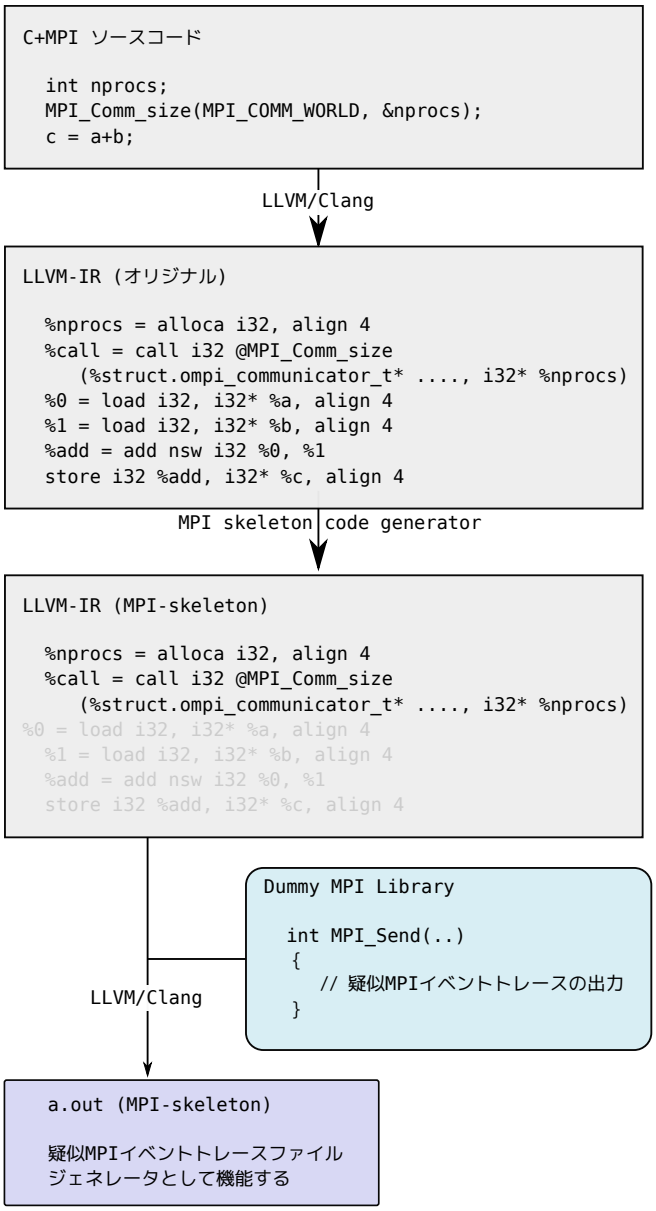
図 4　DUMPI で収集される MPI イベントの例

び終了時間，データの個数，送信元プロセスのランク，タグなどの関数の引数が記録される．本研究では，実際に取得された MPI イベントトレースファイルを編集することで，疑似 MPI イベントトレースファイルを生成する．

2.3　疑似 MPI イベントトレースファイルの自動生成

本節では，コード解析による疑似 MPI イベントトレースファイルの自動生成について述べる．本稿では，オリジナルのソースコードを Clang/LLVM を用いて LLVM-IR と呼ばれる中間表現に変換し，それらを解析することで疑似 MPI イベントトレースファイルを自動生成するジェネレータを開発している．

LLVM プロジェクトでは，コンパイラおよびツールチェーンを開発している [6]．Clang は LLVM の C, C++, Object C 向けのフロントエンドである．LLVM においては，LLVM intermediate representation (LLVM-IR) と呼ばれる中間表現がサポートされている．




```
terminal$ for i in 0,1,...,1023; do
    env MPI_COMM_RANK=$i MPI_COMM_SIZE=1024 \
    ./a.out <input-real-file> <output-pseudo-file>
done
```

図 5 疑似 MPI イベントトレースファイルの自動生成のフロー

図 5 に，LLVM-IR の解析による疑似 MPI イベントトレースファイルの自動生成のフローを示す．まず，もとの C のソースコードを Clang/LLVM コンパイラを用いて，LLVM-IR に変換する．続いて，本稿で開発中の MPI スケルトンコードジェネレータが，LLVM-IR を解析し，MPI 関数呼出し，および MPI 関数呼出しに必要な命令のみを抜き出して，MPI スケルトン LLVM-IR コードを生成する．この MPI スケルトン LLVM-IR コードを，Clang/LLVM コンパイラを用いてコンパイルし，オブジェクトファイルを生成する．オブジェクトファイルは dummy MPI ライブラリとリンクされ，実行可能形式のファイルが生成される．dummy MPI ライブラリは，通信を行うかわりに，対応する疑似 MPI イベントトレースを出力する．この実行可能形式は，疑似 MPI イベントトレースファイルジェネレータとして動作する．将来システムのノード数と MPI ランク番号を環境変数として与えて実行すると，ジェネレータは対応する疑似ファイルを生成する．以下で，疑似ファイルの生成の各ステップの詳細について述べる．

まず，われわれが開発している MPI スケルトンコードジェネレータについて述べる．このツールは，以下のようにして，もとの LLVM-IR を解析し，MPI 関数呼出し，通信性能推定に必要な MPI 関数の引数，これらの引数を計算するために必要な命令を抽出する：

- 各 LLVM-IR の命令を走査し，それが MPI 関数呼出しであれば以下を行う:
  - 通信性能推定に必要な各引数に関して逆定数伝搬を行う．
    引数の値は別のオペランドに対する他の命令によって決定される可能性があるため，すべての必要な命令を再帰的にリストする．
    同様に，もしも MPI 関数呼出しおよび必要な命令が if-then-(else) やループなどのコントロールフローに含まれるとき，これらのコントロールフローも再帰的にリストする．LLVM-IR においては，コントロールフローはブロックとして明示的に定義されている．
    通信性能推定に必要な MPI 関数の引数は，MPI スケルトンコードジェネレータの内部で，各 MPI 関数ごとに明示的に定義される．たとえば，MPI_Allreduce に対しては，データの個数およびコミュニケータが定義され，MPI_Send に対しては，データの個数，送信先プロセスのランク，タグ，コミュニケータが定義されている．

図 6 に MPI_Send の第 4 引数である送信先プロセスのランクを追跡する逆定数伝播の例を示す．この C 言語のソースコードは 図 3 に示したものである．例では，ターゲットとした引数 %12 は，以下のように追跡される：

$\boxed{1}$ %12 → $\boxed{2}$ %p1 → $\boxed{3}$ %rem
→ $\boxed{4}$ %add8
→ $\boxed{5}$ %sub → $\boxed{6}$ %5 → $\boxed{7}$ %myrank
→ $\boxed{5''}$ %6 → $\boxed{6''}$ %nprocs
→ $\boxed{4'}$ %7 → $\boxed{5'}$ %nprocs

追跡処理の過程で複雑な関数呼出し（たとえば MPI 関数の出力をさらに使用する場合など）を含まないのであれば，関数をまたいだ追跡も可能である．

MPI 関数呼出しおよび MPI 関数呼出しに必要な引数の計算のみを行う MPI スケルトン LLVM-IR コードは，コンパイルされ，dummy MPI ライブラリとリンクされて，疑似 MPI イベントトレースファイルジェネレータとして

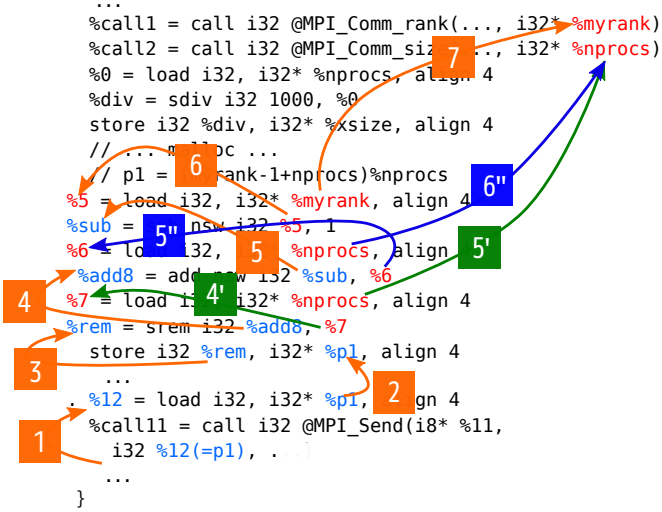


図 6　LLVM-IR 解析に基づく MPI スケルトンの生成

```
int MPI_Send(const void *buf, int count, MPI_Datatype
     datatype, int dest,int tag, MPI_Comm comm)
{
/* for the time stamp, it simply reads the real
     event trace file and writes it without any
     change  */
  fgets(tmp, MAX_BUF, fin);
  fputs(tmp, fout);
   // MPI_Send entering at walltime 8835524.347071757,
      cputime 0.271717400 seconds in thread 0.

   /* for the number of elemements in a buffer,
      it reads the real event trace file but
      ignore it, and writes the count-arguments
      provided by the MPI skeleton code */
  fgets(tmp, MAX_BUF, fin);
  fprintf(fout, "int count=%d\n",count);
  // int count=1048576

  fgets(tmp, MAX_BUF, fin);
  fputs(tmp, fout);
  // MPI_Datatype datatype=9 (MPI_INT)

  fgets(tmp, MAX_BUF, fin);
  fprintf(fout, "int dest=%d\n",dest);
   // int dest=1

  fgets(tmp, MAX_BUF, fin);
  fprintf(fout, "int tag=%d\n",tag);
   // int tag=0

  fgets(tmp, MAX_BUF, fin);
  fputs(tmp, fout);
  // MPI_Comm comm=2 (MPI_COMM_WORLD)
```

図 7　dummy MPI ライブラリにおける MPI_Send 関数．疑似 MPI イベントトレースを出力する

動作する．図 5 で示したように，将来システムのノード数と MPI ランクは環境変数として，実 MPI イベントトレースファイルは引数としてジェネレータに与えられる．

われわれの dummy MPI ライブラリで定義される各 MPI 関数は，それが呼び出された時，通信のかわりに，入力された実 MPI イベントトレースおよび MPI スケルトンコード

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
{
  ......
  *rank=atoi(getenv("MPI_COMM_RANK"));
  fgets(tmp, MAX_BUF, fin);
  fprintf(fout, "int rank=%d\n" ,*rank);
   ......
  return 0;
}
```

図 8　dummy MPI ライブラリにおける MPI_Comm_rank 関数．環境変数から仮想の MPI ランクを取得し，疑似 MPI イベントトレースを出力

により計算された引数を用いて，対応する疑似 MPI イベントトレースを出力する．例として dummy MPI ライブラリにおいて定義される MPI_Send 関数を 図 7 に示す．MPI_Comm_size および MPI_Comm_rank では，MPI イベントの出力に加えて，環境変数で指定され得た仮想サイズおよび MPI ランクを取得する (図 8)．

MPI スケルトンコードから生成された疑似 MPI イベントトレースファイルジェネレータが実行され，MPI 関数呼出しに遭遇したとき，ジェネレータは対応する dummy MPI 関数に引数をわたして，対応する疑似 MPI イベントトレースを出力させる．

現在のわれわれの実装では，C 言語のみをサポートしている．これは，Fortran に向けた LLVM コンパイラである Flang/LLVM が未だに開発途上であることが理由である．

3. 実験

本章では実験によって SCAMP 法および疑似 MPI イベントトレースファイルジェネレータを検証するとともに，疑似ファイルの生成コスト，シミュレーションにかかる時間について議論する．

3.1 実験環境

SCAMP 法の検証のために，少数の実 MPI イベントトレースファイルから多数の疑似 MPI イベントトレースファイルを生成し，シミュレーションを行った結果と，実際に対応する数のノードを用いて行った結果を比較する．比較のために京コンピュータを用いた．京コンピュータの諸元を 1 に示す．

SST/macro はバージョン 7.1.0 を用いた．SST/macro のパラメータについては，予備実験によって簡単に合わせ込みを行った．

3.2 SCAMP 法の性能検証

本節の実験は先行研究 [8] とほぼ同等だが，先行研究ではマニュアルで疑似 MPI イベントトレースファイルを生成したいたのに対して，本節では本稿で開発したジェネレータにより疑似ファイルを自動生成している．

表 1　The specification of the K computer

CPU	Fujitsu SPARC64VIIIfx, 8 core, 2.00 GHz
Memory	16GB , 64GB/s
Cache	L1:32+32KB/core, L2:6MB/core
Network	Tofu (6D mesh/torus) Interconnect 5GB/s x 2

表 2　SST/macro のパラメータ

parameters	value
nic model	simple
switch model	logP
switch ejection latency	3.0us
link bandwidth	4.8GB/s
topology	torus　n, 1, 1

表 3　ステンシルおよび FT における問題サイズ

stencil application				
	# of nodes	Problem size		Problem size
16K	4×4	$(16384 \cdot 4)^2$	24K	$(24576 \cdot 4)^2$
	8×8	$(16384 \cdot 8)^2$		$(24576 \cdot 8)^2$
	16×16	$(16384 \cdot 16)^2$		$(24576 \cdot 16)^2$
	32×32	$(16384 \cdot 32)^2$		$(24576 \cdot 32)^2$
	64×64	$(16384 \cdot 64)^2$		$(24576 \cdot 64)^2$

NPB-FT	
# of nodes	Problem size
4×4	$2048 \times 256 \times 256$
8×8	$2048 \times 512 \times 512$
16×16	$2048 \times 1024 \times 1024$
32×32	$2048 \times 2048 \times 2048$

まず，予備実験として，SST/macro を検証しパラメータの合わせ込みを行うために，ping-pong および MPI_Allreduce のアプリケーションで実験を行った．**図 9** の上および中央に ping-pong および MPI_Allreduece のコードを示す．

ping-pong アプリケーションは，2 ノード間で MPI_Send と MPI_Recv を 100 回繰り返すアプリケーションである．データサイズを 4MB から 1024MB まで変化させている．allreduce アプリケーションは，16 ノードから 512 ノードの上で MPI_Allreduce を 100 回繰り返すアプリケーションである．

実験的にレイテンシおよびネットワークバンド幅のパラメータの合わせ込みを行った．**表 2** にパラメータを示す．

図 10 に，実際および推定された実行時間を示す．SST/macro は ping-pong アプリケーションについては適切な推定を行っているが，allreduce については実際の実行とは異なる結果を返している．これは京コンピュータと SST/macro ではリダクションの実装が異なることが原因であると考えられる．京コンピュータにおいては，ワンワードのリダクションは Tofu バリアインターフェース (TBI)

```
/** ping-pong **/
for(i=0;i<100/2;i++){
  if(myrank==0){
    MPI_Send(buf, n, MPI_INT, 1, ..
  }else{
    MPI_Recv(buf, n, MPI_INT, 0, ..
  }
  usleep(1000);
  if(myrank==1){
    MPI_Send(buf, n, MPI_INT, 0, ..
  }else{
    MPI_Recv(buf, n, MPI_INT, 1, ..
  }
  usleep(1000);
}
```

```
/** allreduce **/
for(i=0;i<100;i++){
  MPI_Allreduce(tmp1, tmp2, 1, MPI_DOUBLE,
                 MPI_SUM, MPI_COMM_WORLD);
  usleep(1000);
}
```

```
/** stencil **/
for(iter=0; iter<16; iter++){
  memcpy(&(b[0][0]), &(a[0][0]),
                 (n+2)*(n+2)*sizeof(double));
  exchange_halo(a, n, myrank, nprocs,
                     up, down, left, right);

  // --- begin the kernel ----------------
  for(i=1; i<n; i++){
    for(j=1; j<n; j++){
      a[i][j] += (c0*b[i][j-1] + c1*b[i][j+1]
               + c2*b[i-1][j] + c3*b[i+1][j]);
  }}
  diff=0.0;
  for(i=1; i<n; i++){
    for(j=1; j<n; j++){
      diff +=  (b[i][j] - a[i][j])
                       * (b[i][j] - a[i][j]);
  }}
  // ---- end the kernel ----------------
  MPI_Allreduce(&diff, &diff2, 1, MPI_DOUBLE,
                     MPI_SUM, MPI_COMM_WORLD);
}
```

図 9　Ping-pong (top), allreduce (middle), およびステンシルアプリケーション

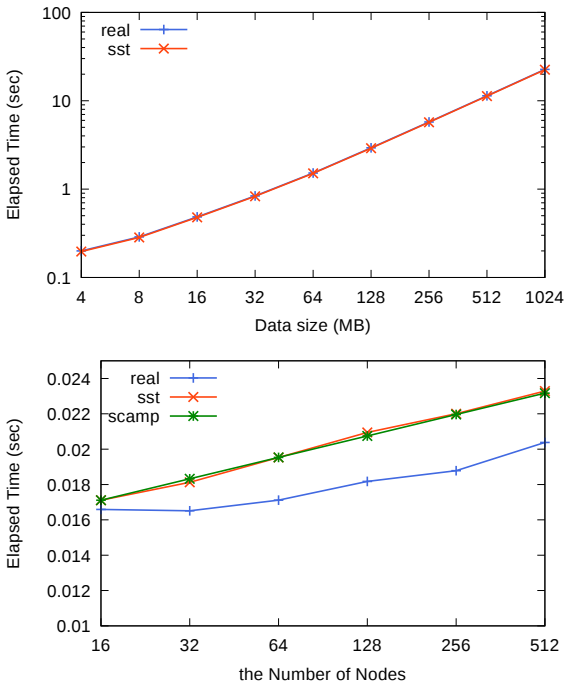


図 10 ping-pong（上）および allreduce（下）アプリケーションに対する実験結果．凡例で “real“ は実測値，“sst“ は 実 MPI イベントトレースファイルを用いた SST/macro による推定結果，”scamp” は最小ノードから生成した疑似 MPI イベントトレースファイルを用いた SCAMP 法＋ SST/macro による推定結果である．ただし，ping-pong は 2 ノードしか用いないアプリケーションのため “scamp“ は行っていない

によってハードウエア的にサポートされている．allreduce において SST/macro と SCAMP+SST/macro を比較すると，SCAMP+SST/macro は SST/macro の結果を適切に再現できている．

次に，SCAMP 法を検証するために，小規模な実実行から得られたトレースファイルから疑似 MPI イベントトレースファイルを生成し，大規模実行時の性能推定を行う SCAMP 法による結果と，実際に対応する規模の実行を実機で行った結果とを比較した．実験には，2 次元のステンシルアプリケーション (図 9, 下) と Nas Parallel Benchmarks[1] から 二次元分割三次元 FFT を行う NPB-FT を C 言語に書き替えたものを用いた．

表 3 に問題サイズおよびノード数を示す．いずれもウィークスケーリングを仮定している．実験の結果は 3 回の独立な試行の平均である．また，簡単のために，1 ノードに 1 つの MPI ランクを割り当てるものとし，スレッド並列は考慮しない．比較のために，実 MPI イベントトレースファイルを用いた SST/macro および，実際の実行を行った．SCAMP+SST/macro の実験では，すべての疑似 MPI イベントトレースファイルは，最小の実験単位である 16 ノードの実実行から得られた実ファイルより 2.3 節で述べた方法で生成された．

図 11–13 にステンシルアプリケーションおよび NPB-FT

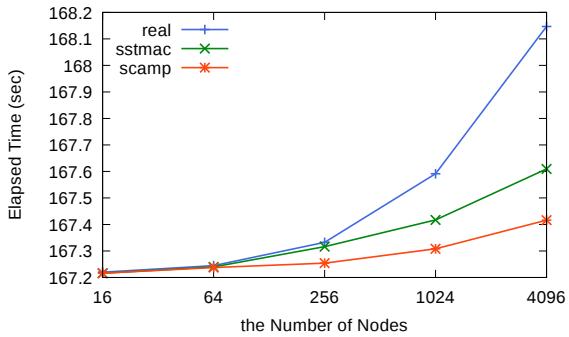


図 11 ステンシルアプリケーション (16K) の実際および推定実行時間

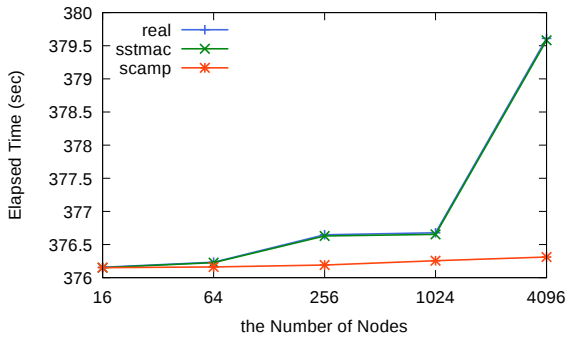


図 12 ステンシルアプリケーション (24K) の実際および推定実行時間

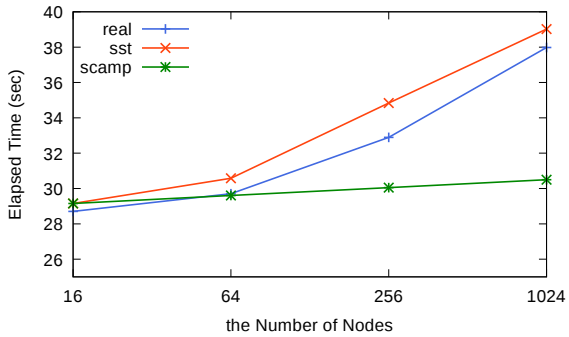


図 13 NPB-FT アプリケーション (24K) の実際および推定実行時間

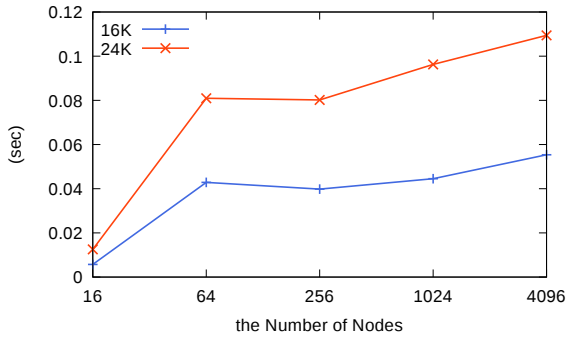


図 14 ステンシルアプリケーションのカーネル部の演算時間のノードごとの差

の実際および推定された実行時間を示す．これらの図において推定時間の差を示すための Y 軸のレンジが，非常に小さいことに注意されたい．

図 11 や予備実験でおこなった allreduce (図 10-bottom)

において，SCAMP+SST/macro は実際の実行に対して適切な推定を行っていないが，シミュレートするシステムと対応する数の実トレースを用いた SST/macro の推定は適切に再現している．ただし，**図 12** および**図 13** においては，SCAMP+SST/macro は，通信性能の傾向についての知見は与えるものの，比較的楽観的な見積もりを与える傾向があることがわかる．これは，ステンシルアプリケーションにおいては，少数のノードの実トレースから得られる疑似 MPI イベントトレースファイルにおいては演算時間のばらつきによる通信の待ち時間などが反映されにくいためである．**図 14** に，各ノード数の実験における，ステンシルアプリケーションのカーネル部の実行時間の差の最大値を示す．ステンシルアプリケーションのカーネル部とは，図 9 において，`begin kernel` と `end kernel` 挟まれているブロックである．ただし，周辺部にあたるノードは無視している．このような演算時間におけるノードごとのばらつきは，OS のジッタなどのさまざまな影響であらわれ，待ち時間として通信時間にあらわれ集積されて実行時間に影響する．しかし，SCAMP では，演算時間は MPI 関数の終了時間および次の MPI 関数の開始時間として暗黙に定義され，これらのタイムスタンプは比較的演算時間のばらつきが少ない少数ノード実行時のものを大規模実行を想定したシミュレーションでも用いている．よって，SCAMP は，楽観的な見積もりを与える傾向があるのの考えられる．

NPB-FT においては，ウィークスケーリングを仮定した場合でも，データの持ちかえなどのための時間が増加する．しかしながら，疑似 MPI イベントトレースファイル作成時にはこのような演算部の変化については考慮していないため，より楽観的な見積もりを与えるものと考えられる．

3.3 疑似 MPI イベントトレースファイルの生成

本節では，本稿で実装した LLVM-IR の解析に基づくジェネレータが適切に疑似 MPI イベントトレースファイルが適切に生成できているかを検証する．また，現状の実装において疑似 MPI イベントトレースファイルが生成できないケースについても議論する．

表 4 に 16 ノードを用いて 2 次元ステンシルアプリケーション 16K を実行したときのランク 5 の MPI イベントログの抜粋と，このイベントログを元に疑似 MPI イベントトレースファイルジェネレータで生成された ランク 9 の MPI イベントログの抜粋を示す．この実験はウィークスケーリングを想定した実験であるため，count はいずれも同じ 16384 になっている．一方，通信の宛先は 6 (right), 1 (top), 9 (down), .. から 10 (right), 1 (top), 17 (down) に適切に置き換えられていることがわかる．

このアプリケーションは非周期的境界条件を仮定しているため，2 次元の角や端に相当するランクは 2 次元の内側に相当するランクと比較して通信回数が少ない．上下左右に通信をするノードの疑似 MPI イベントトレースファイルは，同じように上下左右に通信をするノードから得られた実ファイルを使用して生成される必要があり，角や端にあるノードから生成される実ファイルを使用して生成することはできない．現在の実装では，実トレースと疑似トレースの間のランク番号の対応はマニュアルで指定している．多くのアプリケーションにおいて適切なマッピングは容易であると考えられるが，そもそもマッピングが不可能な場合，たとえばノード数が増加するについれて MPI 関数呼出し回数も増加するような場合，には現状のジェネレータでは疑似ファイルを適切に生成することはできない．将来の課題として，Domain Specific Language (DSL) などを用いてユーザに通信を記述してもらう手法などを検討している．

3.4 疑似 MPI イベントトレースファイル生成およびシミュレーションコスト

本節では疑似 MPI イベントトレースファイル生成のコストおよびシミュレーションそのもののにかかる時間について検討する．

ファイル生成およびシミュレーションに用いた環境は以下である：

- Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
- DDR4-2133 Reg ECC (2GBx6)

今のところ，ファイル生成およびシミュレーションについて，分散並列化およびスレッド並列化は行っておらず，1 コアのみを用いている．

図 15 に SST/macro の実行時間を示す．並列アプリケーションのシミュレーションを 1 コアの逐次実行で行っているため，ノード数の増加にともなってシミュレーションに要する時間も増加している．しかし，数千コア程度までであれば，実際の並列実行よりも高速にシミュレーションを行うことが可能である．

図 16 に疑似 MPI イベントトレースファイルをそれぞれのノード数の分だけ生成するために必要な時間を示す．各 MPI ランクに対するジェネレータは独立に行えるため，生成のための総時間は，ノード数に対して線形に増加する．1 ランクぶんに費される生成時間は 0.05 ∼ 0.15 秒程度と高速であった．NPB-FT はステンシルアプリケーションよりも MPI 関数の呼出し回数は少ないが，コードそのものはより複雑であり，疑似 MPI イベントトレースファイルを生成するための MPI スケルトンコードもやや複雑になり，1 つの疑似 MPI イベントトレースファイルの生成に要する時間も長くなっている．

4. 関連研究

Carrington [3] らは，少数のコアを用いた時のイベント

表 4 ステンシルアプリケーションにおける実 MPI イベントトレースファイル (16 ノード実行のランク 5) と，ジェネレータにより生成された疑似 MPI イベントトレースファイル (64 ノード実行のランク 9)

```
// 実 MPI イベントトレースファイル (rank-5 of 16 nodes)

MPI_Send entering at walltime 3038480.192285198, ...
int count=16384
MPI_Datatype datatype=14 (MPI_DOUBLE)
int dest=6
int tag=0
MPI_Comm comm=2 (MPI_COMM_WORLD)
MPI_Send returning at walltime 3038480.192420677, ...
--
MPI_Send entering at walltime 3038480.193209336, ...
int count=16384
MPI_Datatype datatype=14 (MPI_DOUBLE)
int dest=1
int tag=0
MPI_Comm comm=2 (MPI_COMM_WORLD)
MPI_Send returning at walltime 3038480.195791038, ...
--
MPI_Send entering at walltime 3038480.195851373, ...
int count=16384
MPI_Datatype datatype=14 (MPI_DOUBLE)
int dest=9
int tag=0
MPI_Comm comm=2 (MPI_COMM_WORLD)
MPI_Send returning at walltime 3038480.195998959,
```

```
//疑似 MPI イベントトレースファイル (rank-9 of 64 nodes)

MPI_Send entering at walltime 3038480.192285198,
int count=16384
MPI_Datatype datatype=14 (MPI_DOUBLE)
int dest=10
int tag=0
MPI_Comm comm=2 (MPI_COMM_WORLD)
MPI_Send returning at walltime 3038480.192420677,
--
MPI_Send entering at walltime 3038480.193209336,
int count=16384
MPI_Datatype datatype=14 (MPI_DOUBLE)
int dest=1
int tag=0
MPI_Comm comm=2 (MPI_COMM_WORLD)
MPI_Send returning at walltime 3038480.195791038,
--
MPI_Send entering at walltime 3038480.195851373,
int count=16384
MPI_Datatype datatype=14 (MPI_DOUBLE)
int dest=17
int tag=0
MPI_Comm comm=2 (MPI_COMM_WORLD)
MPI_Send returning at walltime 3038480.195998959,
```

トレースから，MPI アプリケーションの大規模実行時の振る舞いを予測する方法論を提案した．ただし，彼らの研究はノード内イベント（キャッシュミス，メモリアクセスなど）に着目しており，われわれの研究はノード間の MPI イベントに焦点をあてている．

Wu [7] らは，ScalaExtrap と呼ばれる少数のイベントトレースから大規模ノードでの振る舞いを外挿する手法を提案した．彼らのアイディアは，SCAMP により近いが，彼らが row-major のメッシュトポロジのみを対象としてトポロジの解析から外挿を行うのに対して，われわれの手法はソースコードの解析から MPI スケルトンプログラムを作成して疑似 MPI トレースファイルが生成できるため，メッシュトポロジに対する推定も行えるが，ネットワークのトポロジを限定しない．

また，われわれのアプローチの特徴として，本稿では SST/macro および DUMPI ライブラリを用いたものの，SCAMP および疑似 MPI イベントトレースファイルの自動生成手法は，ネットワークシミュレータとは独立なものであり，さまざまなネットワークシミュレータに対して適用することができる．

5. おわりに

本研究では，MPI イベントトレースに基づく通信性能推定を，既存のシステムでは十分な数のトレースが取得できないような大規模将来システムに適用するための方法論 SCAlable Mpi Profiler (SCAMP) を提案している．本稿では，SCAMP 法で用いられる疑似トレースファイル作成手法の自動化について検討を行い，ソースコードの中間表現である LLVM-IR の解析による疑似 MPI イベントトレースファイルジェネレータのプロトタイプを実装した．

実験を行い，SCAMP 法について再検証すると同時に，実装した疑似 MPI イベントトレースファイルジェネレータによるファイル生成のコストやシミュレーションそのものにかかる時間について検証した．SCAMP については，先行研究で明らかになっていたように，実験で用いた SST/macro を実ファイルを用いて実行した場合を適切に

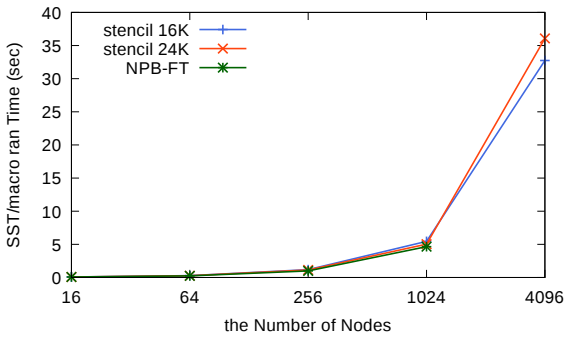


図 15 SST/macro シミュレーション実行時間

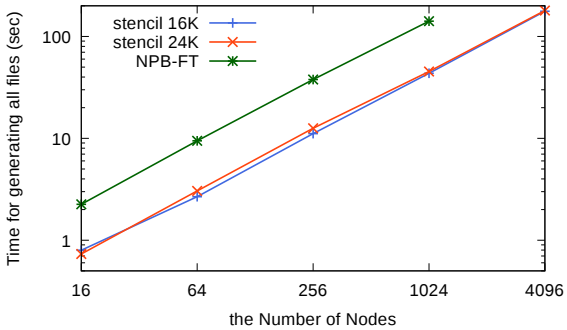


図 16 疑似 MPI イベントトレースファイルを生成するためにかかる実行時間

再現するものの，大規模ノード上での演算時間のばらつきなどを考慮しないため，より楽観的な見積もりを与えることも多かった．アプリケーションソースコードの解析から MPI スケルトンコードを生成し，これを疑似イベントトレースを出力するための dummy MPI ライブラリにリンクした，疑似 MPI イベントトレースファイルジェネレータは，実際の MPI 実行と比較しても十分に短い時間で疑似ファイルを生成することが可能であった．

今後の課題として

- ウィークスケーリングであっても演算数が変化する場合があるため，これを勘案する拡張
- 演算数が一定であっても，さまざまな影響で演算時間にばらつきがでることが想定されるため，これを勘案する拡張
- ノード数の増加により，実トレースファイルと疑似トレースファイルで通信呼出しが 1 対 1 に対応しない場合など，より複雑なアプリケーションに向けた疑似 MPI イベントトレースファイルの生成手法の検討

などが挙げられる．

謝辞

本研究は JSPS 科研費 17K12693 の助成を受けたものです．本論文の結果の一部は，理化学研究所のスーパーコンピュータ「京」を利用して得られたものです．

参考文献

[1] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The nas parallel benchmarks 2.0. Technical Report NAS Technical Report NAS-95-020, 1995.

[2] R. F. Barrett, S. Borkar, S. S. Dosanjh, S. D. Hammond, M. A. Heroux, X. S. Hu, J. Luitjens, S. G. Parker, J. Shalf, and L. Tang. On the role of co-design in. high performance computing. In *Advances in Parallel Computing /Volume 24: Transition of HPC Towards Exascale Computing*, pp. 141 – 155. 2013.

[3] L. Carrington, M. A. Laurenzano, and A. Tiwari. Inferring large-scale computation behavior via trace extrapolation. In *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*, 2013.

[4] G. Hendry and A. Rodrigues. SST: A simulator for exascale co-design. In *ASCR/ASC Exascale Research Conference*, pp. –, 2012.

[5] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo. A simulator for large-scale parallel architectures. *International Journal of Parallel and Distributed Systems*, 1(2):57–73, 2010.

[6] The LLVM Compiler Infrastructure. https://llvm.org/.

[7] X. Wu and F. Mueller. Scalaextrap: trace-based communication extrapolation for SPMD programs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pp. 113–122, 2011.

[8] 辻, 李, 朴, 佐藤. SCAMP: MPI 疑似通信プロファイルによるネットワーク性能推定手法の提案. 情報処理学会研究報告, No. 2016-HPC-153, p. On Line. 情報処理学会, 2016.

ⓒ 2017 Information Processing Society of Japan