Vol.2013-HPC-138 No.14
2013/2/21

DMATP-MPI: MPI 向け動的メモリ割当分析ツール

秋元秀行[†1, †2]　安島雄一郎[†1, †2]　安達知也[†1]
岡本高幸[†1, †2]　三浦健一[†1, †2]　住元真司[†1, †2]

我々は JST-CREST において，性能を犠牲としない省メモリ型の通信ライブラリの研究開発を進めている．なぜならば，ポストペタスケールのスーパーコンピュータではプロセス数が数千万規模に達する一方，計算性能に比例したメモリ量の増加は望めず，通信ライブラリの使用するメモリがシステムを圧迫する懸念があるためである．省メモリ化に着目した場合，通信ライブラリ単体のメモリ使用量を定量的に測定・評価する必要があるが，その方法は確立されておらず，手間とスキルを要する．さらには，巨大な MPI ライブラリからメモリ使用量の大きい部分を特定し削減するためには，ターゲットの効率的な絞り込みや特定区間のメモリ獲得挙動，データ構造の把握が不可欠である．我々は，これら目的の使用に適したツールとして，ライブラリやその内部関数毎の動的メモリ使用量を簡便かつ定量的に評価・分析するツールとして，Dynamic Memory Allocation Tracing Profiler for MPI(DMATP-MPI)を開発したので，その詳細を報告する．

DMATP-MPI: Dynamic Memory Allocation Tracing Profiler for MPI

HIDEYUKI AKIMOTO[†1, †2]　YUICHIRO AJIMA[†1, †2]　TOMOYA ADACHI[†1]
TAKAYUKI OKAMOTO[†1, †2]　KENICHI MIURA[†1, †2]　SHINJI SUMIMOTO[†1, †2]

Assuming post peta FLOPS supercomputer, it is thought that number of process will reach to 10-100 million and it seems to be difficult to increase the system memory with increase of the calculation performance. A message passing interface (MPI) is widely used for the supercomputer systems as a communication library. It is one of the key issues that the memory usage of communication library, which is proportional to the number of process, suppresses the system memory. Therefore, we have been studying to solve this issue as a project of JST-CREST. When accomplishment of research, it is necessary to measure quantitative memory usage of the communication library repeatedly. Additionally, it will be helpful to analyze the memory usage of each function for efficient development in the MPI library because it is large and complicated. Here, we have proposed and developed the memory usage measurement and analysis tool which called DMATP-MPI: Dynamic Memory Allocation Tracing Profiler for MPI. This manuscript reports detail of DMATP-MPI.

1. はじめに

Linpack 演算性能を用いたスーパーコンピュータ（スパコン）のランキングサイトである Top500 によれば，計算速度は年率約 1.9 倍で向上しており，近年はスカラ型スパコンがその上位を占めている[1]．スカラ型スパコンの各計算ノードは, 個別のメモリを持ち互いに共有されないため，計算結果を必要に応じてインターコネクトを介して交換する必要がある．この目的に使用する通信ライブラリとして Message Passing Interface（MPI）[2], [3]がもっとも広く利用され，事実上のスタンダードとなっている．スカラ型スパコンの計算速度の向上は，主に計算ノード・CPU コア数の増加によってもたらされており，今後もその傾向は維持されるものと考えられる[4], [5], [6]．その一方で計算性能に比例したメモリ量の増加は期待できず，MPI 等のシステムソフトウェアによるユーザメモリの圧迫が懸念され，ポストペタスケールのスパコンを実現する上での問題点の一つと認識している[7]．我々は性能を犠牲としない省メモリ型の通信ライブラリの実現に向け，JST-CREST プロジェクトの一つとして研究・開発に着手している[8], [9]．省メモリ化に着目した場合，通信ライブラリ単体のメモリ使用量を定量的に評価する必要があるが，既存の方法ではプロセス単位でしか測定することができず，その定量評価には労力とスキルを要する．また巨大な MPI ライブラリのメモリ使用量の削減のためには，その対象を効率的に絞り込むための分析ツールも必要である．我々は，これらの目的の使用に適したツールとして Dynamic Memory Allocation Tracing Profiler for MPI(DMATP-MPI)を開発したので，その詳細を報告する．

本研究報告の構成は 2 章で既存のメモリ使用量の測定方法と問題点について述べる．3 章では省メモリ型の通信ライブラリの開発に必要なメモリ使用量の評価・分析ツールの要件を述べる．4 章，5 章では，それぞれ我々の提案する DMATP-MPI の開発方針，およびその詳細・実装について述べる．6 章では DMATP-MPI を用いたい動的メモリ使用量の分析結果を示し，7 章でまとめる．

2. 既存のメモリ使用量測定方法と問題点

本章ではよく知られているメモリ使用量の測定方法や動

†1 富士通株式会社　次世代テクニカルコンピューティング開発本部
Fujitsu Limited., Next Generation Technical Computing Unit
†2 (独)科学技術振興機構　戦略的創造研究推進機能
Japan Science and Technology Agency (JST),
Core Research for Evolutional Science and Technology (CREST)

的メモリの獲得･開放に関する分析ツールについて述べる．プロセス単位のメモリ使用量を取得する方法として，“ps alx”や“top –b –n1”コマンドを実行することや，“/proc”内のプロセス ID フォルダの特定ファイルにアクセスすることによって得ることが可能であることが知られている．しかしながら，上記の方法ではプロセス単位のメモリ使用量しか取得することはできず，MPI 等の個別ライブラリのメモリ使用量を得ることはできない．

一方，動的メモリの使用量については主にリークの有無を調べるツールを一部応用できる．その代表に，GNU libc の拡張関数である mtrace[10]や valgrind[11]などがあるが，やはり個別ライブラリの動的メモリ使用量を容易に集計することはできない．

3. メモリ使用量の評価・分析ツールの要件

本章ではライブラリ開発，特に我々の目指す省メモリ型の通信ライブラリの開発に必要なメモリ使用量の評価・分析ツールの要件について述べる．第一は評価ツールとしての側面で，通信ライブラリのメモリ使用量を定量的に測定・評価できることである．第二は分析ツールとしての側面で，ライブラリ内のメモリ使用量の大きい部分を特定できること，詳細なメモリ獲得挙動をトレースできることである．

第一の要件は，我々の目指す通信ライブラリが省メモリ型であることを特徴の一つとしていることから，ライブラリのメモリ使用量を繰り返し，定量的に評価する必要があるためである．また，省メモリ化は特定条件だけではなく，様々なアプリケーションとの組合せや異なる条件で実現されていることが望ましく，既存のアプリケーションやライブラリに手を加えずに容易に測定可能であることが望まれる．

第二の要件は，省メモリ型の通信ライブラリの研究・開発を網羅的・効率的に進めるために必要なものである．ソースコードを直接確認し修正個所を絞り込み，改良する手法では，既存の MPI ライブラリは手に余る程大きく複雑になっている．そこで，メモリ使用量をライブラリよりも小さい単位（関数毎）で分類・集計することによって，改良が必要な個所を効率よく特定することなどが必要であると考える．また，詳細な分析や実際の改良に当たってはソースの特定個所のメモリ使用量をトレースできるようなツールも必要と考える．

4. DMATP-MPI の開発方針

省メモリ型の通信ライブラリの開発にあたり，以上に述べた要件を満たすメモリ使用量の評価・分析ツールの開発として，以下の方針に従い DMATP-MPI を開発した．

1. ユーザプログラムやライブラリの変更なしに簡便かつ実動作状況におけるメモリ使用量の測定を実現する．
2. 低粒度，すなわちスレッド，ライブラリおよびライブラリ内の関数毎のメモリ使用量を分類・集計する．
3. メモリ使用量は動的メモリを対象とし，OS が割り当てる実際のメモリ量を集計する．
4. 集計値としては，最大・最小を含むメモリ使用量，および malloc 系関数の呼び出し回数とする．
5. ソースに修正が必要となるが，必要に応じ任意区間のメモリ使用量に関する情報を取得可能とする．

5. DMATP-MPI の詳細および実装

本章では 4 章で述べた動的メモリ使用量の評価・分析ツールの開発方針に従った実現方法および実装について述べる．図 1 に動的メモリ使用量の分類・集計処理における，被測定プログラムと DMATP-MPI の処理フロー概略を示す．DMATP-MPI は動的ライブラリとして提供し，環境変数 LD_PRELOAD によって事前にロードした上で，被測定プログラムを実行する．本ツールは C 言語の動的メモリの獲得・開放機能を提供する malloc 系の 4 関数（malloc, realloc, memalign, free）の動作を変更し，動的メモリの獲得・開放が行われるたびに，その呼び出し元ライブラリ・関数の特定，および割り当て・開放量の確認を行い，分類・集計する．そして被測定プログラムの終了時に，その結果を出力するものである．以降にその詳細を順に述べる．

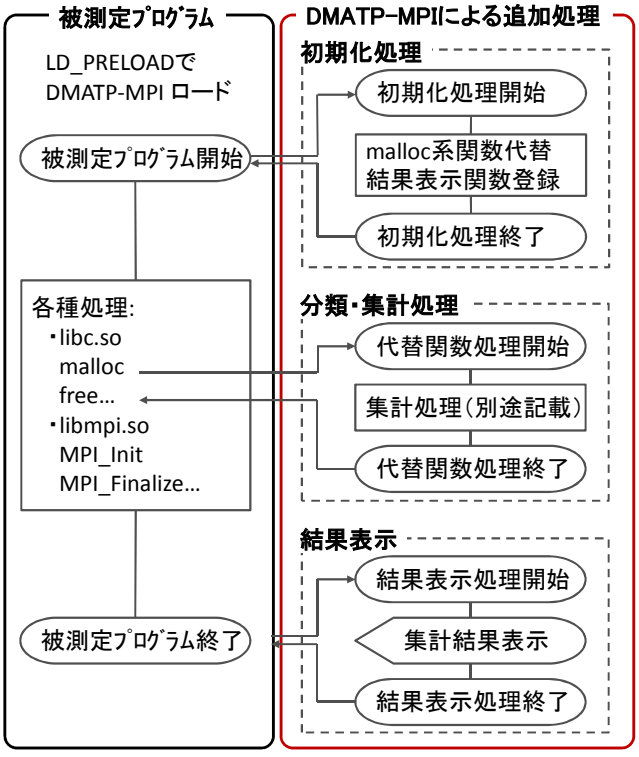


図 1　DMATP-MPI を用いた被測定プログラムのメモリ使用量の分類・集計処理の概略フロー

Figure 1 Outline flow chart of testee program and DMATP-MPI which measures classification memory usage.

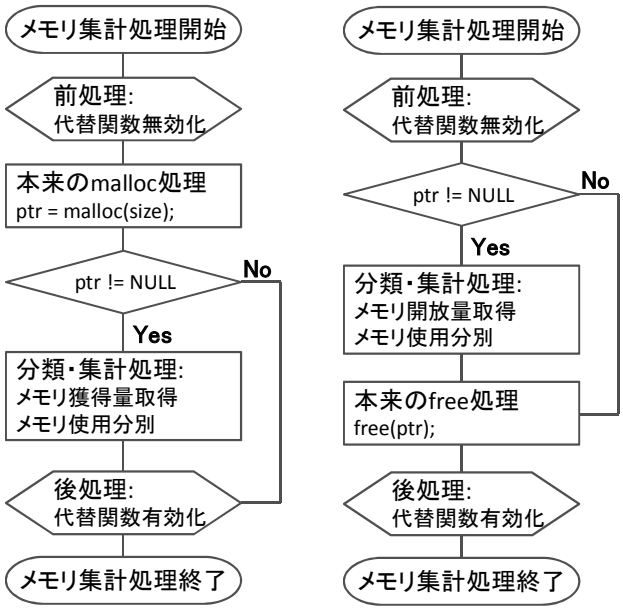
表 1　malloc 系関数代替の為のポインタ変数と代替関数型・引数一覧
Table 1 Function pointer variable for alternation of malloc family functions and those types and arguments.

Func. pointer	malloc_usable_size (B)
void *__malloc_hook	void *_function_(size_t _size_, const void *_caller_)
void *__memalign_hook	void *_function_(size_t _alignment_, size_t _size_, const void *_caller_)
void *__realloc_hook	void *_function_(void *_ptr_, size_t _size_, const void *_caller_)
void __free_hook	void _function_(void *_ptr_, const void *_caller_)

5.1 malloc 系 4 関数の代替および内部処理における一時的な無効化方法

DMATP-MPI では malloc 系の関数を本来のメモリ獲得・開放機能に加え，動的メモリの分類・集計処理を加えた関数に代替する. GNU libc には malloc 系関数の動作を変更させるための仕組みが用意されており，それを利用する[12], [13]. 例えば動的メモリ獲得関数である void *malloc(size_t *size*) の代替は，void* 型の関数ポインタ変数である __malloc_hook に有効な関数ポインタを設定することによって行う. 表 1 は malloc 系 4 関数を代替するための関数ポインタ変数および代替関数の型・引数の一覧である．これらの変数への代入処理は，DMATP-MPI ライブラリを LD_PRELOAD し，非測定プログラムを実行する際の初期化時に行われる様にした. なお, 詳細は 5.4 節で述べるが，初期化処理においては，被測定プログラムの終了時に分類・集計結果の出力関数の登録も行う.

図 2 は malloc および free の代替関数の処理内容を簡易的に示したものである．非測定プログラムの動作に支障を来さずに動的メモリ使用量の分類・集計を行うためには，代替関数内部においても代替前の関数と同じ動作をさせる必要がある．つまり，代替関数内部では本来の malloc 系関数を呼び出し，適宜動的メモリの獲得・開放を行う必要がある．GNU libc に用意されている代替関数を使用する仕組みでは，元の malloc 系関数を呼び出すためには __malloc_hook 変数を元の関数ポインタ（通常は NULL）に置き換える必要がある. 代替関数の無効/有効化は本来の関数を呼び出す前後で, 該当の関数に対してのみ行えば良い. しかし，現在の実装では図 2 に示す様に，代替関数の大部分において無効化を行っている．また代替関数の無効/有効化は malloc 系の 4 関数全てに対して行っている．これは，代替関数の内部処理で使用している関数（C の標準関数を含む）が，malloc 系の関数を内部使用している場合の再帰呼び出しや，これによる集計結果への悪影響を避けるため



(a) malloc 代替処理　　(b) free 代替処理

図 2　malloc 系関数代替の内部処理フロー図
Figure 2 Flow chart of alternative malloc family functions.

である.

また，本方式による本来の malloc 系の関数の呼び出し方法は，“The GNU C Library Reference Manual”にも記載されているが[13]，一時的に__malloc_hook 関数ポインタを書き換え無効化するため，スレッドセーフでない点が指摘されており, 本ツールも現在はスレッドセーフとなっていない. なお GNU libc の拡張関数である mtrace/muntrace によるメモリリーク有無の確認も類似の方法を使用しており，やはりスレッドセーフとはなっていない.

5.2 動的メモリの獲得・開放量の取得方法

次に図 2 に示した動的メモリ使用量の分類・集計処理の詳細について述べる．本節では malloc 系関数が獲得・開放する動的メモリ量について，次節ではそのメモリ使用を分別する方法について述べる.

malloc, memalign 関数ではメモリの獲得，free 関数では開放，realloc 関数では状況によりメモリの獲得または開放がそれぞれ行われる．malloc, memalign, realloc 関数では最終的に確保されるメモリ量が引数 size_t *size* によって指定される．各関数は引数で指定された *size* の動的メモリを確保するが，メモリアライメントや最低サイズによる制限のため，実際の割り当てメモリ量は必ずしも *size* に等しいとは限らない. 前述の malloc 系の関数によって割り当てられた実際のメモリ量は，GNU libc 拡張関数である size_t malloc_usable_size(void **ptr*)で取得できる．ここで，引数の **ptr* には malloc 系の関数で割り当てられた動的メモリのアドレスを指定する必要がある．表 2 は x86_64 の環境における，malloc の *size* 引数と“malloc_usable_size”によって返された実際の割り当てメモリサイズの関係である．本環境

表 2　malloc 関数に与えた引数: size と malloc_usable_size の返り値の関係

Table 2 Relationship between given argument: *size* of malloc and returned value of malloc_usable_size.

malloc(*size*) (B)	malloc_usable_size (B)
1 ～ 24	24
25 ～ 40	40
41 ～ 33,656,792 (16 step)	16 step
33,656,793 ～ 33,660,888	33,660,912
33,660,889 ～ 33,664,984	33,665,008
33,664,985 ～ (4, 096 step)	4,096 step

測定環境: CentOS 6.0, Kernel 2.6.32-71.29.1.el6.x86_64

では最低割り当てサイズが 24 B，以後特定サイズ（size = 33,656,792 B）までは 16 B ステップで増加し，それ以降は 4,096 B 単位で増加する振る舞いであった．特定サイズは動的メモリの使用状況や環境によって異なる．

“malloc_usable_size”関数の返り値を動的メモリ割り当て量の集計に用いることは，free や realloc によって既に確保されたメモリ使用量を知る上でも都合が良い．例えば void free(void *prt)関数により開放されるメモリ使用量は，*ptr を引数に“malloc_usable_size”を呼び出すことで容易に得ることができる．“malloc_usable_size”を用いない実装では，malloc 等のメモリ獲得関数で割り当てられたメモリアドレスと割り当て要求量をテーブルとして保存し，解放時には該当するメモリアドレスから解放されるメモリ量を検索・取得しなければならず，煩雑な処理となる．

5.3 動的メモリ使用量の分別方法

次に動的メモリ使用量の分別方法について述べる．メモリ使用量はスレッド，動的ライブラリおよびその内部の個別関数に分別する．

初めにスレッドの分別方法について述べる．スレッドは malloc 系の代替関数内において，スレッド ID を調べることで容易に識別することができる．スレッド ID の取得にはシステムコールの syscall(SYS_gettid)により行った．但し，既に述べたが，GNU libc が用意する malloc 系関数を代替する仕組みではスレッドセーフな作りにすることが困難で，現在の DMATP-MPI の実装もスレッドセーフにできておらず，今後の改善課題である．

次にメモリ獲得・開放を行った動的ライブラリおよびその内部関数を特定する方法について述べる．malloc 系の関数の呼び出しに至るまでの，ライブラリ・関数の呼び出し履歴はスタックフレームの解析により得ることができる．GNU libc では，ユーザレベルのプログラムからスタックフレームに関する情報の取得方法として，backtrace 拡張関数群が提供されている[14]. “backtrace”は関数呼び出しの履歴情報をスタックからアドレスベースで取得する関数である．一方，“backtrace_symbols”はアドレスベースの履歴情報を文字列へ翻訳する関数である．DMATP-MPI では両者をセットで呼び出し，文字列情報からイブラリ・関数の分別を行う．図 3 は malloc 代替関数内で呼び出した“backtrace_symbols”による翻訳結果の一例である．図の例では 9 段（9 行）の関数呼び出し履歴が示されている．図中の斜体文字は説明の都合上加えたもので，前記の関数による出力ではない．再下段の 2 行（要素 7, 8）はプログラムを実行するためのスタック情報である．実際の関数呼び出し履歴は要素 0 から 6 であり，その呼び出しは要素 6 から 0 の順である．翻訳結果の各行は“PorgLib(Func+Offset)[Addr]”の形式を持ち，“ProgLib”はプログラムまたは動的ライブラリ名，以降順に関数名，関数の先頭からのオフセット位置，返りアドレスである．要素 6 は最初の関数呼び出しが，被測定プログラム“IMB-MPI1”の“main”であることを表している．要素 5 は動的ライブラリである“libmpi.so”の“MPI_Init”関数が呼ばれていることを表しており，呼び出し元は前述の“main”関数である．その後，“MPI_Init”からは“libmpi.so”内の 4 関数が順次呼ばれ（要素: 4～1），最終的に“opal_class_initialize”関数からメモリ割当関数である malloc が呼ばれていることが分かる．DMATP-MPI では被測定プログラムから最初に呼ばれたライブラリおよびその内部関数を基にメモリ使用量を分類・集計する．このため，要素番号の最大－2 すなわち 6 から 0

[backtrace_symbolsの出力例]

```
0: ./libdmatpmpi.so(malloc_hook+0x91)[0x7f42a4315ad3]
1: /home/akimoto/OpenMPI/lib/libmpi.so.1(opal_class_initialize+0x8f)[0x7f42a408af4f]
2: /home/akimoto/OpenMPI/lib/libmpi.so.1(opal_output_init+0x1a6)[0x7f42a408d266]
3: /home/akimoto/OpenMPI/lib/libmpi.so.1(opal_init_util+0x39)[0x7f42a408a569]
4: /home/akimoto/OpenMPI/lib/libmpi.so.1(ompi_mpi_init+0x6c)[0x7f42a3fd8fdc]
5: /home/akimoto/OpenMPI/lib/libmpi.so.1(MPI_Init+0xf0)[0x7f42a3fee540]
6: ./IMB-MPI1(main+0x2f)[0x4033bf]   ← ユーザプログラムの main から MPI_Init 関数コール
7: /lib64/libc.so.6(__libc_start_main+0xfd)[0x3b2ac1ec5d]  ┐ 固定部
8: ./IMB-MPI1[0x4032d9]              ← ユーザプログラム    ┘
```

図 3　関数呼び出し履歴を示す backtrace-symbols 出力の例（被測定プログラム”IMB-MPI1”）

Figure 3 A sample of backtrace-symbols which represents order of called functions. Testee program is “IMB-MPI1”.

に向かって，呼び出し履歴を確認し，最初に呼ばれたライブラリ・内部関数を特定する．すなわち，図 3 に示した malloc によるメモリ獲得は“libmpi.so”の“MPI_Init”によって確保されたものとして分別する．malloc 系関数の全ての呼び出しに対して同様の解析を行い，メモリを獲得・開放した基となったライブラリ・関数を特定する．これらの情報と 5.2 節で取得したメモリ獲得・開放量を用いて分類・集計を行う．なお，上記で示した分別方法を変更することにより，最終的にメモリを獲得・開放したライブラリやその内部関数を特定し，それらに関する分類・集計を行うこともできる．

5.4 分類・集計内容および結果の出力

最後に分類・集計項目，および結果を被測定プログラムの終了と同時に出力する方法について述べる．DMATP-MPI では動的メモリ使用量の集計項目として，表 3 に示す値を分類・集計する．最小値についてはプロセス全体を通した集計では負になることはないが，例えば“MPI_Init”, “MPI_Finalize”の様に初期化関数と終了関数が別の場合，動的メモリの開放処理が主に行われる“MPI_Finalize”のメモリ使用量は負値になりえる．

測定結果の出力は，被測定プログラムの初期化時に C の標準関数で提供されている“atexit”関数を用いて，分類・集計結果の表示関数を登録する．これにより非測定プログラムが正常終了すると集計結果が出力される．なお，この方法では各種のシグナルによるプログラムの異常終了では集計結果は出力されない．

また，プログラムソースに改変を加え特定区間におけるメモリ使用量のトレースについては，メモリ使用量を任意に表示する“print_mem”関数を定義し，プロセス全体の集計結果を用い適宜表示することで対応した．

6. 分類・集計結果の例

DMATP-MPI を用いた MPI ライブラリのメモリ使用量の測定結果を 2 つ示す．測定はインターコネクトとして Connect X DDR InfiniBand を用いた X86_64 PC クラスタを用いて行った．ソフトウェア環境は OS に CentOS 6, Kernel 2.6.32-71.29.1.el6.x86_64，MPI に Open MPI 1.6 を用いた．初めに，被測定プログラムとして IMB-MPI1[15]の Alltoall ベンチマークを 64 並列で実行し，ランク 0 にて動的メモリ使用量を採取した結果を図 4 に示す．Alltoall のメッセージサイズは 4 MB である．図中の①はプロセス全体のメモリ使用量であり，終了時点の動的メモリとして，538 KB のメモリを消費していることを示している．一方，動作中の最大メモリ使用量としては 557 MB であったことを示している．その下には，被測定プログラムの実行中の malloc 系関数の呼び出し回数が集計されている．②はメインスレッドに関する情報である．プロセス全体とメインスレッドの集計値はほぼ一致しており，動的メモリ獲得・開放の大部分がメインスレッドで行われていることが分かる．③, ④は MPI ライブラリおよびその内部関数のメモリ使用量の集計結果である．被測定プログラム終了時点の MPI ライブラリのメモリ使用量は 556 KB, 測定中の最大値は 19.8 MB である．IMB-MPI1 では実行するベンチマークの種類や結果を集計するために，ベンチマーク対象の関数である MPI_Alltoall 以外の関数も呼び出されていることが測定結果から読み取れる．MPI 通信関数のメモリ使用量はプログ

表 3　DMATP-MPI の動的メモリ集計値
Table 3 Dynamic Memory Profile of DMATP-MPI.

変数名	集計内容
mem_size	現在の動的メモリ使用量
mem_min	動的メモリ使用量の最小値
mem_max	動的メモリ使用量の最大値
malloc_cnt	malloc 関数の呼び出し回数
realloc_cnt	realloc 関数の呼び出し回数
memalign_cnt	memalign 関数の呼び出し回数
free_cnt	free 関数の呼び出し回数

```
==== Statistics of overall memory usage ====================
Program name: ./IMB-MPI1                                    ①
mem_size = 537672, mem_min = 0, mem_max = 556699408
malloc: 15188, realloc: 526, memalign: 546, free: 12731
============================================================
---- Statistics of individual thread memory usage ----------
Thread ID: 19995                                            ②
mem_size = 537672, mem_min = 0, mem_max = 556699168
malloc: 15186, realloc: 526, memalign: 546, free: 12729
  ---- Statistics of individual library memory usage -------
  Library: /home/akimoto/OpenMPI/lib/libmpi.so.1            ③
  mem_size = 555632, mem_min = 0, mem_max = 19817224
  malloc: 15093, realloc: 526, memalign: 546, free: 12641
    ---- Statistics of individual function memory usage ----
    Function: MPI_Init                                      ④
    mem_size = 2375192, mem_min = 0, mem_max = 4302480
    malloc: 8407, realloc: 525, memalign: 11, free: 2289
    Function: MPI_Bcast
    mem_size = 6058008, mem_min = 0, mem_max = 6058064
    malloc: 390, realloc: 0, memalign: 56, free: 103
    Function: MPI_Recv
    mem_size = 7613048, mem_min = 0, mem_max = 7613048
    malloc: 3402, realloc: 0, memalign: 462, free: 969
    Function: MPI_Alltoall
    mem_size = 2596480, mem_min = 0, mem_max = 2596496
    malloc: 1092, realloc: 0, memalign: 12, free: 53
    Function: MPI_Barrier
    mem_size = 1086000, mem_min = 0, mem_max = 1086016
    malloc: 15, realloc: 0, memalign: 5, free: 5
    (一部 省略)
    Function: MPI_Finalize
    mem_size = -19242464, mem_min = -19242464, mem_max = 472
    malloc: 326, realloc: 1, memalign: 0, free: 8910
    ------------------------------------------------------
  Library: /lib64/libc.so.6                                 ⑤
  mem_size = 384, mem_min = 0, mem_max = 536881832
  malloc: 89, realloc: 0, memalign: 0, free: 83
```

図 4　DMATP-MPI による IMB-MPI1 Alltoall(4MB)の動的メモリ使用量の分類・集計結果
Figure 4 Result of memory usage while executing 4MB Alltoall benchmark in IMB-MPI1 by using DMATP-MPI.

ラム終了時と最大値がほぼ一致している. これは Open MPI の通信関数はメモリを獲得すると，MPI_Finalize まで開放しない作りとなっているためである．また，malloc 関数の呼び出し回数はプロセス全体で 15,188 回に対し，MPI ライブラリで 15,093 回呼ばれており，大多数を占める．更に MPI ライブラリの malloc 呼び出しの半数以上は MPI_Init 関数で行われていることも分かる．一方，⑤は libc.so（IMB-MPI1 が直接呼び出し）のメモリ使用量で，その最大値は 537 MB ある. これは Alltoall の送受信データとして，ノード当たり 4MB, 64 並列, 送受信の各バッファサイズとほぼ一致している.

次に MPI_Init 関数に着目し，より詳細な測定・分析を行った結果を図 5 に示す. 本測定は Open MPI 1.6 のソースコードを修正し，MPI_Init で呼ばれる内部関数を呼び出すたびに DMATP-MPI のメモリ集計値を出力する関数である“print_mem”を埋め込み測定した. 横軸は MPI_Init 関数から呼ばれる内部関数の一覧であり，左から右に時系列で呼び出されている．縦軸は各内部関数における動的メモリ使用量の増加量をログスケールで示している．プロセス数として 1 から 1920 まで変化させて測定したところ，MPI_Init の内部関数において，プロセス数に依存して大きくメモリ使用量が変化する 5 個所，7 関数を特定した．また malloc 回数の変化から，malloc 回数がプロセス数に比例しメモリ使用量が大きくなっている関数と，malloc 回数は変わらずに獲得量がプロセス数に依存する内部関数があることが分かった．更に詳細な測定結果は別途報告する[16].

7. まとめ

ポストペタスケールのスパコン向けの省メモリ・低遅延型の通信ライブラリの研究開発を進めるにあたり，ライブラリ毎のメモリ使用量を容易かつ定量的に評価・分析するツールとして Dynamic Memory Allocation Tracing Profiler for MPI(DMATP-MPI)を提案・実装した．同ツールは動的ライブラリ毎のメモリ使用量の分類・集計に加え，個別関数の集計も可能であり，巨大な通信ライブラリからメモリ使用量の多い部分を特定する上でも有用なツールである．今後 DMATP-MPI を有効に活用し，MPI ライブラリを含む省メモリ型の通信ライブラリの研究開発を推進していく予定である. なお本ツールは MPI に限らず動的ライブラリのメモリ使用量の評価・分析に広く使用することができる点を付記する.

参考文献

1) Super Computer TOP500
http://www.top500.org/
2) Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (1995).
http://www.mpi-forum.org/

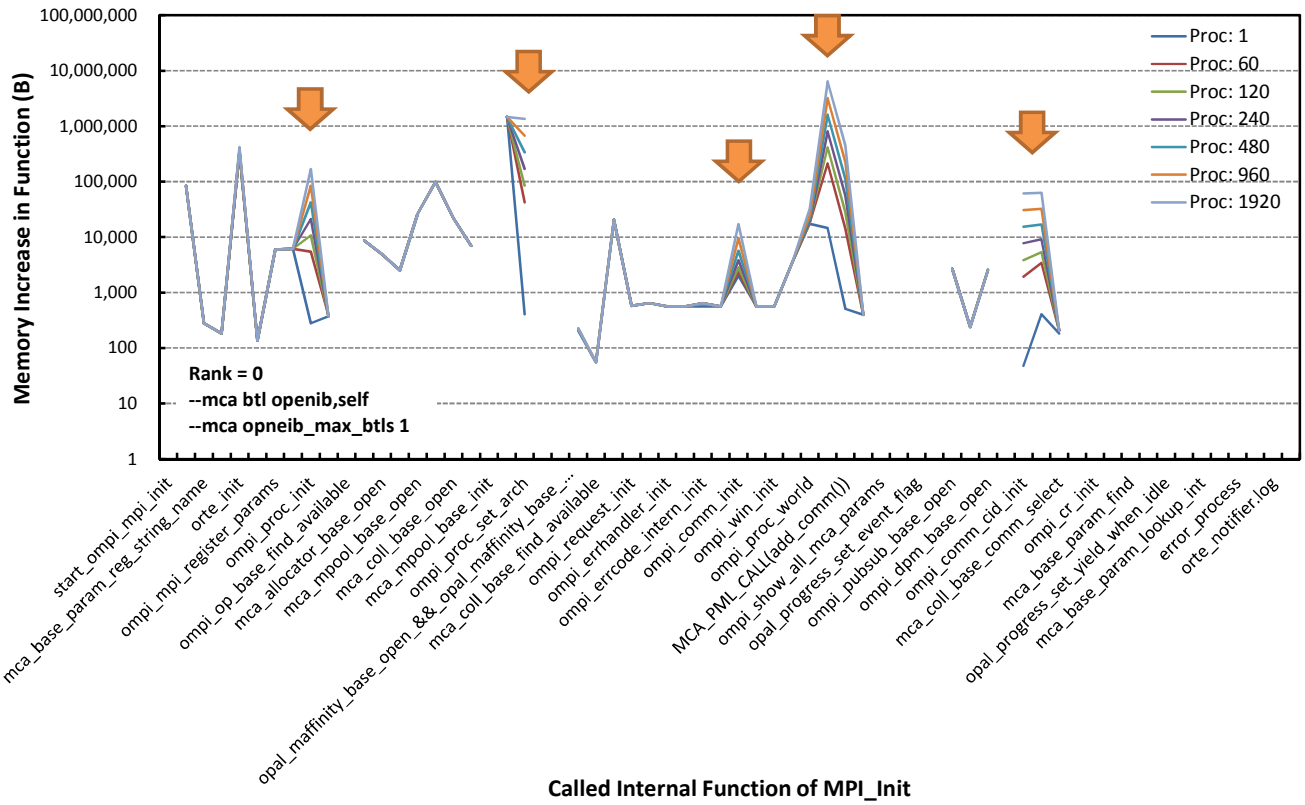


図 5　MIP_Init 内部呼び出し関数毎の動的メモリ増加量

Figure 5 Memory increase in internal functions which are called by MPI_Init.

3) Message Passing Interface Forum: MPI-2: Extensions of the Message-Passing Interface (1997).
http://www.mpi-forum.org/
4) Kogge, P., et al.: Exascale computing study: technology challenges in achieving exascale systems, DARPA Information Processing Technologies Office Sponsored Study (2008).
5) Torrelans, J.: Architectures for exteme-scale computing, IEEE Computer, 43(11), pp. 28-35 (2009).
6) Dongarra, J., et al.: The international exasccale software project roadmap, The international journal of high performance computing applications, 25(1), pp. 3-60 (2011).
7) 三浦健一, 秋元秀行, 安島雄一郎, 岡本高幸, 住元真司: エクサスケールコンピューティングに向けた省メモリ通信ライブラリの検討, 情報処理学会研究報告, 2012-HPC-133(14), pp. 1-6 (2012).
8) 安島雄一郎, 秋元秀行, 岡本高幸, 三浦健一, 住元真司: 片側通信による，グローバルデータ構造の効率的な操作方法の検討, 情報処学会理研究報告, 2012-HPC-133(7), pp. 1-8 (2012).
9) 秋元秀行, 三浦健一, 岡本高幸, 安島雄一郎, 住元真司: InfiniBand Atomic Operation の性能評価, 情報処理学会研究報告, 2012-HPC-133(8), pp. 1-6 (2012).
10) Loosemore, S., Stallman, R.M., McGrath, R., Oram, A., and Dreooer, U.: The GNU C Library Reference Manual for version 2.16, p.44 (2012).
http://www.gnu.org/software/libc/manual/pdf/libc.pdf
11) The Valgrind Developers: Valgrind-3.8.1 (2012).
http://valgrind.org/
12) Free Software Foundation: The GNU C Library (2012).
http:www.gnu.org/ software/libc
13) Loosemore, S., Stallman, R.M., McGrath, R., Oram, A., and Dreooer, U.: The GNU C Library Reference Manual for version 2.16, p.40 (2012).
http://www.gnu.org/software/libc/manual/pdf/libc.pdf
14) Loosemore, S., Stallman, R.M., McGrath, R., Oram, A., and Dreooer, U.: The GNU C Library Reference Manual for version 2.16, p.787 (2012).
http://www.gnu.org/software/libc/manual/pdf/libc.pdf
15) Intel: Intel MPI Benchmarks 3.2.3 (2011).
http://software.intel.com/en-us/articles/intel-mpi-benchmarks
16) 住元真司, 秋元秀行, 安島雄一郎, 安達知也, 岡本高幸, 三浦健一: DMATP-MPI を用いた MPI ライブラリの関数別メモリ使用量評価, to be published 情報処理研究報告, 2013-HPC-138 (2013).