Vol.2017-HPC-158 No.32
2017/3/10

処理粒度に応じたデータ分割・配置を行う多次元データ処理フレームワーク

滝澤 真一朗[1,a)]　松田 元彦[1]　丸山 直也[1]　中村 宜文[1]

概要：大規模並列システムで実行される計算科学アプリケーションは複数のタスクから構成されており，タスクごとに要求する並列数，データ配置，実行数が異なる．そのためタスク実行時には，並列数やデータアクセスパターンを考慮して，処理対象のデータを分散配置する必要があるが，手作業での実装は困難である．本研究では，シンプルな指示によりデータの分割・配置規則，処理単位規則を指定できる，Multi-View データモデルを提案する．処理系にて指示に従い，利用者から透過的にデータ配置，局所性を考慮したタスク配置を行う．格子 QCD アプリケーションを用いた評価の結果，提案を用いたデータ配置のコストは微々たるものであり実行時間への影響が軽微であること，生産性の向上を確認した．

1. はじめに

大規模並列システムでは，気象予測や分子動力学，天文学などの分野の計算科学アプリケーションが多数実行されている．これらのアプリケーションは，計算カーネルや独立したサブプログラムとして実装される複数のタスクからなるワークフローとして構成されており，各タスクごとに要求する並列度やデータアクセスパターン，実行数が異なる．例えば，データ同化を行う気象予測アプリケーションでは，異なるパラメータを入力とした気象モデルを複数実行（アンサンブル）し，その全ての結果と観測値を用いて統計処理を行い，気象予測を行う．前段の気象モデルのアンサンブル実行では，各アンサンブルを複数プロセスを用いて，シミュレーション領域をプロセス間分割して計算する．後段の統計処理は個々の座標の計算であり，該当座標のすべてのアンサンブル出力と観測値を用いて計算する．1 座標の計算は 1 プロセスで行えるが，座標数は膨大である．

このようなアプリケーションにおいて，タスク要求に沿わない場合には，タスク実行中にリモートへのデータアクセスが発生したり，負荷分散が不均一になり計算資源利用効率の低下が起こる．そのため，タスクの要求に応じた並列数を確保し，処理対象データの分割・配置を行う必要がある．これにはタスクごとに，(1) 必要な数のプロセスからなる実行環境を作り，(2) 通信や IO を通じて，各プロセスに必要なデータを配置し，(3) タスクを実行することが求められる．タスク数が少ない場合はこの実装コストは多くはないが，(2) のデータの配置を行うにあたり，処理対象データを適切にプロセス間分割して配置することは，面倒で，かつ間違えを起こしやす作業である．また，処理対象データサイズが大きい場合には，効率の良いデータ管理や転送方法も求められる．

上記の問題を解決するために，本研究では Multi-View データモデルを提案する．これは処理対象データを多次元配列として表現し，利用者はそのプロセス間分割配置規則，処理単位規則を単純な記法で指示する．処理系はその指示に従い，利用者から透過的にデータ分割・再配置を行い，データアクセスの局所性を考慮したタスク配置を行う．本フレームワークをプロセス管理・通信用に MPI を用いて C++にて実装し，C/C++/Fortran アプリケーションから利用できるよう API を整備した．格子 QCD アプリケーションの 1 実装である BQCD [1] を用いて評価を行った．その結果，データ再配置・タスク配置を MPI 関数で実装した場合に対して，再配置のための転送時間は増加したが，タスク実行時間に対する比率は 6%以下であり，性能への影響は軽微であることが確認できた．また，提案を用いた方が実装時に検討する項目と実装コード行数が少なく，生産性が高いことも確認できた．

2. Motivating Examples

2.1 格子 QCD

格子 QCD シミュレーションは，強い相互作用がかかわる物理現象や物理量を計算する手法である．このシミュレーションは，モンテカルロ法によるゲージ場の配位生成と生成されたゲージ場の配位の解析からなる．モンテカル

[1] 理化学研究所 計算科学研究機構, RIKEN AICS
[a)] shinichiro.takizawa@riken.jp

ロ法では逐次的にゲージ場の配位を生成し，各モンテカルロステップにて，異なる入力ベクトルに対する Dirac 方程式を解く必要がある．一般的な実装では，入力ベクトルごとに逐次計算され，個々の入力ベクトルではプロセス間で領域分割され並列計算される．しかしながら，この方法では強スケールしないため，BQCD [1] では入力ベクトル群をプロセス間で再配置して，領域だけでなく，入力ベクトルに対しても並列に計算する方法（アンサンブル並列）が提案されている．

Dirac 方程式を解くタスクを $DiracEq$，その前後のタスクを $Other$ とした場合の，タスク毎の並列数，タスク数，データ配置の違いを図 1 に示す．$DiracEq$ は入力ベクトルに対して複数実行する必要があるため，入力ベクトル数を E とした場合，タスク実行数は E となる（図では $E = 2$）．$DiracEq$ は並列に実行できるので，格子 QCD シミュレーション全体を P プロセスで実行する場合（図では $P = 4$），個々のタスクは P/E プロセスで並列に処理できる．入力ベクトル長が N の場合（図では N は 4 の倍数），プロセス間で領域分割し，各プロセスは $\frac{N}{(P/E)}$ 個の要素の処理を担当することになる．一方，$Other$ は全ての $DiracEq$ の入出力ベクトルを一括して処理する必要があるため，タスク実行数は入力ベクトル数に対して 1 となる．P プロセスで処理する場合，各プロセスは E 個の入出力ベクトルについてそれぞれ $\frac{N}{P}$ の要素を担当するため，合計 $E \times \frac{N}{P}$ 個の要素を担当する．この数は $DiracEq$ と等しいが，ベクトル中の担当領域が異なる．このように，タスクごとの実行数，タスクが要求する並列数，プロセスが要求するデータ配置が異なる．

2.2 データ同化を用いた気象予測

データ同化はシミュレーションの精度向上のために，気象予測分野で広く用いられている手法である．データ同化気象予測アプリケーションの 1 つである，NICAM-LETKF [2] では，気象モデルである NICAM を複数の入力に対してアンサンブル実行し，その結果と観測データをもとに，LETKF にて格子点ごとに統計処理を行う．この処理をタイムステップ数分繰り返す．

NICAM-LETKF ではデータアクセスの局所化を目的に，NICAM と LETKF を同じプロセスを用いて実行する．NICAM-LETKF のタスク毎の並列数，タスク数，データ配置の違いを図 2 に示す．プロセス数を P（図では $P = 4$），NICAM のアンサンブル数を E とすると，NICAM タスク数は E（図では $E = 2$），各 NICAM は P/E プロセスで並列に実行できる．NICAM のシミュレーション領域が N 格子点（図では $N = 8$）からなる場合，各プロセスは $\frac{N}{P/E}$ 格子を処理することになる．一方，LETKF は 1 格子点ごとの処理であり，LETKF タスク数は N，1 プロセス/タスクで処理できる．LETKF では，1 格子点の E 個の NICAM 出力を処理するため，入力要素数は E となる．また，一般に $N \gg P$ なので，各プロセス N/P 格子点の LETKF タスクを実行することになり，各プロセスは総計 $E \times (N/P)$ 要素を持つことになる．この数は NICAM と等しいが，NICAM では「担当アンサンブルの担当領域の格子点データ」であるのに対して，LETKF では「担当格子点の全アンサンブルデータ」となり，種類が異なる．このように，NICAM と LETKF において，タスク数，タスクが要求する並列数，プロセスが要求するデータ配置が異なる．

2.2.1 課題

上記アプリケーションではタスクごとに，タスク数，タスクのプロセス数，データ配置パターンが異なるという特徴がある．このため，タスクごとにデータの再配置を行い，必要プロセス数からなる実行環境を整備する必要がある．従来は，後続タスク実行前に，以下の手順からなる作業を手作業で実装していた．

(1) 入力データの再配置のため，各プロセスにて，入力ベクトルの各要素の送信先プロセスを計算する．
(2) プロセス間でデータ再配置を行う．
(3) 再配置されたデータを持つプロセス上でタスクを実行する．

(1) と (3) は，後続タスクのデータ配置パターンとタスク並列度を考慮して実装する必要があり，面倒かつ間違えを起こしやすい作業である．特に前述の格子 QCD や気象シミュレーションでは入力が多次元配列となるため，分割の実装がより面倒になる．(2) には再配置手段の切り替えと，高い効率が求められる．プロセス数が少なく，データがメモリに収まる場合には `MPI_Alltoallv` による再配置で十分であるが，プロセス数が多くなった場合には，関連するデータを持つプロセス間でのみグループを組み通信を行う等の，通信の局所化が求められる．データがメモリに収まらない場合には IO は避けられないが，ボトルネックとなり得る共有ファイルシステムは用いず，通信とローカル IO で実現する最適化が求められる．NICAM-LETKF では問題サイズに応じて，数 MB〜数 TB のデータを処理するため，再配置手段の切り替えは特に重要となる．上記を考慮した再配置の実装を手作業を行うのは困難である．

以上の実装の困難さは，アプリケーションの分野をまたぎ共通の課題であるが，共通する有効な手段はなく，個々のアプリケーションごとに独自に実装されているのが現実である．

3. Multi-View Data Model

本研究では上記の問題を解決するために，単純な指示でデータの再配置規則，処理単位規則を指定できる Multi-View データモデル，及び，そのモデルを実装し，利用者から透過的にデータの再配置，タスク配置を行うフレーム

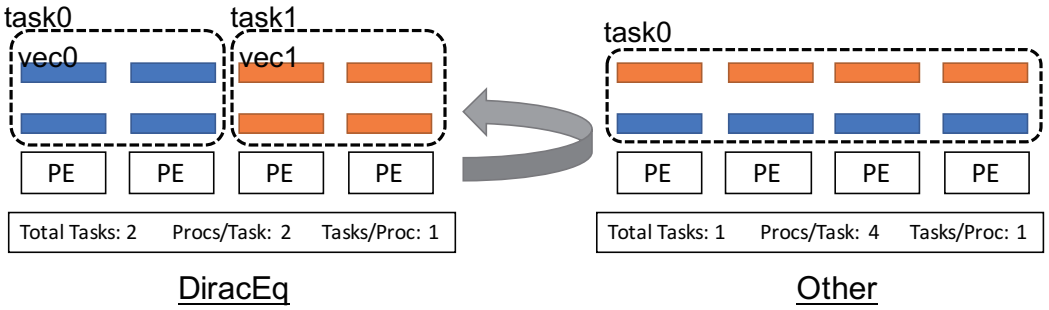


図 1: 格子 QCD でのタスク毎の並列数，タスク数，データ配置の違い

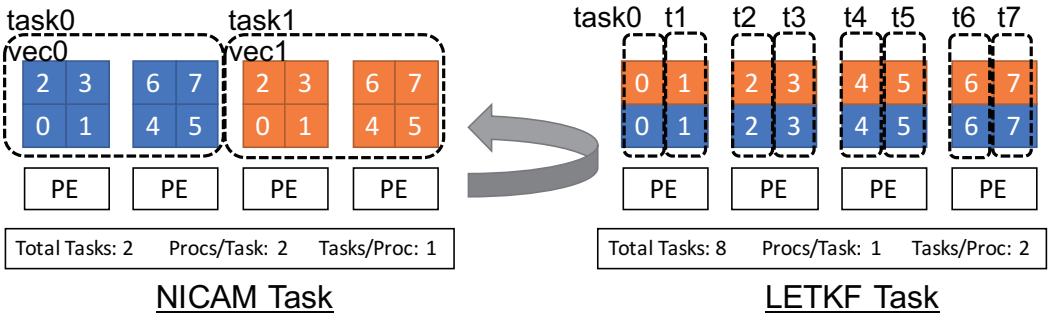


図 2: NICAM-LETKF でのタスク毎の並列数，タスク数，データ配置の違い

ワークを提案する．本フレームワークは多次元データを処理対象データ形式とし，それを処理する API セットを提供する．多次元配列データのみを対象とするためアプリケーションは限定されるが，上記のような空間分割を行う計算科学シミュレーションだけでなく，画像や動画を扱う機械学習分野などへの応用も期待でき，適用分野は広いと考えている．

本章ではモデルの説明を行い，フレームワークの実装を 4 章にて述べる．

3.1 データモデル

本フレームワークで扱うデータ（以降，*DataStore (DS)*）は多次元配列で表現される．各次元の要素（以降，*DataElement (DE)*）数は等しく，第 i 次元における要素数を n_i とした場合，総要素数は $\prod n_i$（全 n_i の積）となる．

DS を処理するタスクを複数プロセスで実行する場合，各 DE はプロセス間で分散配置されなければならない．DE のプロセス間配置規則を *Split* で指定する．Split は DS の，異なる要素数からなる別 DS への射影を定義する．Split は DS の次元数と同じ数の要素からなるタプルであり，タプルの要素は ALL(A)，NONE(N)，あるいは 1 以上の整数値からなる．タプルの i 番目の要素は DS の第 i 次元の分割規則を指定する．1 以上の整数値 m_i が指定された場合，その次元を m_i 個のブロックに分割する．ALL は $m_i = n_i$ の場合であり，NONE は $m_i = 1$ の場合である．Split により射影された DS の第 i 次元の要素数 S_i は以下で定義される．

$$S_i = \begin{cases} n_i & (Split_i = ALL) \\ 1 & (Split_i = NONE) \\ m_i & (Split_i = Other) \end{cases} \quad (1)$$

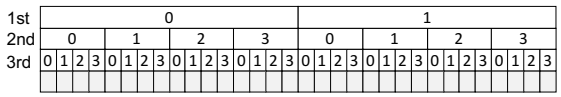
S_i の全次元の積がデータ分割数となり，それらをプロセス間でブロック分割して配置する．Split は DS に対して set_split() 関数を呼ぶことにより設定できる．

図 3 に，ALL と NONE からなる Split による DS のデータ分割を示す．図 3a は，3 次元で各次元の要素数がそれぞれ 2，4，4 である DS（以降，DS(2,4,4)）をフラットに図示したものである．合計要素数は各次元のサイズの積であり 32 となる．図 3b は DS(2,4,4) を Split<A,N,N>で分割した時のデータ配置である．Split<A,N,N>は第 1 次元でのみデータを分割するので，分割されたデータ数は 2 となり，システムに 2 プロセス以上存在する場合，2 つのプロセスに分割して配置される．

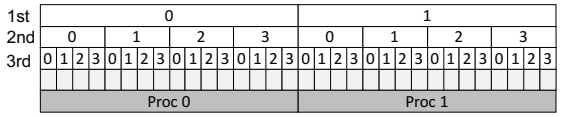
3.2 View によるデータ処理

API セットとして様々な機能を実現する関数群を提供するが，ここでは DS を入力としたタスク実行を行う唯一の関数である map() について説明する．その他の関数としては，DS に座標を指定してデータを追加する関数，DS から座標を指定してデータを取得・削除する関数，DS を分割・結合する関数などある．

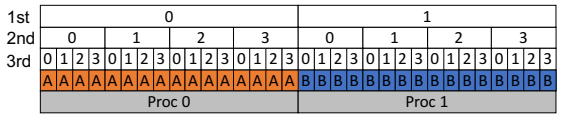
map() は 1 つの DS を入力とし，1 つの DS を出力とする，利用者定義タスクを実行する関数である．この際，利



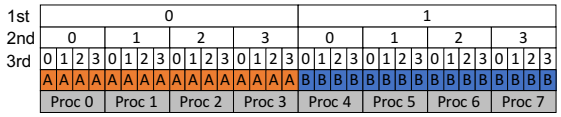
(a) Layout of `DS(2,4,4)`



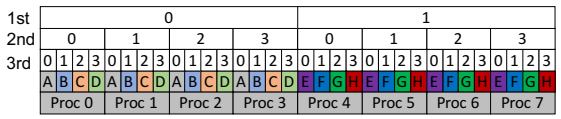
(b) Split by `Split<A,N,N>`



(c) Map by `Split<A,N,N>` and `View<A,N,N>`



(d) Map by `Split<A,A,N>` and `View<A,N,N>`



(e) Map by `Split<A,A,N>` and `View<A,N,A>`

図 3: Data and task layout of a DataStore

用者が指示する処理単位分割規則に従い DS 内要素を分割して，個々の断片ごとにタスクが実行される．`map()` の出力 DS を別の `map()` の入力 DS として繰り返しタスクを実行することができる．このように DS の処理の遷移を記述することにより，アプリケーションワークフローを構築する．最初の `map()` の入力 DS は，処理対象データを持つプロセスが，座標を指定して個々のデータを DS に追加することにより作成できる．

処理単位分割規則は *View* により指定する．View は Split と同じ表記からなり，DS の分割方法も同じである．タスク実行時のデータ読み込みを局所化するために，Split により分割配置されたデータ断片を持つプロセス上で，View により定義されたタスクが実行される．

図 3c に `DS(2,4,4)` を `Split<A,N,N>`によりデータ分割し，`View<A,N,N>`によりタスク分割した場合のデータ配置とタスク配置を示す．この場合，DS は第 1 次元にて分割され，16 個の DE からなる 2 つの断片に分けられる．View による分割では，それぞれ 2 つのタスクの入力として扱われる（図中 A と B）．プロセス数が 2 以上あれば，Split による分割で 2 プロセスにデータが分割配置されているので，その 2 つのプロセス上で，並列して 2 つのタスクが実行される．以上を行うには，以下の通りに `map()` を呼び出す．`ds1` は出力 DS である．

```
ds0 = DataStore(2, 4, 4)
ds1 = DataStore(2, 2)
ds0.set_split(<A, N, N>)
ds0.map(ds1, user_task, <A, N, N>)
```

図 3d は View は同じだが，`Split<A,A,N>`を指定した場合の例である．この Split では第 1 次元と第 2 次元によりデータを分割するため，4 個の DE からなる 8 個のデータブロックに分割される．プロセス数が 8 以上ある場合，データは 8 プロセスに分散配置され，各タスクには 4 プロセス提供され，それぞれ並列に処理することができる．

図 3e はさらに `View<A,N,A>`を指定した場合の例である．この View では第 1 次元と第 3 次元によりデータを分割するため，4 個の DE を処理する 8 個のタスクが定義される (図中 A〜H)．8 プロセス以上ある場合，各タスク 4 プロセスを用いて処理されるが，各プロセス 4 タスク担当しているため，それらは逐次に処理される（A,B,C,D を逐次に，それと並列して E,F,G,H を逐次に実行)．

以上のデータ配置パターンとタスク実行パターンは大きく異なるが，View と Split の変更のみで対応できる．

`map()` で実行されるタスクが満たす制約を述べる．各タスクには実行時に，システムから適切な数のプロセス群からなる実行環境を与えられる．そのため，実行場所を前もって知ることはできない．実行場所を意識せずプログラミングできるメリットがある一方，タスクから DS 外のデータにアクセスする際には特別に対応する必要がある．各タスクには，そのタスクで処理する DE とその座標の組の集合が入力として渡される．複数プロセスで実行する場合には分割され，重複のないサブ集合が入力として各プロセスに渡される．タスク処理中に他プロセスのデータが必要になった場合には，利用者が送受信関数を用いて転送を行う．タスクの結果を後続タスクに渡す場合には，出力 DS に座標を指定してデータを追加する．

3.3 タスク実行時の各種パラメータの関係

タスクの総数は DS の各次元のサイズと View により以下で定義される．

$$Task\_Count = \prod V_i \tag{2}$$

ここで V_i は View により射影された DS の第 i 次元の要素数であり，式 1 同様に定義される．Split により分割される，データ断片の数も式 1 を用いて同様に定義できる．

$$Data\_Count = \prod S_i \tag{3}$$

システムにある総プロセス数が式 3 に等しい場合，全プロセスに均一にデータが配置される．システムにそれ以上のプロセスが存在する場合，一部のプロセスにはデータが配置されず，タスク実行にも参加しない．

各プロセスが実行するタスク数は以下の式により定義さ

れる．

$$Tasks\_Each\_Proc = \prod \lceil \frac{V_i}{S_i} \rceil \quad (4)$$

一方，各タスクに割り当てられるプロセス数は以下により定義される．

$$Procs\_in\_Task = \prod \lceil \frac{S_i}{V_i} \rceil \quad (5)$$

しかしながら，式 4 と 5 が成立するのは，システム内のプロセス数が式 3 と等しい場合である．式 3 より大きくなる場合には，タスクが実行されないプロセスが存在し，資源利用効率が悪くなる．式 3 未満となる場合には，他プロセスよりも多くのデータを処理するプロセスが存在し，負荷分散が悪くなる．効率よくタスクを実行するためには，利用者はシステム内の総プロセス数と DataStore，View，Split の関係を考慮する必要がある．

4. 実装

上記の提案を C++言語にて，プロセス管理・プロセス間通信に MPI を用いて実装した．提案システムを，システムと利用者定義タスクが同一プロセス群上で動作するように設計した．そのため，各タスクは，View の指示に従って `MPI_Comm_split()` によりシステムが動作する `MPI_COMM_WORLD` を分割したサブコミュニケータ上で実行される．個々のタスクは並列実行されるが，ワークフローを記述する main プログラムは逐次プログラミングモデルで動作するように設計した．これを実現するために，タスク外の処理は対象データを持つプロセス，あるいはランク 0 プロセスで実行され，結果のみをプロセス間で共有するよう実装した．現在の実装では 2 種類のストレージターゲットがあり，DS の内容はインメモリか，ファイルのいずれかに保存される．C/C++/Fortran API を提供する．

`map()` の実装について述べる．各プロセスが直前のタスクで DS に追加したデータは，そのプロセスローカルに配置される．その結果，Split により指定された，後続タスクが要求するデータ配置規則に合わない可能性が高い．そこで `map()` ではタスク実行前に，現在のデータ配置が Split の規則に従っているか確認し，従っていない場合には再配置を行う．確認・再配置のために，各プロセスは自分が持つ DS 内の全 DE の座標と Split から 0 -「プロセス数-1」の数値を生成する．全プロセスにおいて，この数値がランク値と 1 つでも異なる場合には再配置を行う．再配置はこの数値を宛先として `MPI_Alltoallv` で行う．タスクを実行する際には，処理対象のデータブロックを持つプロセス群が同一サブコミュニケータを構成するよう `MPI_Comm_split()` を実行する．プロセス上に，異なるタスクで実行される複数のデータブロックが存在する場合，そのプロセスは複数のタスクを逐次に実行する．タスク実行時に，そのタスクにそのプロセスで利用できる全ての計算機資源を提供するためである．

利用者定義タスクの実装について説明する．C++ API を用いる場合，利用者定義タスクは以下のクラスを継承する関数オブジェクトとして実装する．

```
class Mapper {
public:
  virtual int operator()(
          DataStore* inds, DataStore* outds,
          Key& key,  vector<DataPack>& dps,
          MapEnvironment& env);
};
```

`inds`，`outds` はそれぞれ入出力 DS であり，タスクの計算結果は `outds` に追加する．`key` はそのタスクで処理されるデータ集合の識別子であり，DS 内の DE の座標と View から計算される．`dps` にはこのプロセスで処理される DE とその座標の組みからなる配列である．`env` にはこのタスクを実行するプロセス群を含む `MPI_Comm`，タスク実行時に指定された View や Split からなる実行環境情報を含む．利用者がタスクを実装する際には，`dps` 中の各データを，`env` 中の `MPI_Comm` を用いて必要に応じて通信を行いながら計算し，結果を `outds` に書き込むよう実装する．C と Fortran ではタスクをコールバック関数として実装する．Fortran の場合には，以下のインターフェースを持つ関数として実装する．各引数は C++のインターフェースと同等の意味である．

```
integer(c_int) function mapfn_t(inds, outds,
                                key, dps, env)
end function
```

利用者定義タスクはシステムと同じプロセス空間で動作するため，Abort する処理は記述できない．異常時には異常を表す戻り値をタスクから返す必要がある．

5. 評価

格子 QCD アプリケーションの BQCD を用いて，性能と生産性の評価を行なった．BQCD は主に Fortran で記述された，格子 QCD 計算を行う 15 万行強のプログラムであるが，本評価では Dirac 方程式をアンサンブル実行する箇所を対象に次の 2 実装を評価した．1 つ目はオリジナルの BQCD 実装（以降，MPI Native）で，該当箇所の前後で `MPI_Alltoallv` にて再配置のコードを記述している．もう 1 つは該当箇所を提案フレームワークを用いて実装したものである（以降，Proposal）．どちらも計算内容は等しいが，データ再配置手法が異なる．

評価に用いた Proposal の疑似コードを図 4 に示す．BQCD は Dirac 方程式を解くタスク（*DiracEq*）を含むワークフローを繰り返し実行する．図 4 は *DiracEq* の 1 度の実行とその前後のデータ再配置を行う．

格子 QCD は 4 次元空間のシミュレーションを行い，ここでは各次元 64 格子点から構成されるとする．

```
   # Create a DataStore
   # 1st      : ensemble
   # 2nd - 5th: simulation space (x, y, z, t)
   # Default Split: <N, 8, 8, 8, 8>
1  ds0 = DataStore(4, 64, 64, 64, 64)
2  ds0.load_parallel(task_load)
   # Execute Dirac equation solver
3  ds0.set_split(<A, 8, 8, 4, 4>)
4  ds0.map(ds0, task_dirac, <A, N, N, N, N>)
   # Restore to the default layout
5  ds0.set_split(<N, 8, 8, 8, 8>)
6  ds0.map(NULL, task_restore, <N, N, N, N, N>)
```

図 4: BQCD pseudo-code

DiracEq のアンサンブル数は 4 であるとする．このデータを DS に読み込ませるために，1 行目では 5 次元の DS(4, 64, 64, 64, 64) を宣言している．2 行目で，データ読み込み用の専用の map() を実行している．task_load にて，各プロセスの入力配列から，ds0 にデータを読み込ませる．BQCD では空間データを次元ごとに分割数を指定して，プロセスに分散配置する．各プロセスは担当領域の全アンサンブルのデータを持つ．4096 プロセスを用いて，各次元 8 分割した場合，DS に読み込ませた結果は Split<N,8,8,8,8>が指定された場合と同じ配置になる．

3，4 行目で *DiracEq*（task_dirac）を実行している．アンサンブル並列で実行するには，各アンサンブルの入力を異なるプロセスに配備する必要があるため，Split ではアンサンブル次元には ALL を指定している．アンサンブル数は 4 なので，1 アンサンブルあたり 1024 プロセスで実行される．空間データの分割数が 1024 になるよう，Split の 2 次元目以降を設定している．*DiracEq* はアンサンブルごとに実行さるので，map() の View の指定では，アンサンブル軸でのみ分割する View<A,N,N,N,N>を指定している．

5，6 行目では後続タスクを実行するために，データレイアウトをもとに戻し，出力配列に結果を書き込む task_restore を実行している．データを元のレイアウトに再配置できてしまえば，個々のプロセスごとにコピーを行うだけであるので，map() の View は任意で構わない．

5.1 性能評価

京コンピュータを用いて最大 1024 ノードまでの強スケール，弱スケールの評価を行った．1 MPI プロセス/ノードで実行し，MPI Native と Proposal とで同じノード集合を用いた．評価に用いた，アンサンブル並列実行に関連するパラメータをそれぞれ表 1 と表 2 にまとめる．各格子点 156 個の倍精度複素数の値を持つため，例えば 1024 プロセスを用いた強スケール評価の場合，各プロセスは約 5MB をインメモリに持ち（1024 格子点 × 2 アンサンブル × 156 倍精度複素数値），全体で 5GB が処理される．*DiracEq* の実行時間はモンテカルロステップの進捗につれ，熱平衡状態に達するまで増加する．実際，$24 \times 24 \times 24 \times 48$ 格子を 384 ノードで実行した際（1728 格子点/ノード）には 1 秒から 12 秒と実行時間が増加したことを確認した．熱平衡に達するには長時間の実行を要するため，本評価では *DiracEq* 実行時間は評価対象外とし，*DiracEq* アンサンブル実行のための (1) 準備（Prepare），(2) 実行前再配置（Relocation 0），(3) 実行後再配置（Relocation 1），(4) 後処理（Post）の合計実行時間を評価対象とした．

表 1: 強スケール評価で用いたパラメータ

Processes	128	256	512	1024
Lattice Size	32x32x32x32			
Ensembles	2			
Grid Point/Process	8192	4096	2048	1024

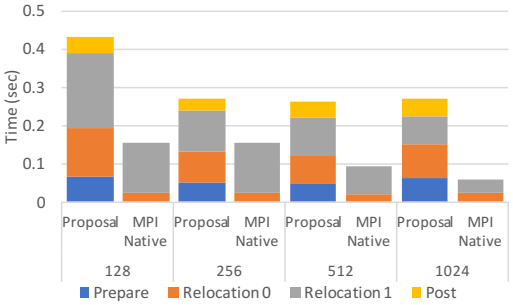


図 5: Evaluation of Strong Scalability

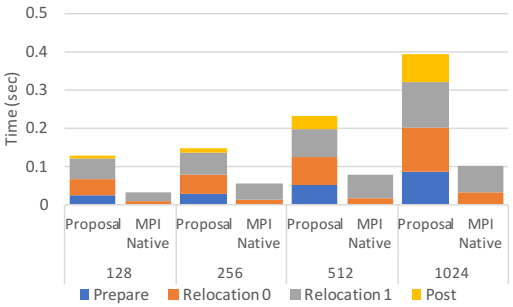


図 6: Evaluation of Weak Scalability

強スケールの結果を図 5 に，弱スケールの結果を図 6 に示す．Proposal では Prepare として対象データを DS に読み込ませる処理が，Post として DS を解放する処理を行う必要があるが，MPI Native ではそれらは必要なく，それぞれの時間は 0 である．強スケール評価では，いずれの結果もプロセス数が増えるにつれ時間が短縮されているが，提案の方が最大で 4.5 倍遅い．これには 2 つ理由がある．1 つは Prepare と Post の時間であり，強スケール評価では格子サイズが固定なため，プロセス数によらずほぼ一定の時間を要している．もう 1 点は再配置実施前の，再配置の要不要の確認処理である．BQCD では必ず再配置が実施

表 2: 弱スケール評価で用いたパラメータ

Processes	128	256	512	1024
Lattice Size	32x32x16x16	32x32x32x16	32x32x32x32	64x32x32x32
Ensembles	2			
Grid Point/Process	2048			

されるが，その前提知識がないため，毎回確認を行っている．確認処理実装にあたり，配列への排他制御を行なっており，実際そのコストが大きく出ていることを確認している．弱スケール評価では，MPI Native に対して Proposal が最大で 4 倍実行時間が大きい．格子サイズ増加に伴う，Prepare と Post における DS 処理時間の増加，再配置要不要の確認処理時間の増加が主な原因である．

DiracEq 実行時間に対する，Proposal による再配置のオーバーヘッドを外挿により評価する．*DiracEq* の実行時間は熱平衡状態に達したのちは，プロセスごとの格子点数に正比例して増加する．ストロングスケールすることを想定し，1728 格子点/プロセスの結果をもとに外挿すると，本評価に用いた設定では *DiracEq* の実行時間は 7〜57 秒となる．一方，Proposal による再配置時間は最大で 0.43 秒である．オーバーヘッドは最大でも 6%程度であり十分小さく，BQCD に関しては性能面の問題はないと言える．

5.2 生産性評価

提案フレームワークはデータ再配置と，プロセスへのタスク割り当てを抽象化し，利用者からは透過的にそれらを行う．一方で，MPI Native の場合同様，利用者は対象データの分割規則と，タスク割り当て規則を考えて実装する必要がある．提案フレームワークを用いる場合，利用者に求められるのは，(P1) 対象データの分割規則とタスク実行規則を定義すること，(P2) 対象データを DS に読み込ませること，(P3) 対象計算を `map()` のタスクとして定義し，View と Split を用いて実行することが求められる．MPI Native の場合は，(M1) 対象データの分割規則とタスク実行規則を定義すること，(M2) 対象データの送信先プロセスを求め，送信バッファを作成し転送すること，(M3) データを持つプロセス群を含むコミュニケータを作成すること，(M4) 対象計算の実行をスケジュールすることが求められる．P1 と M1 は共通する．P3 は，M2 + M3 + M4 に相当する．P3 は対象計算を定められたインターフェースで実装し，P1 で定義した規則に従い View と Split を指定するだけで十分である．一方の M2 は各プロセスが持つ全データについて，宛先プロセスを計算し送信バッファの適切な場所にデータをセットしなければならず，面倒かつ間違えやすい作業である．M3，M4 についても，再配置後のデータ配置を考慮してコミュニケータ分割を行い，タスクの実行をスケジュールする必要があり，面倒な実装となる．これら面倒な作業を抽象化し，利用者から隠蔽できている点において，提案フレームワークの生産性は MPI を直に用いる場合に対して高いと言える．P2 に関しては冗長な実装コストではあるが，タスクの入力バッファのデータを DS に座標を指定してコピーするだけで十分であり，M2 で求められる作業と比較すると少ない．実際，Proposal と MPI Native にてコード行数をカウントしたところ，共通部分を除くと，Porposal は 650 行に対して MPI Native は 732 行となった．

以上をまとめると，提案手法は性能面に関しては MPI Native より大きいオーバーヘッドが生じるが，実装コストを下げるメリットがある．タスクの実行時間が転送時間よりも十分に大きいことが前提となるが，有用な手法であると言える．

6. 関連研究

本研究で提案した Multi-View モデルでの多次元配列の分割は，分散配列をサポートする言語や処理系に共通する．High Performance Fortran (HPF) は配列要素のプロセス間分割を定義するコンパイラ指示文を提供する [3]．HPF による 2 次元配列のブロック分割は以下のように記述できる．

```
      REAL A(100, 100)
!HPF$  PROCESSORS PROCS(10)
!HPF$  DISTRIBUTE A(BLOCK, BLOCK) ONTO PROCS
```

`DISTRIBUTE` の指示が，本研究の Split に相当する．Partitioned Global Address Space (PGAS) モデルを提供するシステムでは，複数プロセスにまたがる局所性を考慮した共有アドレス空間を利用でき，Co-array Fortran や Unified Parallel C (UPC) [4]，UPC++ [5]，X10 [6]，Chapel [7] など高性能計算向けの言語でサポートされている．例えば，X10 の分散配列のブロック分割は以下のように記述できる．

```
val R : Region = 1..100;
val D = Dist.makeBlock(R);
va distArr = DistArray.make[Int](D, ([i]:Point)=>i);
```

`Dist.makeBlock` の指定が，本研究の Split に相当する．これらには本研究の View に相当する概念がない．そのため，分割配置されたサブ領域を持つ一部のプロセス間で連携して処理する場合には，利用者はプロセスへのデータ配置を意識したタスクマッピングを記述しなければならない．X10 の場合にはデータを持つプロセスを識別したのちに，`at()` でそのプロセスに移動させて処理を行う必要が

ある．Hierarchically Tiled Array (HTA) はマルチコアシステムの性能向上のための局所性と並列性向上のためのデータモデルである [8]．HTA は階層的なタイルからなる多次元データであり，タイル単位でプロセス間配置や処理をおこなう．HTA を用いることで本研究と同等のデータ分割・計算実行ができると考えられるが，タイルが分割される階層とインデックスを考慮して実装する必要がある．本研究は HTA のアプローチをより抽象化したと言える．Havanero-Java は開発者の生産性向上を目的に，1 次元配列を任意の多次元配列として扱うことができる Array-View を提供する [9]．プログラムからのデータの見方を変更できる点において本研究の View に相当するが，データ分散を定義する方法はない．

MapReduce [10] は，計算科学分野でもデータ並列計算を行うプログラミングモデルとして広く用いられている．計算科学アプリケーション向けに，繰り返し処理の効率化を目的とした処理系である Twister [11] や，MPI アプリケーションからの利用を目的とした MR-MPI [12] や K MapReduce [13] などの処理系がある．Apache Hadoop [14] やこれら処理系により，ゲノム解析 [15] や天気予報，地球科学 [16]，分子動力学 [17] などのワークフローが MapReduce モデルにて構築できることが実証されている．本研究の提案と MapReduce モデルの違いはデータの抽象表現と，並列タスクの実行方式にある．MapReduce ではデータを Key-Value で表現し，Key には任意の値を取ることができる．一方提案では，データは多次元配列として管理し，個々のデータは座標を Key として識別される．提案では Key の表現に制約があるが，Key の階層性（座標の次元）を用いてデータを複数ノードに分散配置し，データ局所性を考慮した MPI 並列タスクの実行を実現している．一方 MapReduce では，データ局所性を考慮したタスク実行は実現されているが，タスクは逐次プログラムである．MPI 並列プログラムの実行も不可能ではないが，その際の各プロセスにおけるデータアクセスの局所性は考慮されていない．

高性能計算向けのワークフローツールやアプリケーションフレームワークにはデータアクセスを抽象化しているものもある．Swift [18] と Swift/T [19] はデータの位置とデバイス透過性を実現する，メニータスクデータ処理ワークフローツールである．これらはタスク自身が MPI 並列プログラムであることを考慮していない点，およびそのタスク内部でのデータアクセスの局所性を考慮していない点で，本研究とは異なる．Kokkos [20] は性能可搬性を実現するアプリケーション構築のためのプログラミングモデルであり，ホストメモリとデバイスメモリを含む多層のメモリを管理する．Kokkos はメモリ階層の抽象化を行い，透過的にデバイスに適したレイアウトで（row-major か column-major か）多次元配列を格納する．Kokkos におけるデータアクセス抽象化の目的は 1MPI プロセス内部での性能可搬性を実現することに対し，我々の研究は透過的に複数ノード上でのデータアクセス局所性を向上させることにあるため，目的が異なる．

7. まとめ

並列システムで実行される計算科学アプリケーションのワークフローは，異なる並列度とデータアクセスパターン，実行数の要求を持つ複数のタスクから構成され，高性能実現のためには，タスクごとの要求に合わせる必要がある．この実装コスト削減を目的に，本研究では多次元配列データを対象に，シンプルな指示でプロセス間データ再配置を行い，要求されたタスク粒度で局所性の高いデータアクセスを行えるようタスク配置を行うフレームワークを提案した．提案フレームワークでは Multi-View データモデルを提供し，データ分割規則（*Split*）と処理単位分割規則（*View*）をタプルとして与えるのみで，透過的にデータ配置，タスク配置を行う．格子 QCD アプリケーションの 1 実装である BQCD にて評価を行なった．データ配置とタスク配置を MPI で実装した場合に比べると，提案の方がデータ再配置に要するオーバーヘッドが最大で 4.5 倍大きいことを確認したが，タスク実行時間に対する割合は最大でも 6%以下であり，性能に与える影響は軽微であることを確認した．また，提案を用いた方が実装時に検討する項目も少なく，コード記述量も少なくなることを確認した．

今後の課題として，データ再配置の性能向上と，前述の NICAM-LETKF や機械学習等へのアプリケーション適用事例を増やすことを考えている．並行して，アプリケーション要求に合わせて，データモデルや API セットの修正・改善を行う予定である．また，多階層からなるストレージ階層を考慮したデータ管理を行うことも考えている．

謝辞 本論文の結果（の一部）は，理化学研究所のスーパーコンピュータ「京」を利用して得られたものです．

参考文献

[1] Nakamura, Y. and Stuben, H.: BQCD - Berlin quantum chromodynamics program, *The 28th International Symposium on Lattice Field Theory (Lattice2010)* (2010).
[2] Yashiro, H., Terasaki, K., Miyoshi, T. and Tomita, H.: Performance evaluation of a throughput-aware framework for ensemble data assimilation: the case of NICAM-LETKF, *Geoscientific Model Development*, Vol. 9, No. 7, pp. 2293–2300 (2016).
[3] Loveman, D. B.: High performance Fortran, *IEEE Parallel Distributed Technology: Systems Applications*, Vol. 1, No. 1, pp. 25–42 (1993).
[4] Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F., El-Ghazawi, T., Mohanti, A., Yao, Y. and Chavarría-Miranda, D.: An Evaluation of Global Address Space Languages: Co-array Fortran and Unified Parallel C, *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Pro-*

gramming, pp. 36–47 (2005).

[5] Zheng, Y., Kamil, A., Driscoll, M. B., Shan, H. and Yelick, K.: UPC++: A PGAS Extension for C++, *28th IEEE International Parallel and Distributed Processing Symposium*, pp. 1105–1114 (2014).

[6] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An Object-oriented Approach to Non-uniform Cluster Computing, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp. 519–538 (2005).

[7] Chamberlain, B. L., Deitz, S. J., Iten, D. and Choi, S.-E.: User-Defined Distributions and Layouts in Chapel: Philosophy and Framework, *2nd USENIX Workshop on Hot Topics in Parallelism* (2010).

[8] Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B. B., Garzarán, M. J., Padua, D. and von Praun, C.: Programming for Parallelism and Locality with Hierarchically Tiled Arrays, *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 48–57 (2006).

[9] Cavé, V., Zhao, J., Shirako, J. and Sarkar, V.: Habanero-Java: The New Adventures of Old X10, *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pp. 51–61 (2011).

[10] Dean, J. and Ghemawat, S.: MapReduce : Simplified Data Processing on Large Clusters, *Communications of the ACM*, Vol. 51, No. 1, pp. 1–13 (2008).

[11] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J. and Fox, G.: Twister: a runtime for iterative MapReduce, *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 810–818 (2010).

[12] Plimpton, S. J. and Devine, K. D.: MapReduce in MPI for Large-scale graph algorithms, *Parallel Computing*, Vol. 37, No. 9, pp. 610–632 (2011).

[13] Matsuda, M., Maruyama, N. and Takizawa, S.: K MapReduce: A Scalable Tool for Data-Processing and Search/Ensemble Applications on Large-Scale Supercomputers, *IEEE Cluster 2013 Conference*, Indianapolis (2013).

[14] : Welcome to Apache Hadoop, `http://hadoop.apache.org/`.

[15] McKenna, A., Hanna, M., Banks, E., Sivachenko, A., Cibulskis, K., Kernytsky, A., Garimella, K., Altshuler, D., Gabriel, S., Daly, M. and DePristo, M. A.: The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data, *Genome Research*, Vol. 20, No. 9, pp. 1297–1303 (2010).

[16] Chen, Q., Wang, L. and Shang, Z.: MRGIS: A MapReduce-Enabled High Performance Workflow System for GIS, *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pp. 646–651 (2008).

[17] Tu, T., Rendleman, C. A., Borhani, D. W., Dror, R. O., Gullingsrud, J., Jensen, M. O., Klepeis, J. L., Maragakis, P., Miller, P., Stafford, K. A. and Shaw, D. E.: A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories, *International Conference for High Performance Computing Networking Storage and Analysis (SC08)*, No. November (2008).

[18] Wilde, M., Hategan, M., Wozniak, J. M., Clifford, B., Katz, D. S. and Foster, I.: Swift: A language for distributed parallel scripting, *Parallel Computing*, Vol. 37, No. 9, pp. 633–652 (2011).

[19] Wozniak, J. M., Armstrong, T. G., Wilde, M., Katz, D. S., Lusk, E. and Foster, I. T.: Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing, *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 95–102 (2013).

[20] Edwards, H. C., Trott, C. R. and Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, *Journal of Parallel and Distributed Computing*, Vol. 74, No. 12, pp. 3202–3216 (2014).