Vol.2015-HPC-150 No.33
2015/8/6

ノード単体性能評価のための MPIアプリケーションリプレイ環境の作成

辻 美和子[1,a)]　佐藤 三久[1]

概要：並列アプリケーションの開発や移植においては，通信部に加えて，ノード単体性能の評価や最適化が必要である．本稿では，並列アプリケーションの通信性能評価・最適化のためのライブラリである DUMPI パッケージを拡張し，ノード単体性能評価時に並列実行時の振る舞いを再現可能なライブラリを作成した．このライブラリは，トレース収集のための並列実行時に，通信バッファを取得しておき，単体再現時には取得した内容を読み込むことで逐次再現時にも並列実行時の振る舞いを再現する．これにより，他ノードの結果に依存して処理が変わるようなアプリケーションであっても，逐次ノード上で再現することができる．

1. はじめに

近年のスーパーコンピュータ開発においては，アプリケーションとアーキテクチャの協調設計である「コデザイン」により科学的成果の創出を加速する動きがある [1]．コデザインの主要な過程のひとつは，多様な仮想アーキテクチャ上でのアプリケーションの性能予測と最適化である．アプリケーションの性能予測においては，通信性能およびノード単体性能の予測や評価が不可欠である．ノード単体性能評価および最適化は，コデザインプロセスのみならず，一般的なアプリケーションの移植の際にも重要である．

しかしながら，並列アプリケーションの中には，2 つ以上のノードで動作することを前提に記述されており，ノード単体性能評価のためにソースコードを大きく書き換える必要があるものがある．また，他ノードの実行結果に依存して別のノードでの実行パスが変化するアプリケーションなど，単純にプロセス数を 1 に設定しても，並列実行時の一般的な挙動が再現できない場合もある．

本稿では，MPI で記述された並列アプリケーションを，ソースコードを改変することなく，ノード単体性能評価時に並列実行時の振る舞いを再現できるようにするために，通信メッセージ取得ライブラリおよびリプレイライブラリを開発した．前者のライブラリを用いて実際にアプリケーションを並列に実行し，MPI プロファイルおよび通信メッセージを取得する．続いて，後者のライブラリを用いてこれらのログを読み込みながら，単体ノードでのリプレイを行う．単体ノードでは動作しないアプリケーションや，他ノードの結果に依存して処理が変わるようなアプリケーションであっても，通信メッセージをあらかじめ取得しておくことで，単体ノード上で再現し，ノード単体性能を評価することができる．

また，単体ノード実行のプロファイルログとして，並列実行時の通信時間ログから通信時間が外挿し，並列実行時と同等のものを生成することができる．

[1] 理化学研究所計算科学研究機構
RIKEN Advanced Institute for Computational Science, JAPAN
[a)] miwako.tsuji@riken.jp

2. MPI アプリケーションのプロファイル取得・リプレイ環境

提案ライブラリは，SST/macro の MPI プロファイル取得用ライブラリである DUMPI を拡張して実装されている．本章では，まず SST, SST/macro および DUMPI について述べ，続いて提案ライブラリの概要および実装について述べる．

2.1 研究背景と関連研究 – SST

The Structural Simulation Toolkit (SST) [3] は，Sandia National Laboratories で開発されている HPC システムのコデザインのためのツールキットである．SST は，ネットワークアーキテクチャなどのマクロ的評価のための SST/macro と，CPU シミュレータなどのミクロ的分析のための SST/micro からなる．

SST/macro は，分散メモリシステムのためのマクロスケールシミュレータであり，MPI を用いたアプリケーションのシミュレーションが可能である．SST/macro のシミュレーションには，オンラインシミュレーションとオフライ

ンシミュレーションがある．

オンラインシミュレーションは，実際のアプリケーションをシミュレータ上で実行する．通信部は，実際の通信関数を呼び出すかりに，SST/macro にリダイレクトされる．現実的には，数万プロセスからなる並列アプリケーションをそのままエミュレートすることは不可能である．その場合には，演算カーネル部分をスケルトン化したスケルトンアプリケーションを用いることもできる．

オフラインシミュレーションは，まずフルアプリケーションを実機上で並列実行し，MPI 関数の呼び出しごとにそのプロファイルを記録する．SST/macro は，これらの MPI トレースをあつめて，実機と同じシステムもしくは新たなマシンやレイアウトのシステム上での並列実行をシミュレートする．SST/macro では，トレースファイルとして標準的な形式 (OTF)　と独自の DUMPI 形式がサポートされている．DUMPI 形式は OTF 形式よりも詳細な情報を得ることができる．

以下では DUMPI ライブラリについて述べる．多くの MPI ライブラリの実装において，MPI 関数は実質的な機能を提供する PMPI 関数のラッパーであるウィークシンボルとして実装されている．ウィークシンボルとして定義された関数は，オブジェクトファイルをリンクして実行形式ファイルを作成する過程で，ストロングシンボルとして定義されている同名の関数に置き換えることができる．

DUMPI は MPI ライブラリにおけるウィークシンボルを上書きすることでアプリケーションのソースコードの改変なしに MPI 関数のプロファイルを可能にする．DUMPI は以下のように新たな MPI 関数のラッパーを定義し，プロファイル情報を集め，PMPI 関数を呼び出す：

```
int MPI_Send(...)
{
  /** Start profiling work **/
  ...
  int rval = PMPI_Send(...);
  /** Finish profling work */
  ...
  /** Write profiles **/
  ...
   return retval;
}
```

DUMPI ライブラリをリンクして作成したアプリケーションを並列に実行すると，以下のようになファイルが得られる：

```
dumpi-2015.07.08.11.29.36-0000.bin
dumpi-2015.07.08.11.29.36-0001.bin
....
dumpi-2015.07.08.11.29.36-0007.bin
dumpi-2015.07.08.11.29.36.meta
```

dumpi からはじまり，実行日時，ランク番号で識別され，拡張子 bin でおわるファイルは，各ランク番号における MPI のプロファイルが書き込まれる．DUMPI ライブラリは，MPI 関数が呼び出される度に，以下の情報を記録して，トレースファイルに書き込む：

- 関数名
- 通信バッファを除く MPI 関数への入力変数（データの個数，データ型，送信先ランク，受信元ランク，タグ，コミュニケータなど）
- 通信バッファを除く MPI 関数からの出力変数（ランク番号，コミュニケータのサイズ，生成されたコミュニケータなど）
- MPI 関数の開始時刻，MPI 関数の終了時刻
- 呼び出したスレッド
- （存在する場合）PAPI 情報

拡張子 meta でおわるファイルは，トレースリプレイを構成するために用いられるメタデータである．たとえば，以下のような内容を含む

```
hostname=node1
numprocs=8
username=<none>
startime=1436322576
fileprefix=dumpi-2015.07.08.11.29.36
version=1
subversion=1
subsubversion=0
```

本稿では，DUMPI ライブラリを拡張し，

- 逐次ノード上での単体性能評価（オフラインシミュレーション）のための通信メッセージの記録

を可能にした．また，逐次ノード上でのリプレイ用ライブラリを新たに作成し

- 上述の通信メッセージを読み込みながらのリプレイ
- 並列実行時に取得した MPI 関数のプロファイリングから外挿された通信時間および，リプレイ時の演算実行時間に基づく，新たなプロファイル・ファイルの作成

を可能にした．後者のファイルは，SST/macro オフラインシミュレータの入力としても利用可能である．すなわち，単体ノードもしくは単体 CPU シミュレータによる出力を，さらなるシミュレーションの入力として用いることができる．

2.2 概要

図 1 に提案ライブラリの概要を示す．まず，対象アプリケーションを実機にて並列に実行する．この際に，ライブラリ関数は MPI 関数呼び出しのプロファイルおよび通

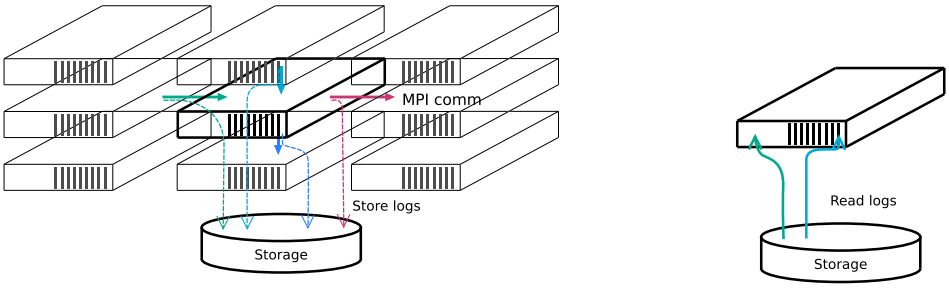

図 1　ログ取得 (左) と単体ノードにおけるリプレイ (右)

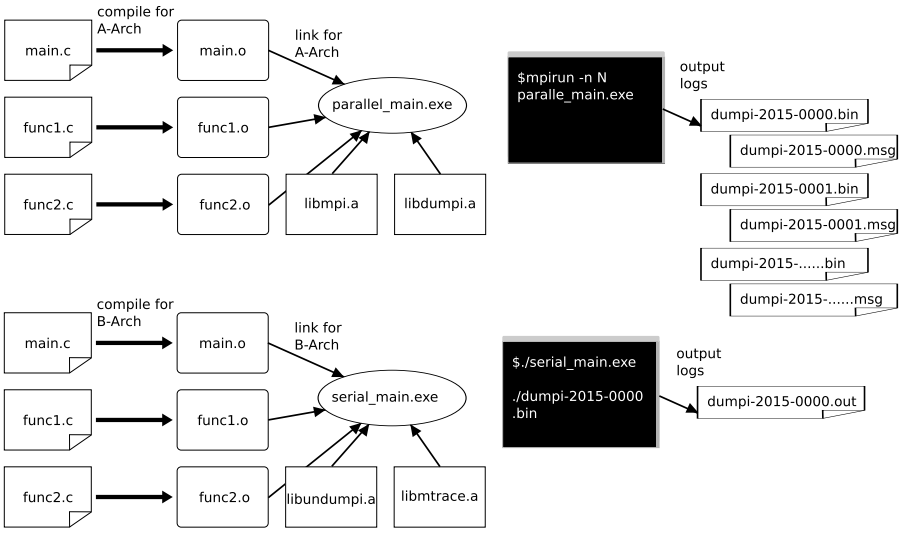

図 2　提案ライブラリの使用イメージ

信メッセージを記録しておく．続いて，単体ノード上でこれらのログを読みながらリプレイを行う．リプレイは，実ノードのみならず，CPU シミュレータなどの上でも行うことも想定される．

図 2 に提案ライブラリの使用イメージを示す．基本的にはソースコードの変更は不要である．ソースコードをオブジェクトファイルにコンパイルし，リンク時に，MPI ライブラリに加えて，ログ取得用の dumpi ライブラリをリンクする．このようにして作成された実行形式ファイルは，実機にて並列に実行される．実行後は，以下のようなファイルが出力される：

```
dumpi-2015.07.08.11.29.36-0000.bin
dumpi-2015.07.08.11.29.36-0000.msg
dumpi-2015.07.08.11.29.36-0001.bin
dumpi-2015.07.08.11.29.36-0001.msg
  ....
dumpi-2015.07.08.11.29.36-0007.bin
dumpi-2015.07.08.11.29.36-0007.msg
dumpi-2015.07.08.11.29.36.meta
```

ここで，bin および meta は，既存の DUMPI ライブラリを使用した場合と同じである．追加で得られるファイル msg には，各ランクにおけるすべての通信メッセージが記録されている．

つづいて，別の環境（単体ノードやシミュレータ）でアプリケーションをコンパイルする．アプリケーションのオブジェクトファイルに，MPI ライブラリと dumpi ライブラリの代わりに DUMPI プロファイルログ読み込みライブラリである undumpi ライブラリ，および MPI 通信トレース読み込みライブラリ mtrace ライブラリをリンクして，実行形式ファイルを生成する．生成した実行形式ファイルは逐次実行され，リプレイ時のプロファイル .out が生成される．

並列実行時およびリプレイ時の MPI 関数呼び出しプロファイルにおけるタイムスタンプは，プロファイル取得やログの読み書きにかかるオーバーヘッドが修正された上で記録される．

2.3　実装

本節では実装について述べる．

ログ取得時

2.1 章で述べたプロファイルに加えて，通信メッセージを取得するために，DUMPI ライブラリにおける MPI 関数のラッパーは以下のように拡張される：

```
int MPI_Recv(void *buf...)
{
  /** Start profiling work **/
  ...
  int rval = PMPI_Recv(buf, ...);
  /** Finish profling work **/
  ...
  /** Write profiles **/
  ...
   /** Write messages (buf) **/
  ...
   return retval;
}
```

ただし，MPI_Irecv などのノンブロッキング通信の場合は，PMPI 関数呼び出し終了直後にメッセージが届いているとは限らない．このため，ノンブロッキング通信呼び出し時には，MPI プロファイルのみをログに記録する．同時に，MPI_Request とバッファへのポインタをセットで登録しておき，MPI_Waitall などの通信終了確認時に対応する通信メッセージの記録を行う．

通信メッセージはビックエンディアンとリトルエンディアンのいずれかの形式で書き込まれる．ログ取得環境と，後に通信メッセージを読み込むリプレイ環境とでエンディアンが異なる場合には，オプションとして読み込みもしくは書き込み時のいずれかにバイトリオーダを指定する．また，メッセージ取得は，MPI_Type_vector などのユーザ定義のデータの並べ替えにも対応している．

拡張されたライブラリは，従来の DUMPI と同様に MPI プロファイル bin を生成する．しかし，すべての通信メッセージを記録するためのオーバヘッドは大きく，そのままでは通常のプロファイルや通信メッセージ取得を行わない場合と比較して，総実行時間が長めに出てしまう（図 3，上）．図 3 で，s_i は MPI 関数 i の開始時刻を，t_i は終了時刻を示す．図において，MPI 関数 1 と 2 の間の演算部分は，プロファイルからは $(s_2 - t_1)$ 秒と見積もられるが，この見積りには MPI 関数 1 を記録するためのオーバヘッド o_1 が含まれるため，実際には $(s_2 - t_1 - o_1)$ 秒しかかかっていない．そこで，より精確なプロファイルを提供するために，MPI 関数 2 の開始時刻，すなわち演算 2 の終了時刻，を

$$s_2' := s_2 - o_1$$

とする．同様に，

$$t_2' := t_2 - o_1$$
$$s_3' := s_3 - o_1 - o_2$$
$$t_3' := t_3 - o_1 - o_2$$
$$\cdots$$

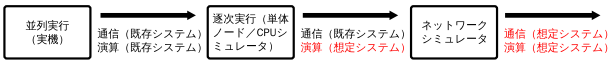

図 5　リプレイログからの全体性能推定

として，MPI 関数の開始および終了時刻から累積オーバーヘッドを減ずる．このようにログ取得のオーバヘッドの影響を考慮することで，ログ取得を行わない場合と近いかたちのプロファイルを得ることができる．

リプレイ時

上述のようにして取得されたプロファイルと通信メッセージを用いてノード単体性能評価を行うために，MPI トレースリプレイライブラリ libmtrace を作成した．このライブラリは，DUMPI と同様に，MPI 関数へのラッパー関数を含む：

```
int MPI_Recv(void *buf...)
{
  int retval= ..._read_...(buf,..)
  ...
  return retval;
}
```

リプレイ時の入力ファイル（通信メッセージログなど）は，実行時の引数もしくは環境変数として与えられる．

また，リプレイ時にも取得時と同様にして，プロファイルの書き込みを行う．しかし，単体ノードでのリプレイ時には通信のかわりに MPI メッセージファイルの読み込みを行うため，図 4 に示すように，プロファイルの作成時に

- メッセージファイル読み込みオーバーヘッドの修正
- 通信時間の外挿

を行う．本稿では単体ノードの演算性能評価に主眼をおいているため，通信時間はログ取得時の値をそのまま代入している．

さまざまなネットワークを想定したより複雑な通信時間の評価は，リプレイ時に得られたプロファイルログを SST/macro や別のシミュレータに入力してさらなるシミュレーションを行うことで得られると考えられる（図 5）．このようにして，未知のシステムでの性能評価が，コードの書き換えやカーネルの切り出しなどをすることなく，平易に可能になると期待される．

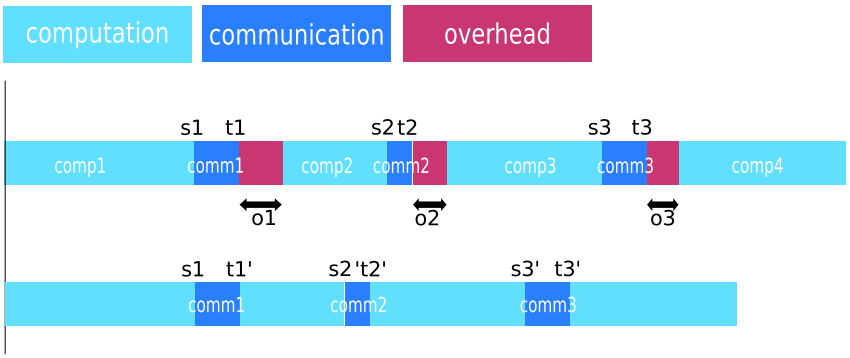

図 3　ログ取得オーバーヘッドの修正．図では簡単のためにより負荷の大きい通信後のオーバーヘッドを示したが，通信前のプロファイル取得のオーバーヘッドも同様に修正される

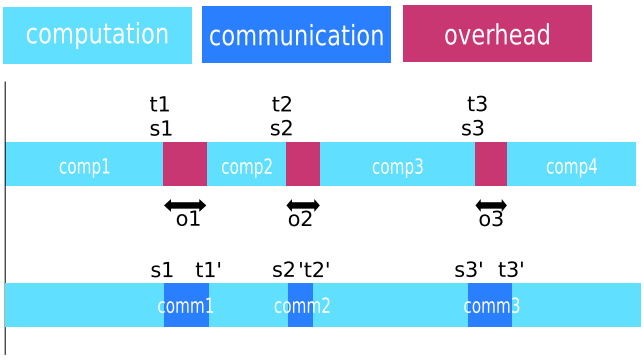

図 4　リプレイログ作成．演算はリプレイ環境の時間を用い，通信はログ取得時の数値が代入される．仮想ネットワーク上での通信時間推定は，このログを用いて別のシミュレータで行うことができる．また，オーバヘッドは修正される

表 1　FX10 ノード諸元

CPU	Fujitsu SPARC64IXfx, 16 core, 1.65 GHz
Memory	32GB
Compiler	Fujitsu Compiler 1.2.1
Option	-Kfast

表 2　Haswell 機ノード諸元

CPU	Intel Xeon E5-2680 v3 , 12core x 2socket, 2.50GHz
Memory	384GB
Compiler	Intel Compiler 15.0.2
Option	-O3

3. 実験

提案ライブラリを用いて計算実験を行った．ベンチマークとしては，NAS Parallel Benchmarks (NPB) [2] から，Scalar Penta-diagonal Multi-Zone version (SP-MZ) を用いた．SP-MZ は，MPI と OpenMP のハイブリッド並列プログラミングモデルに基づいて記述されている．

また，実験環境としては FX10 (表 1) および Haswell 機 (表 2) を用いた．FX10 は複数ノードをそなえた並列環境であり，Haswell 機は 1 ノードのみの逐次環境である．

SP-MZ の問題サイズをクラス B およびクラス C とし，ノード数はそれぞれ 8，16 とした．ノード内のスレッド並列では，FX10 では常に 16 スレッドを用い，Haswell 機では 16 スレッドおよび 24 スレッドを用いた．

実験の結果を表 3 に示す．

ビッグエンディアン環境 (FX10) でログを取得し，ビッグエンディアン環境 (FX10) でリプレイを行う場合，ログの取得や読み込みにかかるオーバーヘッドの全実行時間に対する比率は，通信頻度や量などアプリケーションの性質によって大きく左右されるであろうものの，全体の数パーセントから 20 パーセント程度であった．また，オーバーヘッドの通信時間に対する比率は，ログ取得時は通信時間と同程度，リプレイ時は 3 ～ 4 倍程度であり，極端に通信の占める割合の高いアプリケーション以外ではユーザの許容範囲内でログの取得や読み込みが可能であると考えられる．

一方，ビッグエンディアン環境 (FX10) でリトルエンディアン環境でリプレイを行うためのログを取得する場合，通信メッセージを記録する際にバイトオーダを変更する必要があるため，通信時間の十数倍という比較的大きな

表 3 実験結果

		実実行時間	通信時間	推定演算時間	オーバーヘッド	推定実行時間
SP-MZ.B.8	トレースなし (FX10)	3.294	-	-	-	3.294
	ログ取得時 (FX10)	3.529	0.1875	3.150	0.1912	3.337
	リプレイ時 (FX10)	3.810	(0.1875)	3.215	0.5955	3.402
	ログ取得時 (FX10→Haswell)	5.406	(0.1814)	3.178	2.044	3.359
	リプレイ時 (Haswell,16thread)	1.666	(0.1814)	1.581	0.0854	1.762
	リプレイ時 (Haswell,24thread)	1.724	(0.1814)	1.641	0.0831	1.822
SP-MZ.C.16	トレースなし (FX10)	7.599	-	-	-	7.599
	ログ取得時 (FX10)	8.102	0.4070	7.317	0.3781	7.724
	リプレイ時 (FX10)	8.931	(0.4070)	7.301	1.630	7.708
	ログ取得時 (FX10→Haswell)	12.92	0.3484	7.342	5.230	7.690
	リプレイ時 (Haswell,16thread)	3.485	(0.3484)	3.294	0.1914	3.642
	リプレイ時 (Haswell,24thread)	3.627	(0.3484)	3.442	0.1852	3.790

オーバーヘッドがかかる．

推定実行時間はログ取得時には，実実行時間からオーバーヘッドを減じた時間である．リプレイ時には実実行時間からオーバーヘッドを減じて通信時間を足した時間である．これらはトレースを行わない場合よりも 1.3 ～ 1.6 パーセント程度大きな数字となっている．これは，オーバヘッド取得自体にかかるオーバーヘッド（計時のための時間），および，プロファイルや通信メッセージの書き込み・読み込みを行うことによるキャッシュ内容の変化などの影響であると考えられる．

Haswell 機におけるリプレイでは，ベンチマークプログラムをソースコードの改変なしに Intel コンパイラで再コンパイルし，リプレイライブラリとリンクするだけで，並列実行時の振る舞いを再現することが確認された．

4. おわりに

本稿では，並列アプリケーションのノード単体性能評価を，並列実行時の振る舞いを再現しながら行えるように，通信メッセージ取得ライブラリとリプレイ用ライブラリを作成した．これらのライブラリ組を用いて，SPARC アーキテクチャである FX10 でアプリケーションを並列実行し，取得したログを x86 アーキテクチャである Haswell 機でリプレイすることができた．これにより並列システムが利用可能でないときでも，単体ノードを用いてノード内性能の最適化を行うことが可能になる．

また，単体ノードにおけるリプレイ時に，並列実行時のプロファイルに基づいて通信時間を外挿し，並列実行時のようなプロファイルログの生成を可能にした．このログを用いて，既存の通信シミュレータ上でさらなるシミュレーションを行うことで，リプレイされた単体ノードで構成された並列システムや，リプレイされた CPU シミュレータで構成された並列システムの全体性能の予測が可能になることが期待される．

参考文献

[1] R. F. Barrett, S. Borkar, S. S. Dosanjh, S. D. Hammond, M. A. Heroux, X. S. Hu, and J. Luitjens. On the role of co-design in high performance computing. In *Transition of HPC Towards Exascale Computing*, pp. 141--155, 2013.

[2] R. F. V. der Wijngaart and H. Jin. Nas parallel benchmarks, multi-zone versions. Technical Report NAS Technical Report NAS-03-010.

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