B-003

マルチコアプログラムにノード並列機能を加える API の提案
Preliminary Implementation of Incremental Programming Model for Multicore-node Parallel Programs

直木 三華† 緑川 博子† 甲斐 宗徳†
Mika Naoki Hiroko Midorikawa Munenori Kai

1. はじめに

近年，1つのプロセッサに複数のコアを備えたマルチコアが一般的になってきた．GPU に関しても，優れた演算処理能力をグラフィック以外の計算処理に利用されており，GPU やマルチコア CPU を用いた並列プログラムが普及しつつある．スーパーコンピュータやクラウドでは，複数のコンピュータを高速ネットワークで繋いだクラスタがすでに広く用いられており，高速処理応用には，マルチコアクラスタをいかに有効に利用するかが重要になっている．

これまで多くの科学技術計算並列プログラムは，プログラムの複雑さを犠牲にしても，実行時間性能を重視し開発されてきた．しかし，最近のノード数コア数の著しい増加は，プログラムを複雑化し，開発・保守コストを増大させて，プログラム生産性を悪化させている．このため，高速実行だけでなく，プログラム開発・保守時間や可読性・記述性も含め，ユーザにかかる総コストを減らすべく，記述が容易な並列言語・モデルへの要求が高まってきている．

そこで本論文では，GPU やマルチコア向けの汎用 API などを用いたノード内並列プログラムに，新たなプラグマ文を追加記述することで，容易にノード間並列を引き出せるような API を提案する．ノード内のマルチコア並列における API には，既に OpenMP[1]や pthread などがある．更に近年では，GPU 並列においてもインクリメンタルに記述可能な OpenACC[2]なども開発されている．

ここで提案する API では，ユーザはこれらの汎用 API を自由に選択出来る仕様となっている．試作実装には，ソフトウェア分散メモリに基づいて我々の開発した MpC[3]言語を用いている．これにより、ソフトウェア分散システム上でマッピング情報を含んだ共有データを導入可能とし，インクリメンタルにグローバル記述を可能としたモデルを提案する．

2. 並列言語の現状・動向

現状，複数ノード間の並列プログラムの記述の大半はメッセージパッシングモデルである MPI[4]，ノード内並列プログラムでは OpenMP・Pthread，GPU 並列では OpenCL[5]・CUDA などが主に用いられている．

2.1 MPI の現状と問題点

MPI は処理の規則性の有無によらず柔軟なプログラム記述が可能である．これは，メッセージパッシングモデルがローカル変数とその通信を明示的に記述するローカルビューに基づいた記述を必要とするためである．ユーザが細かく通信を指示することで通信を効率化し実行時間を短縮化できるが，コードの複雑さが増し，可読性が悪く，開発コストも高いため，ユーザに多大な負担がかかるという難点もある．現在，マルチコアクラスタ向けの並列プログラムでは，クラスタ間並列を MPI，ノード内マルチコア並列を OpenMP で記述することが多い．しかし，最近では，MPI によるプログラム開発における低生産性が，特に問題視されるようになってきている．

2.2 グローバルビュー型言語モデル

このような現状から，幾つかの新しいグローバルビュー型言語モデルが提案・開発されている． MPI などでノード内ローカル変数により処理を記述するのとは対照的に，グローバルビュー型では，ノード全体にグローバルな共有変数が存在するかのように処理を記述できる．たとえば，実際にはノードに分散されて存在する配列でも，記述上はノードを指定せずに通常のローカル配列のようにアクセスを可能にする機能を提供するモデルである．

近年，グローバルビュー記述を可能にしつつ，ノードとデータの関連性やマッピング情報なども付加して記述でき，性能向上を目指す PGAS(Partitioned Global Address Space) モデルが注目されている．最近の PGAS モデルの代表例には Chapel[6]，X10[7]，XcalableMP[8]などがあるが，MPI などのローカルビューより記述が容易であり，可読性が高いとされている．しかし，言語によってはデータ配置や処理の柔軟性に制限があることもあり，ノード間で共通の変数・配列名でアクセスできるデータ範囲が限定され，データの所在を把握する必要が生じる時もあったりと，必ずしもユーザが容易に記述できる環境とはなってはいない．

3. API の設計

本研究では，科学技術計算で最も使われ，処理時間が多くかかるループ構文について着目し，マルチコアクラスタでの並列プログラミングを可能にする API のプロトタイプを作成した．今回の API では，MPI のような柔軟性がある機能面については多少犠牲にするが，新たな知識をユーザに必要とせず，最も単純でユーザビリティの高い記述方式の API を提供する．

この API では，ノード間並列の指示を#pragma 文でインクリメンタルに記述することが可能となり，マルチコア並列・GPU 並列に関してはユーザが好みの汎用 API を選択可能な仕様とする．以下の 3 つの並列性を，それぞれ段階的に記述することで，容易に並列化を実現可能となる環境を構築していく．

・コア並列…既存のスレッド API(OpenMP,pthread など)
・GPU 並列…既存のスレッド API(OpenACC,OpenCL など)
・ノード間並列…提案する API(SMint)

例えば図 1 のように，ループ計算を含む逐次コードにこの API を使用すると，ノード間並列プログラムになる．ここで，OpenMP のようなインクリメンタルなマルチコア並

†成蹊大学理工学研究科理工学専攻 Graduate School of Science and Technology, Seikei University

Copyright © 2012 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

列プログラムにこの API を使用するとノード＋コア並列プログラムに， OpenACC のような GPU 並列プログラムにこの API を使用するとノード+GPU 並列プログラムとして活用する．またノード間並列を使用したくない場合は，今回提供する SMint(distributed Shared Memory interface) の #pragma 文を無効のままコンパイルすることで，もとの逐次コードやマルチコアコード，GPU コードのプログラムとして動かすことが可能である．

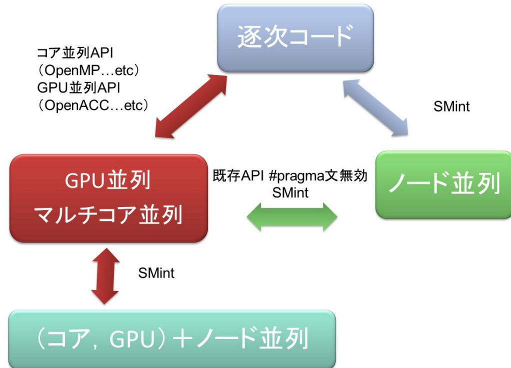

図 1　API 使用方針

具体的には，使用したい共有データの直前に，#pragma SMint 文を図 2 のようにユーザが数行だけ追加記述することによって，ノード間の並列性を実現させる．

また，ユーザは追加記述したプログラムを専用コンパイラに通すだけで，計算資源をユーザが指定した通りにマッピング処理を行う．そして，プログラマはノードを意識せずに，マッピングされた共有の変数を簡単にアクセス可能（グローバルビュー型 PGAS）にする．今回，このことを実現可能にするために，ソフトウェア分散メモリシステム上で実装された MpC を利用して，SMint のプロトタイプを構築とした．

これらの点を踏まえて，API の実装は，図 2 に示すような pragma 文を挿入することによって実現していく．

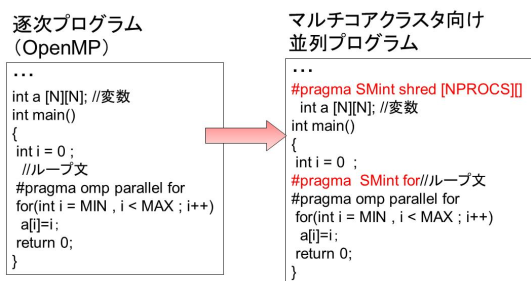

図 2 API 使用例

上記の使用方針のもと，逐次プログラムにこの API を適用した際の並列動作の例を図 3・図 4 に示す．

図 3 では通常の逐次プログラム for 文処理の配列データの様子を表している．このループ文に今回の SMint を記述すると，2 行加えるだけでコア＋ノード両方の並列性を実現することを可能にする．図 4 は，4 コア 4 ノードクラスタの環境で SMint を利用する際，16 並列でデータ処理を行うときの記述とマッピングによる動作の例である.図 4 では，データを縦に 4 分割した際の分散の様子を図に示しているが，この時の分割割り当ては，ユーザが自由に指示できる仕様となっている．

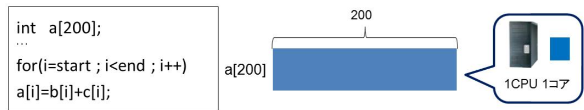

図 3　逐次プログラムの for 文処理

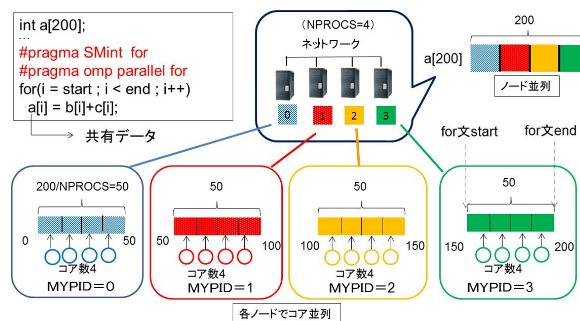

図 4　SMint API 使用時の for 文処理

4.　API の実現方式

ノードを意識せずに，共有の変数をアクセス可能にし，またユーザ指定した通りに共有データの分割割り当てを行う PGAS 型データを容易に可能にするために， MpC（Meta Process C）[3]言語に拡張を加え，既存の逐次プログラム・スレッド並列プログラムの繰り返し文をノード間並列可能にする SMint 言語の設計を行った．MpC 言語では，共有データを使用可能にし，共有データを分散マッピングでノードに割り付け，トランスレータで C 言語プログラムに変換するサポートがある．今回 SMint では，通常のプログラムにユーザが共有データを使いたい箇所の直前に #pragma 文を書き加えることで，共有データを使用可能にし，必要な MpC プログラムに変換・変更を行う．下記の構文を記述したプログラムをコンパイラに通すと，トランスレータでプログラムに変更を加え，ループ文箇所を等分並列実行コードとして変換する．

ユーザには，以下の表 1 の API を提供する．

表 1　SMint API の MpC プログラムへの変換方式

	#pragma SMint 構文	MpC 表記（関数）
(ⅰ) 宣言	#pragma SMint shared (割り付け方式 : 図 5)	Share(型)（変数)::(割り付け方式 : 図 5)
(ⅱ) ループ	#pragma SMint for	(並列部のデータ領域割り当てを自動挿入)

共有変数をプロセスに割り付ける際は，MpC 言語で提供されている下記の方式で記述する．

Copyright © 2012 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.

Shared 型名 変数名[sm]...[s1] ...[s0]::[dm] ...[d1] ...[d0](st,0)

変数名[sm]...[s1] ...[s0] → 変数の各次元サイズ
[dm] ...[d1] ...[d0] → 各次元の分割数（省略時は分割数1）
st → 割り付け開始プロセス番号（省略時は0か任意プロセス）
n → 割り付けプロセス数（省略時は全使用プロセス数）

図 5　共有変数とプロセスとを関連づける割り付け方式

5.　SMint コンパイラの構造

今回 SMint コンパイラでは従来の MpC コンパイラに手を加えて，2 段階のトランスレータに通すことで，#pragma SMint 構文を使用可能にした．このコンパイラでは，パス 1 で必要な情報をファイルに取得し，コードの挿入及びシステムライブラリ関数へ変換する．そしてパス 2 では，パス 1 で取得したデータを使ってループ文に必要な条件式の計算を行い，コードを挿入する．そして，変数リネーミングを行い，C プログラムに変換する．

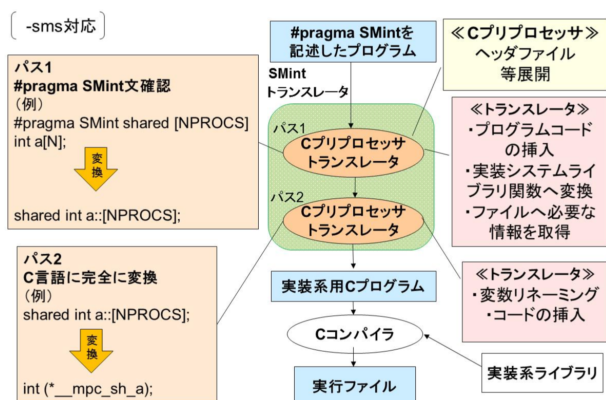

図 6　SMint コンパイラの構造

6.　SMint トランスレータ

図 7 では，表 1 の API を使用して記述したプログラムから，SMint コンパイラに通して MpC プログラムへ変換した例を表している．

共有したい変数データの直前に「#pragma SMint shared (割り付け方式)」を記入してコンパイラに通すと，1 段階目のトランスレータ（パス 1）で shared 型の変数に変換する．for 文の直前に「#pragma SMint parallel for」と追加記入してトランスレータに通すと，パス 1 でノード毎の繰り返し文の開始条件・終了条件に関する必要な情報を取得し，変数の宣言及びコードへ対応させる．

そしてパス 2 で，取得した情報から開始&終了条件を得る計算式のコードを挿入する．今回図 7 では，斜線部分がパス 2 で挿入したコードに対応する．また，for 文の計算を各ノードが実行した後に同期をとらせるために，「#pragma SMint parallel for」構文の最後には mpc_barrier(0) を自動挿入させ，バリア同期をさせる．

また，トランスレータでは main 関数の最初に起動関数である mpc_init を，main 関数の最後の括弧と return 文のあとに，終了関数である mpc_exit 関数を自動挿入する．

```
#include <stdio.h>
#include <unistd.h>
#include<mpc.h>
#include<omp.h>
#define N 1024

#pragma SMint shared [NPROCS](0,NPROCS);
int a[N];
int main(int argc,char **argv)
{
  int i;
#pragma SMint parallel for
 #pragma omp parallel for
  for(i=0;i<N;i++){
       a[i]=i;
 }
 for(i=0;i<N;i++)
       printf("a[%d]=%d , ",i,a[i]);
  return 0;
}
```

```
...
int (*__mpc_sh_a);
 int main(int argc,char **argv)
{
  mpc_init(argc,argv);
  int i;
{
 int __mpc_start , __mpc_end;
      __mpc_start = (0)+(((1024)/NPROCS)*(MYPID));
      __mpc_end = __mpc_start+((1024)/NPROCS);
#pragma omp parallel for
for(i=__mpc_start;i<__mpc_end;i++){
    __mpc_sh_a[i]=i;
 }
    mpc_barrier(0);
}
 for(i=0;i<1024;i++)
 printf("a[%d]=%d , ",i,__mpc_sh_a[i]);
mpc_exit();
 return 0;
  mpc_exit();
}
```

図 7　SMint プログラムから MpC プログラムへの変換

7.　おわりに

今回は，プロトタイプとしてループの並列化と単純なデータ分割の最小限な機能のみ実装し，翻訳結果の動作確認と，初期システムでの簡単な動作実験を行った．

今後は，OpenMP の private 文や OpenACC の copyin・copyout 文などに対応するような，ノード共有データ（shared データ）とノード局所データ間でのデータコピー機能も追加する予定である．データコピー機能を拡張することにより，並列セクションにおいて，ノード内の並列処理の高速化を実現させる方針とした．また，OpenMP にあるような reduction 文などの collective 関数についても対応させ，機能を拡張していく予定である．

参考文献

[1] http://openmp.org/wp/
[2] http://www.openacc-standard.org/
[3] 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)
[4] http://www.mpiweb.org/Home
[5] http://www.khronos.org/opencl/
[6] http://chapel.cray.com/
[7] http://x10-lang.org/
[8] http://www.xcalablemp.org/

Copyright © 2012 by
The Institute of Electronics, Information and Communication Engineers and
Information Processing Society of Japan All rights reserved.