Vol.2021-HPC-178 No.24
2021/3/16

分散共有メモリシステム mSMS における マルチノード・マルチ CPU・マルチ GPU プログラミング

緑川 博子[†1]，阪口 裕梧[†1]

概要：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（MpC, SMint, SMS ライブラリ関数）と，既存の OpenMP, pthread, OpenACC など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．すでに，マルチノードマルチ GPU による 27 点ステンシル計算において，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 倍の性能を獲得できることを示した．本報告では，１ノードに複数 GPU を有し GPU 間にハードウエアリンク（NV Link）を持つシステムにおいて， mSMS＋OpenMP＋OpenACC＋CUDA により，マルチノード上の大域配列データに対して，GPUDirect P2P による GPU 間直接通信利用を記述でき，ホスト-GPU 間通信オーバーヘッドを軽減できることを示す．

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，マルチコア，GPU，マルチスレッド，マルチノード，クラスタ，ソフトウェア分散共有メモリ，GPUDirect，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，現在も MPI が広く用いられ，分散メモリモデルによるプログラム開発の低生産性が未だ解決されたとは言えない．これを軽減するため，PGAS（Partitioned Global Address Space）モデルと総称される様々な言語・API が提案されてきたが[18]，大域データ配列や大域インデックスを利用可能とするものはあるものの，可能なアクセス範囲がノード隣接データ領域に限らていたり，範囲を超えた大域データアクセスには明示的な MPI のような局所的記述が必須であったり，定義できる大域データサイズに制限があり大規模計算には利用できないなど，多くの不自由さが存在する[7][8]．

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 (multithreaded Shared Memory System) [1,2,3]は，図１のように，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するランタイムシステムを実現する．多くの PGAS が，その専用コンパイラを使って，大域データアクセスを静的なノード間通信などに変換するのとは対照的に，mSMS では，大域空間へのアクセスは，複数の SMS 内部システムスレッドによる効率的な通信制御機能を備えたランタイムシステムがリアルタイムで処理する[2]．

このため，mSMS は，以下のような特徴を持つ．

1) 大域データすべての領域に，どのノード（プロセス）のどのコア（スレッド）からも制限なくアクセスできる．
2) 静的なコンパイラでは変換できない実行時にアクセス先や挙動が決まる動的な応用にも，ユーザが付加的なデータ・通信制御コードを加えることなく実行できる．
3) SMS ランタイムは，各ノードに大規模共通アドレス空間を持つ Unix プロセスとして実行されるため，汎用の C ポインタによる記述や，不規則構造を持つ大域ツリーなどの動的データ生成削除も可能となっている[3]．
4) 提供できる大域データサイズに制限がなく，利用可能なノード物理メモリ容量 ×利用ノード数までの共有メモリが構築できる．資源規模に応じた処理，大規模システムによる巨大データの処理も可能である[2]．
5) アクセスする大域データが事前にわかっている静的処理や，同じデータが繰り返しアクセスされる処理には，遠隔データ事前一括フェッチ（preload 機能），既存キャッシュ更新フェッチ（overload 機能）など，高効率関数が利用でき，さらに性能を向上させることもできる[10]．

5)の機能は，大域配列に更新を繰り返す処理に効果的である．一方，多くのシミュレーションで不可欠な複雑な境界条件の設定では，大域配列と大域インデックスにより，どのノードコアからも自由に値の設定ができることが，プログラム開発上の大きなメリットとなっている[4][5]．

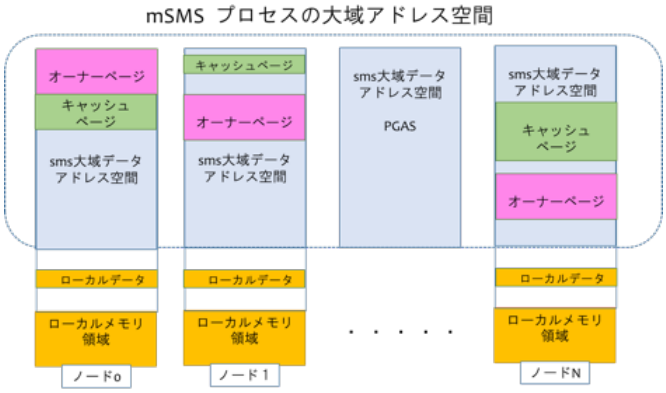


図１　クラスタ上で大域アドレス空間を実現する ソフトウエア分散メモリ mSMS

†1 成蹊大学　Seikei University.

このように，mSMS では，マルチノード・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．SMS ハイブリッドプログラミングモデルでは，図２に示すように，SMS が提供する 3 つのマルチノード並列インタフェース（MpC, SMint, SMS ライブラリ関数）と，既存のマルチコア汎用並列インタフェース(OpenMP[16], pthread, OpenACC[17], CUDA[22])を自由に組み合わせることができる．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，応用処理の特性，ユーザの知識経験などによって，利用するインタフェースをユーザが自由に組み合わせてプログラムを作成する．

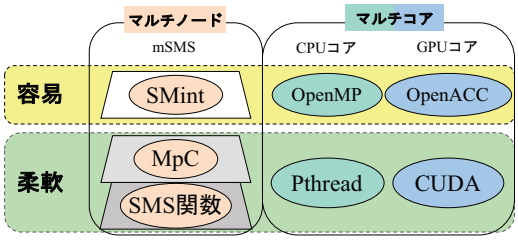


図２　SMS ハイブリッドプログラミングモデル

本報告では，mSMS が提供するプログラミング API とその記述例を示した後，ステンシル計算を題材に，mSMS における GPU プログラミングの記述と性能を示す．すでに，2 つの GPU クラスタにおいて，mSMS＋OpenMP＋OpenACC を用い，既存の mSMS＋OpenMP による性能[2]に比べ，およそ 10 倍の性能を獲得できることを示した[10]. ここでは，さらに 1 ノードに搭載される複数 GPU 間にハードウエアリンク（NVLink）を持つシステムにおいて，GPUDirect Peer to Peer（DirectP2P）を用い，mSMS＋OpenMP＋OpenACC＋CUDA による処理記述例と性能を示す．

2. mSMS ハイブリッドプログラミングモデル

2.1 mSMS における並列プログラミングインタフェース

分散共有メモリシステム mSMS では，図 3 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 [6]と MpC[14,15]は，トランスレータにより，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．他の PGAS 言語のような専用コンパイラは不要で移植性が高い．

2.2 ディレクティブベース API：SMint

図３の最上位にある SMint [6]は，OpenMP や OpenACC と同様に，逐次プログラムに，pragma SMint 文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，図４のように逐次コードに組み合わせて付加することにより，インクリメンタルプログラミングを実現する．ループ文をマルチノード並列にする機能に加え，OpenACC のように，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（事前のデータローカライズ，遠隔データ prefetch と処理結果の一括反映，またはキャッシュの廃棄など）を行う指示句（copyin, copyout, copy, create など）を

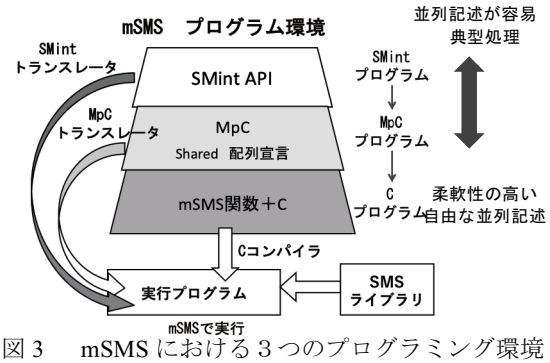


図３　mSMS における３つのプログラミング環境

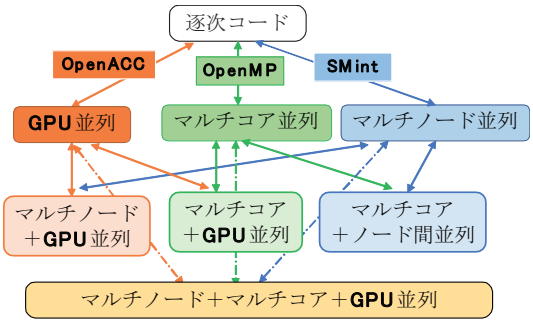


図４　SMint は他の API を組み合わせて逐次コードを容易に並列化

表１　逐次コードとの行数比較と MPI に対する性能

各種言語	プログラムの記述性		性能	特徴
	行数	逐次に対する行数増加率	MPIに対する相対実行時間＊	扱えるグローバルデータサイズ、アクセス可能領域、記述の制限など。
UPC (Global view model)	29	1.07	4.0 ～ 10.4	グローバルデータサイズに制限有 小さい
UPC (Local view model)	57	2.11	1.3 ～ 8.9	MPIと同様のノードを意識したローカル記述
UPC (動的確保 upc_alloc)	71	2.63	9.4 ～ 77.0	コードが複雑
XcalableMP	40	1.48	1.0 ～ 1.3	グローバルデータのアクセス可能領域に制限有
SMint	31	1.15	0.9 - 1.0	グローバルデータのサイズ、アクセス制限なし
MPI	73	2.70	1	
Sequential (OpenMP)	27	1.00		

＊ノード、スレッド数により異なる

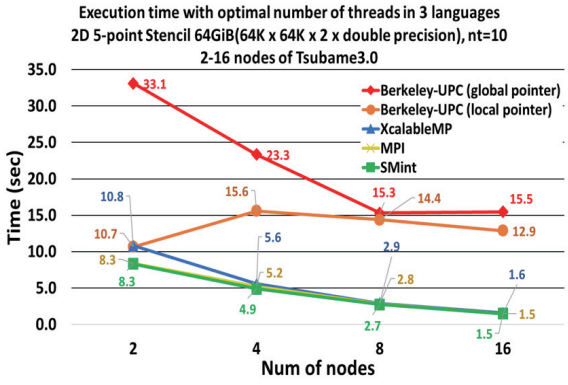


図５　各種言語（UPC, XcalableMP, MPI, SMint）の５点ステンシル計算の性能比較（Tsubame3.0）
（各言語の最適スレッド数で実行）

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．また，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[6]．予め，アクセス領域が決まっていない動的な応用や，これらの指示句を用いない場合は，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を検知し，遠隔ノードからデータフェッチをする．

表 1 は，C 言語を基本とする代表的な PGAS 言語である XcalableMP [19]，UPC [20][21]と，SMint を用い，Tsubame3.0 [24]において，ステンシル計算の記述性と性能を比較した表である．図 5 はノード数の違いによる性能の違いを示す．他言語に比べ，SMint (SMS)が，性能，記述の柔軟性において，最も優れていた[7][8].

2.3 大域データ宣言による MpC プログラム

第二の手法は，データ分散マップ付き多次元大域配列を利用できる MpC プログラムである．MpC の開発は古く，元々ソフトウエア分散共有メモリシステム（SMS[11], TreadMarks[12], JIAJIA[13]）やマルチコア pthread プログラム向けに，同一インタフェースで共有メモリプログラミングができる言語として，C に最小限の拡張を施した拡張 C である[14][15]．MpC トランスレータは，上述の 4 つシステムの中からユーザが指定ものに合わせ，下層実装システムのライブラリ関数などを利用した C コードに変換する．

図 6 は，OpenMP と MpC によるマルチノードマルチコアステンシル計算プログラムである．現在は，mSMS[1]専用の後継 Mp C トランスレータを用いている．テンポラルブロッキングアルゴリズム[9]を採用し，袖領域の幅を bt (テンポラルブロックサイズ) に増やし，bt 時間ステップ分のデータ更新を各ノードがまとめて行う．これにより，ノード間の袖領域通信回数を減らすことができる．

図 6 の 10,11 行目のように，C 配列宣言に shared を付加すると大域配列が利用可能で，分散マッピング指定を付加することもできる．この例では，3 次元配列を z 方向に等分割して全実行ノードに割り当て，各ノードでのローカルノードアクセスを高めて性能向上を図っている．shared 配列宣言は，MpC トランスレータにより，分散マッピング付き大域データ動的割付 sms_mapalloc 関数呼び出しに変換され，プログラム中の大域配列へのアクセス記述は，全てポインタアクセスに変換される．MPI の rank 番号とプロセス数に相当する MYPID, NPROCS 定数が利用できる．次節で述べる SMS ライブラリ関数との混在記述も可能である．

2.4 SMS ライブラリ関数利用による C プログラム

図 3 の最下層にある SMS ライブラリ関数を用いた C

```
1  …
2  #include <mpc.h>                                   マルチノード・CPUコア並列
3  #include <omp.h>                                   OpenMP + MpC（SMS）
4  #define NZ 2048 // for 4 nodes
5  #define NY 2048
6  #define NX 2048
7  #define NT 128
8  #define BT 4
9  //  大域配列  分散マップ
10 shared double A[NZ][NY][NX]::[NPROCS][1][1](0,NPROCS);
11 shared double B[NZ][NY][NX]::[NPROCS][1][1](0,NPROCS);
12
13 int main(int argc, char **argv)
14 { int z,y,x,bt=BT;
15  int size,st,ed;
16  mpc_init(&arg, &argv); //マルチノード並列開始
17
18  //各ノードの担当領域の設定(Z方向分割)
19  size = NZ / NPROCS;  st = size * MYPID;  ed = size * (MYPID +1);
20
21  データ初期化 or 入力
22  mpc_barrier();
23
24   double (*src)[NY][NX] = A;   // 2次元配列へのポインタ   src[NZ][NY][NX]としてアクセス可能
25   double (*dst)[NY][NX] = B;   // 2次元配列へのポインタ   dst[NZ][NY][NX]としてアクセス可能
26   double (*tmp)[NY][NX];
27   int t, tt;
28   int stz, edz;    // テンポラルブロッキングのための時間ステップ毎の計算範囲
29
30  for(t=0; t<NT; t+=bt){//time step loop
31
32    for(tt=bt-1; tt>=0; tt--){   // bt step loop  冗長計算ありテンポラルブロッキング
33     stz = MAX(st-tt, 1);
34     edz = MIN(ed+tt, NZ-1)      計算範囲 stz, edz の設定(③)
35
36    #pragma omp parallel for
37      for(z=stz; z<edz; z++){
38       for(y=1; y<NY-1; y++){
39         for(x=1; x<NX-1; x++){ //7点ステンシルの例
40           dst[z][y][x] = 0.4*src[z][y][x] + 0.1*(src[z-1][y][x] + src[z+1][y][x]
41                        + src[z][y-1][x] + src[z][y+1][x] + src[z][y][x-1] + src[z][y][x+1]);
42       } } }
43     tmp = src; src = dst; dst = tmp;  //src dst 交換
44    }//bt step loop end
45
46    sms_sync_drop(); //隣接ノードからの袖領域データのキャッシュを廃棄
47
48   }//time step loop end
49   mpc_exit(); //マルチノード並列終了
50 }
```


図 6　MpC によるステンシル計算記述例

```
1  …
2  #include <omp.h>                                   マルチノード・CPUコア並列
3  #include <openacc.h>                               OpenMP + SMS関数
4  #include <sms.h>
5  #define NZ 2048 // for 4 nodes
6  #define NY 2048
7  #define NX 2048
8  #define NT 128
9  #define BT 4
10 double (*A)[NY][NX]; //  2次元配列へのポインタ，  A[NZ][NY][NX]としてアクセス可
11 double (*B)[NY][NX]; //  2次元配列へのポインタ，  B[NZ][NY][NX]としてアクセス可
12
13
14 int main(int argc, char **argv)
15 { int z,y,x,bt=BT;
16  int size,st,ed;
17  sms_startup(&argc, &argv); //マルチノード並列開始
18
19  int dim[4] = {NZ, NY, NX, -1};      // 配列の次元
20  int div[4] = {sms_nproc, 1, 1, -1}; //データ配列のマルチノード分散マッピング，次元毎の分割数
21 //   大域配列   A[NZ][NY][NX], B[NZ][NY][NX] の確保と分散マップ
22  A   = (double (*)[NY][NX])sms_mapalloc(dim, div, sizeof(double), 0, sms_nprocs);
23  B   = (double (*)[NY][NX])sms_mapalloc(dim, div, sizeof(double), 0, sms_nprocs);
24
25  //各ノードの担当領域の設定(Z方向分割)
26  size = NZ / sms_nprocs;  st = size * sms_rank;  ed = size * (sms_rank+1);
27
28  データ初期化 or 入力
29  sms_barrier();
30
31   double (*src)[NY][NX] = A;   // 2次元配列へのポインタ src[NZ][NY][NX]としてアクセス可能
32   double (*dst)[NY][NX] = B;   // 2次元配列へのポインタ dst[NZ][NY][NX]としてアクセス可能
33   double (*tmp)[NY][NX];
34   int t, tt;
35   int stz, edz;    // テンポラルブロッキングのための時間ステップ毎の計算範囲
36
37  for(t=0; t<NT; t+=bt){//time step loop
38
39    for(tt=bt-1; tt>=0; tt--){// bt step loop  冗長計算ありテンポラルブロッキング
40     stz = MAX(st-tt, 1);
41     edz = MIN(ed+tt, NZ-1)      計算範囲 stz, edz の設定(③)
42
43    #pragma omp parallel for
44      for(z=stz; z<edz; z++){
45       for(y=1; y<NY-1; y++){
46         for(x=1; x<NX-1; x++){ //7点ステンシルの例
47           dst[z][y][x] = 0.4*src[z][y][x] + 0.1*(src[z-1][y][x] + src[z+1][y][x]
48                        + src[z][y-1][x] + src[z][y+1][x] + src[z][y][x-1] + src[z][y][x+1]);
49       } } }
50     tmp = src; src = dst; dst = tmp;  //src dst 交換
51    }//bt step loop end
52    sms_sync_drop(); //隣接ノードからの袖領域データのキャッシュを廃棄
53
54   }//time step loop end
55   sms_shutdown(); //マルチノード並列終了
56 }
```


図 7　SMS ライブラリ関数によるステンシル計算記述例

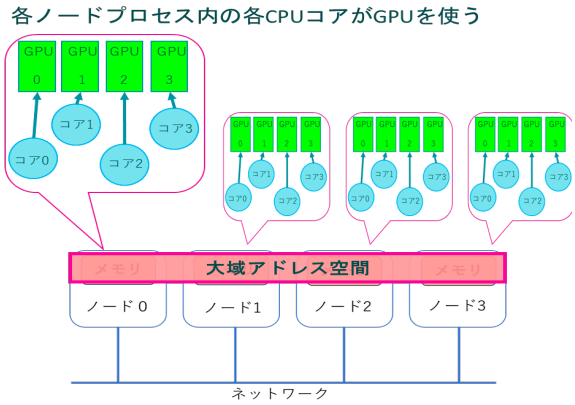
プログラムでは，MPI の rank 番号とプロセス数に対応する sms_rank，sms_nprocs 定数が利用できる．マルチノード上で共有される大域データは，sms_alloc あるいは，sms_mapalloc を用いて，動的に確保する．sms_alloc は一つの指定ノードに指定サイズの大域データを割り付ける．sms_mapalloc は，前述の MpC の大域配列宣言の実装に用いられている関数で，MpC の大域配列宣言の分散マッピング指定にあるパラメタ（配列次元毎の分割数とデータを割り付けるノード範囲）を引数にして，サイクリックに複数ノードにデータを分散マップする[14]．図 6 と同じ処理を SMS ライブラリ関数で記述した例を図 7 に示す．

3. マルチノードマルチ GPU プログラミング

ここでは，3.1 でマルチ GPU クラスタでの典型的記述方式を示す．3.2 で，マルチ GPU によるステンシル計算記述例を示す．ここでは，ホスト経由の GPU のデータ交換を行う．3.3 ではノード内 GPU 間で GPUDirectP2P[23] を用いた場合のステンシル計算記述例を示す．

3.1 mSMS におけるマルチ GPU 処理の典型的記述

図 8(a)のようなマルチ GPU クラスタにおいて，SMS 関数 ＋OpenMP＋OpenACC による典型的な記述方式を示す．図 8(b)では，(1) sms_startup 関数によってマルチノード並列を開始し，(2) 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 数を取得し，(3) OpenMP 並列セクションにより，GPU 個数と同数スレッドを生成し，(4) 各スレッドが OpenACC の acc_set_device_num 関数により利用 GPU を選択し，マルチノード・マルチ GPU 並列処理を行う．



(a)　マルチ GPU クラスタ

```
#include <stdio.h>
#include <omp.h>
#include <openacc.h>
#include <sms.h>
…

int main(int argc, char **argv)
{
  sms_startup(&argc, &argv);
  int numgpus = acc_get_num_devices( acc_device_nvidia );
  …
#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
    {
      …
    }
    #pragma acc exit data copyout(…)
  }
  sms_shutdown();
}
```

ヘッダインクルード
マルチノード実行
マルチコア実行
GPU実行

(b)　SMS 関数 ＋OpenMP＋OpenACC による記述

図 8　マルチ GPU クラスタにおける典型的記述方式

3.2 mSMS におけるマルチ GPU ステンシル計算

図 9 は，マルチ GPU によるテンポラルブロッキングステンシル計算の記述例である．３次元配列を z 軸方向で分割してノードで分担し，さらに 1 ノードの担当領域を 4 つの GPU に割り当てて計算する．テンポラルブロッキングにより，ノード間の袖領域通信と，CPU-GPU 間の袖領域通信の回数を減らすことができる．反面，本来必要のない袖領域の計算が増加するため，通信時間と計算時間のトレードオフが存在する．

```
…
#include <omp.h>
#include <openacc.h>
#include <sms.h>
#define NZ 2048 // for 4 nodes
#define NY 2048
#define NX 2048
#define NT 128
#define BT 4

double (*A)[NY][NX];
double (*B)[NY][NX];

int main(int argc, char **argv)
{ int z, y, x, bt = BT;
  int size, st, ed;

  sms_startup(&argc, &argv); //マルチノード並列開始
  int numgpus = acc_get_num_devices( acc_device_nvidia ); //搭載GPU数取得

  int dim[4] = {NZ, NY, NX, -1};
  int div[4] = {sms_nproc, 1, 1, -1}; //データ配列のマルチノード分散マッピング
  A   = (double (*)[NY][NX]) sms_mapalloc( dim, div, sizeof(double), 0, sms_nprocs);
  B   = (double (*)[NY][NX]) sms_mapalloc( dim, div, sizeof(double), 0, sms_nprocs);

  //各ノードの担当領域の設定(Z方向分割)
  size = NZ / sms_nprocs;  st = size * sms_rank;  ed = size * (sms_rank+1);

  データ初期化 or 入力

  sms_barrier();        //全ノード実行同期，データー貫性同期

  //搭載GPU数と同数スレッドを起動
  #pragma omp parallel num_threads(numgpus) private(x, y, z)  firstprivate(bt, numgpus, st, size..)
  {  int gpuid = omp_get_thread_num();                  //スレッド番号を利用GPU番号にする
     acc_set_device_num(gpuid, acc_device_nvidia);    // 各スレッドの利用GPU番号を指定

     double (*src)[NY][NX] = A;   // src , dst ポインタに処理する配列A,Bを設定
     double (*dst)[NY][NX] = B;
     double (*tmp)[NY][NX];
     int t, tt;
     int laststst,l asted, lastsize;      // ①各GPUにおける最終的な計算結果の範囲
     int maxst, maxed, maxsize;    // ②各GPUにおける袖領域（bt)を含む最大計算範囲
     int stz, edz;                 // ③各GPUにおける時間ステップ毎の計算範囲

     各GPUの計算範囲を計算，上の①②変数に設定

     <1> 各GPUへ初期データを転送

  for(t=0; t<NT; t+=bt){// time step loop
    for(tt=0; tt<bt; tt++){// bt step loop  冗長計算ありテンポラルブロッキング
      各GPUの計算範囲を，上の③変数に設定
       #pragma acc kernels loop independent gang
                 present(src[maxst:maxsize][0:NY][0:NX],dst[maxst:maxsize][0:NY][0:NX])
      for(z=stz; z<edz; z++){
        #pragma acc loop independent seq
        for(y=1; y<NY-1; y++){
          #pragma acc loop independent vector(NX)
          for(x=1; x<NX-1; x++){ //7点ステンシルの例
              dst[z][y][x] = 0.4*src[z][y][x] + 0.1*(src[z-1][y][x] + src[z+1][y][x]
                           + src[z][y-1][x] + src[z][y+1][x] + src[z][y][x-1] + src[z][y][x+1]);
    } } }
      tmp = src; src = dst; dst = tmp;  //src dst 交換
    }//bt step loop end

    <2>  btステップ毎のGPU間袖領域の交換

  } // time step t loop end

    最終結果をホストに転送

 } //omp parallel end
  sms_sync_drop(); //全ノード同期，キャッシュ廃棄
  sms_shutdown(); //マルチノード並列終了
}
```


図 9　SMS におけるマルチ GPU ステンシル計算記述例

mSMS によるマルチノード処理

大域データ配列 A,B は, 図 9 中 23,24 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. 28, 29 行目でデータの初期化（あるいは入力）を行った後, 30 行目でデータのノード間のデータ一貫性同期を行い，全ノードに担当する大域データの値を反映させる.

33 行目で搭載 GPU 数と同数のスレッドを起動し，33～71 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，計算結果の GPU からホストへの転送を行う．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，各ノードの袖領域データのキャッシュを廃棄する．最後に，73 行で sms_shutdown 関数によりマルチノード処理を終了する.

各スレッドにおける GPU 処理

図 10 に各スレッドにおける GPU 処理の詳細（図 9 の 33～71 行目の OpenMP parallel セクション内部）を示す．図 9 の「<1> 各 GPU への初期データ転送部分」は，図 10 の 15～19 行目に対応し，図 9 の「<2> bt ステップ毎の GPU 間の袖領域の交換」は，図 10 の 38～57 行目に対応する．各

```
1  #pragma omp parallel num_threads(numgpus) firstprivate(bt, numgpus, sms_rank, sms_nprocs, st, size)
2  {
3    int gpuid = omp_get_thread_num();                  //スレッド番号を利用GPU番号にする
4    acc_set_device_num(gpuid, acc_device_nvidia);    // 各スレッドの利用GPU番号を指定
5
6    double (*src)[NY][NX] = A;  // src , dst ポインタに処理する配列A,Bを設定
7    double (*dst)[NY][NX] = B;
8    double (*tmp)[NY][NX];
9    int t, tt;
10   int laststst,l asted, lastsize;     // ①各GPUにおける最終的な計算結果の範囲
11   int maxst, maxed, maxsize;    // ②各GPUにおける袖領域（bt)を含む最大計算範囲
12   int stz, edz;                 // ③各GPUにおける時間ステップ毎の計算範囲
13
14   各GPUの計算範囲を計算，上の①②変数に設定
15                                          <1> > 各GPUへの初期データ転送
16   #pragma acc enter data copyin(src[maxst:maxsize][0:NY][0:NX])  //src配列をホストからGPUへ転送
17   #pragma acc enter data create(dst[maxst:maxsize][0:NY][0:NX]) //dst配列をGPUに作成
18
19   全スレッド，全ノード実行同期   #pragma omp barrier, sms_sync();
20                                          ノード内の両端のGPUは，暗黙に隣接ノードデータをアクセス
21   for(t=0; t<NT; t+=bt){//time step loop
22     for(tt=0; tt<bt; tt++){// bt step loop  冗長計算ありテンポラルブロッキング
23   各GPUの計算範囲を，上の③変数に設定
24     #pragma acc kernels loop independent gang
25                        present(src[maxst:maxsize][0:NY][0:NX],dst[maxst:maxsize][0:NY][0:NX])
26       for(z=stz; z<edz; z++){
27         #pragma acc loop independent seq
28         for(y=1; y<NY-1; y++){
29           #pragma acc loop independent vector(NX)
30           for(x=1; x<NX-1; x++){ //7点ステンシルの例
31             dst[z][y][x] = 0.4*src[z][y][x] + 0.1*(src[z-1][y][x] + src[z+1][y][x]
32                          + src[z][y-1][x] + src[z][y+1][x] + src[z][y][x-1] + src[z][y][x+1]);
33         } } }
34         tmp = src; src = dst; dst = tmp;  //src dst 交換
35     }//bt step loop end
36                                          <2> btステップ毎のGPU間袖領域の交換
37   if(t!=NT-bt){  //最終イテレーション以外}
38     // 1. 袖領域(内側)をGPUからホストに転送 update host
39       //左隣に計算GPUがあるGPUは，左データをホストに転送
40       #pragma acc update host (src[laststst:bt][0:NY][0:NX]) if(!(sms_rank==0 && gpuid==0))
41       //右隣に計算GPUがあるGPUは，右データをホストに転送
42       #pragma acc update host (src[lasted-bt:bt][0:NY][0:NX]) if(!(sms_rank==sms_nprocs-1 &&
43       gpuid==numgpus-1))
44
45       全スレッド，全ノード実行同期   #pragma omp barrier, sms_sync();
46
47       // 2. 袖領域(外側)をホストからGPUに転送 update device
48         //左隣に計算GPUがあるGPUは，ホストから左袖領域を取得
49         #pragma acc update device (src[maxst:bt][0:NY][0:NX]) if(!(sms_rank==0 && gpuid==0))
50           //右隣に計算GPUがあるGPUは，ホストから右袖領域を取得
51           #pragma acc update device (src[lasted:bt][0:NY][0:NX]) if(!(sms_rank==sms_nprocs-1 &&
52           gpuid==numgpus-1))
53
54       全スレッド，全ノード実行同期   #pragma omp barrier, sms_sync_drop();
55                                          ノード内の両端のGPUは，暗黙に隣接ノードデータをアクセス
56   } //最終イテレーション以外 終わり
57
58   } // time step t loop end
59   #pragma acc exit data copyout(src[laststst:lastsize][0:NY][0:NX])  //最終結果をホストに転送
60   } //omp parallel end
61
```

図 10　図 9 のスレッド並列セクションの GPU 処理の詳細（ホスト経由 GPU 間データ交換）

ノードが担当するデータのうち，両端のデータ領域を担当する GPU では，片方の袖領域データが隣接ノードに存在する．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ノード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．（16 行目の acc data copyin や，52 行目の acc update_device ）

図 10 の 21～59 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 22～36 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 時間ステップ分の繰り返しである.

15～19 行目<1>の初期データ転送部分では，src 大域配列のうち各 GPU が担当するデータ部分全体を各 GPU へコピーする．一方，38～57 行目は，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 回の更新後に必要な袖領域データのみをホストメモリ経由で転送する．38～57 行目の<2>部分では，まず，担当データ領域のうち，各 GPU の両隣の GPU の計算に必要な袖領域データ（当該 GPU 担当領域の両端内側）のみを GPU からホストに転送し，全スレッド・全ノードの終了後，今度は，各 GPU 自身の計算に必要な袖領域データ（担当領域の両端外側）のみをホストから各 GPU に転送する．必要な袖領域のみを CPU-GPU 間で通信することにより，データ転送量を減らし，高速化している[10].

21～59 行目の時間ステップの繰り返しが終了後，60 行目の acc exit data copyout によって，各 GPU 上の計算結果をすべてホストに転送する．この後, 61 行目で, 各 GPU における処理（スレッド並列セクション）が終了する.

ステンシル計算では, あらかじめアクセスする遠隔データ領域が既知であるため，計算の前に，データプリフェッチ関数（preload, overload）[10]を利用し，隣接ノードにある袖領域を，事前にフェッチして，さらに高速化することもできる[4][5]. 付録 1 に，この関数の説明とこれを追加したプログラムコードを示す. また，図 10 にある全スレッド・全ノード実行同期の実際のコードは，付録 2 に示す.

3.3　GPUDirectP2P による GPU 間データ転送

図 11 は, マルチノードマルチ GPU 処理をさらに高速化するために，ノード内 GPU 間の直接通信（GPUDirectP2P）と，隣接ノードからの袖領域の一括データプリフェッチ preload, overload を加えたコードである．図中の星印★の部分が，図 9 に加えられた部分で，GPUDirectP2P と，ノード間袖領域 preload, overload の記述である.

このうち GPUDirect に関わる部分（図 11 の<1G> と <2G>の部分）は，付録 3 の付録図 3 に示す. GPUDirectP2P は，CUDA コードで書く必要があるた

め， SMS 関数，OpenMP，OpenACC，CUDA で記述した関数を利用している．GPU 間通信を利用することにより，同一ノード内の GPU 間の袖領域の交換は，ホストメモリを経由することなく，高速にデータ交換が実行できる． 図 11 の<1P>の preload と<2G>の中の overload の部分は，付録 1 と同様である．

本節で示した図 6 から図 11 までのプログラムにおいて，マルチノードにおける大域データが一貫した形式やアドレスで，記述，アクセスできていることがわかる．

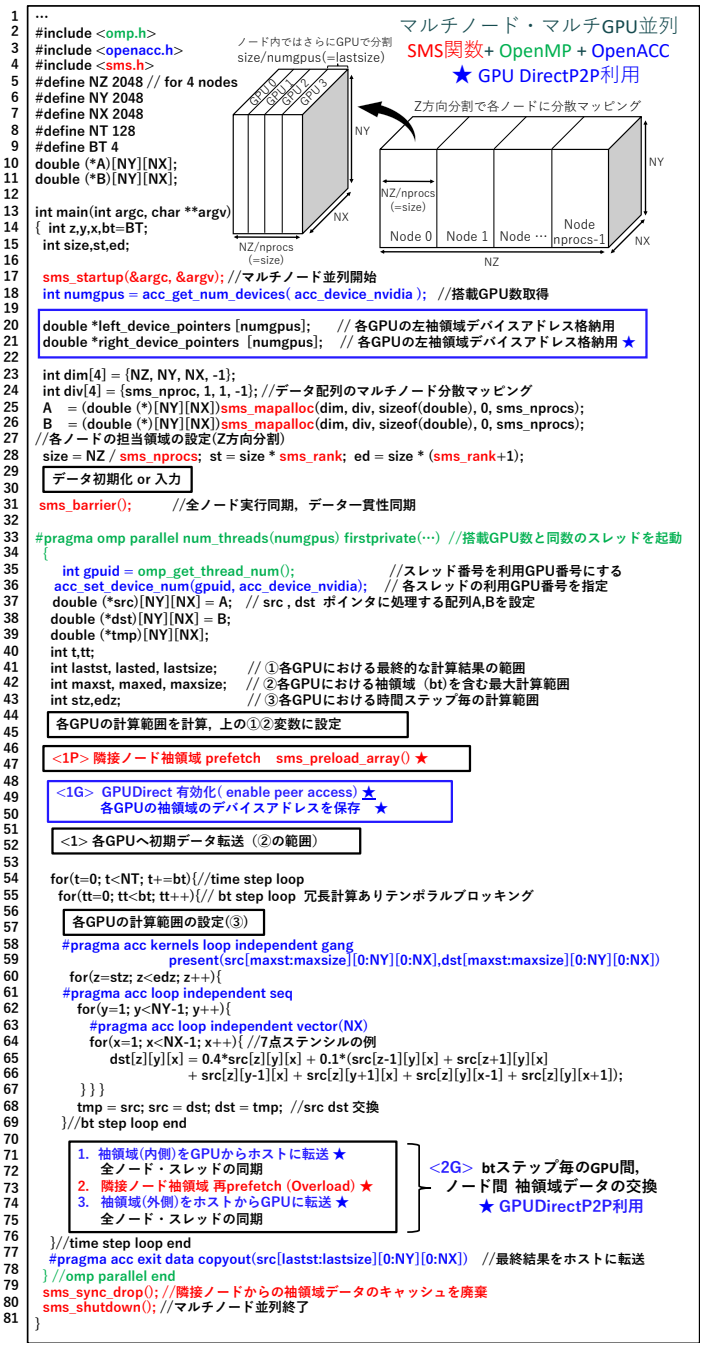


図 11　SMS 関数 ＋OpenMP＋OpenACC＋CUDA 関数による GPUDirectP2P マルチ GPU ステンシル計算の記述

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 [25]（表 1）と東工大 Tsubame3.0 [24]（表 2）を用いた．また，C コンパイラには，OpenACC を利用するため PGI コンパイラを用いた．コンパイルオプションとしては，-O2 -acc -mp –ta=tesla,cc60 -Minfo=accel を指定した. PGI コンパイラは両者とも, LLVM 版を用いている．

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とその内訳を図 12 と図 13 に示す. GPU（Tesla P100）のメモリは 16GB であるが, 袖領域を含む 2 つの配列を格納するために, 1 ノードあたり 32GB（8GB x 4GPU）の配列データ(2048 x 2048 x 512 x 2 配列 x 8B)を計算する. 図 12 と図 13 は, (a)がホストメモリ経由で GPU の袖領域データ交換をした場合，(b)が GPUDirectP2P を用いた場合のそれぞれの weak scaling 性能を表す. 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．2 ノード以上の計測では，両側の隣接ノードとの通信がある rank 1 のスレッド 0 における計測結果を示しており，1 ノードの計測では rank 0 のスレッド 0 における計測結果を示している．このため，どのノードのどのスレッドかで，時間成分の分布は多少異なる．

weak scaling 性能

図 12(a)の Reedbush-L のホストメモリ経由の GPU データ交換では, 全体実行時間は 1 ノード利用時 13.6 秒, 16 ノード利用時では 18.7 秒で，1 ノードに対し 72.7%の並列効率となる. 図 12(b)の GPUDirect 利用時では, 1 ノード利用時 11.0 秒, 16 ノード利用時 15.6 秒で，1 ノードに対し 83.4%の並列効率となる．

図 13(a)の Tsubame3.0 における全体実行時間は，1 ノード利用時 15.5 秒，64 ノード利用時 18.8 秒で，1 ノードに対し 82.4%の並列効率となる．図 13(b)の GPUDirect 利用時では，1 ノード利用時 11.2 秒, 64 ノード利用時 16.5 秒で，1 ノードに対し 67.9%の並列効率となる．ノード数が増えるに従い，ノード間同期の占める割合が増し，相対的に GPU-ホスト転送時間の低減効果が少なくなる．

処理時間成分

1 ノード実行の場合は，いずれのクラスタでも，純粋な計算時間は 50%程度しかなく，ホストと GPU 間のデータ移動にコストがかかっている．DirectP2P を利用すると, 初回と最終回のデータ転送はあるものの，

実行中のデータ移動コストはほぼ無視できるほどに低下する．実行中のデータ移動コストはほぼ無視できるほどに低下する．

複数ノード実行では，純粋な計算時間の割合はノード数の増加とともに小さくなる．ノード間実行同期がノード数に応じて増加し，全体の 5-18 %を占める．(ただし，各ノードで処理タイミングが違うので観測される同期時間はノード毎にそれぞれ異なる．) mSMS によるリモートノードからの袖領域の一括 preload 時間はノードによりばらつくが全体の 3-10%程度で，全ノードの実行同期よりは低い．

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．DirectP2P を利用しても，1 ノード内の両側の GPU はホストメモリにある preload ずみの袖データを転送しているため，1 ノード実行の場合ほど劇的な低下は見られない．

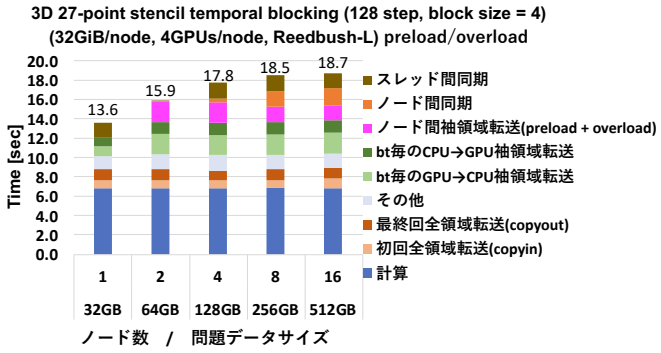
もし GPU の搭載メモリ量を超える処理を，1 ノード内で処理する場合には，ホストメモリから GPU へデータを複数回送付して処理し，ホストに戻すことになるが，ホスト・GPU 間のデータ転送コストが問題となる．

表 1　Reedbush-L

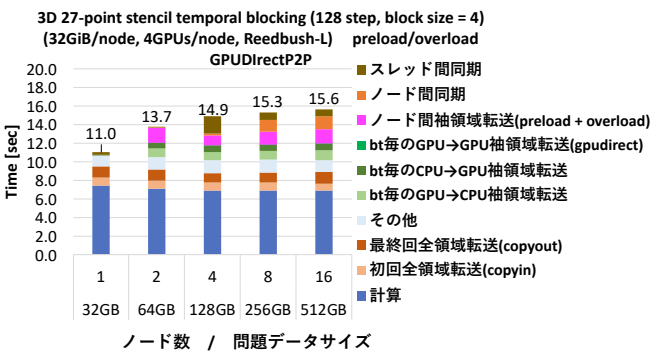
CPU	Intel Xeon CPU E5-2695 v4 @ 2.1GHz * 2
Num of Core / Threads	18 Core * 2
Memory	256GiB
Network	InfiniBand EDR 100Gbps 4*2リンク
GPU	Tesla P100 16GB * 4
OS	Red Hat Enterprise Linux7
modules	cuda9/9.2.148, pgi/19.3, intel-mpi/2018.1.163

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



(a)　ホストメモリ経由の GPU データ交換

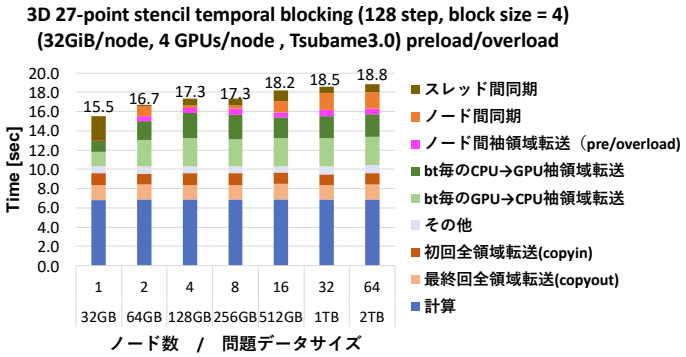


(b)　ノード内 GPU-DirectP2P 利用時によるデータ交換

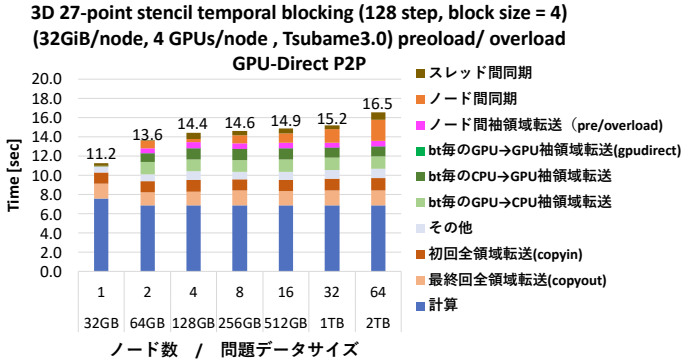
図 12　Reedbush-L における実行時間

GPUDirectP2P の効果

GPUDirect により CPU-GPU データ転送時間が短縮され，ホスト経由のデータ転送に比べ，図 12 の Reedbush-L では



(a)　ホストメモリ経由の GPU データ交換



(b)　ノード内 GPU-DirectP2P 利用時によるデータ交換

図 13　Tsubame3.0　におけるステンシル計算実行時間

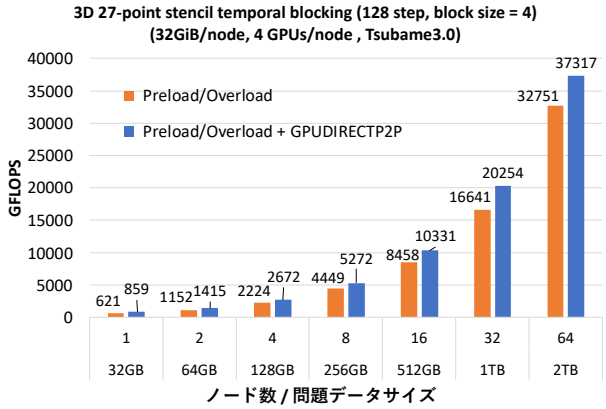


図 14　Tsubame3.0　におけるステンシル計算性能

19%(1 ノード)～17%(64 ノード)程度，図 13 の Tsubame3.0 では 28%（1 ノード）～ 12%（64 ノード）程度，全体実行時間が短縮化されている．

図 14 に，Tsubame3.0 における GPUDirectP2P の有無による性能の違いを示す．GPUDirectP2P 利用により，全体性能は 38.4% (1 ノード) ～13.9% (64 ノード) 向上しているが，ノード数が増えるに従い向上率は鈍る．

Tsubame3.0 における過去に行った mSMS + OpenMP による 27 点ステンシル計算[2]では，本実験と同じ 128 ステップを bt=16 で，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．この時の 64 ノードにおける性能は, 3364 Gflops (mSMS+preload), 3378 Gflops (mSMS), 3434 Gflops (MPI)であった．図 14 の今回の 64 ノードの性能, 37817 Gflops (preoad+ GPUDirect), 32751 Gflops (preload+GPU)と比べると，それぞれ 9.7 倍，11.24 倍と，およそ 10 倍程度，GPU 利用によって高速になっている．1 ノードで扱う問題サイズ，ソフトウエア実行環境，mSMS のバージョンが異なるため, 単純な比較はできないものの，GPU メモリサイズに見合った処理であれば，効果は高い．

5. おわりに

本報告では，mSMS ハイブリッドプログラミングモデルの概要を述べ，ステンシル計算を典型的かつ単純な例題として, 実際の記述例を提示した. 本モデルの利点の一つは，すでに世の中で広く利用され，日々，機能拡張や改良が行われている既存の並列 API はそのままの形で組み合わせて，その恩恵を利用できるようにしていることである．多くの PGAS 言語と異なり，mSMS は実行ランタイムの提供を主体とし，特殊な言語文法や記述をなるべく排除している．C 文法と最小限の SMS ライブラリ関数により, クラスタ上に容易に大域データを構築し処理が記述できる．

一方，OpenMP や OpenACC が広く利用される理由の一つは，逐次プログラムにディレクティブを追加し，典型的並列処理を容易に記述できるという点である．この点については，よく使われる並列セクションに限定し，ディレクティブベースの SMint を提供している．

GPU プログラミングでは，CUDA に対する OpenACC の性能が向上し，プログラミング生産性の観点から,「どうしても書けないものだけ CUDA で記述する」というスタイルが定着しつつある. 本報告では, mSMS, OpenMP, OpwnACC, CUDA 関数を組み合わせて利用するプログラム記述を示した. CUDA では, ホストメモリと GPU メモリの違いが明白で，明示的なデータ転送やアドレス変換が必要であったのに対し，OpenACC では，data copy などの API により，ホストコードと同様な形式で，GPU の処理を記述でき，SMS との併用も容易である．

一方，GPUDirectP2P の記述には CUDA による記述が必要なため，OpenACC の host_data_use_devide により，ホストアドレスをデバイスアドレスに変換し, 作成した cuda 関数に渡し，cuda 関数内部以外では，OpenACC により，見通しよく記述できる．

現在，GPU 関連技術として様々な動きがある．遠隔ノード GPU 間直接通信 GPUDirectRDMA やその実装，GPU の少ないメモリを，ホストメモリを利用して仮想的に大きな一つのメモリにみせる Unified Memory など．現在の SMS で利用している機構と競合する可能性があるが，共存可能な技術であれば，組み合わせも検討する．

謝辞

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] Hiroko Midorikawa, Kenji Kitagawa, Yugo Sakaguchi: "mSMS : PGAS Runtime with Efficient Thread-based Communication for Global-view Programming", 2019 IEEE International Conference on Cluster Computing Cluster2019 , pp.1-2 , DOI: 10.1109/CLUSTER.2019.8891009, 2019
[2] 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, 2018.7
[3] 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, 2019.7
[4] Ryoya Tabata, Hiroko Midorikawa, Ki'nya 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，Poster-52 & Abstract, 2020.1 http://sighpc.ipsj.or.jp/HPCAsia2020/program.html
[5] 緑川博子: "高性能，高生産性を実現する大規模メモリ・並列処理システムソフトウエアの研究"，公募型共同研究 平成 31 年度採択課題，JHPCN 学際大規模情報基盤共同利用・共同研究 第 12 回シンポジウム, 2020.7
スライド http://www.ci.seikei.ac.jp/midori/paper/JHPCN-Sympo12th_jh190039-ISH.pdf
https://jhpcn-kyoten.itc.u-tokyo.ac.jp/abstract/jh190039-ISH
[6] 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, 2018.12
[7] 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 2019.7
[8] 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
[9] 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.
[10] 阪口裕悟，緑川博子:″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″，情処学会，HPC 研究会報告, Vol.2020-HPC-173,No.7,pp.1-8, 2020.3
[11] 緑川博子，飯塚肇 : "ユーザレベル・ソフトウエア分散共有

メモリ SMS の設計と実装", 情報処理学会論文誌ハイパフォーマンスコンピューティングシステム,Vol.42, No.SIG9 (HPS 3), pp.170-190, 2001.8
[12] C. Amza; A.L. Cox; S. Dwarkadas; P. Keleher; et.al." TreadMarks: shared memory computing on networks of workstations", IEEE Computer, Vol. 29, No.2, pp.18-28,1996.2
[13] Weiwu Hu, Weisong Shi, Zhimin Tang : JIAJIA: An SVM System Based on A New Cache Coherence Protocol,Proc. of the High Performance Computing and Networking (HPCN'99), LNCS 1593, pp.463-472, 1999
[14] 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
[15] 緑川博子, 飯塚肇:"メタプロセスモデルに基づくポータブルな並列プログラミングインターフェース MpC",情報処理学会論文誌：コンピューテイングシステム,Vol.46 No.SIG4(ACS9), pp.69-85, 2005.3
[16] OpenMP https://www.openmp.org/
[17] OpenACC https://www.openacc.org/specification
[18] M.D. Wael, et al.: "Partitioned Global Address Space Languages", Journal of ACM Computing Surveys (CSUR), Vol.47, No.62, 2015
[19] Xcalable MP http://www.xcalablemp.org/ja/
[20] UPC Consortium, UPC Language Specifications Version 1.3, https://upc.lbl.gov/docs/user/upc-lang-spec-1.3.pdf
[21] Berkeley UPC http://upc.lbl.gov/ ver.2.28.9, (2018.7.20)
[22] CUDA https://developer.nvidia.com/cuda-zone
[23] NVIDIA GPU Direct https://developer.nvidia.com/gpudirect
[24] Tsubame3.0 http://www.gsic.titech.ac.jp/tsubame3
[25] Reedbush-L https://www.cc.u-tokyo.ac.jp/supercomputer/reedbush/system.php

付録

付録１． 遠隔データのプリフェッチ機能

ステンシル計算は，遠隔データアクセス領域とタイミングが予め分かっている典型的な処理である．このため，アクセス毎にページ単位でデータをフェッチするのではなく，予め，必要なデータを一括してプリフェッチすることもできる．sms_preload_array()は大域データの任意の矩形領域に必要なページを解析し，連続アドレスの複数ページは，一括して送受信し，領域に含まれる全てのページをあらかじめフェッチする．

sms_overload_array()は，sms_preload_array()でRW データとして，すでにローカルノードに読み込まれていることがわかっている同一の矩形領域に，同じ大域データ(値が更新されている)を再度読み込む時に利用する．単に一括フェッチができるだけでなく，すでに存在する遠隔ページキャッシュ領域を，sms_sync_drop()もしくは，sm_barrier()などにより廃棄することなく，利用する．すなわち，各ページの状態（RO, RW, NO）チェックや，ReadWrite への属性変更を省略し，効率よく上書きする．

したがって，この２つの関数を用いる場合には，実行同期時に，ローカルノードにあるキャッシュの廃棄作業（すなわち，キャッシュページを RW（または RO）から NOAccess にする）とする作業は行わない（実行同期 sms_sync() のみを行う）．これにより，次の overload のデータ読み込み時に，すてられたキャッシュを再度属性変更（RW または RO への設定)する手間も省略でき，Segv signal も発生しない．

テンポラルブロッキングステンシル計算のような応用では，利用した隣接袖領域は，ノード自身の計算に必要な冗長計算のための途中計算結果が格納されているだけで，キャッシュの内容をもとのホームノードのデータを更新する必要がない．このため，segv アクセスで処理を行う場合には，bt ステップ毎に，sms_sync_drop()で，実行同期の時

```
#pragma omp parallel num_threads(numgpus) firstprivate(bt, numgpus, sms_rank, sms_nprocs, st, size)
{
  int gpuid = omp_get_thread_num();                  //スレッド番号を利用GPU番号にする
  acc_set_device_num(gpuid, acc_device_nvidia);   // 各スレッドの利用GPU番号を指定

  double (*src)[NY][NX] = A;   // src , dst ポインタに処理する配列A,Bを設定
  double (*dst)[NY][NX] = B;
  double (*tmp)[NY][NX];
  int t, tt;
  int laststst,l asted, lastsize;      // ①各GPUにおける最終的な計算結果の範囲
  int maxst, maxed, maxsize;    // ②各GPUにおける袖領域（bt)を含む最大計算範囲
  int stz, edz;                        // ③各GPUにおける時間ステップ毎の計算範囲
  各GPUの計算範囲を計算，上の①②変数に設定
                                                    <1> > 各GPUへの初期データ転送
   // ★1 preload 隣接ノードから，袖領域をホストにprefetch
   size_t glbstrd[3] = { NZ, NY, NX };   size_t prel_size[3] = { bt, NY, NX };
   if(sms_rank!=0 && gpuid==0) //left
        sms_preload_array( &A[maxst][0][0], glbstrd, prel_size, 3, sizeof(double), 1);
    else if(sms_rank!=sms_nprocs-1 && gpuid==numgpus-1)  //right
        sms_preload_array( &A[lasted][0][0], glbstrd, prel_size, 3, sizeof(double), 1);
  #pragma acc enter data copyin(src[maxst:maxsize][0:NY][0:NX])  //src配列をホストからGPUへ転送
  #pragma acc enter data create(dst[maxst:maxsize][0:NY][0:NX]) //dst配列をGPUに作成
   全スレッド，全ノード実行同期  #pragma omp barrier, sms_sync();

  for(t=0; t<NT; t+=bt){//time step loop
   for(tt=0; tt<bt; tt++){// bt step loop 冗長計算ありテンポラルブロッキング
    各GPUの計算範囲を，上の③変数に設定
   #pragma acc kernels loop independent gang
                        present(src[maxst:maxsize][0:NY][0:NX],dst[maxst:maxsize][0:NY][0:NX])
     for(z=stz; z<edz; z++){
      #pragma acc loop independent seq
      for(y=1; y<NY-1; y++){
        #pragma acc loop independent vector(NX)
        for(x=1; x<NX-1; x++){ //7点ステンシルの例
            dst[z][y][x] = 0.4*src[z][y][x] + 0.1*(src[z-1][y][x] + src[z+1][y][x]
                         + src[z][y-1][x] + src[z][y+1][x] + src[z][y][x-1] + src[z][y][x+1]);
      }}}
      tmp = src; src = dst; dst = tmp;  //src dst 交換
   }//bt step loop end
                                                    <2> btステップ毎のGPU間袖領域の交換
  if(t!=NT-bt){  //最終イテレーション以外}
   // 1. 袖領域(内側)をGPUからホストに転送 update host
     //左隣に計算GPUがあるGPUは，左データをホストに転送
    #pragma acc update host (src[laststst: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))
     全スレッド，全ノード実行同期  #pragma omp barrier, sms_sync();

     // ★2 overload 隣接ノードから，袖領域をホストに再prefetch(overload)
       if(sms_rank!=0 && gpuid==0)  //left
           sms_overload_array( &A[maxst][0][0], glbstrd, prel_size, 3, sizeof(double), 1);
       else if(sms_rank!=sms_nprocs-1 && gpuid==numgpus-1)  //right
           sms_overload_array( &A[lasted][0][0], glbstrd, prel_size, 3, sizeof(double), 1);

    // 2. 袖領域(外側)をホストからGPUに転送 update device
     //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))
     //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))
     全スレッド，全ノード実行同期のみ，キャッシュは維持  #pragma omp barrier, sms_sync();
  } //最終イテレーション以外 終わり

  } // time step t loop end
  #pragma acc exit data copyout(src[laststst:lastsize][0:NY][0:NX])  //最終結果をホストに転送
} //omp parallel end
```

付録図１　図 10 のスレッド並列セクションにプリフェッチ機能を加え

にキャッシュを廃棄する．

しかし，初回に sms_preload_array()を用い，二回目以降に sms_overload_array()で用いて，同じ領域のキャッシュページに上書きする場合には，sms_sync（）で実行同期のみを行い．キャッシュページは廃棄処理を省略する．

付録図１は，本文中の図 10 のコードに，隣接ノードからの袖領域プリフェッチを加えたコードである．隣接ノードにある袖領域（z 方向にテンポラルブロックサイズ bt 分）を，事前にフェッチして，データアクセスを高速化している．

付録２．　全スレッド・全ノードの同期

付録図２は，本文中の各プログラム例で単純化して記述した全スレッド・全ノード同期の実際のコードである．実行同期とデータ一貫性同期，キャッシュの扱いがそれぞれ異なる．ノード間の同期は，マスタースレッドが行っている．

```
全スレッド，全ノード実行同期
#pragma omp barrier
#pragma omp master
sms_sync();
#pragma omp barrier
```

全スレッド・全ノード実行同期

```
全スレッド，全ノード実行同期，キャッシュ廃棄
#pragma omp barrier
#pragma omp master
sms_sync_drop();
#pragma omp barrier
```

全スレッド・全ノード実行同期，キャッシュ廃棄

```
全スレッド，全ノード実行同期, 全ノードデーター貫性同期
#pragma omp barrier
#pragma omp master
sms_barrier();
#pragma omp barrier
```

全スレッド・全ノード実行同期，データ一貫性同期
（キャッシュ廃棄）

付録図２　全スレッド・全ノード同期の実際のコード

付録３　GPUDirectP2P のコード

図 12 における GPUDirectP2P に関連するコードの<1G>と<2G>の実際を，付録図３に示す．

<1G>は，GPUDirectP2P の相手（pear）の設定を行う CUDA 関数 enable_gpudirect を呼ぶ．後半は，<2G>で GPUDirectP2P 通信を行う際の各 GPU の袖領域のデバイスアドレスを予め配列に格納している．

<2G>は，３ステップに分かれ，最初の２ステップは，ノードの両端の GPU の処理で，ホストメモリを介して，袖領域を交換する．最後の３ステップ目で，ノード内の GPU 間で直接通信を行う．memcpy_P2P は，CUDA で書かれており，内部で cudaMemcpy を呼ぶ．

<1G> 各GPUの袖領域のデバイスアドレスを保存

```
enable_gpudirect(gpuid, numgpus, sms_rank); // GPUDirect 有効化( enable peer access)  cuda関数

//  各GPUの袖領域のデバイスアドレスを保存
 if(gpuid != 0){   //各ノードの左端GPU以外
    double *tmp1 = &src[maxst][0][0];
   #pragma acc host_data use_device(tmp1)
    {       left_device_pointers[gpuid] = tmp1;  }   // 各GPUが，左袖領域デバイスアドレスを格納
  }
 if(gpuid != numgpus-1){   //各ノードの右端GPU以外
    double *tmp2 = &src[lasted][0][0];
     #pragma acc host_data use_device(tmp2)
    {       right_device_pointers[gpuid] = tmp2;  }  //各GPUが，右袖領域デバイスアドレスを格納
  }
```

<2G> btステップ毎のGPU間袖領域の交換

```
if(t!=NT-bt){   //最終イテレーション以外}

 // 1. 各ノードの両端GPUは，袖領域(内側)をGPUからホストに転送 update host
       //左端ノード 0 以外の   各ノードの左端GPUは，左データをホストに転送
      #pragma acc update host (src[laststst:bt][0:NY][0:NX]) if( sms_rank !=0 && gpuid ==0))
      //右端ノード以外の   各ノードの右端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))

       全スレッド，全ノード実行同期   #pragma omp barrier, sms_sync();

// 2.各ノードの両端GPUは，袖領域(外側)をホストからGPUへ転送 update device
       // 左端ノード 0 以外の，各ノードの左端の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 )
        //右端ノード以外で，   各ノードの右端の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 )

// 3.  各ノードで，左右の隣GPUから   GPUDirectP2Pにより，袖領域（外側）を得る

     double *tmp1 = &src[laststst][0][0];      double *tmp2 = &src[lasted-bt][0][0];
    #pragma acc host_data use_device( tmp1, tmp2)      // ホストの袖領域をデバイスアドレスtmp1, tmp2とする
    {                                                  // cuda 関数 memcpy_P2Pで，cudaMemcpy（cudaMemcpyDeviceToDevice）
      if(gpuid != 0)  //各ノードの左端GPU以外
        memcpy_P2P( right_device_pointers[gpuid-1], tmp1, bt*NY*NX*sizeof(double) );  //左GPUの右袖領域を自分GPUへ
      if(gpuid != numgpus-1)   //各ノードの右端GPU以外
        memcpy_P2P( left_device_pointers[gpuid+1], tmp2, bt*NY*NX*sizeof(double) );  //右GPUの左袖領域を自分GPUへ
    }

     全スレッド，全ノード実行同期   #pragma omp barrier, sms_sync_drop();

} //最終イテレーション以外 終わり
```

ノード内の両端のGPUは，暗黙に隣接ノードデータをアクセス

付録図３　図 11 の GPUDirecP2Pt 関連の記述