Vol.2019-HPC-170 No.41
2019/7/26

# 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

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 を基本とした分散メモリ型のプログラミングモデルに対し，複数の計算ノードにおいても，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，データのグローバルビューを確保し，より高生産なプログラム開発ができる．しかし，一定レベルの性能を確保するために，グローバルビューモデルを提供しながらも，現実には，従来の MPI によるプログラムと同等なローカルビューによる記述や MPI 片側通信 GET/PUT とほぼ同等な計算ノード間のデータ通信記述を必要とする場合も多い．筆者らは，ソフトウエア分散共有メモリ mSMS をランタイムとして，マルチノードマルチコア並列処理におけるグローバルビューによるプログラミング環境を提供している．本報告では，この mSMS が提供する 3 つのプログラミングインターフェースのうち，インクリメンタルプログラミングを実現する API，SMint と，同じく C 言語を基本としてグローバルビューをサポートする PGAS 言語，XcalableMP と UPC について，プログラム記述性と性能を比較した．XcalableMP と UPC において唯一，グローバルビュー記述が可能であった小規模なステンシル計算処理を例に，３つの言語・API において，記述における制限やグローバルビュープログラムの性能について明らかにした．

**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，ソフトウエア分散共有メモリ，グローバルビュー，大域アドレス空間，並列言語

## 1. はじめに

高性能計算応用においては，高速ネットワークで結ばれた複数計算ノードとノード内マルチコアを有効利用するため，MPI＋X（OpenMP[1]，OpenACC[2] など）によるプログラミング手法が広く用いられている．また，分散メモリモデルである MPI によるプログラム開発の生産性の低さを改善するために，PGAS（Partitioned Global Address Space）モデルで総称される様々な言語・API が提案されている[3][4][5]．多くの PGAS では，MPI と異なり，大域データ配列宣言や大域インデックスを利用できることでグローバルビューを実現する．しかし，各ノードプロセスがアクセスできる遠隔データの範囲に制限があったり，遠隔データアクセスには，MPI 片側通信（get/put）記述と同様に，アクセスしたいデータを所有するノードの番号などを指定してデータにアクセスする coarray 記述[6]などが必要となる．多くの場合，実行プロセスに対し，真の意味で，同一共有メモリアドレス空間を提供しているわけではなく，特定のコンパイラを介して，上記のような特殊な API を下層通信実装関数（MPI など）に変換している．すなわち，全ノードプロセスが，同一のアドレス空間を共有して，C 言語のポインタ変数やアドレスを使ったアクセスを可能とするシステムはほとんどない．また，性能上や実装上の理由により，現実には，多くの PGAS で，MPI と同等なローカルビューによる記述や MPI 片側通信と同等なデータ通信の明示的記述を必要とする場合がある

PGAS の一つである XcalableMP（XMP）[4]は，C（もしくは Fortran）に拡張を加えた言語であり，ディレクティブベースの API を持つ．ループ文の並列化機能や，データの分散マッピング機能，配列のブロック代入文など C を拡張している．同じく C 言語をベースにした PGAS の一つに Unified Parallel C（UPC）[5]がある．upc_forall 文などの新たな構文・文法も追加しており，ループ文の並列化機能や，データの分散マッピング機能がある．グローバル配列データを分散マッピングするための宣言文 shared も備えている．

これに対し，我々は分散共有メモリシステム mSMS[12]をランタイムとして用い，マルチノードにおけるグローバルビュープログラミング環境を実現している．mSMS では 3 つのプログラミング API が利用可能である．(1)SMS ライブラリ関数を使った C プログラム，(2) マルチノードにグローバル共有データを，容易に分散マッピングするためのグローバルデータ配列宣言文 shared を導入した MpC 言語による記述[13]，(3)逐次 C プログラムに#pragma を追加するだけで容易にマルチノードマルチコア並列処理が可能となるディレクティブベースの API，SMint[14]である．

表 1 に３つの言語の比較を示す．XMP ではグローバルビューにより記述できる処理が限られており，C であってもグローバルデータへのポインタが利用できない．一方，SMint，UPC ではグローバルデータに対するアクセス領域制限はなく，ポインタを扱う処理も記述可能である．SMint では，通常の C 言語と同様に，グローバルデータもローカルデータも同一のデータ型ポインタでアクセス可能で，グローバル・ローカルを問わないシームレスなデータアクセスを提供する．mSMS ランタイムは，実行時にそのアクセスデータが遠隔データかローカルデータかを判別して適切な遠

†1 成蹊大学 Seikei University.

隔ノードからデータをフェッチする．一方，UPC では，グローバルデータを指すポインタとローカルデータを指すポインタは，明確に別の型として定義されており，ポインタ変数自体も shared データかローカルデータかで区別されるため，4 種類のポインタ型が存在する．この型の区別の存在により，UPC コンパイラは適切な通信機構の使用を伴う処理に静的に変換する．したがって型変換にはオーバーヘッドを伴う．

XMP と UPC はいずれも，主に，専用コンパイラによるプログラムの静的解析に基づいた下層通信機構利用プログラムへの変換を行っており，実行時の動的処理にはあまり対応していないため，ランタイムシステムの負荷は少ないといえる．一方，SMint では，C 言語からの構文拡張がほとんどなく，SMS ライブラリ関数の挿入を主体とした単純な C プログラムへのトランスレータと汎用 C コンパイラ（gcc）を用いる．実行時には，下層の mSMS 分散共有メモリのランタイムが動的な処理を行う．

表 1　UPC と XcalableMP との比較

| | UPC | XcalableMP | mSMS (+MpC) | SMint |
|---|---|---|---|---|
| Access to global data | Anywhere (shared data) | sleeve or gmove | Anywhere | Anywhere |
| Use pointer | Possible | Impossible | Possible | Possible |
| Runtime load | Light | Light | Heavy | Heavy |
| Dedicated Compiler | Required | Required | Unnecessary (Simple translator) | Simple translator |
| Directive-based Programming | × | ○ | × | ○ |

High flexibility

本論文では，C 言語を基本としてグローバルビューをサポートする PGAS 言語として，SMint，XMP，UPC を用い，3 言語がグローバルビューモデルで実装することができるステンシル計算を用いて，記述性とその性能を比較する．mSMS における並列プログラミング

## 2. mSMS における並列プログラミング

分散共有メモリシステム mSMS[12]では，応用や状況に応じて，ユーザが 3 つの API から一つを選択できる．いずれの API も図 1 の最下層の SMS ライブラリ関数使用による C プログラムへ変換される．

### 2.1 SMS ライブラリ関数利用による C プログラム

第 1 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 2 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

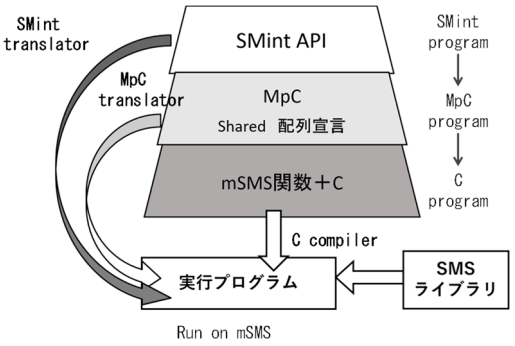



図 1　mSMS におけるプログラミング環境

```
#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();
}
```

Distributed map by using 4 nodes

図 2　SMS ライブラリ関数利用による C プログラム

```
#include <mpc.h>      // C program by using MpC (shared data declaratiion)
#define N …
shared double vec1[N] ::[1](0,1) ;      // Mapping to node0
shared double vec2[N] ::[1](1,1) ;      // Mapping to node1
shared double array[N][N] ::[NPROCS][](0,NPROCS) ;
                                        //Distributed mapping to all nodes
int main(int argc, char *argv[] )
{  int size, st, ed;                    //Area of each node

   mpc_init(&argc, &argv);

   size=N/NPROCS;
   st=size * MYPID; ed=size * (MYPID+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
   }
   mpc_barrier();
   mpc_exit();
}
```

図 3　MpC における shared データ分散マッピング記述

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]．MpC プログラムにおける shared データの配列宣言は，この sms_mapalloc に変換されている．

### 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，図 3 に示すように，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]，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．MpC プログラムは，MpC トランスレータによって，図 2 と同等な C プログラムに自動変換される．

### 2.3 SMint　インクリメンタルプログラミング

第三の手法 SMint 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データ転送を行う指示句（データローカライズ）（copyin，copyout，copy，create など）加えることができ，予め，並列セクション開始前に必要データをローカルノードに preload 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 3 は図 2 を SMint で記述したプログラム例である．図 4 は 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 preload を行ったプログラム例である．

```
#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[] )
{   int size, st, ed;

  // 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] (n0, n1, n2, n3) = array[N][N] (n0, n1, n2, n3) * vec1[N] (n0)

Distributed map by using 4 nodes

図 4　copyin 指示句を用いた SMint プログラム

## 3. 他の PGAS 言語との比較

本節では，比較に用いる 2 つの PGAS 言語，XcalableMP と UPC について概要を述べ，SMint の記述と比較する．

### 3.1 XcalableMP の概要

XcalableMP[4]は，分散メモリシステム向けのディレクティブベースの PGAS 言語で，グローバルビューモデルとローカルビューモデルという 2 つのプログラミングモデルを提供している．

グローバルビューモデルにおけるプログラミングでは，データの分散マッピング（データマッピング），処理の分散マッピング（ワークマッピング）を pragma 指示文の挿入により利用することができる．図 5 に XMP と SMint におけるデータマッピングとワークマッピングの記述例を示す．for 文中，グローバル配列 a[N]のどの領域にもアクセスできるようにみえるが，XMP の場合には，template t[N]で分散マッピングしたローカルノードにある配列の領域にしかアクセスしない．一方，SMint は，a[i]が遠隔ノードにあった場合であってもデータをフェッチして処理する．

```
···
#pragma xmp nodes p[4]
#pragma xmp template t[N]
#pragma xmp distribute t[block] onto p
double a[N];
#pragma xmp align a[i] with t[i]
···
int main(){
  int i;
#pragma xmp loop on t[i] //work mapping
  for(i=0; i<N; i++)
    a[i]=··· //
  return 0;
}
```

data mapping

XMPではa[N]のうち，ローカルにある範囲のみアクセスできる

N, a[N], n0, n1, n2, n3

(a) XMP プログラム

```
···
#pragma SMint shared ::[NPROCS] (0,NPROCS)
double a[N];
···
int main(){
  int i;
#pragma SMint parallel for //work mapping
  for(i=0; i<N; i++)
    a[i]=···
  return 0;
}
```

data mapping

SMintではa[N]の全範囲にアクセスできる

N, a[N], n0, n1, n2, n3

(b) SMint プログラム

図 5　XMP と SMint のデータマッピングとワークマッピング

図 6(a)(b)に典型的なステンシル計算処理の XMP と SMint の記述例を示す．図 6(a)の XMP では，袖領域プリフェッチに shadow 構文，reflect 構文を用い，図 6(b)の SMint は，scopyin 指示句を用いる．XMP と SMint の記述は，いずれも予め並列 for 文で利用するデータをプリフェッチすることで効率を高めている．

図 7 に XMP と SMint におけるグローバルデータのコピーの記述例を示す．図 7(a)のように，XMP では，gmove 構文を用いてグローバルデータをコピーする必要がある．この例ではノード 2 が持つ a[50]～a[59]に，ノード 0 が持つ a[0]～a[9]をコピーしている．グローバルデータのコピーに

おいて，gmove 構文を使うか否かの判断は，a[0]～a[9]がa[50]～a[59]とは違うノードに分散マッピングされているということを意識していなければできない．すなわち，配列名 a とインデックス i のグローバルな名前空間で利用できるものの，各データの所在が遠隔ノードかローカルノードかを常に意識した記述が必要となる，

一方，SMint ではグローバルデータへのアクセスをシームレスに行うことが出来るため，図 7(b)のように，通常の C プログラムと同等の記述で書くことができ，データの所在を意識する必要はない．

すなわち，XMP におけるグローバルビュープログラミングにおいて，他ノードが持っているデータにアクセスすることができるのは，①shadow / reflect 構文による袖領域のアクセス，②gmove 構文による値のコピー，の 2 つの方法だけである．それ以外でグローバル領域にアクセスするような応用は，XMP におけるグローバルビュープログラミングで実装することが出来ない．

一方，XMP のローカルビューモデルにおけるプログラミングでは，図 7 と同様の処理を，図 8 のように記述する．図 8 では，MPI プログラミングと同様に，各ノードのローカル配列とローカルインデックスを用い，:の後にノード番号を付加したアクセス手法（Coarray 記法）を用いる．このようなローカルビュープログラミングでは，グローバルデータのどの部分がどの計算ノードにあるのか，常に意識しなければならず，MPI におけるプログラムの煩雑さと変わらない記述となる．多くの PGAS では，Coarray 記法による代入文は，MPI の片側通信に直接的に変換される．

### 3.2 berkeley-UPC の概要

UPC[5]は分散メモリシステム向けの PGAS 言語の 1 つであり berkeley-UPC[7]はその実装の 1 つである．

UPC では，通常のデータ型の前に，拡張データ型 shared を付けることで，グローバル領域にデータを宣言し，アクセスすることができる．また，グローバル領域のデータへのポインタを利用することも可能である．

配列の分散マッピングの方法は，図 9 に示すように，デフォルトでは，1 要素のサイクリック分散であるが，shared と型名の間にブロックサイズを指定するとブロック単位のサイクリック分散マップが記述できる．UPC では，THREAD と呼ばれる実行 entity で並列処理が行われるが，THREADS はこの THREAD の総数である．THREAD は，実際にはプロセスに近く，同一計算ノードにおけるコアであっても，別プロセスとして実行されるため，THREAD 間のデータアクセスにはグローバルデータとして，遠隔ノードにおける THREAD と同様な手順が必要である．

図 10 に UPC と SMint におけるデータの分散マッピングの違いを示す．UPC は言語上，ローカルノードの THREAD と遠隔ノードの THREAD を区別してプログラムを記述す

```
…
#pragma xmp shadow a[1][0] //Declaration of sleeve area
…
int main(){
  int I,j;
#pragma xmp reflect (a) //Prefetch sleeve area
#pragma xmp loop on t[i] //work mapping
  for(i=0; i<NY; i++)
    if(i==0 || i==NY-1) continue;
    for(j=1; j<NX-1; j++)
      b[i][j] = 0.4*a[i][j] + 0.15*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);

  return 0;
}
```

(a) shadow/reflect 指示文を用いた XMP プログラム

```
…
int main(){
  int i;
#pragma SMint parallel for scopyin(a[1][0]) //work mapping and prefetch sleeve area
  for(i=0; i<NY; i++)
    if(i==0 || i==NY-1) continue;
    for(j=1; j<NX-1; j++)
      b[i][j]= 0.4*a[i][j] + 0.15*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);

  return 0;
}
```

(b) scopy 指示句を用いた SMint プログラム

図 6　XMP と SMint における袖領域のプリフェッチ

```
#pragma xmp nodes p[*]
#pragma xmp template t[100]
#pragma xmp distribute t[block] onto p
int a[100];
#pragma xmp align a[i] with t[i]
…
#pragma xmp gmove
  a[50:10] = a[0:10];
```

(a) XMP プログラム

```
#pragma SMint shared ::[NPROCS] (0,NPROCS)
int a[100];
…
for(i=0; i<10; i++)
  a[50+i] = a[i];
```

(b) SMint プログラム

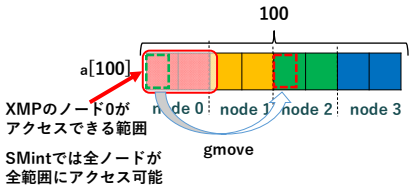



図 7　XMP と SMint のグローバルビューモデルにおけるグローバルデータのコピー

```
int rank = xmpc_this_image();
int a[25]:[*];

if(rank==0)
  a[0:10]:[2] = a[0:10];
```

Coarray 記述
a[index]:[nodeNum]

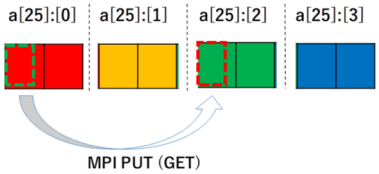


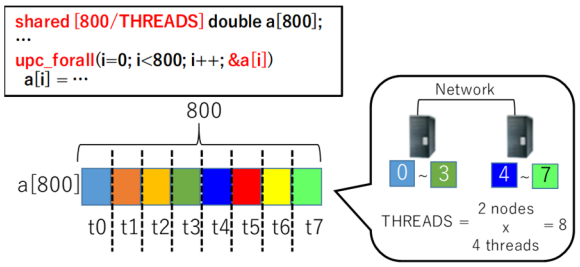

図 8　XMP のローカルビューモデルによる Coarray 記述を用いたデータコピー

ることができない．（コンパイル時に-pthreads=ノード当たりの THREADS 数，実行時に利用する計算ノード数を指定することで，THREAD のレイアウトは決められる．）一方，SMint や，XMP では，マルチノード並列，マルチコア並列という 2 段階の並列処理レベルをそれぞれの pragma で独立に指定でき，同一ノードのマルチコアにおける並列指定に OpenMP などを用い，同一計算ノード内のスレッドは，オーバーヘッドなくデータに効率よくアクセスできる．

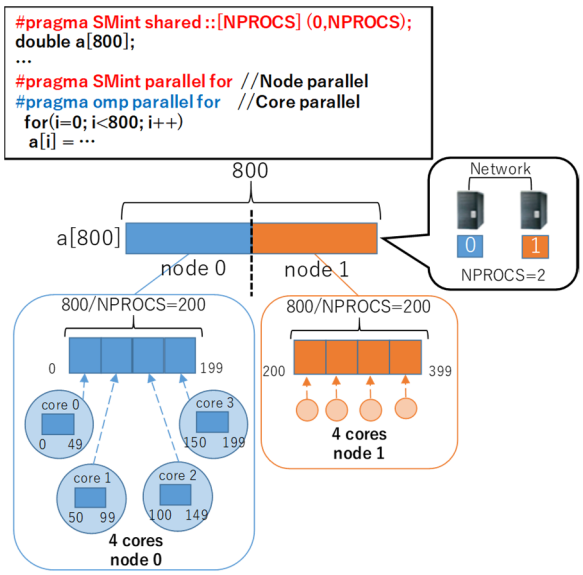
UPC は，UPC ライブラリ関数を利用する以外に，C 言語に新たな構文を加えている．upc_forall では，C 言語の for 文に 4 つ目のセクション（アフィニティ指定）を加え，プロセスがアクセスするデータのアドレスなどを指定する．

```
shared double a[100]; //1 element cyclic
shared [100/THREADS] double b[100]; //block size is 100/THREADS
```

図 9　UPC のグローバルデータ宣言




（a）UPC におけるデータの分散マッピング




（b）SMint におけるデータの分散マッピング

図 10　UPC と SMint におけるデータの分散マッピングの違い

```
//The process holding b[i] executes the iteration
upc_forall(i=0; i<100; i++; &b[i]){
  b[i]=…
}
```

図 11　upc_forall の例

図 11 の例では，b[i]を保持しているプロセスがそのイテレーション i を実行する．

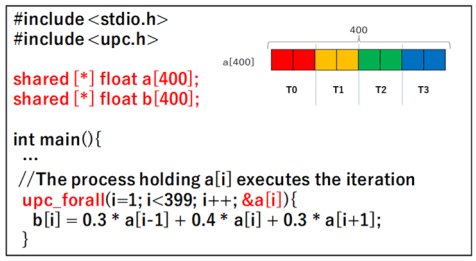
UPC におけるグローバルビュープログラミングとローカルビュープログラミングについて述べる．図 12(a)は，1 次元 3-point ステンシルの処理の記述例である．shared 配列 a[400]に対する処理がグローバルビューモデルで記述されており，プログラムは非常に見やすい．しかし，パフォーマンスを上げるために推奨されているのは図 12(b)のようなローカルポインタを用いたローカルビューモデルによる記述法である．この例は，ローカルデータポインタ型の pa に，shared データ配列 a の各ノードに分散配置された部分の先頭アドレスを代入している．また，for 文では，各ノードにあるローカル配列 pa[100]に対する処理のように記述しており，MPI におけるローカル配列とローカルインデックスによる記述となんら変わりがない．

## 4.　３つの言語の記述性の比較

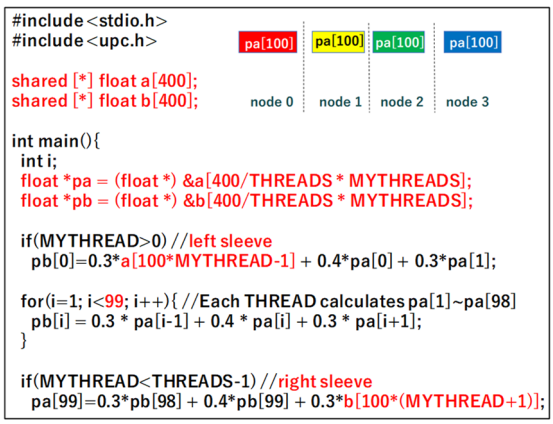
実験では，XMP のグローバルビューモデルで実装することができる応用がステンシル計算のみであったため，今回の比較にはステンシル計算を用いることとした．

### 4.1　UPC におけるデータサイズの制限

UPC では，図 9 の 2 行目で示した shared 宣言において，分散マッピングのブロックサイズには，制限がある．図 13




（a）UPC のグローバルビューモデル




（b）UPC のローカルビューモデル

図 12　UPC による 1 次元 3-point ステンシル

に，UPC におけるグローバルポインタ（shared pointer）の内部表現を示す．デフォルトでは，shared ポインタは 64bit 変数で表現され，内部を，ブロックサイズ（20bit），THREAD 数(10bit)，THREAD 当たりのメモリアドレス空間(34bit)で分割して利用している．この 3 要素への bit 長の配分は，UPC インストール時の configure のオプションとして—with-sptr-packed-bits=phase, thread, addr（phase がブロックサイズ）と指定することで変更可能であるが，いずれも 3 つの合計値は 64bit でなければならないため，大規模なデータを，多ノード多スレッドで実行するのには向かない．そこで今回は，—enable-sptr-struct というオプションを利用した．これは shared ポインタを 64bit 変数表現から，128bit の構造体による表現に変更するものである．しかしこのオプションを利用しても，3 つのフィールドのビット幅を自由に変更することはできない．しかも，現状ではブロックサイズとして 31bit＝2G 要素以上は利用できず，4 THREADS 以上の THREADS 数で同一サイズのデータを処理するには，

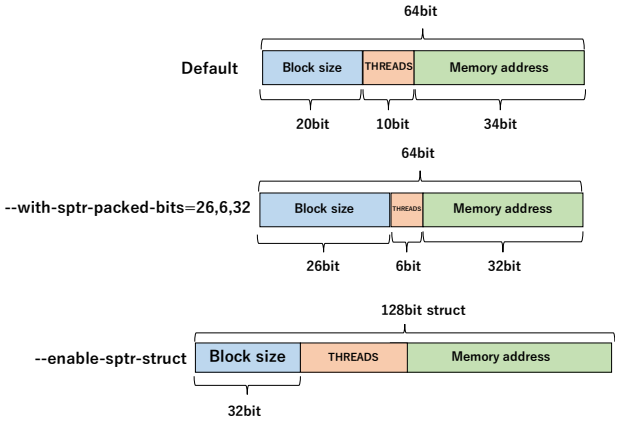



図 13　UPC における shared ポインタの内部表現

```
#include …
#include <smint.h>
…
#pragma SMint shared ::[NPROCS][1] (0, NPROCS);
double a[NY][NX];
#pragma SMint shared ::[NPROCS][1] (0, NPROCS);
double b[NY][NX];

int main(int argc, char **argv){
 int x,y,t;

 //Array Initialize
 …

 for(t=0; t<NT; t++){ //time step
  if(t%2==0){ //phase 1 : b=a
   #pragma SMint parallel for scopyin(a[1][])
   #pragma omp parallel for private(x)
   for(y=0; y<NY; y++){
    if(y==0 || y==NY-1) continue;
    for(x=1; x<NX-1; x++){
     b[y][x] = 0.4*a[y][x] + 0.15*(a[y-1][x] + a[y+1][x] + a[y][x-1] + a[y][x+1]);
    }
   }
  }
  else{ //phase2 : a=b
   #pragma SMint parallel for scopyin(b[1][])
   …
  }
 }// time step end

 return 0;
}
```



図 14　SMint による 2 次元 5 点ステンシルのスケルトンコード

最大 65536 * 65536 要素の 2 次元データを利用するしかない．

したがって，本実験では比較的小規模サイズの 2 次元配列に対する 5-point のステンシル計算処理を対象とせざるをえなかった．

### 4.2　2 次元 5-point ステンシル計算による記述性の比較

2 次元 5-point ステンシル計算を 3 つの言語でそれぞれ実

```
#include …
#include <xmp.h>
…
#pragma xmp nodes p[*] // Declaration of node set
#pragma xmp template t[NY] //Declaration of template (virtual array)
#pragma xmp distribute t[block] onto p
double a[NY][NX];
double b[NY][NX];
#pragma xmp align a[i][*] with t[i]
#pragma xmp align b[i][*] with t[i]
#pragma xmp shadow a[1][0] //Declaration of sleeve area
#pragma xmp shadow b[1][0]

int main(int argc, char **argv){
 int i,j,t;

 //Array Initialize
 …

 for(t=0; t<NT; t++){ //time step
  if(t%2==0){ //phase 1 : b=a
   #pragma xmp reflect (a) //Prefetch sleeve area of a
   #pragma omp parallel for private(j)
   #pragma xmp loop on t[i]
   for(i=0; i<NY; i++){
    if(i==0 || i==NY-1) continue;
    for(j=1; j<NX-1; j++){
     b[i][j] = 0.4*a[i][j] + 0.15*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
    }
   }
  }
  else{ //phase2 : a=b
   #pragma xmp reflect (b) //Prefetch sleeve area of b
   …
  }
#pragma xmp barrier
 }// time step end
 return 0;
}
```



図 15　XMP による 2 次元 5 点ステンシルのスケルトンコード

```
#include …
#include <upc_relaxed.h>
…
#define LY NY/THREADS //local size of Y
shared [LY * NX] double a[NY][NX];
shared [LY * NX] double b[NY][NX];

int main(int argc, char **argv){
 int x,y,t;

 //Array Initialize
 …

 for(t=0; t<NT; t++){ //time step
  if(t%2==0){ //phase 1 : b=a
   upc_forall(y=1; y<NY-1; y++; &a[y][0]){
    for(x=1; x<NX-1; x++){
     b[y][x] = 0.4*a[y][x] + 0.15*(a[y-1][x] + a[y+1][x] + a[y][x-1] + a[y][x+1]);
    }
   }
  }
  else{ //phase2 : a=b
   …
  }
  upc_barrier;
 }// time step end

 return 0;
}
```



図 16　UPC による 2 次元 5 点ステンシルのスケルトンコード（グローバルポインタ版）

装し，記述性を比較する．図 14，15, 16 に SMint，XMP, UPC の各言語で実装したステンシル計算のスケルトンコードを示す． SMint と XMP を比較すると，どちらも pragma 構文の挿入をしているが，SMint に比べ，XMP では，計算システムのネットワークトポロジーなども考慮したマッピングも可能とするために，pragma 記述が多くなっている．図 16 の UPC の記述方法（以降グローバルポインタ版と呼称する）はグローバルビュープログラミングであり，記述量が少なく，簡潔で見やすい．

UPC のプログラムは, 図 16 のグローバルビュー記述に，以下で述べる 2 つの記述法（図 17，18）を加え，性能を比較した．グローバルポインタ版を，パフォーマンス向上のために推奨されているローカルポインタによる記述法に変更したのが図 17 の記述法である（以降ローカルポインタ版と呼称する）．これは shared 配列データのうちローカルにあるデータを指すローカルポインタ（local を指す local ポインタ）を利用した記述である．このため，ローカル以外のデータアクセスが必要な上下の袖領域に関する記述は，別途追加されている．

shared 配列宣言において, 4.1 で述べたブロックサイズの制限を受けないために，図 18 では，upc_alloc 関数を用いている．（upc_alloc 版と呼称する）．upc_alloc 版では， 図のように upc_alloc を利用し，処理配列 a[NY][NX]の各ノードの担当領域 a[LY][NX] (ただし LY=NY/THREADS)を確保している．その後グローバルポインタ配列 a[MYTHREAD] (shared を指す shared ポインタ)と自ノードのローカルポインタ pa（shared を指す local ポインタ）に担当領域の先頭を代入する．対象とするデータ配列のうち，他 THREAD の領域にアクセスするにはポインタ a を用い，ローカル THREAD の領域には pa を用いることで, 高速化している．

パフォーマンスの高い図 17 のローカルデータを指すローカルポインタ版と図 18 のブロックサイズによる制限のない upc_alloc 版はともにローカルビュープログラミングである．いずれも，図 16 のグローバルビュー記述に比べ，記述の複雑性は増しており，特に図 18 の upc_alloc 版は，ブロックサイズの制限をうけずに大規模データ処理をするためには必須であるが，複雑なポインタ型定義は，可読性も悪く，MPI によるステンシル計算の記述と比べても，生産性が高いとは言えない状況にある．

## 5. ３つの言語の性能比較

### 5.1 実験環境について

実験に用いる XMP は Omni XMP Compiler version 1.3.2, UPC は Berkeley-UPC version 2.28.0 とする．また，ノード間の通信方式として，SMint，XMP は MPI，UPC は GASNet [19] の mpi-conduit を利用している．

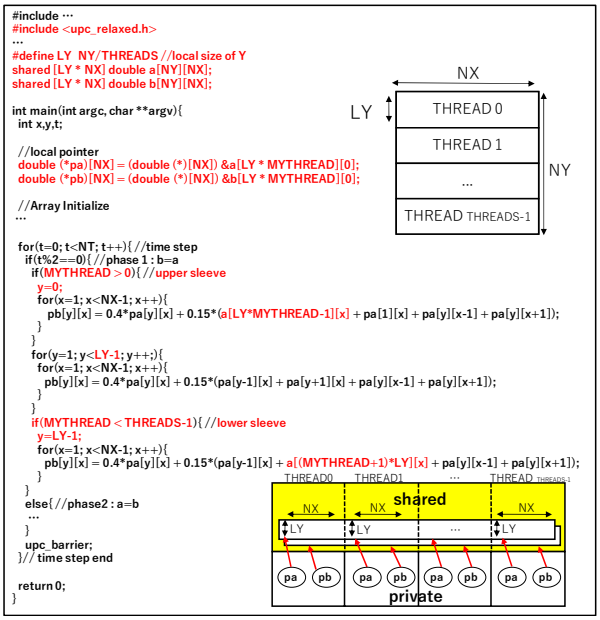



図 17　UPC による 2 次元 5 点ステンシルのスケルトンコード（ローカルポインタ版）

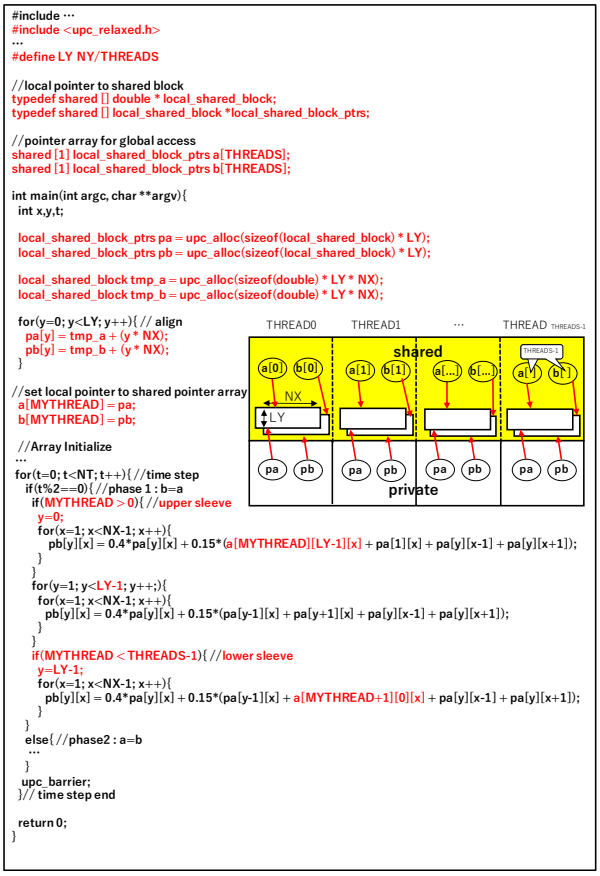



図 18　UPC による 2 次元 5 点ステンシルのスケルトンコード（upc_alloc 版）

測定環境は Tsubame3.0 [7]を用いる（表 2）．各コンパイラで用いるオプションとして，SMint(gcc)は，-O3 –fopenmp –lpthread，XMP(xmpcc)は，-O3 –omp，UPC(upcc)は，-O（これに加え，UPC ではデフォルトで-O3 –param max-inline-insns-single=35000 –param inline-unit-growth=10000 –param large-function-growth=200000 が有効となっている）を指定した．

表 2　実験環境

| | |
|---|---|
| CPU | Intel Xeon CPU E5-2680 v4 @ 2.40GHz * 2CPU |
| Num of Core / Threads | 14 Core / 28 Threads |
| Memory | 256GiB |
| Network | Intel Omni-Path HFI 100Gbps *4 |
| OS | SUSE Linux Enterprise Server 12 SP2 |
| Compiler | gcc 4.8.5 |
| MPI | intel-mpi/18.1.163 |

### 5.2 UPC における記述による性能差

最初に，UPC の 3 つの記述法（図 16, 17, 18）による性能の違いを調査した．図 19 は，Tsubame3.0 の 2 計算ノードにおいて，合計 4~64 個の UPC THREAD を用い，2 次元配列（64K × 64K 要素，double）の 5 点ステンシル計算（繰り返し数 10 回）を行ったときの実行時間を示す．各 THREAD 数において，左，中央，右の順に upc_alloc 版，グローバルポインタ版，ローカルポインタ版のプログラムの実行時間を示す．この図から明らかなように，左端の upc_alloc 版の実行時間は，他の 2 つのプログラム記述に比べ，非常に大きい．4 THREADS 実行時，upc_alloc 版と比べると，グローバルポインタ版では 1.7 倍，ローカルポインタ版では 13.1 倍も高速になる．ただし，いずれの記述でも，4-64 THREADS では，THREAD 数が増えるほど高速化している．

次に，16 計算ノードにおいて，合計 16~512 THREADS で上述と同じ処理を行った結果を図 20 に示す． upc_alloc 版とローカルポインタ版の性能は， 128 THREADS, 256 THREADS でそれぞれ頭打ちとなり，256 THREADS 以降では，むしろ性能は低下していく． これは，strong-scaling の性能評価においては，同一サイズ（64K x 64K 要素）のデータに対するステンシル計算を行うため，THREADS 数が増加するにつれ，1 THREAD 当たりの計算担当領域（ローカルポインタ利用によるアクセス領域）は小さくなり，袖領域におけるグローバルポインタ利用によるアクセスの割合が増加して，オーバーヘッドが増すためである．したがって，upc_alloc 版，ローカルポインタ版 では，THREAD 数が増えることで，高速なローカルポインタによるアクセスに対する低速なグローバルポインタによるアクセスの割合が増加し，性能は低下してゆく．

一方，グローバルポインタ版は，データがローカルであるか否かにかかわらず，全てグローバルポインタによるアクセスを行うため，他 THREAD のデータへのアクセスにかかるオーバーヘッドと，ローカルデータへのアクセスにかかるオーバーヘッドに大きな差がないためと思われる．そのため，THREAD 数を増やし，並列度が上がるほど性能向上している．

しかしすべてグローバルポインタを用いているグローバルポインタ版よりも，ローカルポインタを一部でも用いている upc_alloc 版のほうが，処理時間が長くなっている．この理由は明らかではないが，一つの要因として，upc_alloc 版は，実行時にライブラリ関数 upc_alloc によりグローバルデータを確保していることから，他の二つの静的配列宣言によるプログラム記述と異なり，コンパイラによる静的な解析や最適化が十分できなかった可能性がある．このため，静的配列宣言のプログラムに比べ，高効率な実行コードに変換されておらず，実行時に型変換や間接アクセスなどが生じて，実行時間が増加している可能性も考えられる．

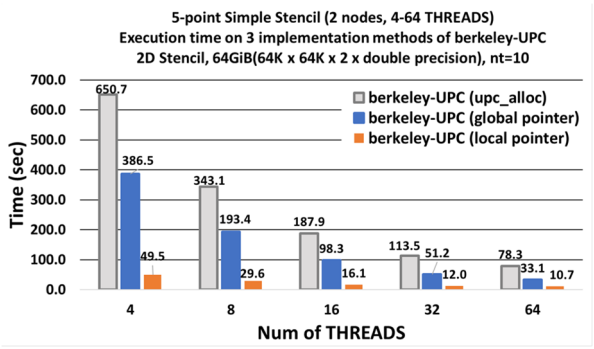



図 19　UPC の 3 つの記述法による実行時間
（2 nodes，4-64 THREADS）

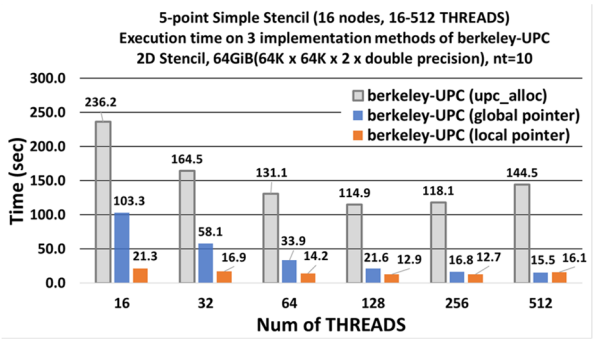



図 20　UPC の 3 つの記述法による実行時間
（16 nodes，16-512 THREADS）

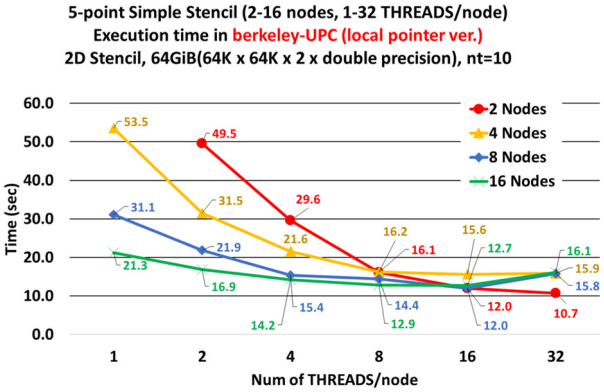
### 5.3 3 つの言語の性能比較

図 21 に UPC, XMP, SMint のプログラムによるそれぞれの実行時間を示す．2～16 計算ノードを用い，各ノード当たりのスレッド数を変化させている．

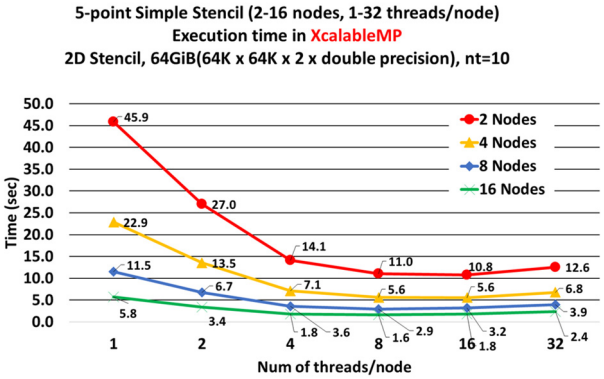
図 21(a) は，UPC プログラム記述の中で，最速であったローカルポインタ版のプログラムの実行時間である．UPC では，計算ノード数が多くなると，1 ノード当たりのスレッド数による性能変化が少なくなる．最速なのは，2 ノー

ド利用でノード当たり 32THREADS を用いた場合で，10.7 秒である．UPC では，THREAD 数が同じである場合，ノード当たりの THREAD 数をふやして，ノード数を減らすほうが高性能となる傾向がある．

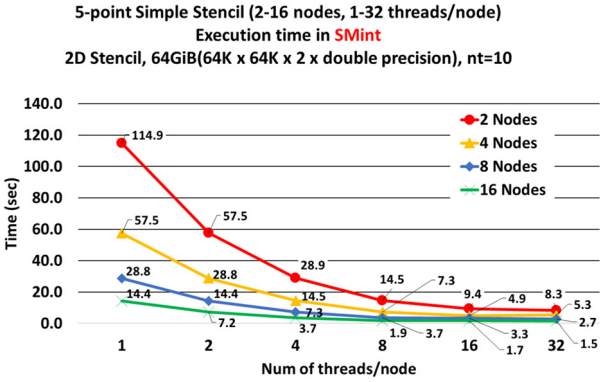
図 21(b) は XMP プログラムの実行時間である．最も高速なのは 16 ノード×8 スレッド実行時で 1.6 秒であり，ノード当たり 8~16 スレッドが最も高速である．XMP では，ノード当たりのスレッド数を増やすと，性能が劣化してくる．




(a) UPC（ローカルポインタ版）




(b) XMP




(c) SMint

図 21　3 言語それぞれの実行時間

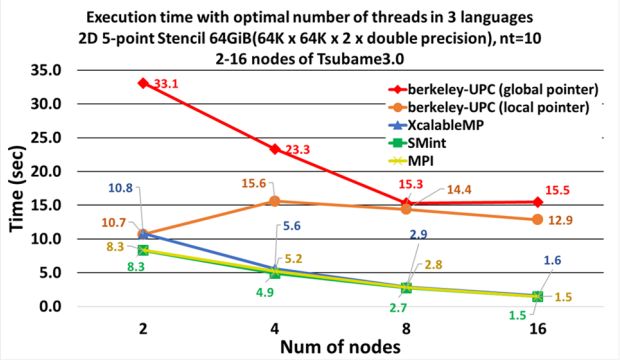



図 22　3 言語+MPI の最適スレッド数による実行時間

表 3　各言語におけるノード当たりの最適スレッド数

| | | Language | | | |
|---|---|---|---|---|---|
| | | XMP | UPC | SMint | MPI |
| Num of Nodes | 2 | 16 | 32 | 32 | 8 |
| | 4 | 16 | 16 | 16 | 16 |
| | 8 | 8 | 8 | 32 | 8 |
| | 16 | 8 | 16 | 32 | 8 |

図 21(c) は SMint プログラムの実行時間である．SMint では，各ノード数で，ノード内のスレッド数が増加するほど，性能は向上しており，他の言語に見られたような性能の低下は見られない．最速は，16 ノード×32 スレッドで 1.5 秒である．

図 22 は，利用ノード数とその時の各言語の最速の実行時間を示している．また，図 22 には MPI プログラムによる実行時間も比較対象として加えている．さらに，図 22 の実行で用いた各言語のノード当たりのスレッド数を表 3 に示す．図 22 によると，UPC の最速版は，2 ノード実行時では 0.9 %程度，XMP よりも高速であるが，4 ノード以上では，XMP と SMint に比べ 3.2 倍～8.6 倍，実行時間がかかっている．XMP と SMint の比較では，いずれのノード数でも 6%~30%程度，SMint のほうが高速になっている．

SMint は，UPC が最も高速となっている 2 ノード実行時で比較しても，UPC の 1.28 倍高速であり，XMP が最も高速となっている 16 ノード実行時で比較しても，XMP の 1.06 倍高速である．さらに，MPI と比較すると，4 ノード実行時，8 ノード実行時に 3~6%程度，SMint のほうが高速になっており，その他の 2 ノード，16 ノード実行時でも同等の性能となっている．

## 6. おわりに

分散共有メモリシステム mSMS 上のディレクティブベース API SMint について，グローバルビュープログラミングをサポートした PGAS 言語である XcalableMP と berkeley-UPC と記述性，性能の比較を行った．

XMP はステンシル計算以外の処理をグローバルビューモデルで実装することは難しく，挿入する pragma 構文も複雑になっている．ステンシル計算における性能としては，SMint とほぼ同等レベルである．

UPC のグローバルビューモデル実装である shared データ宣言文を用いたグローバルポインタ版は，記述性，可読性がともに高い．しかし，その性能は XMP や SMint に比べると劣る．また，コンパイラの静的解析が可能なグローバルデータ配列宣言の利用は，大規模データを対象とした多数ノードへのマッピングが実装上，不可能になっているため，大規模計算には利用することができない．大規模データ計算を行うには，upc_alloc 関数を用い，非常に難解な 4 種類のポインタの型変換などを駆使してプログラムを作成するしかなく，少なくともステンシル計算においては，MPI プログラムのプログラム開発生産性に比べ，低いといわざるを得ない．

結論として，XMP と UPC は PGAS 言語と言われているが，グローバルアドレス空間をサポートするグローバルビューモデルによる記述では，適用できる応用，扱えるグローバルサイズに制限があることがわかる．

SMint は，グローバルデータもローカルデータも単純な C のポインタでアクセスすることが可能で，アクセス領域に制限もない．これは，下層の mSMS ランタイムシステムが，実行時の柔軟な遠隔データアクセス機能を提供しているためである．mSMS では，各ノードの搭載物理メモリサイズを超える大規模仮想アドレス空間を持つプロセスを実際に各計算ノードに走らせて並列実行を行っている．

他の PGAS 言語のように，強力なコンパイラによる静的な最適化や，遠隔データアクセス領域を MPI の GET/PUT 通信に置き換える手法は用いていないが，あらかじめアクセス領域がわかっている場合には，SMint の copyin 指示句をユーザが記述することにより，プリフェッチ関数が挿入され，同等の性能を獲得できる．

現在，配列のような，規則的データ構造をもつ応用だけでなく，ツリーやグラフを用いる不規則構造データを扱う応用についても，ポインタによるアクセスを用いて性能評価を進めている．それぞれの応用で，メモリアクセス局所性を抽出し，実行時のランタイムシステムの助けをかりて，性能とプログラム記述の両面で，生産性の高いプログラミング環境を実現する．

## 謝辞

本報告における XcalableMP の言語文法や仕様に関しては，理化学研究所 計算科学研究機構　研究員 中尾昌広氏にご助言を頂きました．ここで，貴重なご助言に深謝致します．

## 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] Berkeley UPC http://upc.lbl.gov/ ver.2.28.9(2018.7.20)
[8] GASNet https://gasnet.lbl.gov/
[9] Tsubame3 http://www.gsic.titech.ac.jp/tsubame3
[10] Tarek el-ghazasi, et al. “UPC Distributed Shared Memory Programming”, WILEY, 2005, ISBN-10-471-22048-5
[11] H.Midorikawa, U.Ohashi, H.Iizuka : "The Design and Implementation of User-Level Software Distributed Shared Memory System: SMS - Implicit Binding Entry Consistency Model -", Proceeding of IEEE Pacific Rim Conference on Communications Computers and Signal Processing, pp.299-302, 2001-08.(DOI: 10.1109/PACRIM.2001.953582 )
[12] 緑川博子："ソフトウエ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",情報処理学会，ハイパフォーマンスコンピューティング研究会報告（HPC）,Vol 2018-HPC-165,No22,pp.1-9 ， (2018-07-31)
[13] H.Midorikawa : "The Performance Analysis of Portable Parallel Programming Interface MpC for SDSM and pthread",Proceeding of IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2005, IEEE/ACM CCGrid),Vol.2, pp.889-896, Fifth International Workshop on Distributed Shared Memory (DSM2005), 2005-05 ( DOI: 10.1109/CCGRID.2005.155865)
[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"，情報処理学会，研究報告ハイパフォーマンスコンピューティング（HPC）,Vol 2018-HPC-167,No 5,pp.1-9,（2018,12-19）