修士論文中間報告 2010 年 4 月 30 日

計算規模を動的に拡張・縮小可能な並列計算をサポートする PGAS 処理系

A PGAS Framework Supporting a Parallel Computation Expanding and Shrinking its Scale Dynamically

情報理工学系研究科 電子情報学専攻 近山・田浦研究室 修士 2 年 48096419 原健太朗

Abstract

In order to execute a parallel computation on a cloud environment in which the number of available resources can change dynamically, the scale of the parallel computation should expand or shrink dynamically in response to the increase or decrease of available resources. It is, however, difficult to describe most parallel scientific computings such as finite element methods so that they can scale up or down dynamically. Therefore, this paper proposes a PGAS framework named *DMI*, with which if a programmer creates a sufficient number of threads then the framework automatically and dynamically schedules these threads on the available resources at the time. Furthermore, as primary elemental techniques in DMI, this paper proposes techniques for improving the performance of a global address space, techniques for maintaining the consistency of the global address space beyond dynamic joining or leaving of nodes, and techniques for migrating a live thread safely between nodes.

1 序論

1.1 背景と目的

有限要素法による応力解析や地震シミュレーション，粒子法による流体解析や衝突解析など，長時間を要するような大規模な並列科学技術計算への要請が高まっている．

従来，企業や大学などの組織がこのような並列科学技術計算を実行しようと思えば，その組織は自前でデータセンタを構築して管理する必要があった．これに対して，近年注目されているコンピューティング形態がクラウドコンピューティング（以下，クラウド）である．クラウドでは，クラウドプロバイダと呼ばれる組織が大規模なデータセンタを構築し，インフラストラクチャ，プラットフォーム，ソフトウェアなどを整備して，それらをサービスとして利用者に提供する．そして利用者は，それらのサービスを必要なときに必要な量だけ利用することができ，実際に利用した量だけ課金される [13, 5]．このようなクラウドにおいては，煩雑なサーバ管理技術などが不要なうえ，従量制課金のもとで必要なときに必要な量だけ利用できるため，自前でデータセンタを管理するよりもコストパフォーマンスが優れる場合が多い．代表的なクラウドサービスとしては，仮想サーバやストレージなどの計算機資源を提供する Amazon EC2 や Amazon S3，利用者が作成したアプリケーションの実行環境を提供する Google App Engine や Windows Azure，エンドユーザのためのソフトウェアサービスを提供する Google Docs や Salesforce.com の CRM などがある．

このように，クラウドは多種多様なアプリケーション領域に対して効率的な実行環境を提供しているが，実は，クラウド上で大規模な並列科学技術計算を効率的に実行するのは容易ではない．本研究では，クラウド上で大規模な並列科学技術計算を実行することの難しさについて分析したうえで，それを効率的に実行可能な並列分散プログラミング処理系を開発することを目的とする．

1.2 クラウドの特徴

クラウドでは，多数の計算資源を多数の利用者で利用することになる．よって，クラウドサービスを効率的に運用するためには，何らかの形で，利用者間で計算資源を効率的にスケジューリングする仕組みが必要になる．具体例で説明する．あるクラウドの中に利用者 A と利用者 B がいるとし，今利用者 A の負荷が増大したとする．このときこのクラウドを利用している利用者が他にいなければ，利用者 A の計算規模を拡張して負荷分散が図られる．やがて，利用者 B の負荷が利用者 A の負荷より増大したとすると，今度は利用者 A の計算規模を縮小するかわりに利用者 B の計算規模を拡張することで全体としての負荷分散が図られる，というような計算資源のスケジューリングがたとえば起きる．当然，どのような条件が成立したときにどのように利用者の計算規模を拡張・縮小させるかは，各クラウドサービスの課金体系，対象とするアプリケーション，SLA（Service Level Agreement）などに応じてさまざまである．しかし，いずれにせよ，クラウドの本質は，全体の負荷状況に応じて各利用者の計算規模を拡張・縮小することで，多数の計算資源を多数の利用者間で効率的にスケジューリングするという点にある．クラウドでは各利用者が利用可能な計算資源数が動的に増減するのである．

このようなクラウド環境の具体例としては，Amazon EC2 Spot[1] がある．Amazon EC2 Spot は，指示した数の仮想マシンを確実に利用することが可能な Amazon EC2 のクラウド環境よりも，より安価に計算資源を提供することを意図して作られたクラウドサービスである．Amazon EC2 Spot では計算資源を「競り落として」利用する．すなわち，需要と供給に応じて計算資源の時価が動的に変動しており，利用者が決めた入札価格がその時点での時価を上回っていれば，その時価で仮想マシンを利用することができる．たとえば，時価が$0.03/時間のときに$0.05/時間で入札すれば仮想マシンが起動する．やがて

時価が$0.06/時間に上がったとすれば仮想マシンの電源が落とされる．よって，不要不急の計算であれば，時価よりも安い入札価格を設定することで「いつか実行されれば良いから安く実行」することができるし，急を要する計算であれば，時価よりも高い入札価格を設定することで「高くても良いからすぐに実行」することもできる．このように Amazon EC2 Spot では，時価と入札価格との兼ね合いで利用可能な計算資源数が動的に増減しうる，安価なクラウド環境を提供している．

1.3 クラウド上での並列計算

さて，以上のようなクラウドにおいて，並列計算はどのように実行されるべきかを考える．前述のように，クラウドでは利用可能な計算資源が動的に増減する．よって，クラウド上での並列計算は，そのとき利用可能な計算資源に対応して計算規模を動的に拡張・縮小しながら実行される必要がある．たとえば，1000 ノードが利用可能なときにはその 1000 ノードを利用して実行され，やがて 10 ノードしか利用できなくなればその 10 ノードだけを利用して実行される必要がある．ところが，明らかに，利用可能な計算資源の増減に対応して，計算規模を動的に拡張・縮小できるように並列科学技術計算を記述するのは難しい．あるときは 1000 ノードで，2 時間後にはまったく別の 10 ノードで実行されるように並列科学技術計算をプログラムとして記述するのは明らかに困難である．

したがって，長時間を要する大規模な並列科学技術計算をクラウド上で実行するためには，計算規模の拡張・縮小を強力にサポートするようなプログラミング処理系が必須である．具体的には，以下の性質を満足するプログラミング処理系が要請されている：

- プログラマは，（計算規模の拡張・縮小をいっさい考えることなく，単に）アプリケーションの並列性をプログラムに記述するだけで良い．
- あとは処理系が自動的に，そのとき利用可能な計算資源に対応してプログラムの計算規模を動的に拡張・縮小してくれる．

要するに，十分な並列性さえ記述された並列プログラムが与えられたときに，その計算規模を自動的に拡張・縮小してくれるようなプログラミング処理系が要請されている．

計算規模を自動的に拡張・縮小してくれるクラウドサービスとしては，Google App Engine[2] がある．Google App Engine では，利用者が Java もしくは Python で記述した Web アプリケーションを登録しておくと，そのアプリケーションに対するクライアントからのリクエスト数の増減に応じて，計算規模が自動的に拡張・縮小され，利用者が何の意識を払わずとも負荷分散が図られる．2010 年 4 月時点における無料コースでは，1 分間に最大 7400 個ものリクエストが処理可能とされている．しかし，Google App Engine で実行できるアプリケーションは，30 秒以内に終了するアプリケーションに限定されている．よって，Google App Engine では，各リクエストが 30 秒以内に処理できるような多くの Web アプリケーションは実行できるものの，長時間を要する並列科学技術計算を実行することはできない．計算の各部分が 30 秒以内に終了するように並列科学技術計算全体をうまく分割して実行すれば可能かもしれない

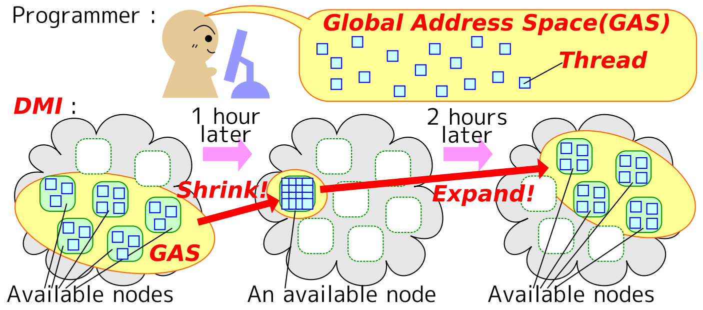

Fig.1 Concepts of DMI.

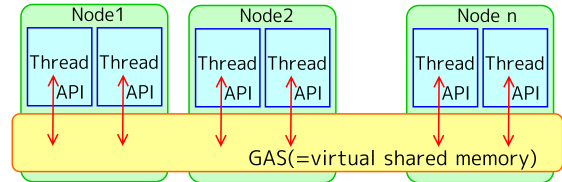

Fig.2 Programming Interfaces of DMI.

が，一般に，有限要素法などの並列科学技術計算をそのように細粒度に分割して並列プログラムを記述するのは困難である．Google App Engine が 30 秒以内という条件を課しているのは，各リクエストを単位として計算資源をスケジューリングしているため，各リクエストが 1 時間も 2 時間も処理を継続していては，利用者全体を通じての応答性を維持できなくなるためだと考えられる．以上のように，Google App Engine は，短時間で完了する計算をスケーラブルにクラウド上で実現する枠組みではあるが，長時間を要する並列科学技術計算をどうクラウド上で実現すれば良いかに関する解決策が提示されているわけではない．

1.4 本研究の提案：DMI

以上の考察に基づき，本研究では，長時間を要する大規模な並列科学技術計算に対しても，計算規模を自動的に拡張・縮小してくれるような並列分散プログラミング処理系として，Distributed Memory Interface（DMI）[17, 19, 16] を提案して実装し評価する．DMI の概要は以下のとおりである（Fig.1）：

- プログラマは，（計算規模の拡張・縮小を考えることなく）十分な数のスレッドを生成するように並列プログラムを記述するだけで良い．
- あとは処理系が自動的に，それら大量のスレッドを，そのとき利用可能な計算資源にスケジューリングすることで，利用可能な計算資源の増減に対応して計算規模を動的に拡張・縮小してくれる．
- スレッド間のデータ共有レイヤーとして，高性能な大域アドレス空間を提供する．

ここで大域アドレス空間とは，（複数のノードに分散された）すべてのスレッドからアクセス可能な仮想的な共有メモリのことである．ソケットプログラミングなどでは，分散環境でデータを共有する場合にはデータを send/recv しなければならないが，大域アドレス空間を持つ処理系では，各データに対してグローバルに一意なアドレスが振られており，各スレッドはそのアドレスを read/write することでデータを共有できる．DMI では，大域アドレス空間に対する mmap/munmap, read/write，排他制御や条件変数などの同期，スレッドの cre-

ate/join/detach などの API を提供しており（Fig.2），概念的にはあたかも通常の共有メモリ環境上の並列プログラミングと同様のプログラミングインタフェースでクラウド上の並列計算を記述することができる．現時点の DMI は，C 言語の共有ライブラリとして約 19000 行で実装されており，73 個の API を提供している．

DMI における主要な要素技術は以下の 3 点である：

- 高性能な並列プログラムをサポートしなければならない．大域アドレス空間に対するアクセス性能を高めるにはどうすれば良いか?
- 計算規模を拡張・縮小するためには，大域アドレス空間の一貫性がノードの動的な参加・脱退を越えて正しく維持されなければならない．どのように一貫性を維持すればよいか?
- 大量のスレッドをそのとき利用可能な計算資源にスケジューリングするには，生きたスレッドをノード間で移動しなければならない．安全なスレッド移動をどう実現するか?

以降の節では，上記の 3 点の要素技術に関して，関連研究，提案手法，性能評価を含めて順に論じる．

2 大域アドレス空間のデザイン

2.1 関連研究：従来の PGAS 処理系の問題点

大域アドレス空間に基づくプログラミングモデルでは，物理的に分散した複数のノード上に存在する各データに対してグローバルに一意なアドレスが振られており，そのアドレスを read/write することでデータにアクセスすることができる．大域アドレス空間を実現する代表的な手法は，PGAS（Partitioned Global Address Space）と呼ばれるもので，UPC[7] や Global Arrays[14] などの処理系が有名である．一般的な PGAS では，データを適当なサイズのページという単位に分割して管理する．そして，各ページに対してオーナーとなるノードを 1 個固定的に決め，そのオーナーにそのページの最新状態と一貫性を管理させる．たとえば，1GB のデータを 250MB のページ p_1, p_2, p_3, p_4 の 4 個に分割し，ページ p_1, p_2 のオーナーをノード 1 に，ページ p_3, p_4 のオーナーをノード 2 に決めたとする．このとき，たとえばノード 2 上のスレッドがページ p_1 を read しようとした場合には read フォルトが発生し，ページ p_1 のオーナーであるノード 1 に対して，ノード 2 からリクエストが送られる．そして，ノード 1 がページ p_1 の最新状態をノード 2 に送ることで read フォルトが解決される．また，ノード 2 上のスレッドがページ p_1 を write しようとした場合には write フォルトが発生し，ページ p_1 のオーナーであるノード 1 に対して，ノード 2 から write すべきデータが送られる．そして，そのデータに基づいてノード 1 がページ p_1 を更新することで write フォルトが解決される．

以上の PGAS の基本原理について議論する．第一に，オーナーは固定で良いかどうかを考える．DMI では，既存の PGAS 処理系とは異なり計算規模の拡張・縮小に対応する必要があるが，計算規模を拡張・縮小すれば各スレッドのページに対するアフィニティが変化することに注意する．たとえば，ノード 1 にスレッド t_1, t_2, t_3，ノード 2 にスレッド t_4, t_5, t_6 が存在し，ページ p_1, p_2, p_3 のオーナーをノード 1，ページ p_4, p_5, p_6 のオーナーをノード 2 とする．さらに，各スレッド t_i はページ p_i に対して頻繁に write を行うというアクセス特性を持っているとする．さて，ここで新たにノード 3 を加えて計算規模を拡張し，各ノード上でのスレッド数を均等化させるために，スレッド t_1, t_2 をノード 1 に，スレッド t_3, t_4 をノード 2 に，スレッド t_5, t_6 をノード 3 にスケジュールしたとする．このときページのオーナーを移動させなければ，スレッド t_3, t_5, t_6 は，他ノードをオーナーとするページに対して頻繁に write を行うことになり，非効率である．したがって，DMI においては，オーナーを固定するのではなく，実際のアクセスパターンに従ってオーナーを移動させる方が望ましいと考えられる．しかし一方で，オーナーを移動させすぎるとオーナーの所在追跡のためのオーバーヘッドが増大することをふまえると，オーナーを移動させない方が良い場合もある．要するに，オーナーを移動させるべきかいなかは，そのページに対するアクセス特性に依存する．第二に，一度アクセスしたページをキャッシュすべきかどうかを考える．通常の PGAS では read フォルトのたびにオーナーにページを要求するが，write の頻度よりも read の頻度の方が大きい場合には，read フォルトのたびにオーナーと通信するのは非効率である．このような場合にはページをキャッシュする方が性能が向上すると考えられるが，一方で，ページをキャッシュすればキャッシュの一貫性管理のためのオーバーヘッドが増大してしまう．要するに，ページをキャッシュすべきかいなかも，そのページに対するアクセス特性に依存する．以上の議論をまとめると，大域アドレス空間の性能を引き出すためには，プログラマが明示的にページに対するアクセス特性を指示できるようにすることが重要である．具体的には，各 read/write のたびに，その read/write が read/write フォルトを引き起こした場合の挙動（オーナーを移動させるかいなか，ページをキャッシュするかいなか）を明示できる機能が必要である．DMI では，このような機能として Selective cache read/write を提案する．

また，私の知るかぎり，既存研究で大域アドレス空間に対するノードの動的な参加・脱退をサポートした PGAS は存在しないが，DMI ではそれらをサポートし，計算規模の動的な拡張・縮小を実現する必要がある．

2.2 提案手法

2.2.1 Selective cache read/write

DMI の提案する Selective cache read/write[17, 19] では，各 read に対して，その read が read フォルトを引き起こした場合の挙動として以下の 3 とおりを指定できる：

GET ページをキャッシュしない．

INVALIDATE read した後で，そのページをキャッシュする．このキャッシュは，次にこのページが更新されたときに無効化される．

UPDATE read した後で，そのページをキャッシュする．このキャッシュは，ページが更新されるたびにその更新が反映され，常に最新状態に保たれる．

実践的な使い分けとしては，各ページへのアクセス特性に応じて，近い将来にそのページを read しないのならば GET を，

比較的頻繁にそのページを read し，かつそのページに対する read の性能よりも write の性能の方が問題になる場合には INVALIDATE を，比較的頻繁にそのページを read し，かつそのページに対する write の性能よりも read の性能の方が問題になる場合には UPDATE を指定するのが良い．

また，各 write に対して，その write が write フォルトを引き起こした場合の挙動として以下の 3 とおりを指定できる：

PUT write すべきデータをオーナーに対して送信し，オーナーに write してもらう．つまりオーナーを移動しない．

EXCLUSIVE まずオーナーからオーナー権を奪って自分がオーナーになった後で，自分でデータを write する．つまりこの write を呼び出したノードにオーナーを移動する．

前述のように，オーナーを移動すればオーナー追跡のための手間が増大するため，むやみにオーナーを移動するのは好ましくない．実践的な使い分けとしては，各ページへのアクセス特性に応じて，特定の 1 つのノードがそのページに対して write しやすいという著しい傾向がある場合には，そのノードが write するときに EXCLUSIVE を，その他のノードが write するときに PUT を指定するのが良い．一方で，特定の 1 つのノードがそのページに対して write しやすいという著しい傾向がない場合には，すべてのノードが PUT を指定して write するのが良い．

さて，read/write フォルトが発生した場合には，オーナーにそれを通知しオーナーに何らかの処理を行ってもらう必要があるため，当然，各ノードは各ページのオーナーがどのノードなのかを知っている必要がある．しかし，オーナーが移動するたびに，新オーナーがどのノードなのかをすべてのノードに通知するのはあまりに非効率である．そこで，DMI では，probable owner[12] という手法を用いてオーナーを管理する．probable owner では，各ノード i は，各ページ p に関して，いずれかのノードをページ p のオーナーだと予測している．以下では，この予測しているノードを $p.prob(i)$ と表す．具体的には，各ノード i において，$p.prob(i)$ は，ノード i がページ p のオーナーから受信したメッセージのうちもっとも直近のメッセージの送信元ノードを指すように管理される．言い換えると，ページ p に関して，ノード i がオーナーからのメッセージを受信するたびに，$p.prob(i)$ の値を，そのメッセージの送信元ノード（＝そのメッセージが送信された時点でのオーナー）に更新する．たとえば，各ノード i は，read フォルトに対する応答メッセージをオーナーから受信したとき，キャッシュしているページに対する無効化要求のメッセージをオーナーから受信したときなどに，$p.prob(i)$ をそのメッセージの送信元ノードに更新する．また，ページ p のオーナーがノード i からノード j に移動する場合にも，$p.prob(i)$ の値を j に更新する．以上からわかるように，probable owner においては，ノード $p.prob(i)$ がページ p の真のオーナーであるとは限らない（のでこの手法を “probable” owner と呼ぶ）．しかし，（一時的な過渡状態を除く）任意の時刻において，すべての i に関して $i \to p.prob(i)$ の参照関係を有向グラフで描くと，この有向グラフは，真のオーナーへと参照関係が収束する有向グラフになることが保証されている．したがって，ノード i で read/write フォルトが発生した場合には，そのノード i から始めて，$i \to p.prob(i) \to p.prob(p.prob(i)) \to \ldots$ へと順次リクエストをフォワーディングすることによって，やがてオーナーにリクエストを届けることができる．以降では，この有向グラフをオーナー追跡グラフと呼ぶ．

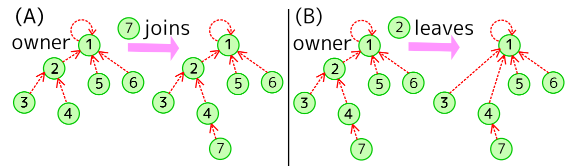

Fig.3 How to change an owner tracing graph when a node joins and leaves.

2.2.2 その他の性能最適化手法

紙面の都合上説明を省略するが，DMI では，Selective cache read/write 以外にも以下のような大域アドレス空間に対する最適化手法を導入しており，さまざまなアプリケーションを高性能に記述することができる[17, 19, 16]：

- 領域分割型の並列科学技術計算における境界点の通信を高生産に記述するための API
- データ転送の自動的な負荷分散
- 離散的なアクセスのグルーピング
- 非同期 read/write
- ユーザ定義のページサイズ
- ユーザ定義のアトミック命令

2.2.3 ノードの動的な参加・脱退への対応

DMI では，計算規模を動的に拡張・縮小する必要があるため，大域アドレス空間に対してノードを動的に参加させたり脱退させたりする必要がある．具体的には，大域アドレス空間へのノード i の参加は以下の手順で行う：

(1) ノード i は，すでに大域アドレス空間に存在しているノード j を 1 個選び，ノード j に対して参加要求を送信する．

(2) 参加要求を受信したノード j はグローバルロックを取得する．このグローバルロックは，大域アドレス空間に対するノードの参加・脱退およびメモリの確保・解放をシリアライズするものである．つまり，このグローバルロックを取得している間は，大域アドレス空間に対するノードの参加・脱退およびメモリの確保・解放が発生しないことを保証できる．

(3) ノード j は，ノード i に対して，その時点で大域アドレス空間に存在するすべてのノードとすべてのページに関するメタデータを送信する．特に，各ページ p に対する $p.prob(j)$ も送信する．

(4) ノード i は，すべてのページを無効化したうえで，各ページ p に関して，$p.prob(i)$ を $p.prob(j)$ に設定する．なお，ここでは $p.prob(i)$ を $p.prob(j)$ に設定したが，実際には $p.prob(i)$ は任意のノードに設定してもオーナー追跡グラフの正しさは維持されることに注意したい（Fig.3(A)）．以上の設定によって，ノード i が参加した後はじめてページ p に read/write したとき，read/write フォルトが発生し，$i \to p.prob(i) \to p.prob(p.prob(i)) \to \ldots$ の順にオーナー追跡が行われて，やがてその read/write フォルトが解決されることになる．

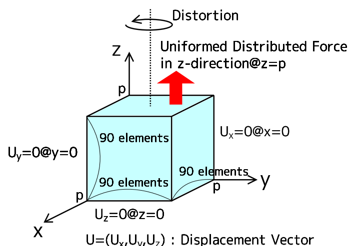

Fig.4 Stress analysis using an FEM.

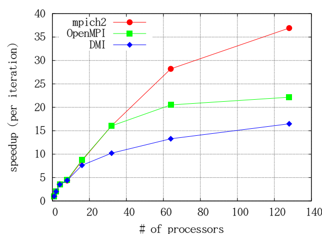

Fig.5 The scalability of an FEM.

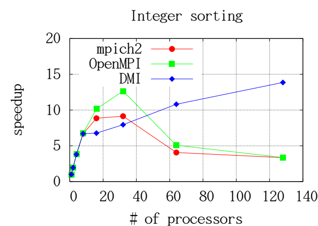

Fig.6 The scalability of integer sorting.

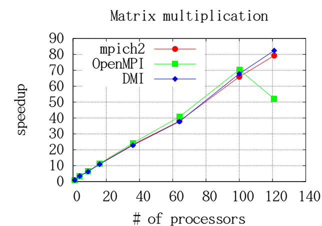

Fig.7 The scalability of matrix multiplication.

(5) ノード i は，すべてのノードに対して接続を確立して参加通知を送り，すべてのノードにノード i の参加を承認してもらう．

(6) ノード i は，グローバルロックを解放する．

一方，大域アドレス空間からのノード i の脱退は以下の手順で行う：

(1) ノード i は，グローバルロックを取得する．

(2) ノード i は，そのノード i がオーナーであるようなページすべてを他のノードに追い出す．キャッシュしているページは単に捨てれば良い．

(3) ノード i は，すべてのノードに対して脱退通知を送信する．このとき各ページ p に関する $p.prob(i)$ も送信する．これを受信した各ノード k は，$p'.prob(k) = i$ であるようなページ p' に関して，$p'.prob(k)$ の値を $p.prob(i)$ に更新する．この作業により，すべてのページのオーナー追跡グラフが，ノード i を含まないオーナー追跡グラフへと再形成される（Fig.3(B)）．すなわち，いつノード i が脱退しても，オーナー追跡を行ううえでは問題のない状態となる．

(4) ノード i は，すべてのノードから脱退通知に対する了承をもらった後で，すべてのノードとの接続を切断する．

(5) ノード i は，その時点で大域アドレス空間に存在している適当なノード j を 1 個選んで，脱退要求を送信する．

(6) 脱退要求を受信したノード j は，グローバルロックを解放する．

以上では，話を簡単化するため，ページの一貫性維持のためのプロトコルの厳密な説明を避けたが，厳密な記述に関しては論文 [17] を参照されたい．

2.3 性能評価

DMI の大域アドレス空間の性能を，高性能な分散プログラミング処理系として代表的な MPI の性能と比較する．実験環境としては，Intel Xeon E5410 2.33GHz（4 プロセッサ[⋆1]）×2 の CPU，32GB のメモリ，カーネル 2.6.18-6-amd64 の Linux で構成されるマシン 16 ノードを 1Gbit イーサネットでネットワーク接続した，合計 128 プロセッサのクラスタ環境を用いた．コンパイラとしては gcc 4.1.2，MPI としては OpenMPI 1.3.3 と mpich2-1.1.1p1，最適化オプションには-O3 を使用した．

第一の実験として，Fig.4 に示すように，3 次元立方体物体に適当な境界条件を課したときの応力解析を並列有限要素法で行った．要素数は 90^3 とした．この問題は，実際の工学に基づく難問であり，第 2 回クラスタシステム上の並列プログラミングコンテスト [3, 18] で使用されたものである．解法アルゴリズムの詳細は省略するが，この並列有限要素法では，立方体物体をプロセッサの数だけ領域分割した後，疎行列を係数行列とする連立方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を，BiCGStab 法という反復法を用いて残差 $||\boldsymbol{b} - A\boldsymbol{x}||^2/||\boldsymbol{b}||^2$ が十分に小さくなるまで反復計算する．本実験では，このコンテストで優勝した MPI のプログラムと，それと同等のアルゴリズムを DMI で記述したプログラムを性能比較した．各反復処理あたりの実行時間に関して，MPI と DMI のスケーラビリティを比較した結果を Fig.5 に示す．Fig.5 における縦軸は，（1 プロセッサで実行した場合の実行時間）/（n プロセッサで実行した場合の実行時間）である．Fig.5 を見ると，DMI のスケーラビリティが mpich2 や OpenMPI よりも劣っているのが読み取れる．この理由は，並列有限要素法のような実用的な並列科学技術計算ではプロセッサ間の負荷バランスが多少乱れるため，send/recv ベースの MPI の方が，read/write ベースの DMI よりも通信コンテンションが生じにくいためであることが，別の実験によりわかっている．

第二の実験として，512GB 個の整数要素を，サンプリング数 1280 個のランダムサンプリングソートで整列する場合のスケーラビリティを MPI と DMI で比較した結果を Fig.6 に示す．Fig.6 では，DMI は mpich2 や OpenMPI よりも高いスケーラビリティを達成している．この理由は，この処理では，サンプリング後に各プロセッサ間で整数要素を全対全交換する部分がボトルネックになっており，かつ，プロセッサ数が増えた場合における mpich2 と OpenMPI の全対全交換の性能が遅いためである．

第三の実験として，6720×6720 のサイズの行列を用いた行列行列積 $AB = C$ を，Fox のアルゴリズムで計算した場合のスケーラビリティを MPI と DMI で比較した結果を Fig.7 に示す．Fig.7 を見ると，DMI は MPI とほとんど同等のスケーラビリティを達成している．これは，Fox のアルゴリズムでは CPU の計算量に対して通信量が相対的に少ないため，通信部分における性能差がスケーラビリティに現れにくいためである．

3 スレッド移動

3.1 関連研究：従来のスレッド移動における問題点

スレッド移動 [4, 11, 10, 9, 15, 6, 8] とは，あるプロセス内で実行しているスレッドを停止させ，そのスレッドのメモリを（特に別のノードの）別のプロセスに移動させてから実行を復帰させることである．DMI の実装に照らし合わせて，以降の議論

⋆1 本稿では，マルチコアにおけるコアのことを統一してプロセッサと呼ぶ．

では以下を仮定する（スレッド移動が行われる瞬間にのみ以下の仮定が成り立てば良い）：

- 各プロセスは多数のスレッドを保持している．
- 各スレッドのメモリは，そのスレッドのレジスタとスタックとヒープから構成される．
- 各スレッドは，そのスレッドのメモリと大域アドレス空間にしかアクセスしない．つまり，他のスレッドのメモリへのアクセス，ファイル I/O，ネットワーク I/O などはない．
- C 言語におけるスレッド移動を考える．

さて，単純に移動元プロセス S 上のスレッド t を移動先プロセス D に移動させるとポインタ無効化の問題が起きる [6]．すなわち，移動先プロセス D では，空いている適当なアドレス空間 a_D にスレッド t のメモリを配置してスレッド t を復帰させることになるが，このときスレッド t のメモリに含まれるポインタは，そのメモリが移動元プロセス S のアドレス空間 a_S に配置されていることを仮定した値になっているため，スレッド t は正しく動作しない．このポインタ無効化の問題に対しては，主に 2 つの解決策が提案されている．第一の解決策は，移動先プロセス D において，スレッド t のメモリに含まれるすべてのポインタをアドレス領域 a_D に合わせて完全に正しく更新する手法 [10, 9] である．この手法では，スレッド移動時にどれがポインタなのかを可能なかぎり正確に発見するための技術が多数提案されているが，本質的に C 言語は型安全な言語ではないため，すべてのポインタを完全に発見することは不可能である．第二の解決策は，iso-address と呼ばれる方法で，アドレス空間全体をあらかじめいくつかに分割し，各スレッドが使用可能なアドレス空間を静的に決め打っておく方法 [4] である．こうすることで，あるスレッドが使用しているアドレス空間が他のいかなるスレッドによっても使用されていないことを保証できるため，スレッド移動時には，移動元と移動先で常に同一アドレスにメモリを割り当てることができる．よって，ポインタ無効化の問題は起こらない．既存のスレッド移動の研究の多くは iso-address を用いている [9, 6]．

しかし，iso-address は，計算規模がアドレス空間全体の大きさに制限されるという問題がある [9]．アドレス空間全体の大きさを w，スレッド数を n，各スレッドが使用可能なメモリの大きさを s とすると，$ns = w$ が成立している必要がある．よって，32 ビットアーキテクチャであれば $w = 2^{32}$ なので，たとえば $n = 1024$ 本ならばわずか $s =$ 4MB，$s =$ 2GB ならばわずか $n = 2$ 本であり，非現実的である．一方，近年の多くの 64 ビットアーキテクチャでは $w = 2^{47}$ のアドレス空間を利用できるため，これをもって iso-address の欠点は解消されたと見る向きもある [11, 15, 8] が，これも楽観的である．なぜなら，$w = 2^{47}$ であっても，$n = 8192$ 本ならば $s =$ 64GB，$n = 1024$ 本ならば $s =$ 512GB であり，これらの数字は 2010 年 4 月現在のマシン性能やクラスタ性能から見れば十分に現実的な数字だからである．以上より，計算規模がアドレス空間全体の大きさに制限されないスレッド移動の手法が要請されていると言える．当然，ハードウェアの進化にともなって $w = 2^{47}$ という数字は今後増える可能性もあるが，そうであっても，計算規模がアドレス空間全体の大きさに制限されないスレッド移動の手法が存在することには価値がある．DMI では，そのようなスレッド移動の手法として random-address を提案する．

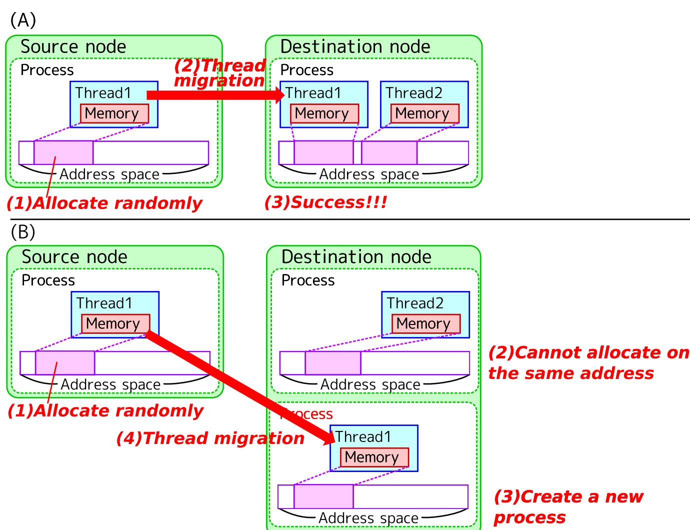

Fig.8　Algorithm of random-address.

3.2　提案手法

3.2.1　random-address

DMI の提案する random-address のアルゴリズムは以下のとおりである：

(1) 各スレッドは，他のスレッドがどのアドレスにメモリを割り当てているかに関する知識をいっさい持たない．各スレッドは，乱数を使ってメモリを割り当てるアドレスを決定する (Fig.8(A))．

(2) スレッド t の移動時に，「運が良ければ」スレッド t が移動元プロセス S において使用しているアドレスは移動先プロセス D では使用されていない．この場合には，移動先プロセス D において，スレッド t のメモリを移動元プロセス S と同一のアドレスに割り当てることで，スレッド移動を完了させる（Fig.8(A)）．

(3) スレッド t の移動時に，「運が悪ければ」スレッド t が移動元プロセス S において使用しているアドレスが移動先プロセス D でも使用されており，移動先プロセス D において，スレッド t のメモリを移動元プロセス S と同一のアドレスに割り当てることができない．この場合には，移動先プロセス D が存在するノード上に新しいプロセス D'（＝新しいアドレス空間）を生成し，新しいプロセス D' の中にスレッド t を移動させる（Fig.8(B)）．

上記の（3）では，新しいプロセス D' を生成したときに，大域アドレス空間を新しいプロセス D' に動的に拡張する必要がある．よって，この random-address は，DMI がノード（つまりプロセス）の動的な参加に対応しているからこそ可能な手法であると言える．

3.2.2　アドレス衝突確率の最小化

random-address では，スレッド移動時にアドレスが衝突する場合には新しいプロセスを生成することでスレッドを移動させるが，1 ノード内のプロセス数が増えると性能が劣化する．なぜなら，DMI の大域アドレス空間はプロセス単位で管理されており，2.2.1 節で述べたページのキャッシュなどもプロセ

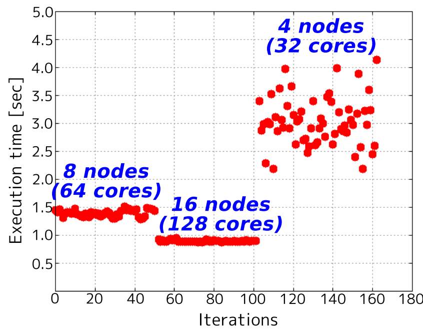

Fig.9 The execution time of each iteration when available resources are changed dynamically.

スを単位として管理されるためである．よって，あるノード内に N 本のスレッドを立てる場合，1 個のプロセス内に N 本すべてのスレッドを格納するのがもっとも望ましい．したがって，random-address においては，本当にランダムにアドレスを割り当てるのではなく，スレッド移動時にアドレスが衝突する確率を最小化するための工夫が必要である．

ここで，考えるべき問題は以下である：

- 各スレッドは，他のスレッドがどのアドレスにメモリを割り当てているかに関する知識をいっさい持たないとする．このとき，あるスレッド i をあるプロセス j の中にスレッド移動させたときのアドレス衝突確率を最小化するためには，各スレッドがどのようなアドレス割り当ての戦略を採用すれば良いか?

証明は省略するが，この問題に対する解答は以下になる：

(1) 利用可能なアドレス空間を $1, 2, ..., w$ とする．ここで，順列 $P = \{1, 2, \ldots, w\}$ に対して，ある置換 σ を作用させた順列 $Q = \{\sigma(1), \sigma(2), \ldots, \sigma(w)\}$ を 1 個構成する（全部で $w!$ 通り存在する）．

(2) 各スレッド i は $0 \le r_i < w$ なる乱数 r_i を発生させる．ここで，$r_i = \sigma(u_i)$ とする（そのような u_i は必ず存在する）．各スレッド i は，$\sigma(u_i), \sigma(u_{(i+1) \bmod w}), \sigma(u_{(i+2) \bmod w}), \ldots$ の順にアドレスを割り当てていく．

この手法のもっとも自明な置換 σ の選び方は σ を恒等置換とする場合，つまり $Q = P$ とする場合である．そしてこれは，「各スレッド i が，乱数 r_i から開始して可能なかぎり連続的にアドレスを割り当てる」戦略が，アドレス衝突確率を最小化する解の 1 つであることを意味している．以上の議論に基づき，DMI では，各スレッドの使用するアドレスが可能なかぎり連続的になるようにアドレスを割り当てる実装を施している．

3.3 性能評価

2.3 節で使用した並列有限要素法による応力解析を題材にして，利用可能な計算資源の増減に伴って計算規模を拡張・縮小する実験を行った．有限要素法における要素数は 150^3 とした．実験環境としては，Intel Xeon E5530 2.40GHz（4 プロセッサ）×2 の CPU，24GB のメモリ，カーネル 2.6.26-2-amd64 の Linux で構成されるマシン 18 ノードを 10Gbit イーサネットでネットワーク接続した，合計 144 プロセッサのクラスタ環境を用いた．この有限要素法では，連立方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を，BiCGStab 法という反復法を用いて，残差 $||\boldsymbol{b} - A\boldsymbol{x}||^2/||\boldsymbol{b}||^2$ が十分に小さくなるまで反復計算する．よって，この各反復計算の先頭に，（スレッド移動の要請があれば）協調的にスレッド移動を行うための関数 `DMI_yield()` を記述した．本実験では，128 本のスレッドを生成し，（1）最初はノード 1～ ノード 8 で実行する（合計 8 ノード，64 プロセッサ），（2）第 50 回目の反復処理終了直後にノード 9～ ノード 16 の 8 ノードを参加させる（合計 16 ノード，128 プロセッサ），（3）第 101 回目の反復処理終了直後にノード 1～ ノード 12 の 12 ノードを脱退させる（合計 4 ノード，32 プロセッサ），というように利用可能な計算資源を動的に増減させた．

各反復処理に要した処理時間を Fig.9 に示す．Fig.9 より，利用可能な計算資源の増減に対応して計算規模を動的に拡張・縮小できていることが読み取れる．合計 4 ノード（つまり 32 プロセッサ）で実行した場合の実行時間が揺れているが，これは，各プロセッサあたり 4 本のカーネルスレッドを割り当てているためカーネルによるスレッドスケジューリングがばらつくためである．本実験では，各スレッドは 510MB～520MB のメモリを消費し，大域アドレス空間は 335MB のメモリを消費した．計算規模の拡張・縮小に要した時間は，（2）における 8 ノード参加時には，8 ノードの参加処理と 120 スレッドのスレッド移動（=57.6GB のメモリ移動）が発生し，17.3 秒を要した．一方，（3）における 12 ノード脱退時には，12 ノードの脱退処理と 120 スレッドのスレッド移動（=57.6GB のメモリ移動）が発生し，30.9 秒を要した．また，これらのスレッド移動に伴うアドレス衝突は発生しなかった．以上の実験結果より，DMI が，実用的な並列科学技術計算に対して，利用可能な計算資源の増減に対応して計算規模を動的に拡張・縮小できることを示せた．

4 結論

4.1 まとめ

本稿では，クラウド環境において，大規模な並列科学技術計算を効率的に実行可能な並列分散プログラミング処理系として，Distributed Memory Interface（DMI）を提案して実装し評価した．DMI のコンセプトは以下のとおりである：

- プログラマは，（計算規模の拡張・縮小を考えることなく）十分な数のスレッドを生成するように並列プログラムを記述するだけで良い．
- あとは処理系が自動的に，それら大量のスレッドを，そのとき利用可能な計算資源にスケジューリングすることで，利用可能な計算資源の増減に対応して計算規模を動的に拡張・縮小してくれる．
- スレッド間のデータ共有レイヤーとして，高性能な大域アドレス空間を提供する．

また，DMI を実現するうえでの要素技術として，大域アドレス空間の高性能化手法，大域アドレス空間に対してノードを動的に参加・脱退させる手法，アドレス空間の大きさに制限されないスレッド移動の手法を提案した．さらに，DMI が，有限要素法による応力解析という実用的な並列科学技術計算に関して，利用可能な計算資源の増減に対応して計算規模を動的に拡

張・縮小できることを示した．

4.2 今後の課題

Amazon EC2 Spot などの実際のクラウド環境において，有限要素法や粒子法などの実用的な並列科学技術計算を題材にして，緻密な性能評価およびそれに基づく DMI の最適化を行う．具体的に最適化すべき点は以下の 3 点である．第一に，現段階で実装している DMI のスレッドスケジューラは，各ノード上で走るスレッド数を単純に均等化することしか行っていないため，スレッド移動のコストやページへのアクセスローカリティを考慮したスケジューリングを検討する必要がある．第二に，利用可能な計算資源数が少ない場合には大量のスレッドを 1 個のノードに詰め込む必要があるため，1 個のノードにプロセッサ数と等しいスレッドを生成する場合と比較すると性能が悪い．よって，この性能劣化を低減する工夫も必要である．第三に，Amazon EC2 Spot のようなクラウド環境においては，事前の予告なく仮想マシンの電源が落とされてしまうため，分散チェックポインティング&リスタートなどを用いた耐故障に対するアプローチも必要である．分散チェックポインティング&リスタートは，DMI にすでに実装しているスレッド移動の自然な延長の機能として実現できると思われる．

参考文献

1) Amazon EC2 [Online]. http://aws.amazon.com/ec2/.
2) Google App Engine [Online]. http://code.google.com /intl/appengine/.
3) 第 2 回クラスタシステム上のプログラミングコンテスト [Online]. https://www2.cc.u-tokyo.ac.jp/procon2009-2/.
4) Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System. *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pp. 496–510, 1999.
5) Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, Vol. 25, pp. 599–616, 12 2008.
6) V. Chaudhary and H.Jiang. Techniques for Migrating Computations on the Grid. *Engineering the Grid: Status and Perspective*, pp. 399–415, Jan 2006.
7) Parry Husbands, Costin Iancu, and Katherine Yelick. A Performance Analysis of the Berkeley UPC Compiler. *Proceedings of the 17th annual international conference on Supercomputing*, pp. 63–73, 2003.
8) Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, Vol. 42, No. 1, pp. 71–87, Jul 1998.
9) Hai Jiang and Vipin Chaudhary. Compile/Run-Time Support for Thread Migration. *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pp. 58–66, 2002.
10) Hai Jiang and Vipin Chaudhary. On Improving Thread Migration: Safety and Performance. *Proceedings of the 9th International Conference on High Performance Computing*, pp. 474–484, 2002.
11) K.Thitikamol and P.Keleher. Thread migration and communication minimization in DSM systems. *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, Vol. 87, pp. 487–497, 3 1999.
12) Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321–359, Nov 1989.
13) Armbrust M., A.Fox, R.Griffith, A.D.Joseph, R.Katz, A.Konwinski, G.Lee, D.A.Patterson, A.Rabkin, I.Stoica, and M.Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2 2009.
14) Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edo Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 203–231, 2006.
15) Boris Weissman, Benedict Gomes, Jurgen W.Quittek, and Michael Holtkamp. Efficient Fine-grain Thread Migration with Active Threads. *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, p. 410, 1998.

発表文献

16) Kentaro Hara and Kenjiro Taura. A Global Address Space Framework for Irregular Applications (accepted, short paper). *High Performance Distributed Computing*, Jun 2010.
17) 原健太朗, 田浦健次朗, 近山隆. DMI：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース. 情報処理学会論文誌（プログラミング）, Vol. 3, No. 1, pp. 1–40, Mar 2010.
18) 原健太朗. 有限要素法における連立方程式ソルバの並列化. 第 9 回 PC クラスタシンポジウム, Dec 2009.
19) 原健太朗, 田浦健次朗, 近山隆. DMI：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース. *SWoPP2009*, Aug 2009.