Vol.2020-HPC-173 No.7
2020/3/17

マルチノード・マルチ GPU プログラミングを容易にする分散共有メモリシステム

阪口 裕梧[†1]，緑川 博子[†1]

概要：高性能計算におけるプログラミング生産性を向上することを目的に，これまで様々な PGAS 言語が提案されている．従来の MPI による分散メモリ型プログラミングモデルに対し，複数の計算ノード上に，大域アドレス空間，大域データを仮想的に提供することで，より高生産なプログラム開発ができる．しかし，その多くは，アクセスできる大域データ範囲に制限があったり，大域データアクセス時に明示的な通信記述が必要であるなど，多くの不自由さが未だ存在する．筆者らは，ソフトウェア分散共有メモリ mSMS をランタイムとして，クラスタの全ノードにアクセス制限のない同一の共有大域アドレス空間を実現し，mSMS が提供する 3 つのマルチノード並列インタフェースと，OpenMP, pthread, OpenACC, CUDA などの既存のマルチコア並列インタフェースを，自由に組み合わせることができる mSMS ハイブリッドプログラミングモデルを提案している．本報告では，マルチノードマルチ GPU によるステンシル計算を例として，プログラム記述とその性能について述べる．ハードウエア環境の異なる 2 つのクラスタシステムにおいて，mSMS＋OpenMP＋OpenACC を用い，既存の mSMS + OpenMP による性能に比べ，およそ 10 倍の性能を獲得した．

キーワード：PGAS，ディレクティブベース，API，共有メモリプログラミングモデル，マルチスレッド，マルチノードプログラミング，クラスタ，ソフトウェア分散共有メモリ，マルチ GPU，大域アドレス空間，OpenACC, OpenMP

1. はじめに

クラスタ利用による高性能計算では，MPI が広く用いられるが，分散メモリモデルによる開発の低生産性が問題となっており，現在，PGAS（Partitioned Global Address Space）モデルと総称される様々な言語・API が提案されている[3][4][5]．これらは，大域データ配列や大域インデックスを利用可能とし，高生産性を実現する．しかし，その多くは，アクセスできる大域データ範囲に制限がある，大域データアクセス時に明示的な通信記述が必要であるなど，多くの不自由さが存在する．

クラスタの各ノードにおける CPU マルチコア処理には，多くの場合 OpenMP[1]が用いられる．OpenMP はディレクティブベースの API で，#pragma 文を逐次プログラムに追加することで簡単に並列化できる．一方，GPU マルチコア処理では，これまで CUDA[7]が主流であったが，OpenMP と同様のディレクティブベース API である OpenACC[2]が広く使われるようになってきた．クラスタにおけるプログラミングは，現状，分散メモリモデルによる MPI と，共有メモリモデルによる OpenMP, OpenACC(および CUDA)を組み合わせることが，一般的である．

これに対し，我々が開発している分散共有メモリシステム mSMS[12]では，クラスタの全ノードプロセスに同一の共有大域アドレス空間を提供し，マルチノード並列プログラムとノード内マルチコア並列プログラムを，共有メモリモデルを用いてシームレスに記述できる．

我々は，図 1 に示すような，(1)mSMS で提供している 3 つのマルチノード並列インタフェースと，(2)既存のマルチコア並列インタフェースを，自由に組み合わせることができるプログラミング環境，mSMS ハイブリッドプログラミングモデルを提案する．

このモデルでは，利用可能なハードウェア・ソフトウェア環境，それぞれの応用処理の特性，ユーザの知識経験によって，それぞれのインタフェースをユーザが自由に組み合わせて開発することができ，それによって最大限の開発生産性を実現する．

本報告では，mSMS ハイブリッドプログラミングモデルにおける SMS 関数 + OpenMP + OpenACC を用い，容易な記述でマルチノードマルチ GPU ステンシル計算処理を実装し，複数のクラスタシステムにおける性能を報告する．

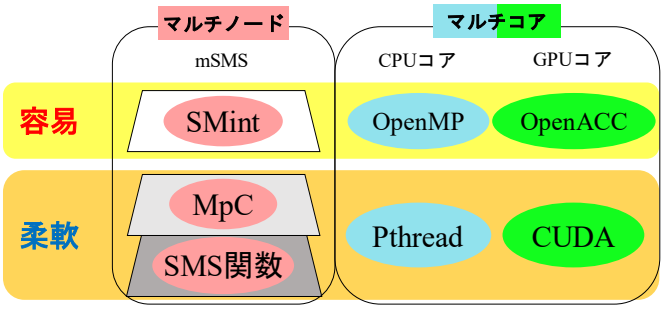


図 1　mSMS ハイブリッドプログラミングモデル

2. mSMS ハイブリッドプログラミングモデル

本節では，mSMS ハイブリッドプログラミングモデルの基本方針を述べ，このモデルのマルチノード並列に用いている分散共有メモリシステム mSMS における並列プログラミングについて述べる．

†1 成蹊大学　Seikei University.

2.1 基本方針

mSMS ハイブリッドプログラミングモデルにおける基本方針は以下である．

① 新しい言語，独自文法・インタフェースを新たに学習するコストを最小限にする．

② 既に多くのユーザが有効利用し，現在も機能向上している既存インタフェースをそのまま利用できる．

③ マルチノードにおいて，大域アドレス空間，共有データを実現し，従来プログラムと同様に，共有データに各ノードから自由にアクセスができる．

④ 利用可能なハードウェア・ソフトウェア環境，応用処理の特性，ユーザの知識・経験，性能と生産性に応じて，自由にプログラミングインタフェースを選べる．たとえば，最高性能を得るため，記述の柔軟性は高いが複雑なプログラミングインタフェースを選ぶか，多少の性能低下よりもプログラム開発の容易さを重視したプログラミングインタフェースを選ぶかの選択が可能であること．

これらの方針をすべて満たすことで，多様で異なる開発生産性を最大限に引き出すことができる．

2.2 mSMS における並列プログラミング

mSMS ハイブリッドプログラミングモデルのうち，マルチノード並列に用いる分散共有メモリシステム mSMS [12]では，図 2 のような 3 つのインタフェースが利用可能である．このうち SMint と MpC はいずれも最下層の SMS ライブラリ関数による C プログラムへ変換される．

2.2.1 SMS ライブラリ関数利用による C プログラム

第一の手法は，SMS ライブラリ関数を用いた C プログラムである．MPI の rank 番号とプロセス数に対応する sms_rank，sms_nprocs 定数が利用できるので，ノード毎に異なる処理をさせることもできる．図 3 に行列ベクトル積のプログラム例を示す．マルチノード上で共有される大域データは，sms_alloc あるいは，sms_mapalloc を用いて，動的に確保する．sms_alloc は一つの指定ノードに指定サイズの大域データを割り付ける．sms_mapalloc は，主に大域データ配列の分散マッピング用で，配列次元毎に分割数を指定し，指定した複数の割り付けノードに，サイクリックにデータを分散マッピングする[13].

2.2.2 大域データ型配列宣言による MpC プログラム

第二の手法は，データ分散マッピング指定付き多次元配列宣言を利用することができる MpC プログラムである．MpC は，C に対し最小限の拡張を施したもので[13]，図 4 に示すような大域データ型 shared による配列宣言を可能にしている．数値計算などで用いることの多い配列宣言が記述しやすい．図 4 の shared データの配列宣言は，MpC トランスレータによって，図 3 における sms_mapalloc 関数の呼び出しに自動変換される．

2.2.3 SMint インクリメンタルプログラミング

第三の手法 SMint[14]は，ディレクティブベースの API で，OpenMP や OpenACC と同様に，逐次プログラムのループ for 文に，pragma SMint を加えることにより，容易にマルチノード並列処理を記述することが可能で，インクリメンタルプログラミングを実現する．また，ループ文をマルチノード並列にする機能に加え，マルチノード並列セクションの前後での遠隔データの一括転送（データローカライズ）を行う指示句（copyin, copyout, copy, create など）を加えることができ，予め，並列セクション開始前に必要なデータをローカルノードにキャッシュすることができる．並列セクション終了時のデータ一貫性同期とキャッシュの扱いも効率化する．これらの指示句を用いない場合には，mSMS ランタイムが，応用プログラムが実行時に遠隔データアクセスしたことを検知してから，SMS ページサイズ単位で遠隔ノードからデータフェッチをする．図 5 は図 3 を SMint で記述し，配列 vec2 を全ノードに分散させ，上述の copyin 指示句により遠隔データのフェッチを行ったプログラム例

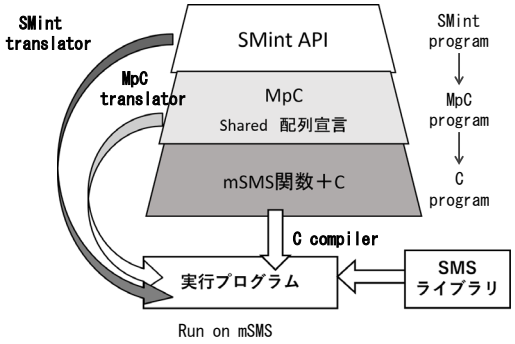


図 2 mSMS におけるプログラミング環境

```
#include <sms.h>          //C program by using SMS library functions
#define N …
int main(int argc, char *argv[] )
{  int size, st, ed;          //Area of each node
   double *vec1, *vec2;  //Pointers for 1D array vec1[N] and vec2[N]
   double (*array) [N];  // Pointer   for 2D array array[N][N]
   // dim: size of array, div: division number of distribution map
   int dim[3]={N, N,-1}, div[3]={1, 1,-1};
   sms_startup(&argc, &argv);

   vec1 = (double*)sms_alloc(sizeof(double), N, 0); // allocation vec1[N] to node0
   vec2 = (double*)sms_alloc(sizeof(double), N, 1); // allocation vec2[N] to node1
   div[0]=sms_nprocs;        //  Split array into bands, distributed mapping to all nodes
   array= (double(*)[N]) sms_mapalloc(dim, div, sizeof(double), 0, sms_nprocs );

   size=N/sms_nprocs;
   st=size * sms_rank; ed=size * (sms_rank+1); //Area of each node
   #pragma omp parallel for          //  Multithreaded parallel execution in each node
   for( i=st; i<ed; i++) {           // Parallel execution of for(i=0; i<N; i++) in all nodes
      for(k=0; k<N; k++)    vec2[i]= array[i][k] * vec1[k];     // Matrix-vector multiplication
   }
   sms_barrier();
   sms_shutdown();
}
```

vec2[N] = array[N][N] * vec1[N]（n1 = n0 n1 n2 n3）
Distributed map by using 4 nodes

図 3 SMS ライブラリ関数利用による C プログラム

```
shared double vec1[N] ::[1](0,1); //node 0 にマッピング
shared double vec2[N] ::[NPROCS](0, NPROCS); //全ノードに分散マッピング
shared double array[N][N] ::[NPROCS][](0, NPROCS); //全ノードに分散マッピング
```

図 4 MpC における大域データ型 shared による配列宣言

である．

2.3 自由な組み合わせによるプログラミング

これまでに，mSMS ハイブリッドプログラミングモデルのうち，mSMS と CPU マルチコア(OpenMP)の組み合わせにより，プログラム記述の容易さと性能の両面で成果を得てきた．大規模クラスタ（東工大 Tsubame3.0）では，180 ノード x 52 コア（合計 9360 コア）を利用し，各ノードに 128GB

```
#include <smint.h>        // Program by using #pragma SMint copyin
#define N …
#pragma SMint shared ::[](0,1) ;      // Mapping to node0
double vec1[N] ;
#pragma SMint shared ::[NPROCS](0, NPROCS ) ; //Distributed mapping to all nodes
double vec2[N] ;
#pragma SMint shared ::[NPROCS][](0,NPROCS) ; // Distributed mapping to all nodes
double array[N][N] ;

int main(int argc, char *argv[] )
{
  // Prefetch vec1(mapped to node0) to local on all nodes before parallel execution
  #pragma SMint parallel for  copyin (vec1[ ])    //Parallel execution on all nodes
  #pragma omp parallel for
  for( i=0; i<N; i++) {
    for(k=0; k<N; k++)
        vec2[i]= array[i][k] * vec1[k];    // Matrix-vector multiplication
  }
}
```

vec2[N] = array[N][N] * vec1[N]　Distributed map by using 4 nodes

図 5　copyin 利用による SMint プログラム

のデータを分散配置して，全体として 22.5TB の大域共有データを確保し，約 30TB の巨大な仮想アドレス空間を持つプロセスを各ノードに稼働させて，大規模ステンシル計算を行い，MPI と同等レベルの性能を得ている[12]．また，SMint と既存 PGAS 言語の XcalableMP[4]，Berkeley-UPC[9]をステンシル計算で比較したところ，記述性，性能の両面で SMint が最も優れていた[15]．最近では，高速な CPU(Intel skylake)クラスタシステム（九大 ITO-A，64 ノード）を利用し，複雑で計算量の多い音響解析処理（FDTD）を行い，成果を得た[17]．さらに配列アクセスのみの応用にとどまらず，従来，クラスタ上での記述・実装が困難であったポインタアクセスによる大規模グローバルツリーを用いた Barnes-Hut アルゴリズムの実装[18]も行っている．

そこで，本報告では，CPU に比べ高速な GPU を搭載した GPU クラスタ向けのプログラムの記述と性能について述べる．マルチノードで 1 ノードあたり 1 台の GPU を用いる場合には，mSMS と OpenACC の組み合わせで記述できるが，GPU メモリはホストのメインメモリに比べ小さく，大規模データを扱うのに適していない．そのため今回は mSMS と OpenMP と OpenACC を組み合わせ，図 6(a)に示すような，1 ノードあたり複数の GPU を利用する処理について述べる．

3. マルチノードマルチ GPU プログラミング

本節では，マルチノードマルチ GPU プログラムの，典型的な記述例について述べたのち，ステンシル計算の実装について述べる．

3.1 SMS 関数 ＋OpenMP＋OpenACC による記述

図 6(a)のようなマルチノードマルチ GPU システムを利用する処理の典型的な記述を図 6(b)に示す．図 6(b)では，sms_startup 関数によってマルチノード並列を開始し，複数 GPU を同時に利用するため，各ノードで OpenACC の acc_get_device_num 関数によって 1 ノードに搭載されている GPU の数を取得し，OpenMP によって GPU 個数と同じ CPU コアを利用，さらに各 CPU コアが OpenACC の acc_set_device_num 関数によって異なる GPU を選択し，利用するという方式である．このように，mSMS ハイブリッドプログラミングモデルでは，マルチノードマルチ GPU に

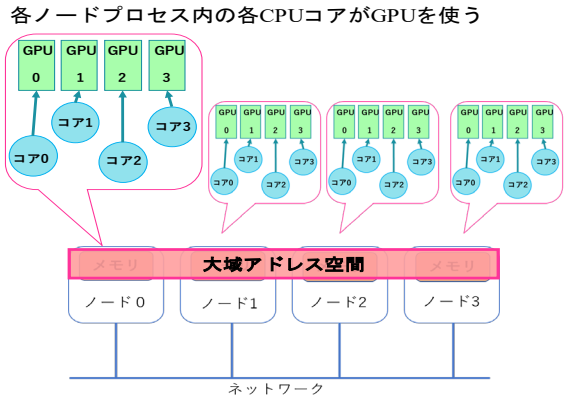


図 6(a)　マルチノードマルチ GPU システムの利用例

```
#include <stdio.h>
#include <omp.h>
#include <openacc.h>
#include <sms.h>                          ← ヘッダインクルード
…
int main(int argc, char **argv)           ← マルチノード実行
{
  sms_startup(&argc, &argv);
  int numgpus = acc_get_num_devices(acc_device_nvidia );
…                                         ← マルチコア実行
#pragma omp parallel num_threads(numgpus)
 {
   int tid = omp_get_thread_num();
   acc_set_device_num(tid, acc_device_nvidia);
   #pragma acc enter data copyin(…) create(…)
   #pragma acc kernels
   {                                      ← GPU実行
    …
   }
   #pragma acc exit data copyout(…)
 }
 sms_shutdown();
}
```

図 6(b)　SMS 関数 ＋OpenMP＋OpenACC による記述

よる並列処理が，SMS 関数 ＋ OpenMP ＋ OpenACC の組み合わせにより，従来の MPI＋CUDA による記述と比べ，容易に記述できる．

3.2 マルチノードマルチ GPU によるステンシル計算

今回，性能評価には，ステンシル計算処理を用いる．ステンシル計算は，典型的な配列データアクセス処理であり，近傍データを使用して各要素のデータ更新を繰り返す処理である．各要素のデータ更新は独立に処理できることから，並列性が高く，GPU を利用するのに適している．また，ステンシル計算の通常のアルゴリズムでは，時間ステップ 1 回ごとにデータ更新を行うが，テンポラルブロッキングア

ルゴリズム[19]では，袖領域の幅を増やし，複数時間ステップのデータ更新をまとめて行う．これにより，マルチノードマルチ GPU ステンシル計算では，ノード間の袖領域通信の回数，CPU-GPU 間の袖領域通信の回数を減らすことができる．その一方で，本来必要のない袖領域の計算も増加するため，計算時間は増加する．つまり，通信時間と計算時間のトレードオフが存在する．

3.3 マルチノードマルチ GPU テンポラルブロッキングステンシル計算の実装

本節では，3.3.1 でマルチ GPU テンポラルブロッキングステンシル計算の概要と mSMS によるマルチノード処理について述べたのち，3.3.2 では各 GPU における処理，3.3.3 では CPU-GPU 間データ転送部分の具体的な記述について述べる．

3.3.1 mSMS によるマルチノード処理

図 7 にマルチノードマルチ GPU ステンシル計算処理の記述の典型を示す．この記述例では 3 次元データ配列 A,B の Z 方向をマルチノードで分割し，さらにノード内の処理を，4 台の GPU によって，同じく Z 方向で分割して処理を行う．大域データ配列 A,B は，図 7 中 22, 23 行目の sms_mapalloc 関数によって Z 方向で各ノードに分散マッピングされる．26, 27 行目でデータの初期化（あるいは入力）を行った後，28 行目でデータのノード間のデータ一貫性同期を行う．

30 行目で搭載 GPU 数と同数のスレッドを起動し，30～71 行目の OpenMP parallel セクション内で，各スレッドが，担当する GPU へのデータ転送， GPU での計算，GPU から計算結果の転送を行う．GPU 計算終了後，72 行目で sms_sync_drop 関数によりノード間の実行同期，各ノード間の袖領域データのキャッシュを廃棄する．最後に，73 行目で sms_shutdown 関数によりマルチノード処理を終了する．（本コードでは結果出力は省略している．実際には，データ検証コードを実行している．）

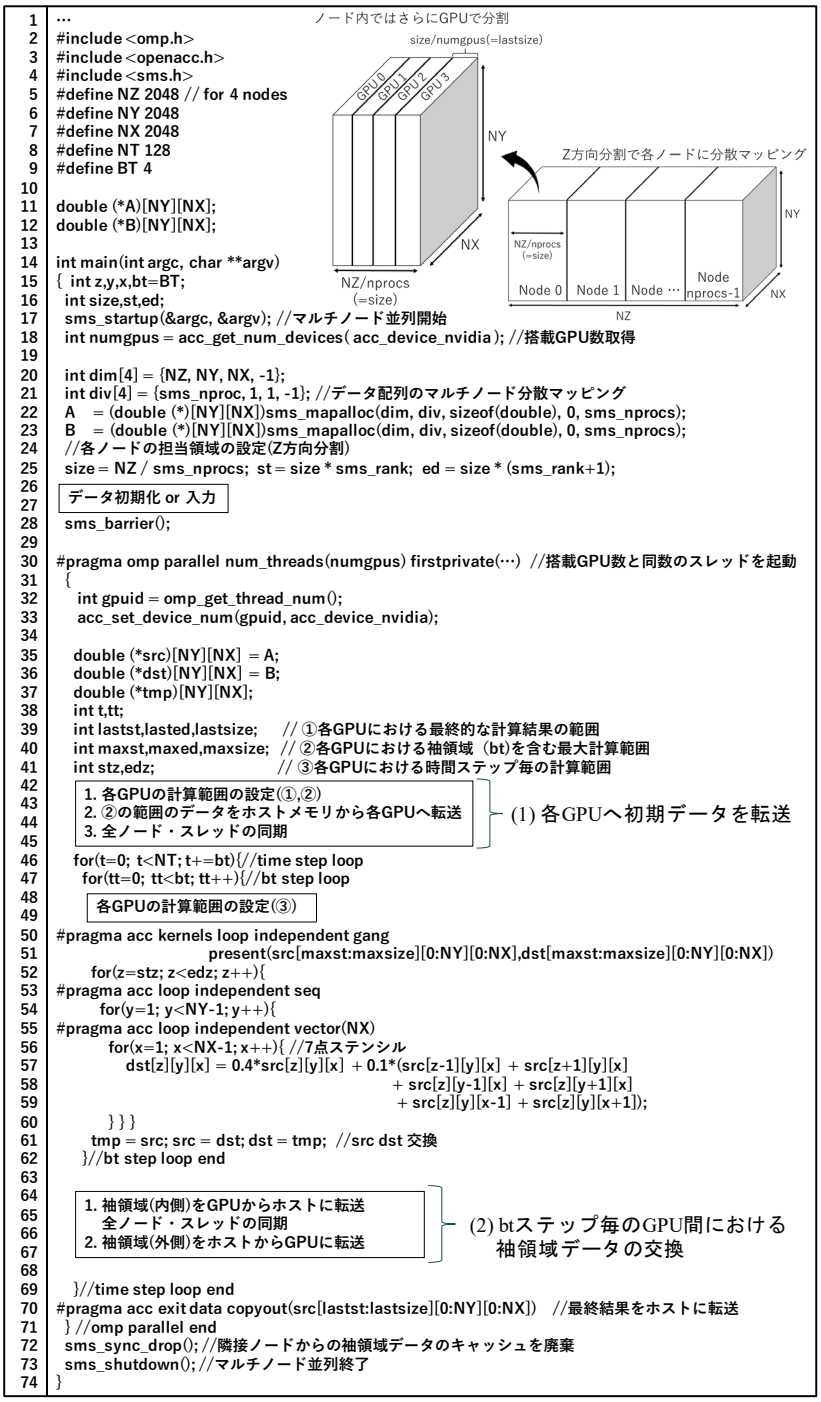


図 7　SMS 関数 + OpenMP + OpenACC による
マルチノードマルチ GPU ステンシル計算の記述

3.3.2 各 GPU における処理

図 7 の 30～71 行目の OpenMP の parallel セクション内には，各スレッドの担当する GPU における処理が記述されている．

まず，42～45 行目の(1)部分では，各 GPU の担当データ領域全体を，ホストから各 GPU へ転送する．この部分の具体的な記述については 3.3.3 に後述する．図 8 は，ローカルノード内における各 GPU の担当領域と，ホストから各 GPU へのデータ転送の様子を示す．図 8 の両端の斜線部分の袖領域データは隣接ノードに存在するが，mSMS では，アクセス時に遠隔ノードからデータが透過的に転送されローカルにキャッシュされるので，特別な記述をせずに袖領域にアクセスできる．

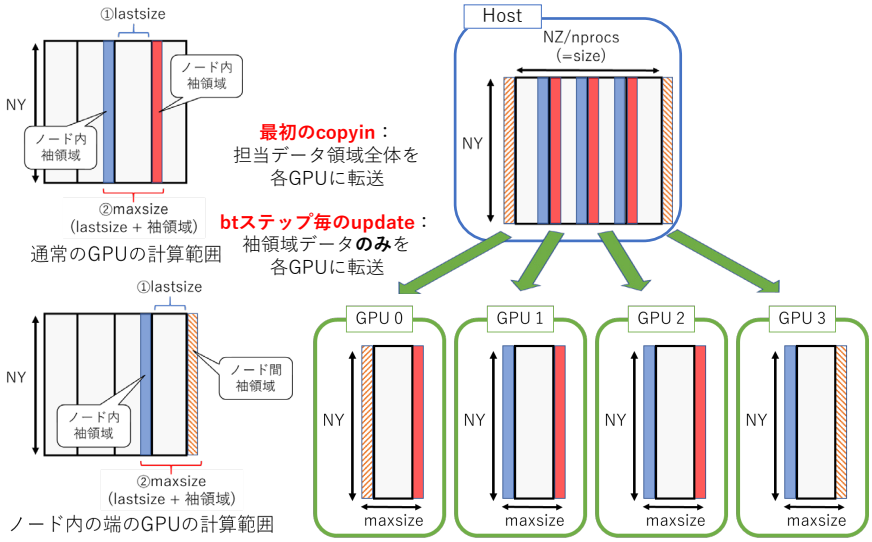


図 8　ローカルノード内における各 GPU の担当領域と，各 GPU へのデータ転送

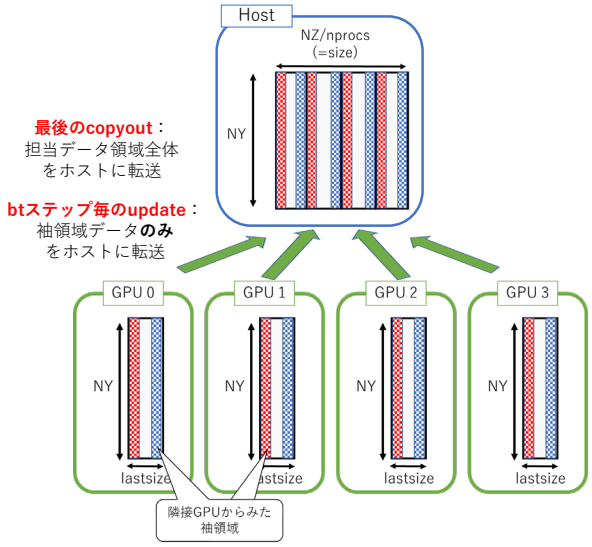


図 9　各 GPU からホストへのデータ転送

次に，46～69 行目はステンシル計算における時間ステップの繰り返し処理に対応し，その中の 47～62 行目はテンポラルブロッキングアルゴリズムにおける，時間ブロックサイズ bt の時間ステップ分の繰り返しとなっている．

64～67 行目の(2)部分では bt ステップ毎に GPU 間で袖領域データの交換を行うため，bt 回更新後の袖領域データをホストメモリ経由で転送する．この部分の具体的な記述は 3.3.3 に後述する．42～45 行目の(1)部分で各 GPU の担当データ領域全体を転送していたのに対し，64～67 行目の(2)部分では，図 9 に示すように，まず，担当データ領域のうち，隣接 GPU が必要とする袖領域データのみを GPU からホストに転送し，その後，図 8 に示すように，袖領域データのみをホストから各 GPU に転送することで，CPU-GPU 間のデータ転送量を減らし，高速化している．

46～69 行目の時間ステップの繰り返しが終わると，70 行目の OpenACC の copyout によって，各 GPU 上の計算結果をホストに転送したのち，各 GPU における処理を終了する．

3.3.3　各 GPU への初期データ転送と GPU 間袖領域データ交換

図 7 42～45 行目(1)の各 GPU への初期データ転送部分の記述は図 10 のようになっている．(1)-1 は，図 8,9 に示すデータサイズや座標を計算する．(1)-2 の src 配列の copyin で，ノード内の端を担当する GPU（図 8 の GPU0 と GPU3）が隣接ノードの袖領域データにアクセスするが，隣接ノードからの袖領域データ転送は，mSMS の機能によって実行時に，自動的に行われるため，この部分にノード間通信の記述は必要がない．

図 7 の 64～67 行目(2) bt ステップ毎の GPU 間袖領域データ交換部分の記述は図 11 のようになっている．(2)-1 OpenACC の update host によって，隣接 GPU が必要とする袖領域データのみをホストに転送し，(2)-2 update device によって(2)-1 で更新された隣接 GPU の袖領域データのみを各 GPU へ転送している．(2)-2 では(1)-2 と同様に，隣接ノードからの袖領域データ転送は，mSMS により行われるため，特別な記述は不要である．

```
(1)-1  計算範囲の設定：①各GPUにおける最終的な計算結果の範囲(laststt,lasted,lastsize)
       計算範囲の設定：②各GPUにおける袖領域（bt)を含む最大計算範囲(maxst,maxed,maxsize)
(1)-2  #pragma acc enter data copyin(src[maxst:maxsize][0:NY][0:NX]) //src配列はGPUへ転送
       #pragma acc enter data create(dst[maxst:maxsize][0:NY][0:NX]) //dst配列はGPUでcreate
       //dst配列はGPU上でsrc配列からコピーする
(1)-3  #pragma omp barrier  //ローカルノードの全GPU制御スレッドの同期
       #pragma omp master  //マスタースレッドによる全ノードの同期
         sms_sync();
```

図 10　各 GPU への初期データ転送の記述

```
       if(t!=NT-bt){ //最後の時間ステップ以外
       //左隣に計算GPUがあるGPUは，左データをホストに転送
(2)-1  #pragma acc update host(src[laststt:bt][0:NY][0:NX]) if(!(sms_rank==0 && gpuid==0))
       //右隣に計算GPUがあるGPUは，右データをホストに転送
       #pragma acc update host(src[lasted-bt:bt][0:NY][0:NX]) if(!(sms_rank==sms_nprocs-1 && gpuid==numgpus-1))
       全ノード・スレッドの同期
       //左隣に計算GPUがあるGPUは，ホストから左袖領域を取得
       #pragma acc update device(src[maxst:bt][0:NY][0:NX]) if(!(sms_rank==0 && gpuid==0))
(2)-2  //右隣に計算GPUがあるGPUは，ホストから右袖領域を取得
       #pragma acc update device(src[lasted:bt][0:NY][0:NX]) if(!(sms_rank==sms_nprocs-1 && gpuid==numgpus-1))
       同期
       }//t!=NT-bt
```

図 11　bt ステップ毎の GPU 間における袖領域データの交換の記述

3.4 遠隔データフェッチを効率化する preload と overload

mSMS では，MpC による shared 配列宣言や sms_alloc，sms_mapalloc 関数を利用して確保された大域データには，どのノードのどのスレッドからでも制限なくアクセスすることができる．したがって，前節で述べたように隣接ノードにある袖領域へのアクセスにも特別な記述が不要でプログラムは非常に書きやすい．特に応用において，複雑な境界条件の設定などには，プログラム作成の生産性が向上する[17]．また，実行時にアクセス先が決まるような応用，たとえば，大域ツリーへのマルチノードからの非同期アクセス[18]なども記述できるという利点がある．

一方，ステンシル計算などのように，並列計算部分で用いる遠隔データがあらかじめ「袖領域」として既知である場合，計算開始前に一括して袖領域をローカルノードにキャッシュしておけば，計算の途中でアクセスの度にページ単位に転送するより効率が良い．このような場合，遠隔データの prefetch を行う preload 関連関数が，複数用意されている．計算開始前に，マスター―スレッドなどが呼び出し，必要な大域データを一括してローカルノードにキャッシュしてから，計算を開始することができる．

preload_array 関数は，図 12（a）に示すように，大域多次元配列の部分配列を一度にローカルにキャッシュする汎用関数である．これらの関数では，SEGV シグナルによるページ毎の転送ではなく，遠隔ノードにある各連続領域を一度に転送する．内部では，mSMS 通信スレッドが，preload_array 関数の引数から連続領域を計算し，該当データを所有する遠隔ノード毎に連続ページの要求を複数送る．同時に，内部受信スレッドが，遠隔ノードからの連続ページを次々に受け取る．これにより，計算中に大域データにアクセスしてから，１ページ毎をフェッチするよりも，遠隔ノードとの通信回数が減り，計算の中断がなくなるという利点がある．

さらに，すでにローカルにあるキャッシュ領域に，データを上書きするための overload_array 関数（overwrite の意味）用意している．preload との違いは，preload 内部で行っているページの状態チェック(キャッシュにあるか否か，書き込み可能か)などを省略しているので，すでにキャッシュされている領域への上書き prefetch が高速にできる．

ステンシル計算では，遠隔ノードから繰り返し袖領域を得る必要があるため，初回の各 GPU への袖領域を含む担当データ領域の転送時（図 7(1)，図 10）に，あらかじめ preload_array を用い隣接ノードの袖領域をローカルノードにキャッシュしておくと高速化ができる．同様に，bt 時間ステップ毎の袖領域の交換（図 7(2)，図 11）の際にも，隣接ノードとの間で overload_array を用いて，新しく更新された隣接ノードの袖領域を，ローカルノードのキャッシュに上書きすることができる．

具体的には，ホストメモリから各 GPU へ担当領域を転送する際（図 7(1)）に，図 10 の(1)-1 と(1)-2 の間に，図 12(b)に示すような sms_preload_array 関数による記述を挿入することで，予め，隣接ノードの袖領域をローカルノードに prefetch してから，GPU へのデータ転送を行うことができる．prefetch を行うのは，図 8 のローカルノードの担当領域のうち，両端のデータ領域を計算する 2 つの GPU（4 ノードの場合 GPU0 と GPU3）を担当するスレッドで，隣接する遠隔ノードから．斜線で示す袖領域を preload する．

同様に，bt 時間ステップ毎に，各 GPU の新しい袖領域の計算結果がホストメモリに戻された後， 図 10 の(2)-1 と(2)-2 の間に，図 12(c)に示すような記述を挿入することで，sms_overload_array 関数により，隣接ノードから最新の袖領域をローカルノードにキャッシュすることができる．

次節で述べる性能評価ではこの prefetch 機構を用いて評価を行った．

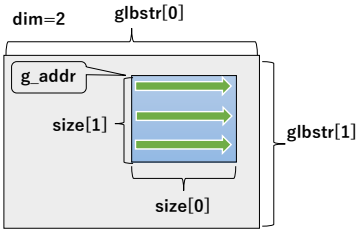
sms_preload_array(), sms_overload_array()

```
int sms_preload_array(void* g_addr, size_t glbstr[], size_t size[], int dim, int elemsize, int mode)
int sms_overload_array(void* g_addr, size_t glbstr[], size_t size[], int dim, int elemsize, int mode)
```

返値　実行結果　0 正常

引数

void *	g_addr	グローバルアドレス,
size_t[]	glbstr	大域配列の全体サイズ
size_t[]	size	prefetchする部分配列サイズ
int	dim	配列次元数
int	elemsize	要素データサイズ（バイト）
int	mode	書き込みフラグ(0:RO,1:RW)



(a) sms_preload_array 関数と sms_overload_array 関数

```
size_t glbstrd[3] = { NZ, NY, NX };  //各次元の全体サイズ
size_t prel_size[3] = { bt, NY, NX }; //各次元の転送したいサイズ

if(sms_rank!=0 && gpuid==0){     //左側の隣接ノードから袖領域を取得
  sms_preload_array( &src[maxst][0][0], glbstrd, prel_size, 3, sizeof(double), 1);
}
else if(sms_rank!=sms_nprocs-1 && gpuid==numgpus-1){//右側の隣接ノードから袖領域を取得
  sms_preload_array( &src[lasted][0][0], glbstrd, prel_size, 3, sizeof(double), 1);
}
```

(b) 隣接ノードから袖領域を prefetch する sms_preload_array 関数

```
if(sms_rank!=0 && gpuid==0){     //左の隣接ノードから袖領域を取得
  sms_overload_array( &src[maxst][0][0], glbstrd, prel_size, 3, sizeof(double), 1);
}
else if(sms_rank!=sms_nprocs-1 && gpuid==numgpus-1){//右の隣接ノードから袖領域を取得
  sms_overload_array( &src[lasted][0][0], glbstrd, prel_size, 3, sizeof(double), 1);
}
```

(c) 同じ領域を再度 prefetch する sms_overload_array 関数

図 12　ステンシル計算を高速化する隣接ノード袖領域一括 prefetch

4. 性能評価

4.1 実験環境

今回の測定では，複数 GPU を 1 ノードに搭載している東大 Reedbush-L[11]（表 1）と東工大 Tsubame3.0[10]（表 2）を用いた．また，OpenACC を利用するため，C コンパイラには PGI コンパイラを用いた．コンパイラオプションとしては，-O2 -acc -mp –ta=tesla,cc60 -Minfo=accel を指定した．

実験当初, Reedbush-L での稼働は確認できていたが，全く同様のプログラムを Tsubame3.0 で動作させようとしたところ，実行時に CUDA のエラーが出力され，稼働に至らなかった．両環境の違いを調査すると，PGI コンパイラが Reedbush-L では LLVM 版，Tsubame3.0 では no-LLVM 版であることが判明した．そこで Tsubame3.0 にも LLVM 版をインストールして頂き，再度稼働実験をしたところ，正常な稼働を確認することができた．詳細は不明であるが，PGI コンパイラが LLVM 版か否かで，生成されるバイナリに何か違いがあったのではないかと考えている．

4.2 マルチノードマルチ GPU ステンシル計算の性能

2 つのクラスタシステム，Reedbush-L と Tsubame3.0 における 3 次元 27 点ステンシル計算処理の実行時間を図 13 と図 14 に示す．1 ノードあたり 32GB（8GB x 4GPU）のデータを処理する場合の性能（weak scaling performance）を表す．テンポラルブロッキングアルゴリズムの時間ブロックサイズには，今回の総時間ステップ数 128 において最速であった bt＝4 を用いている．いずれの性能も，2 ノード以上の計測では隣接ノードと通信をする rank 1 のスレッド 0 における計測結果を示しているが，1 ノードによる計測では rank 0 のスレッド 0 における計測結果を示しているため，2 ノード以上の結果とは時間成分の分布が異なっている．

図 13 は Reedbush-L における実行時間を示す．全体実行時間は 1 ノード利用時 14 秒，最大 16 ノード利用時では 18.7 秒である（1 ノードに対し 74.8%の効率）．図 14 は Tsubame3.0 における実行時間を示す．全体実行時間は 1 ノード利用時 16 秒，最大 64 ノード利用時では 18.9 秒である（1 ノードに対し 84.6%の効率）．いずれのクラスタでも，ノード間同期はノード数が増えると増加し，全体の 5-18 %を占める．

一方，mSMS による袖領域の転送は全体の 3-10%程度と小さい．これに対し，ローカルノード内の CPU-GPU 間の袖領域の転送時間が, 全体のおよそ 20-30%を占めている．GPU の搭載メモリ量を超える処理をする上で，ローカルノードのホストメモリと GPU メモリの間のデータ転送が非常に大きなコストになっていることがわかる．隣接 GPU との袖領域データの交換に関しては，ホストメモリを介さずに，GPU 間のハードウエアリンクを用い直接通信する方式，GPU direct P2P[8]を用いて高速化できると考えている．

Tsubame3.0 における mSMS + OpenMP による 27 点ステンシル計算[12]では, 1 ノードあたり 128GB データ を CPU マルチコア（52 スレッド）で処理していたが, 本実験では，

表 1　Reedbush-L

CPU	Intel Xeon CPU E5-2695 v4 @ 2.1GHz * 2
Num of Core / Threads	18 Core * 2
Memory	256GiB
Network	InfiniBand EDR 100Gbps 4*2リンク
GPU	Tesla P100 16GB * 4
OS	Red Hat Enterprise Linux7
modules	cuda9/9.2.148, pgi/19.3, intel-mpi/2018.1.163

表 2　Tsubame3.0

CPU	Intel Xeon CPU E5-2680 v4 @ 2.40GHz * 2
Num of Core / Threads	14 Core / 28 Threads * 2
Memory	256GiB
Network	Intel Omni-Path HFI 100Gbps * 4
GPU	Tesla P100 16GB * 4
OS	SUSE Linux Enterprise Server 12 SP2
modules	cuda/9.2.148, pgi-llvm, pgi/19.1, intel-mpi/18.1.163

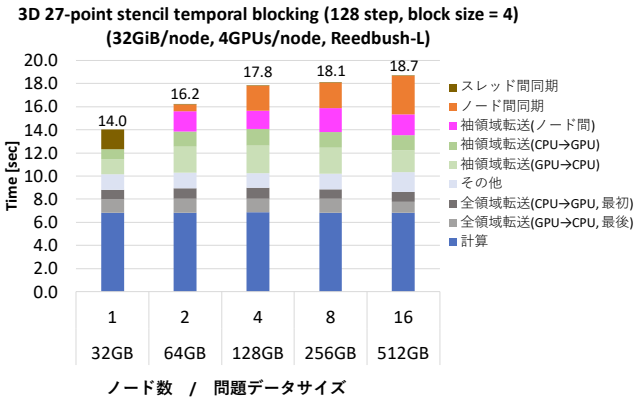


図 13　Reedbush-L における実行時間

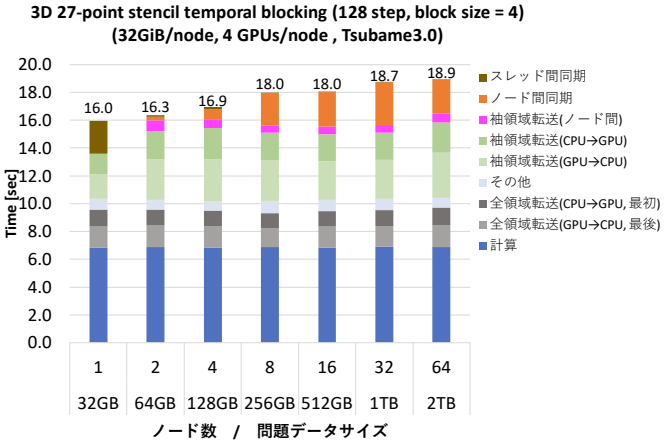


図 14　Tsubame3.0 における実行時間

1 ノードあたり 32GB データを 4GPU で処理している．データ処理量が 1/4 になっているため，前者の mSMS + OpenMP による処理時間の 1/4 と，今回の mSMS＋OpenMP + OpenACC の結果を単純に比較すると，およそ 10 倍高速になっている．扱う問題サイズ，mSMS のバージョンが異なるため，単純な比較はできないが，ステンシル計算において GPU は CPU よりも圧倒的に高速であるといえる．

結果として，SMS 関数 + OpenMP + OpenACC の 3 つのインタフェースを組み合わせた，マルチノードマルチ GPU ステンシル計算のプログラムが，CPU やネットワークなどのハードウエア構成の異なる 2 つのクラスタシステムにおいて正常に稼働し，既存の mSMS + OpenMP による性能に比べ，およそ 10 倍の性能を獲得した．

5. おわりに

mSMS ハイブリッドプログラミングモデルを提案し，そのうち SMS 関数と既存の OpenMP，OpenACC を組み合わせて，マルチ GPU クラスタにおける大域データを用いたマルチノードマルチ GPU ステンシル計算プログラムを，実装した．また，既存の CPU コアを用いた結果と比較して，およそ 10 倍の性能を得ることができた．

さらにこのプログラムを，ハードウエア構成の異なる 2 つのクラスタシステムにおいて稼働させ，mSMS ハイブリッドプログラミングモデルの高移植性を示した．

今回は最も広く利用され，容易な記述が可能な OpenMP と OpenACC を用いたが，mSMS ハイブリッドプログラミングモデルでは，これらとの組み合わせに限定しておらず，Pthread や CUDA など他のインタフェースと自由に組み合わせて利用できることが最大の特徴である．

mSMS ハイブリッドプログラミングモデルは，利用可能なハードウェア・ソフトウェア環境（ノード数，CPU，搭載 GPU 個数，メモリ容量など），応用アルゴリズム（問題データ規模，計算の特性），ユーザの技術・知識量などにより，ユーザが複数のプログラミングインタフェースの中から自由に選択し，組み合わせ，容易にプログラミングできる．

現在，今回のマルチノードマルチ GPU ステンシル計算に，ノード内 GPU-GPU 間データ転送（GPU direct P2P[8]）を組み込んだ実装を行っている．大きなオーバヘッドとなっている CPU-GPU 間データ転送を GPU-GPU 間転送に変更することでさらなる高速化を図る．これには，今回のプログラムのインタフェース（SMS 関数+ OpenMP + OpenACC）にさらに CUDA を加えることで，実現可能である．

謝辞

本報告におけるマルチノードマルチ GPU プログラムの稼働実験に際し，東京工業大学 学術国際情報センターの皆様，及び同センター先端研究部門教授 遠藤敏夫先生には大変お世話になりました．ここで深謝致します．また，常日頃，ご指導，ご支援を頂いている成蹊大学大学院理工学研究科教授 甲斐宗徳先生にも，感謝申し上げます．

本研究は，学際大規模情報基盤共同利用・共同研究拠点の支援(課題番号: jh190039-ISH)及び，JSPS 科研費（課題番号：JP18K11327）の助成を受けたものです．

参考文献

[1] OpenMP https://www.openmp.org/
[2] OpenACC https://www.openacc.org/specification
[3] M.D. Wael, et al.: “Partitioned Global Address Space Languages”, Journal of ACM Computing Surveys (CSUR), Vol.47, No.62 (2015)
[4] Xcalable MP http://www.xcalablemp.org/ja/
[5] UPC Consortium, UPC Language Specifications Version 1.3, https://upc.lbl.gov/docs/user/upc-lang-spec-1.3.pdf
[6] Numwich, R. and Reid, J. ”Co-Array Fortran for parallel programming”, Technical report ral-tr-1998-060, Rutherford Appleton Laboratory(1998).
[7] CUDA https://developer.nvidia.com/cuda-zone
[8] NVIDIA GPU Direct https://developer.nvidia.com/gpudirect
[9] Berkeley UPC http://upc.lbl.gov/ ver.2.28.9(2018.7.20)
[10] Tsubame3.0 http://www.gsic.titech.ac.jp/tsubame3
[11] Reedbush-L https://www.cc.u-tokyo.ac.jp/supercomputer/reedbush/system.php
[12] 緑川博子："ソフトウェア分散共有メモリシステム mSMS による大規模マルチコアノードにおけるステンシル計算",情処学会，HPC 研究会報告,Vol 2018-HPC-165,No22,pp.1-9，2018
[13] H.Midorikawa : "The Performance Analysis of Portable Parallel Programming Interface MpC for SDSM and pthread", IEEE/ACM CCGrid2005 ,Vol.2, pp.889-896, Fifth International Workshop on Distributed Shared Memory (DSM2005), 2005
[14] 阪口裕梧，西矢和生，緑川博子 :"逐次プログラムからマルチコア・マルチノード並列処理への変換を容易にするディレクティブベース API SMint"，情処学会, HPC 研究会報告，Vol 2018-HPC-167,No 5,pp.1-9,2018
[15] 阪口裕梧，緑川博子：”グローバルビュープログラミングをサポートする PGAS 言語の記述性と性能の比較”，情処学会, HPC 研究会報告，Vol 2019-HPC-170，No 41，pp. 1-10 2019
[16] Y.Sakaguchi, H.Midorikawa : “Programmability and Performance of New Global-View Programming API for Multi-Node and Multi-Core Processing”, IEEE Pacific Rim Conference on Communications Computers and Signal Processing, (2019,8)
[17] R.Tabata, H.Midorikawa, K.Takahashi: "Performance Evaluation of Acoustic FDTD(2,4) Method Using Distributed Shared Memory System mSMS", 2020 International Conference on High Performance Computing in Asia-Pacific Region HPC Asia 2020 , pp.1-2 , Fukuoka, Japan, (2020,1) http://sighpc.ipsj.or.jp/HPCAsia2020/hpcasia2020_posters/poster_52.pdf Poster [2020.2.18 online] http://sighpc.ipsj.or.jp/HPCAsia2020/hpcasia2020_poster_abstracts/poster_abstract_52.pdf abstract [2020.2.18 online]
[18] 緑川博子，柴山悠:"分散共有メモリシステムを利用した Barnes-Hut アルゴリズムの初期並列実装と性能評価"，情処学会, HPC 研究会報告，Vol.2019-HPC-170, No.43,pp.1-8, 2019
[19] G. Jin, T. Endo, and S. Matsuoka, “A Parallel Optimization Method for Stencil Computation on the Domain That Is Bigger Than Memory Capacity of GPUs”, IEEE Cluster2013, 2013.