情報処理学会論文誌　プログラミング　Vol. 4　No. 1　27–65 (Mar. 2011)

アドレス空間の大きさに制限されないスレッド移動を実現するPGAS処理系

原　健太朗[†1]　中島　潤[†1]　田浦　健次朗[†1]

利用可能な計算資源が動的に増減しうる環境で長時間を要する並列計算を実行させるためには，そのとき利用可能な計算資源に対応して並列計算の規模を動的に拡張/縮小させる必要がある．しかし，有限要素法などの高性能並列科学技術計算のプログラムを，計算規模を動的に拡張/縮小できるように記述するのは難しい．そこで本稿では，「プログラマは十分な数のスレッドを生成するだけでよく，あとは処理系が透過的にそれら大量のスレッドをその時点で利用可能な計算資源に対してマッピングする」モデルに基づくPGAS処理系としてDMIを実装して評価する．特に，その要素技術としてスレッド移動手法について検討し，アドレス空間の大きさに制限されないスレッド移動手法としてrandom-addressを提案し，その最適性を証明する．評価の結果，DMIを使うことで，通常のSPMD型のプログラムをわずかに変更するだけで計算規模を動的に拡張/縮小可能な並列プログラムを記述できることを確認した．また，計算資源の増減に追随して，実用的な有限要素法アプリケーションの並列度を効果的に増減できることを確認した．

A PGAS Framework Achieving Thread Migration Unrestricted by the Address Space Size

Kentaro Hara,[†1] Jun Nakashima[†1]
and Kenjiro Taura[†1]

In order to execute a parallel computation on the environment in which available resources can change dynamically, the scale of the parallel computation should expand or shrink dynamically in response to the dynamic increase or decrease of the available resources. It is, however, difficult to program most large-scale parallel scientific computations so that they can scale up or down dynamically. Therefore, this paper implements and evaluates a PGAS framework named *DMI*, with which a programmer just has to create a sufficient number of threads because the framework transparently schedules these threads on the available resources at the time. In particular, as an elemental technique for DMI this paper discusses thread migration and proposes *random-address*, a novel thread migration method that is not restricted by the address space size. This paper also gives the proof of the optimality of this random-address method. We confirmed that in DMI we can easily program parallel computations with dynamic scale-up and scale-down by slightly modifying existing SPMD programs. Furthermore, our evaluation showed that DMI can effectively adapt the parallelism of real-world scientific computations, such as a finite element method, to the dynamic increase or decrease of resources.

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分子シミュレーションなど，計算に長時間を要するような大規模な高性能並列科学技術計算への要請が高まっている．また，これらの大規模な高性能並列科学技術計算を実行するための計算基盤として，クラウドコンピューティング（以下，クラウド）[12),42),43)] が広く注目されている．

クラウドでは，クラウドプロバイダ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．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，急激な負荷変動にも柔軟に対応しやすいため，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IaaS（Infrastructure as a Service）としてのAmazon EC2やAmazon S3[1)]，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PaaS（Platform as a Service）としてのGoogle App Engine[2)]やWindows Azure[6)]，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SaaS（Software as a Service）としてのGoogle Docs[3)]やSalesforce.comのCRM[5)]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，どのような仕組みがあれば，長時間を要するような大規模な高性能並列科学技術計算に対しても効率的な実行環境を提供できるかは自明ではない．ここでは，その仕組みについて考える．クラウドでは，クラウドプロバイダが管理する多数の計算資源を

†1 東京大学大学院情報理工学系研究科
School of Information Science and Technology, The University of Tokyo

多数の利用者に対して提供することになるが，各利用者に対する応答性や公平性を保ちつつ，かつクラウドプロバイダにおける計算資源の利用率を高めるためには，これら多数の計算資源を多数の利用者間で柔軟にスケジューリングすることが求められる．たとえば，あるクラウドを利用している利用者 A と利用者 B がいるとし，いま利用者 B に対する負荷がほとんどない状況で利用者 A に対する負荷が増大したとすると，利用者 A の計算規模を拡張することで負荷分散を図ることができる．やがて，別の利用者 B に対する負荷が利用者 A に対する負荷よりも増大したとすると，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，利用者 A と利用者 B に対する応答性と公平性を保ちつつ，全体としての負荷分散を図ることができる．このように，各利用者に対する応答性と公平性を保ちつつ計算資源の利用率を高めるためには，各利用者に対する負荷変動，全体の負荷変動，課金体系などに基づいて，計算資源を利用者間で短時間単位で柔軟にスケジューリングすることが望ましい．しかし，大規模な並列科学技術計算では 1 つの計算の単位が長時間を要するため，1 つの計算をスケジューリングの単位としていては，このような短時間単位の柔軟なスケジューリングを実現できない．したがって，長時間を要する並列計算を短時間単位のスケジューリングのもとで実行するためには，その並列計算自体が，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ような仕組みが必要であると考えられる．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，しばらくして 100 ノードが利用可能になればその 100 ノードを利用して実行されるよう，計算規模を動的に拡張/縮小できるような並列計算の仕組みが必要である．しかし，このような並列計算を記述するのは明らかに難しいため，それを容易化するような並列分散プログラミング処理系が必須であるといえる．

以上の動機に基づき，本稿では，計算規模を動的に拡張/縮小できる並列計算を簡単に記述できるような並列分散プログラミング処理系として DMI（Distributed Memory Interface）を実装して評価する．本稿では特に，その主要な要素技術としてスレッド移動について検討し，アドレス空間の大きさに制限されないスレッド移動手法として random-address を提案する．

1.2　要請される並列分散プログラミングモデル

第 1 に，計算規模を動的に拡張/縮小できる並列計算をプログラマが簡単に記述できるようにするためには，それらを処理系が透過的に実現してくれるような並列分散プログラミングモデルが必要である．すなわち，以下のような並列分散プログラミングモデルが必要である[31]：

- 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よい．
- あとは処理系が透過的に，それらの並列性を物理的な計算資源にマッピングすることで，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．

この並列分散プログラミングモデルのもとでは，プログラマは並列度の変化を意識する必要がない．よって，たとえば高性能並列科学技術計算にとって代表的な SPMD 型で並列プログラムを記述することも可能である．

第 2 に，並列分散プログラミングモデルを考えるうえで重要となるのがデータ通信モデルである．一般に，データ通信モデルを大きく分類すると，メッセージパッシングモデルと共有メモリモデルが存在する．

メッセージパッシングモデルでは，系内の各プロセスに対して一意なランク（名前）が与えられ，ユーザプログラムではランクを用いたデータの送受信を send/receive 操作として明示的に記述する．メッセージパッシングモデルの利点は，(1) ユーザプログラムに記述された操作（send/receive）が下層ハードウェアで実際に発生する操作（send/receive）にそのまま対応するため，ユーザプログラムにとって本質的に必要とされる通信以外は発生せず通信に無駄が生じない点，(2) データの所在管理や通信形態の決定に関してユーザプログラム側に自由度があるため，明示的なチューニングが行いやすく性能を引き出しやすい点などである．一方で，メッセージパッシングモデルの欠点は，ユーザプログラム側でデータの所在や通信形態を管理しなければならないため，動的で不規則なデータ構造を取り扱うような非定型な処理を非常に記述しにくい点などである．まとめると，メッセージパッシングモデルは，性能を引き出しやすい一方でプログラマビリティが低いデータ通信モデルといえる．メッセージパッシングモデルを採用する代表的な処理系には MPI [25),31),38),44),52),53),56)] がある．

一方，共有メモリモデルでは，物理的には分散した計算資源上に処理系が仮想的な共有メモリを構築することによって，ユーザプログラム側からはあたかも通常の共有メモリ環境と同様の read/write 操作によってデータ通信を実現することができる．共有メモリモデルの利点は，通常の共有メモリ環境上の並列プログラムと同様の read/write ベースの記述が可能なため，プログラマは各ノード上のデータ配置や煩雑なメッセージ通信を意識することなく並列アルゴリズムの開発に専念できる点である．一方で，共有メモリモデルの欠点は，

(1) ユーザプログラムに記述された操作（read/write）と下層ハードウェアで実際に発生する動作（send/receive）が対応しないために，ユーザプログラムから処理系の挙動を把握しにくく明示的なチューニングを施しにくい点，(2) ユーザプログラムにとって本来必要な通信パターンが何なのかを処理系側で把握しにくいために，無駄なメッセージ通信が多量に発生する可能性が高い点などである．まとめると，共有メモリモデルは，プログラマビリティが高い一方で性能を引き出しにくいデータ通信モデルといえる．共有メモリモデルを採用する代表的な処理系には，TreadMarks [8]，DSM-Threads [46),47),51)]，SMS [68)]，CRL [41)]，UPC [11),14),18),21)]，Titanium [23),30),54),55),59),60)]，Co-Array Fortran [20),21),49),58)]，Global Arrays [27),48)] などがある．

以上の特徴をふまえて，並列計算を簡単に記述させることを目標とする DMI では，データ通信モデルとして，プログラミングが容易な共有メモリモデルを採用する．共有メモリモデルをさらに細かく分類すると，分散共有メモリモデル[8),41),46),47),51),68)]，ローカルビュー型の PGAS（Partitioned Global Address Space）モデル[20),21),23),30),49),54),55),58)–60)]，グローバルビュー型の PGAS モデル[11),14),18),21),27),48)] などさまざまなものがあるが，DMI では，read/write に基づくプログラミングの容易さと性能をバランスさせるため，グローバルビュー型の PGAS モデルを採用する．グローバルビュー型の PGAS モデルでは，大域アドレス空間に対する read/write 操作に基づく簡単なプログラミングが行えるうえ，ユーザプログラムからデータの分散配置を明示的に指示できるようになっており，リモートとローカルを明示的に区別できるため，性能のチューニングを行いやすい．グローバルビュー型の PGAS モデルに基づく処理系としては，UPC，Global Arrays などが代表的である．

以上の観察に基づき，本稿では，本節冒頭で述べた並列分散プログラミングモデルをグローバルビュー型の PGAS モデルで実現する並列分散プログラミング処理系として DMI を実装して評価する．DMI のコンセプトは以下のとおりである（**図 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よい．
- あとは処理系が透過的に，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は，グローバルビュー型の大域アドレス空間に対するアクセスを通じて簡単に実現できる．

このような DMI を実現するためには，以下の 3 つの要素技術が必須である：

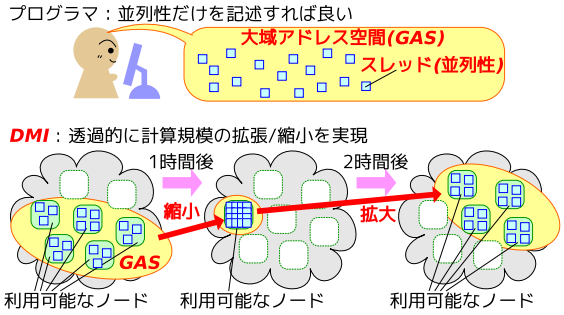

図 1　DMI のコンセプト
Fig. 1　The concept of DMI.

(1) 高性能な並列科学技術計算をサポートするためには，大域アドレス空間に対するアクセスが高性能に行えなければならない．よって，大域アドレス空間に対するアクセスを高性能化する技術．

(2) 計算規模を拡張/縮小するためには，大域アドレス空間のコンシステンシ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．よって，ノードの動的な参加/脱退を越えて大域アドレス空間のコンシステンシを維持する技術．

(3) 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，実行中のスレッドをノード間で移動しなければならない．よって，実行中のスレッドをノード間で安全に移動する技術．

このうち，(1) と (2) に関しては著者らの研究[29),65),66)] を参照されたい．本稿では，(3) のスレッド移動の要素技術について論じる．特に，2.3 節で説明するように，既存のスレッド移動手法[9),10),38),45)] では，生成できるスレッドの本数や各スレッドが使用できるメモリ量がアドレス空間の大きさに制限されるのに対して，本稿では，それらがアドレス空間の大きさに制限されないスレッド移動手法として random-address を提案する．

なお，本稿で提案するスレッド移動手法は，ホモジニアスな Linux 環境を前提にする．実際のクラウドではヘテロジニアスな環境が多いと思われるが，その場合には，仮想マシンを利用してホモジニアスな Linux 環境を作り出すことで DMI を利用できるようになる．

1.3　本稿の構成

2 章では，関連研究について述べ，特に既存のスレッド移動手法の問題点について言及する．3 章では，DMI のシステムを概観する．4 章では，DMI のプログラムのアウトライ

ンを説明する．5 章では，スレッド移動にともなうプログラミング制約について説明する．6 章では，random-address に基づいたアドレス空間管理とスレッド移動手法を説明する．7 章では，DMI におけるスレッド移動の実装について説明する．8 章では，シミュレーションによる random-address の評価とアプリケーションベンチマークによる DMI の評価を行う．9 章では，結論と今後の課題を述べる．付録 A.1 では，random-address のアルゴリズムの最適性を証明する．

2. 関 連 研 究

2.1 クラウドサービスにおける計算規模の拡張/縮小

まず，既存のクラウドサービスにおいて，並列科学技術計算の計算規模を拡張/縮小させる場合の問題点を観察する．計算規模の拡張/縮小を考えるうえでは，何を粒度として拡張/縮小するかが重要になる．粒度としては，大きい方から順に，仮想マシン，プロセス，スレッドがある[16),17)]．

第 1 に，仮想マシンを粒度とする場合には，仮想マシンを起動/停止/移動することで計算規模の拡張/縮小を実現する．仮想マシンを粒度とするクラウドサービスとしては，Amazon EC2，Windows Azure などがあり，利用者は必要なときに必要な量だけ仮想マシンを利用することができる．これらのクラウドサービスの利点は，利用者に対して仮想マシンという汎用的な計算環境が提供されるため，利用者にとっての自由度が大きく，実行可能なアプリケーション領域が広いという点である．一方で，第 1 の欠点は，課金の粒度が CPU の使用時間などではなく仮想マシンの起動時間に基づいて行われてしまう点である．これは，仮想マシンは存在しているだけで無視できない量のメモリ資源を消費することに起因している．第 2 の欠点は，仮想マシンは 1 つの OS として動作するためメモリ消費量が多く，起動/停止/移動には数分を要するので，計算規模の拡張/縮小の要求に対する応答性が悪い点である．たとえば，これらのクラウドサービスが仮想マシンのライブマイグレーション[19),50)]の技術をサポートすることにより，仮想マシンの移動時間を削減して応答性を改善することは可能かもしれないが，いずれにせよ，仮想マシンが使用しているメモリ全体を移動させる必要があることには変わりない．DMI が対象とするような並列科学技術計算にとっては，その並列科学技術計算が使用しているメモリだけが確保/解放/移動されれば十分な場合が多く，OS などを含めた仮想マシンの実行環境すべてが起動/停止/移動されることは要求されておらず，仮想マシン粒度での計算規模の拡張/縮小は不必要に重すぎる場合が多い．

第 2 に，プロセスを粒度とする場合には，プロセスを生成/破棄することで計算規模の拡張/縮小/移動が実現される．プロセスを粒度とするクラウドサービスとしては，Google App Engine などが代表的である．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 Web 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，計算規模が透過的に拡張/縮小され，利用者が何の意識を払わなくても負荷分散が図られる．Google App Engine の第 1 の利点は，計算規模の拡張/縮小の要求に対する応答性の良さである．たとえば，2010 年 7 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．この応答性の良さは，プロセスが消費するメモリ量は仮想マシンより少なく，生成/破棄などの取扱いを高速に実現できることに起因している．第 2 の利点は，実際の CPU 使用時間に基づいた細粒度な課金を行える点である．これも，プロセスのメモリ消費量が少なく，取扱いが軽量であることに起因している．一方で，第 1 の欠点は，Google App Engine は Web アプリケーションに特化した作りになっており，高性能並列科学技術計算のようにプロセスどうしが密に結合して動作するアプリケーションには向いていない点である．たとえば，Google App Engine では，プロセス間のデータ共有は BigTable[15)] というデータベース経由で行われるが，このようなデータベースを利用して，高性能並列科学技術計算に要求されるようなプロセスどうしの密で複雑なデータ共有を効率的に実現することは難しいと考えられる．第 2 の欠点は，短時間で終了するアプリケーションしか実行できない点である．Google App Engine の場合，各プロセスの処理は 30 秒以内に終了させなければならない．しかし，有限要素法などの並列科学技術計算を各計算部分が短時間で終了するように分割して記述することは困難である．

以上のように，既存のクラウドサービスを利用して，並列科学技術計算の計算規模を透過的かつ効率的に拡張/縮小させるのは難しい．

2.2 並列科学技術計算におけるプロセス/スレッド移動

前節で指摘したように，並列科学技術計算の計算規模を拡張/縮小させるためには，プロセス/スレッドの粒度が適している．そこで本節では，クラウドサービスへの応用性は指摘されていないものの，それらに自然に応用可能だと思われる要素技術として，並列科学技術計算におけるプロセス/スレッド移動に関する既存研究を観察する．

まず，BLCR[26)] は，Linux のプロセスをチェックポイント/リスタートするためのカーネルモジュールである．BLCR を用いて，あるノードでチェックポイントしたプロセスを別のノードでリスタートさせることで，ノード間のプロセス移動を実現できる．ただし，プロセス移動を通じて IP アドレスなどの情報を透過的に移動させることは難しいため，BLCR

では TCP/UDP ソケットのチェックポイント/リスタートには対応していない．そこで，研究 52) では，BLCR に対して，MPI プロセスのチェックポイント時に on the fly な MPI メッセージをすべて回収し，リスタート時にそれらを復旧させる機能を付け加えることで，MPI プロセスのプロセス移動を実現している．また，MPI-Mitten[25] では，プロセス移動を越えて MPI における集合通信をサポートするための分散アルゴリズムが提案されている．さらに，Tern[38] や Adaptive MPI[31] では，MPI の各インスタンスをプロセスではなくスレッドとして実装することで，プロセス移動よりも細粒度なスレッド移動を実現している．

このような MPI におけるプロセス/スレッド移動の要素技術は，さまざまな分野に応用されている．第 1 の応用は耐故障な並列計算の実現である．MPI プロセスのチェックポイント/リスタートを行うことで，MPI を用いた耐故障な並列計算を実現できる．さらに，研究 56) では，故障しそうなノード上の MPI プロセスを，故障が起きる前に健康なノードへとプロセス移動させることによって，耐故障性確保のためのチェックポイントの回数を削減している．第 2 の応用は動的な環境における並列計算の負荷分散である．クラウドにかぎらず，多数の利用者が共同利用する時分割方式のクラスタ環境などでは，利用可能な計算資源やその性能が動的に変化するため，その動的な変化に追随して並列計算を適応させることが重要になる．MPI Process Swapping[53] では，n 個のプロセッサを利用する MPI のプログラムに対してあらかじめ $n+m$ 個のプロセッサを割り当てておき，計算資源の性能の動的な変化に追随して，各時点で最も高性能な n 個のプロセッサを選択して実行することを提案している．また，Adaptive MPI[31] では，DMI と同様に，「プログラマは十分な数のスレッドを生成するだけでよく，あとは処理系が透過的にそれら大量のスレッドを物理的な計算資源にマッピングする」ことで動的な負荷分散を行う処理系を実現している．

しかし，以上で述べたようなプロセス/スレッド移動およびその各種応用は，すべてメッセージパッシングモデルに基づくものである．著者らの知るかぎり，DMI のように，共有メモリモデルに基づいて，プログラマから透過的に計算規模の拡張/縮小を実現した研究は存在しない．

2.3　既存のスレッド移動手法とその問題点

最後に，スレッド移動に関する既存研究を観察する．

スレッド移動[9),10),16),22),24),32)–37),39),45),57),61)–63)] 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．このとき，各スレッドのスタック領域やヒープ領域などのメモリ領域をプロセス間で移動させることになるが，これらのメモリ領域にはそのメモリ領域自身へのポインタが含まれている可能性がある．よって，移動先プロセスにおいて，単純に適当なアドレスにスレッドのメモリ領域を割り当ててしまうと，ポインタが無効化してしまい，プログラムの正しい実行を保証できなくなる．この問題に対しては，主に 2 つの解決策が提案されている．

第 1 の解決策は，移動元プロセスと移動先プロセスとで異なるアドレスにメモリ領域を配置することを許すかわりに，スレッド移動の直後に，スレッドのメモリ領域に含まれるすべてのポインタを，移動先プロセスのアドレス領域に合わせて完全に正しく更新する手法[22),33)–36)] 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処理系が完全に把握する必要があるため，どれがポインタなのかをプログラマに明示的に指定させたり，データフロー解析などのコンパイラ的手法を用いてポインタを自動的に発見したりする．しかし，前者の方法はプログラミングの負担を増大させるという問題があり，後者の方法は，本質的に C 言語は型安全な言語ではないのですべてのポインタを自動的に完全に発見することはできないという問題がある．

第 2 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ておき，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[9),10),45)] である．これにより，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つねに保証できるため，スレッド移動時には，移動元プロセスと移動先プロセスとでつねに同一アドレスにメモリを割り当てることができる．既存のスレッド移動の研究の多くは iso-address を用いている[16),33)]．しかし，iso-address では，計算規模がアドレス空間全体の大きさに制限されてしまうという問題がある．アドレス空間全体の大きさを w バイト，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 バイトとすると，iso-address では $ns = w$ 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 $n = 1024$ 個のスレッドを生成するならば各スレッドが使用できるメモリ量はわずか $s = 4\,\mathrm{MB}$ であり，各スレッドが $s = 2\,\mathrm{GB}$ のメモリ量を使用するならばスレッドはわずか $n = 2$ 個しか生成できず，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（CPU の実装に依存するが）$w = 2^{47}$ 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[32),39),57)] が，これも楽観的である．なぜなら，$w = 2^{47}$ であっても，$n = 8192$ 個ならば $s = 64\,\mathrm{GB}$，$s = 512\,\mathrm{GB}$ ならば $n = 1024$ 個であり，これらの数字は 2010 年 7 月現在のクラスタ規模や各ノードの搭載メモリ量から見れば十分に現実的な数字だからである．以上の考察より，今後ますます増大する計算規模に対応するためには，iso-address では不

十分であり，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手法が要請されているといえる．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手法が存在することには価値がある．そこで DMI では，そのようなスレッド移動手法として random-address を提案する（6 章）．

3. 基本設計

本章では，DMI の基本設計について概観する．詳細に関しては著者らの論文[29),65),66)] を参照されたい．

3.1 DMI の概要

DMI のシステム構成を**図 2** に示す．DMI では各ノード上に任意個のプロセスを生成し，各プロセス内に任意個のスレッドを生成することができる．ただし，性能上は，1 ノードあたり 1 個のプロセスを，1 プロセッサあたり 1 個のスレッドを生成するのが望ましい．

DMI における各プロセスは，メモリプールと呼ばれる一定量のメモリを DMI に対して提供する．このメモリプールの量は各プロセスを生成するときに指定できる．すると，DMI はこれらのメモリプールをメモリ資源として，ページテーブルなどのメモリ管理機構をユーザレベルで実装することによって，分散環境上に大域アドレス空間を構築する．これにより，各スレッドは，大域アドレス空間に対する read/write を通じて，すべてのプロセスが提供するメモリプールに透過的にアクセスすることができる．このとき，アクセス対象のデータがそのプロセスのメモリプールに存在しない場合には，ページフォルトが発生して，そのページを持っているプロセスからページが転送される．さらに，必要であれば，転送されてきたページをそのプロセスのメモリプールにキャッシュすることができる．このように

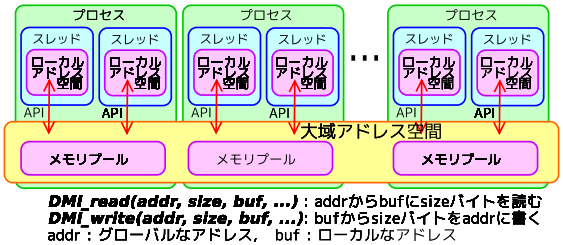

図 2　DMI のシステム構成
Fig. 2　The system architecture of DMI.

DMI では，Co-Array Fortran や UPC などの get/put 操作に基づく PGAS 処理系とは異なり，大域アドレス空間のデータをキャッシュすることができる．また，DMI では同一プロセス内の複数のスレッドがメモリプールを共有キャッシュ的に利用する構成となっており，同一プロセス内のスレッド間のデータ共有は物理的な共有メモリ経由で実現される．すなわち，DMI では，分散レベルの並列性とマルチコアレベルの並列性を統合的に活用したハイブリッドプログラミングを透過的に実現している．さらに，多数のプロセスを利用することで巨大な大域アドレス空間を構築し，DMI を遠隔スワップシステムとして動作させることもできる．ページングを繰り返すうちにメモリプールの使用量が指定量を超えてしまう場合があるが，その場合には，ページ置換アルゴリズムに基づいて他のプロセスへのページアウトが行われる．

DMI は C 言語の静的ライブラリとして実装されており，コンパイラや OS にはいっさい手を加えていないため移植性が高い．DMI の処理系は約 22000 行からなる C 言語で実装されており，86 個の API を提供している．具体的な API としては，メモリ確保・解放・read/write のための基本的な API のほかに，排他制御のための API，ユーザ定義のアトミック命令を作り出す API，非同期な read/write のための API，大域アドレス空間上の離散的な領域に対するアクセスを集約する API，非定型な領域分割に基づく領域間通信をグローバルビューで簡単に記述できる API などを提供しており，多くの高性能な並列科学技術計算を高生産に記述することができる．

3.2 大域アドレス空間に対するアクセス

図 2 に示すように，DMI では，各スレッドのローカルアドレス空間と大域アドレス空間を明確に分離している．ローカルアドレス空間は通常の共有メモリであり，`mmap()` 関数/`munmap()` 関数を通じて確保/解放し，通常の変数参照や配列参照などによって read/write できる．一方，大域アドレス空間は `DMI_mmap()` 関数/`DMI_munmap()` 関数によって確保/解放し，`DMI_read()` 関数/`DMI_write()` 関数によって read/write する．本節ではこれらの API について詳しく述べる．

DMI では，region-based[41)] な分散共有メモリ処理系と同様に，ユーザプログラムの振舞いに合致した任意のコンシステンシ粒度を指定して大域アドレス空間を確保することができる．DMI では，コンシステンシ粒度のことをページ，そのサイズをページサイズと呼ぶ．具体的には，`DMI_mmap(int64_t *addr,int64_t page_size,int64_t page_num,...)` 関数を呼び出すことによって，ページサイズが `page_size` のページを `page_num` 個持つ大域アドレス空間を確保できる．つまり，任意のブロックサイズに基づくブロックサイクリック

なデータ分散を指定することができる．たとえば，巨大な行列行列積をブロック分割によって並列に行いたい場合には，各行列ブロックの大きさをページサイズに指定して大域アドレス空間を割り当てればよい．このように，DMI ではコンシステンシ粒度を明示的に調節することでデータ転送の単位をユーザプログラムにとって必要十分な大きさまで巨大化させられるため，OS のメモリ保護機構を利用する分散共有メモリ処理系や PGAS 処理系と比較すると，ページフォルトの回数を大幅に抑制でき，効率的な通信を実現できる．

大域アドレス空間に対して read/write するには，`DMI_read(int64_t addr,int64_t size,void *buf,...)` 関数/`DMI_write(int64_t addr,int64_t size,void *buf,...)` 関数を使う．`DMI_read(int64_t addr,int64_t size,void *buf,...)` 関数は，大域アドレス空間上のアドレス `addr` から `size` バイトをローカルアドレス空間上のアドレス `buf` に read する．一方で，`DMI_write(int64_t addr,int64_t size,void *buf,...)` 関数は，ローカルアドレス空間上のアドレス `buf` から `size` バイトを大域アドレス空間上のアドレス `addr` に write する．DMI では，アドレス領域 [`addr`, `addr` + `size`) が 1 ページに収まるような `DMI_read()` 関数/`DMI_write()` 関数に対する Sequential Consistency を保証しており，直観的に理解しやすいコンシステンシモデルのもとで並列プログラムを記述できる．複数のページにまたがって `DMI_read()` 関数/`DMI_write()` 関数を呼び出すことも可能であるが，この場合には，要求されたアドレス領域全体がページ単位のアドレス領域に内部で分割され，その分割された各アドレス領域に対して独立に `DMI_read()` 関数/`DMI_write()` 関数が呼び出されるのと同様の結果になる．よって，複数のページにまたがる場合には，必要に応じて排他制御を行う必要がある．さらに DMI では，非同期な `DMI_read()` 関数/`DMI_write()` 関数もサポートしており，ユーザプログラムが要求するコンシステンシ強度に応じて，DMI によって保証されている Sequential Consistency を明示的に緩和させることもできる．

DMI では，大域アドレス空間上のアクセスローカリティを最適化するための強力な手段を提供している．DMI において read フォルトが発生するのは，そのプロセスがそのページのキャッシュを保持していない場合であり，write フォルトが発生するのは，そのプロセスがオーナでないか，またはそのプロセス以外にページのキャッシュを保持しているプロセスが存在する場合である．ここでオーナとは，各ページごとに 1 個ずつ存在する，そのページの最新状態とコヒーレンシの管理を担当するプロセスのことである．DMI では，各 read/write ごとに，その read/write が read/write フォルトを引き起こしたときに，どのようにページを転送するのか，転送されたページをどのようにキャッシュするのか，オーナをどのように移動するのかを明示的に指示することができ，これによってユーザプログラムのアクセスローカリティを最適化することができる．具体的には，`DMI_read(int64_t addr,int64_t size,void *buf,int mode)` 関数の引数 `mode` として次の 3 通りを指定できる：

INVALIDATE　オーナからページを取得したあとでメモリプールにキャッシュする．このキャッシュはそのページが次に更新された際に無効化される．

UPDATE　オーナからページを取得したあとでメモリプールにキャッシュする．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，つねに最新状態に保たれる．

GET　ページ全体ではなく，この read によって要求された部分のデータのみをオーナから取得する．メモリプールには何もキャッシュしない．

また，`DMI_write(int64_t addr,int64_t size,void *buf,int mode)` 関数の引数 `mode` として次の 2 通りを指定できる：

EXCLUSIVE　まず現在のオーナからオーナ権を奪って自分がオーナになったあとで，自分でデータを write する．つまりこの write を呼び出したプロセスにオーナを移動する．

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

計算規模の増減にともなってスレッドが移動すると各スレッドのアクセスローカリティが変化してしまうが，read ローカリティが高いデータに関しては，INVALIDATE あるいは UPDATE を指定することで，スレッドの動的な移動に追随して read ローカリティを適応させることができる．また，write ローカリティが高いデータに関しては，EXCLUSIVE を利用することで write ローカリティを適応させることができる．逆にアクセスローカリティのないデータに関しては，GET や PUT を利用することで，キャッシュコヒーレンシ管理のためのオーバヘッドや余分なページ転送を省くことができる．

3.3　プロセスの動的な参加/脱退

DMI では，プロセスの動的な参加/脱退を越えて大域アドレス空間のコンシステンシを維持するプロトコルを実装しており，並列計算を実行中に動的にプロセス（ノード）を参加/脱退させることができる．プログラミングインタフェースとしては，参加/脱退しようとしているプロセスの存在をポーリングする API，それらの参加/脱退を承認する API，スレッドを生成/破棄する API などが提供されており，参加プロセスに対してスレッドを生成したり脱退プロセスからスレッドを破棄したりすることで，プロセス（ノード）の動的な参加/脱退に対応して計算規模を動的に拡張/縮小させることができる．いい換えると，DMI では，計算規模を拡張/縮小させるためには，プログラマがスレッドの生成/破棄を通じて明示的にスレッド数を増減させる必要がある．

一般に，タスクパラレルなアルゴリズムでは，並列計算を実行中に明示的にスレッド数を増減させるのが比較的容易である．なぜなら，タスクパラレルなアルゴリズムでは，アルゴリズム上の論理的な並列度と実際の物理的な並列度とが概念的に分離されているため，アルゴリズム上はいつ何個のスレッドが参加していてもよく，各タスクの粒度において自由なタイミングでスレッドを増減させられるからである．一方で，SPMD 型のアルゴリズムでは，並列計算を実行中に明示的にスレッド数を増減させるのが困難である．なぜなら，SPMD 型のアルゴリズムでは，アルゴリズム上の論理的な並列度と実際の物理的な並列度が一致しているからである．SPMD 型のアルゴリズムでは，すべてのスレッドが同期的に動作するよう記述されるため，並列計算の途中でスレッド数を増減させるためには，(1) すべてのスレッドが同期をとり，(2) スレッド数を増減させ，(3) 新たなスレッド数において各スレッドの担当範囲を分割し直す，という手順を行う必要があるからである．これはプログラミング上の負担を増大させるだけでなく，担当範囲の再分割に時間がかかる場合には性能上の問題も引き起こす．たとえば，8.2.2 項に示す有限要素法においては，担当範囲の分割は物体全体の領域分割に相当するが，この領域分割は計算時間を要する処理である．まとめると，現状の DMI では，計算規模を拡張/縮小させるためにはプログラマがスレッドを明示的に生成/破棄する必要があり，特に並列科学技術計算に要求される SPMD 型のアルゴリズムに対しては，プログラマビリティと性能の両面において問題がある．

以上をふまえて，本稿では，現状の DMI に対してスレッド移動の技術を付け加え，1.2 節で述べた並列分散プログラミングモデルに基づき，計算規模を動的に拡張/縮小できる並列プログラムをより簡単に記述できるプログラミングインタフェースを整備する．

4. プログラムのアウトライン

1.2 節で述べたように，DMI では，プログラマは十分な数のスレッドを生成するだけでよく，あとは処理系が透過的に，それらのスレッドをそのとき利用可能な計算資源に対して動的にマッピングしてくれる．DMI におけるプログラムのアウトラインを**図 3** に示す．図 3 において，DMI が起動されると `DMI_main()` 関数が実行される．そして，`DMI_scheduler_init()` 関数を呼び出して DMI のスレッドスケジューラを初期化したあと，`DMI_scheduler_create()` 関数を呼び出すことで任意個のスレッドを生成できる．生成されたスレッドは `DMI_thread()` 関数として実行され始める．また，終了したスレッドを `DMI_scheduler_join()` 関数で回収したり，`DMI_scheduler_detach()` 関数でデタッチしたりできる．

ここで，スレッドスケジューリングは DMI によって透過的に行われるが，スレッドスケジューリングのために必要となるスレッド移動はプリエンプティブではなく協調的に行われる．具体的には，DMI によってスレッド移動が必要であると判断された任意のタイミングでスレッド移動が起きるわけではなく，それ以降で，移動対象のスレッドが初めて `DMI_yield()` 関数を呼び出した時点で，その `DMI_yield()` 関数の内部で協調的なスレッド移動が起きる．この `DMI_yield()` 関数は，DMI によってスレッド移動の指示が届いていなければ何も行わずにすぐに返り，スレッド移動の指示が届いていれば内部でスレッド移動を行って移動先の

```
typedef struct arg_t {
  ...; /* 各スレッドに渡す引数 */
}arg_t;

void DMI_main(int argc, char **argv)
{
  arg_t arg;
  int rank, pnum;
  int64_t sched_addr, arg_addr, handle[THREAD_MAX];
  ...;
  pnum = atoi(argv[1]); /* スレッド数 */
  DMI_mmap(&sched_addr, sizeof(DMI_scheduler_t), 1); /* スレッドスケジューラを管理する大域アドレス空間を確保 */
  DMI_mmap(&arg_addr, sizeof(arg_t) * pnum, 1); /* 各スレッドへの引数を格納する大域アドレス空間を確保 */
  for(rank = 0; rank < pnum; rank++) {
    arg = ...; /* 各スレッドに渡す引数を設定 */
    DMI_write(arg_addr + rank * sizeof(arg_t), sizeof(arg_t), &arg, EXCLUSIVE); /* 大域アドレス空間に書き込む */
  }
  DMI_scheduler_init(sched_addr); /* スレッドスケジューラを初期化 */
  for(rank = 0; rank < pnum; rank++) { /* スレッドを生成 */
    DMI_scheduler_create(sched_addr, &handle[rank], arg_addr + rank * sizeof(arg_t));
  }
  for(rank = 0; rank < pnum; rank++) { /* スレッドを回収 */
    DMI_scheduler_join(sched_addr, handle[rank]);
  }
  DMI_scheduler_destroy(sched_addr); /* スレッドスケジューラを破棄 */
  DMI_munmap(sched_addr);
  DMI_munmap(arg_addr);
  ...;
}

int64_t DMI_thread(int64_t arg_addr) /* 各スレッドの処理 */
{
  arg_t arg;
  int iter;
  DMI_read(arg_addr, sizeof(arg_t), &arg, GET); /* 大域アドレス空間から引数を読み込む */
  for(iter = 0; iter < ITER_MAX; iter++) {
    DMI_yield(); /* スレッドスケジューラにスケジューリングのチャンスを与える */
    ...; /* 各イテレーションの処理 */
  }
}
```

図 3　DMI のプログラムのアウトライン
Fig. 3　The outline of a DMI program.

プロセスで返る．したがって，外的な計算資源の変化に追随してスレッド移動を応答性良く行うためには，プログラマは，移動される可能性のあるスレッドがある程度短い間隔で DMI_yield() 関数を呼び出すようにプログラムを記述しておく必要がある．反復計算を行うプログラムであれば，たとえば図 3 のように，各イテレーションの先頭に DMI_yield() 関数を記述すればよい．

このように DMI では，通常の共有メモリ環境におけるスレッド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によって，計算規模を拡張/縮小可能な並列プログラムを簡単に記述できる．

5. プログラミング制約

5.1 制約が必要になる理由

DMI のように，1 個のプロセス内に多数のスレッドが存在するマルチスレッド型のモデルにおいて，各スレッドを粒度としたスレッド移動を行うためには，各スレッドの使用するアドレス領域が独立している必要がある．たとえば，プロセス p の中にスレッド i とスレッド j があり，スレッド i はスレッド j のスタック領域のどこかへのポインタ d を使用しているとする．このとき，スレッド j だけを別のプロセス q にスレッド移動させたとすると，スレッド i が使用しているポインタ d が無効化されてしまい，実行を正しく継続できなくなる．別の例として，プロセス p 内のスレッド i とスレッド j が，プロセス p のグローバル変数 g を使用している状況で，スレッド j だけを別のプロセス q に移動させることを考える．この場合には，グローバル変数 g をプロセス q に移動させてもさせなくても，スレッド i とスレッド j のいずれか一方の実行を正しく継続できなくなる．また，そもそも，プロセス q にもグローバル変数 g がすでに存在しているとすれば，プロセス p のグローバル変数 g をプロセス q へ移動することによって，プロセス q のグローバル変数 g の値が書きつぶされてしまう．さらに，ローカルなファイル IO などはスレッド移動を越えてサポートできない．以上から分かるように，各スレッドを粒度としたスレッド移動を安全に行うためには，C 言語で記述できることすべてをサポートできるわけではなく，アドレス領域の使用に何らかの制約を加える必要がある．よって本章では，まず 5.2 節でアドレス領域をモデル化したうえで，そのモデルに基づいて 5.3 節でプログラミング制約を記述する．さらに，5.5 節でそのプログラミング制約を緩和する．

5.2 アドレス領域のモデル化

まず，アドレス領域をモデル化する．DMI では，アドレス領域全体を以下の 6 種類に分類して扱う（**図 4**）：

$register_i^p$　プロセス p 内のスレッド i のレジスタ領域．$register_i^p$ はマシンによってスレッドローカルに管理される．

$stack_i^p$　プロセス p 内のスレッド i のスタック領域．$stack_i^p$ はマシンによってスレッドローカルに管理される．

$threadheap_i^p$　プロセス p 内のスレッド i のヒープ領域．$threadheap_i^p$ の定義は，プロセス p 内のスレッド i が DMI_thread_mmap() 関数/DMI_thread_mremap() 関数/DMI_thread_munmap() 関数を呼び出すことで確保/解放されるアドレス領域である．$threadheap_i^p$ は DMI によってスレッドローカルに管理される．DMI_thread_mmap() 関数および DMI_thread_mremap() 関数は，返り値として通常のポインタを返すので，このアドレス領域は通常のメモリアクセスによって使用できる[⋆1]．

$static^p$　プロセス p の静的変数領域．$static^p$ はマシンによってプロセスローカルに管理される．

$processheap^p$　プロセス p のヒープ領域．$processheap^p$ の定義は，プロセス p に含まれるいずれかのスレッドが（malloc() 関数/free() 関数などを経由して）システムコールの mmap() 関数/mremap() 関数/munmap() 関数を呼び出すことで確保/解放されるアドレス領域である．$processheap^p$ はマシンによってプロセスローカルに管理される．

dmi　DMI が実現する大域アドレス空間領域．dmi の定義は，DMI_mmap() 関数/DMI_munmap() 関数を呼び出すことで確保/解放されるアドレス領域である．dmi は DMI によって全プロセスで共有されるように管理される．

5.3 プログラミング制約

以上で述べたアドレス領域のモデルに基づき，プログラミング制約について述べる．4 章で述べたように，DMI では，ユーザプログラムに DMI_yield() 関数を呼び出してもらうことで協調的なスレッド移動を行う．このとき，この DMI_yield() 関数に関して以下のプログラミング制約が守られる必要がある：

プログラミング制約　プロセス p 内のスレッド i が DMI_yield() 関数を呼び出す時点において，そのスレッド i の実行を正しく継続するために必要なすべてのデータは，$register_i^p$，

⋆1 なお，プロセス p 内のスレッド i が DMI_thread_mmap() 関数によって確保したアドレス領域を，プロセス p 内の別のスレッド j が DMI_thread_munmap() 関数で解放することはできない．また，プログラミングの便宜のため，DMI_thread_mmap() 関数/DMI_thread_munmap() 関数/DMI_thread_mremap() 関数の上位関数として，DMI_thread_malloc() 関数/DMI_thread_free() 関数/DMI_thread_realloc() 関数を提供している．

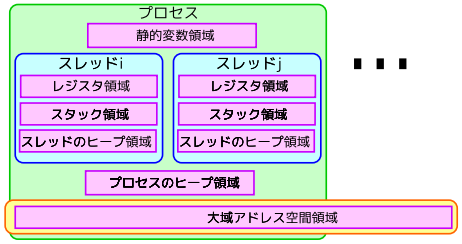

図 4　DMI におけるアドレス領域のモデル化
Fig. 4　The model of address regions on DMI.

```
int printf(format, ...)
{
  static char *buf;
  static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

  pthread_mutex_lock(&mutex);
  if(buf == NULL) {
    buf = malloc(4096);
  }
  ...; /* buf を使って文字列 format を値で埋める */
  write(1, buf, strlen(buf));
  pthread_mutex_unlock(&mutex);
}
```

図 5　printf() 関数の実装例
Fig. 5　An example implementation of printf().

$stack_i^p$，$threadheap_i^p$，dmi のいずれかのアドレス領域に含まれている[⋆1]．

したがって，DMI_yield() 関数を呼び出す時点で，グローバル変数（$static^p$）を使用していたり，malloc() 関数で確保したアドレス領域（$processheap^p$）を使用していたりすると，スレッド移動後の実行の正しさは保証されない．

ここで注意すべき点は，このプログラミング制約は，DMI_yield() 関数を呼び出す時点についてしか言及していないという点である．よって，DMI_yield() 関数を呼び出す時点でプログラミング制約が守られてさえいれば，DMI_yield() 関数を呼び出していない時点においては，$static^p$，$processheap^p$ のアドレス領域を使用しても問題はない．たとえば，(1) p=malloc(...) 関数によりアドレス領域 p を確保し，(2) アドレス領域 p を使用して計算を行い，(3) free(p) 関数によりアドレス領域 p を解放する，という処理を行う関数として f() 関数を考える．このとき，DMI_yield() 関数を呼び出す前に f() 関数を呼び出したとしても，プログラミング制約には違反しない．別の例としては，DMI_read() 関数や DMI_write() 関数などの DMI 関数は，内部的には malloc() 関数などを使用しているが，すべての DMI 関数は，その DMI 関数の実行が終了した時点で $static^p$，$processheap^p$ にはスレッドの実行を継続するために必要なデータを残さないように実装されているため，DMI_yield() 関数を呼び出す前に DMI 関数を呼び出しても，当然プログラミング制約には違反しない[⋆2]．

⋆1 なお，このプログラミング制約は 5.5 節において少し緩和していい直される．
⋆2 ただし，非同期 DMI 関数に関しては，その非同期操作が完了するまでは，スレッドの実行を継続するために必要なデータが $static^p$，$processheap^p$ に残されているため，非同期操作の完了を回収するまでは DMI_yield() 関数を呼び出すことはできない．

以上の観察をまとめると，スレッド移動を行うユーザプログラムに対しては一定のプログラミング制約が課せられるものの，このプログラミング制約のもとでは以下の記述が許されているため，8.2 節で評価するような並列科学技術計算を記述できるだけの記述力は保たれているといえる：

- DMI_thread_mmap() 関数/DMI_thread_munmap() 関数/DMI_thread_mremap() 関数によるスレッドローカルなアドレス領域の確保/解放と，そのアドレス領域に対する通常のメモリアクセス．
- DMI_mmap() 関数/DMI_munmap() 関数/DMI_mremap() 関数による大域アドレス空間の確保/解放と，そのアドレス領域に対する DMI_read() 関数/DMI_write() 関数などによるメモリアクセス．
- 大域アドレス空間に対する同期操作，プロセスの参加/脱退操作などの各種 DMI 関数の呼び出し．

5.4　スレッド移動の手順

前節で述べたプログラミング制約のもとでは，以下の手順により，プロセス p 内のスレッド i をプロセス q へと移動させることができる：

(1) スレッド i が DMI_yield() 関数を呼び出したとき，スレッド i の移動が指示されていれば，スレッド i を停止させる．
(2) プロセス p における $register_i^p$，$stack_i^p$，$threadheap_i^p$ のアドレス領域を，プロセス q に対して送信する．
(3) プロセス q は，受信した $register_i^p$，$stack_i^p$，$threadheap_i^p$ のアドレス領域を，プロセス p で使用されていたアドレス領域とまったく同一のアドレス領域に割り当てる．
(4) プロセス q はスレッド i を復旧させ，DMI_yield() 関数を返す．

dmi のアドレス領域に関しては，スレッド移動にともなって何もする必要はない．なぜなら，大域アドレス空間に関しては，どの時点でどのスレッドからアクセスされても問題がないようにコンシステンシが維持されているためである．なお，上記の (3) において，プロセス p において $register_i^p$，$stack_i^p$，$threadheap_i^p$ が使用していたアドレス領域がプロセス q で使用されていない保証はないため，まったく同一のアドレス領域に割り当てることができる保証はない．この対策に関しては 6 章で述べる．

5.5　プログラミング制約の緩和

5.3 節において，DMI におけるスレッド移動時のプログラミング制約は，並列科学技術計算を記述するための記述力を保っていると述べた．しかし，グローバル変数を使用できな

いことはプログラマに対して一定の不便を強いると考えられる．なぜなら，グローバル変数を使用できないということは，グローバル変数を使用しうるすべてのライブラリ関数をユーザプログラムから呼び出せないことを意味するからである．したがって，（実装に依存するが）printf() 関数，malloc() 関数などの，内部でグローバル変数を使用する libc 共有ライブラリの多くのライブラリ関数は呼び出せないことになる．ところが，実は，5.3 節で述べたプログラミング制約は少し緩和させることができ，特定の条件を満たすライブラリ関数であれば，内部でグローバル変数を使用していても安全に呼び出すことができる．以下では，グローバル変数を使用する printf() 関数を例にとり，プログラミング制約がどう緩和できるかを議論する．

図 5 に示すような実装の printf() 関数を考える．libc 共有ライブラリの printf() 関数の実装では，任意長のフォーマット文字列を許可したり，出力のバッファリングなどを行っていたりすると思われるが，簡単のため省略する．この printf() 関数では，1 回目の呼び出しでグローバル変数 buf にメモリを確保し，2 回目以降の呼び出しでは 1 回目の呼び出し時に確保したメモリを再利用する仕様になっている．いい換えると，グローバル変数 buf にスレッドの実行を継続するために必要なデータを格納したまま，関数が返る仕様になっている．よって，プロセス p 内のスレッド i を別のプロセス q に移動させることを考えたとき，スレッド i が DMI_yield() 関数を呼び出す前に 1 度でも printf() 関数を呼び出しているならば，DMI_yield() 関数を呼び出す時点で，スレッド i の実行を継続するために必要なデータが $static^p$ に格納されていることになり，プログラミング制約に違反してしまう．具体的には，DMI ではスレッド移動の際に $static^p$ を移動させないため，スレッド i が移動先プロセス q で最初に printf() 関数を呼び出した時点でグローバル変数の不一致が起き，問題が生じるように思われる．ところが，実際には何の問題も生じない．なぜなら，図 5 における printf() 関数の実装を注意深く観察すると分かるように，スレッド i が移動先プロセス q で呼び出した printf() 関数は，移動元プロセス p のグローバル変数 buf の値とは無関係に，その時点での移動先プロセス q のグローバル変数 buf の値に基づいて正しく実行されるからである．すなわち，この printf() 関数に関しては，グローバル変数を使用しているにもかかわらず，スレッド移動にともなってグローバル変数を移動させなくても問題は生じない．

以上のような現象が生じる理由は，この printf() 関数は，実際にはグローバル変数を使用してはいるものの，任意のスレッドによって任意の順序で呼び出されても正しく実行されるようなセマンティクスでグローバル変数を使用しているためである．いい換えると，セマンティクスとしては，スレッドの実行を継続するために必要なデータがグローバル変数に入っていないと見なすことができるためである．以上の議論より，先ほど定義したプログラミング制約は以下のように緩和することができる：

プログラミング制約　プロセス p 内のスレッド i が DMI_yield() 関数を呼び出す時点において，そのスレッド i の実行を正しく継続するために必要なすべてのデータは，$register_i^p$，$stack_i^p$，$threadheap_i^p$，dmi のいずれかの領域に含まれているセマンティクスになっている．

ここで問題なのは，各ライブラリ関数が上記のプログラミング制約を満たすかどうかは，通常は仕様として規定されていないため，逐一ライブラリ関数の実装を調べなければならない点である．しかし，本節で主張すべきことは，ライブラリ関数の実装を注意深く検討しさえすれば，仮にそのライブラリ関数が内部で $static^p$，$processheap^p$ のアドレス領域を使用しているとしても，スレッド移動の安全性に影響を与えないようにそれらのライブラリ関数を呼び出すことができる場合がある，ということである．実際，スレッドセーフな libc 共有ライブラリの関数の中には安全に呼び出せるものも多い．

ここまでグローバル変数に関連する問題を議論したが，既存研究においても，グローバル変数の取扱いはスレッド移動を行ううえで深刻な問題とされてきた．第 1 に，PM2[9),10)]，Adaptive MPI[31)]，MigThread[34)–36)]，Arachne[24)] など多くの既存研究はそもそもグローバル変数の使用を禁止している．第 2 に，Windows 環境においてスレッド移動を実現する Tern[38)] では，スレッド移動にともなうスレッドローカルストレージの移動をサポートしており，プログラマは「各スレッドにとってグローバルな」変数を利用できる．しかし，Windows 環境とは異なり，DMI が想定している Linux 環境では，ユーザレベルからランタイムにスレッドローカルストレージを抽出するのは難しい．第 3 に，Java においてスレッド移動を実現している JESSICA2[37),62),63)] では，Delta Execution というマスタワーカ型の手法を用いて，グローバル変数のサポートを実現している．Delta Execution では，系内に存在するすべてのスレッド i は，マスタノードに親スレッド i' を持つ．各スレッド i はマスタノードおよび各ワーカノードを自由にスレッド移動できるが，スレッド i がワーカノードにおいてグローバル変数へのアクセスやファイルアクセスなどプロセス依存な操作を行おうとした場合には，プログラムの実行をマスタノードに存在する親スレッド i' に引き渡し，マスタノード上でそのプロセス依存な操作を実行する．そして，それらのプロセス依存な操作が完了したあと，再びプログラムの実行をワーカノード上のスレッド i に引き戻す．これにより，プロセス依存な操作はすべてマスタノードで実行されることになるため，ユーザプ

ログラムでグローバル変数などを使用しても差し支えない．しかし，この Delta Execution は Java のバイトコードに手を加えることで実現されており，DMI のような C 言語における処理系に応用させるのは難しい．

6. アドレス空間の大きさに制限されないスレッド移動

6.1 基本アイディア

DMI では，計算規模がアドレス空間の大きさに制限されないスレッド移動手法として，random-address を提案する．random-address の基本アイディアは次のとおりである：

(1) 各スレッドは，自分以外のスレッドがどのアドレス領域にローカルアドレス空間を割り当てているかに関する知識を持たない．ここで，プロセス p 内のスレッド i のローカルアドレス空間とは，$register_i^p$，$stack_i^p$，$threadheap_i^p$ を意味するものとする．各スレッドは，乱数を使って，ローカルアドレス空間を割り当てるアドレスを決定する（**図 6** (A)）．よって，各スレッドがローカルアドレス空間を割り当てる操作（`DMI_thread_mmap()` 関数/`DMI_thread_munmap()` 関数/`DMI_thread_mremap()` 関数）は，他のスレッドとの通信をいっさい必要とせず，完全に独立に実行できる．

(2) いま，ノード P 上のプロセス p に存在するスレッド i を，ノード Q 上のプロセス q へと移動させるとする．このとき，「運が良ければ」，移動元プロセス p においてスレッド i が使用しているアドレス領域は，移動先プロセス q では使用されていない．この場合には，移動先プロセス q において，スレッド i のローカルアドレス空間を移動元プロセス p と同一のアドレス領域に割り当てることで，スレッド移動を完了させる（図 6 (A)）．

(3) スレッド i の移動時に，「運が悪ければ」，スレッド i が移動元プロセス p において使用しているアドレス領域が，すでに移動先プロセス q でも使用されている．この場合には，当然，移動先プロセス q において，スレッド i のローカルアドレス空間を移動元プロセス p と同一のアドレス領域に割り当てることができない．そこで，移動先プロセス q が存在するノード Q 上に新しいプロセス q'（要するに新しいアドレス空間）を生成し，新しいプロセス q' の中にスレッド i を移動させる（図 6 (B)）．

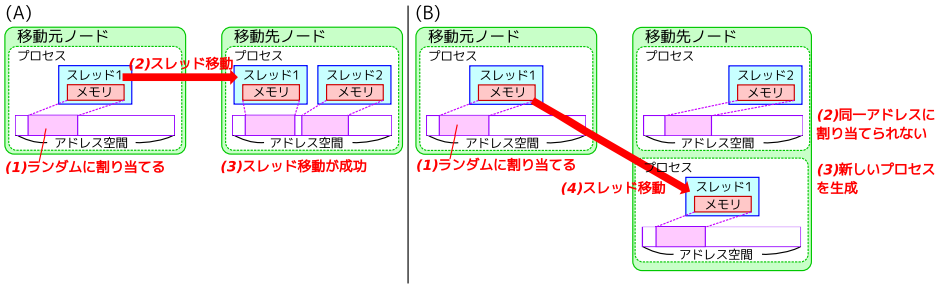

図 6　random-address のアルゴリズム．(A) アドレスが衝突しない場合，(B) アドレスが衝突する場合
Fig. 6　An algorithm of random-address. (A) When addresses collide, (B) When addresses do not collide

この random-address は，DMI がプロセスの動的な参加/脱退に対応しているからこそ可能な手法である．random-address ではアドレスが衝突した場合にプロセス数が増えるが，スレッドスケジューリングを適切に行ってスレッドが存在しなくなったプロセスを破棄するようにすれば，スレッド移動を繰り返してもプロセス数が増え続けることはない．たとえば，ノード P 上に存在するスレッド i とスレッド j に関して，ある時刻 t において，スレッド i が使用しているアドレス領域とスレッド j が使用しているアドレス領域に重なりがあり，スレッド i はノード P 上のプロセス p にスレッド j はノード P 上のプロセス p' に入っている状況を考える．このとき，仮に，スレッド j が使用しているアドレス領域と，別のノード Q 上にすでに存在しているプロセス q が使用しているアドレス領域に重なりがなければ，スレッド j をプロセス q の中へ移動させることで，プロセス p' を破棄することができる．あるいは，時刻 $t+\Delta t$ において，スレッド i またはスレッド j が使用しているローカルアドレス空間が変化して，スレッド i が使用しているアドレス領域とスレッド j が使用しているアドレス領域に重なりがなくなったとすれば，スレッド j をプロセス p の中にスレッド移動させることで，プロセス p' を破棄することができる．理論的には，m 個のスレッド x_0，x_1，…，x_{m-1} に関して，ある時刻 t において，各スレッド x_i が使用しているアドレスの集合を S_0^t，S_1^t，…，S_{m-1}^t とするとき，これら m 個のスレッドは，最小 $f(S_0^t, S_1^t, \ldots, S_{m-1}^t)$ 個のプロセスに格納することができる．ここで $f(S_0^t, S_1^t, \ldots, S_{m-1}^t)$ とは，以下の条件を満たす F のうち最小の値とする：

条件　m 個の集合 S_0^t，S_1^t，…，S_{m-1}^t を，F 個のグループ G_0，G_1，…，G_{F-1} に分類したとする．つまり，

$$\forall i\ (0 \le i \le m-1), \exists j\ (0 \le j \le F-1), \forall k\ (0 \le k \le F-1 \wedge k \ne j):$$
$$S_i^t \in G_j \wedge S_i^t \notin G_k$$

となるように各 S_i^t を分類したとする．このとき，

$$\forall i\ (0 \le i \le F-1), \forall S_j^t\ (S_j^t \in G_i), \forall S_k^t\ (S_k^t \in G_i \wedge k \ne j) : S_j^t \cap S_k^t = \emptyset$$

が成り立つ．

しかし，当然ながら，任意の時刻 t において，m 個のスレッドを $f(S_0^t, S_1^t, \ldots, S_{m-1}^t)$ 個のプロセスに格納するようにスレッドスケジューリングを行うのは現実的ではない．実際には，ノード間のスレッドの負荷バランス，スレッド移動に要する時間，プロセス数を増やすことによるオーバヘッド，各スレッド間でのデータ共有の度合いなどの要素を総合的に考慮して，スレッドスケジューリングを最適化する必要がある．ただし，本稿ではスレッドスケジューリングの最適化は考察の対象外とする．

6.2　アドレス衝突確率の最小化

前節で述べたように，random-address では，スレッド移動時に移動先プロセスでアドレス領域が衝突した場合には，そのプロセスが存在するノード上に新しいプロセスを生成することによってスレッドを移動させる．しかし，一般論として，スレッド間のデータ共有の方がプロセス間のデータ共有よりも高速であり，協調動作するインスタンスはプロセスとして実装するよりもスレッドとして実装する方が性能上望ましいことをふまえると，アドレス衝突を理由として同一ノード内に多数のプロセスを生成することは性能上不利である．

DMI に即していえば，3.1 節で述べたように，DMI ではプロセスを単位として大域アドレス空間のコンシステンシ管理を行っているため，1 ノード内のプロセス数が増えると性能が劣化してしまう．具体的には，第 1 に，各プロセスにつき，他ノードからのメッセージを受信する receiver スレッド，それらのメッセージを処理する handler スレッド，ページの追い出しを担当する sweeper スレッドなどの多数の管理用スレッドが存在している．よって，1 ノード内のプロセス数を増やせば，1 ノード内に存在する管理用スレッドも増えてしまい，計算本体を行うスレッドの性能が劣化してしまう．第 2 に，スレッド i とスレッド j が同一のプロセスに属していればスレッド i とスレッド j とでページのキャッシュを共有できるのに対して，別のプロセスに属している場合にはページのキャッシュを共有できない．たとえば，スレッド $i \rightarrow$ スレッド j の順序でページをキャッシュしようとする場合，スレッド i とスレッド j が同一のプロセスに属していればページフォルトは 1 回で済むのに対して，別のプロセスに属している場合にはページフォルトが 2 回発生してしまう．このように，1 ノード内のプロセス数が増えると性能が劣化する．

以上の観察より，アドレス衝突を理由として同一ノード内のプロセス数を増やさないようにするために，できるだけアドレス衝突を起きにくくする手法が必要だといえる．すなわち，random-address においては，乱数を使うとはいえ，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おおよそ以下である（問題の正確な定義は付録 A.1 で与える）：

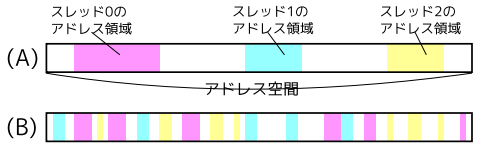

図 7　アドレス領域の連続的な使用と離散的な使用．(A) 連続的な使用，(B) 離散的な使用
Fig. 7　Patterns of how to use an address space. (A) Sequential use, (B) Discrete use

問題　m 個のスレッド x_0，x_1，…，x_{m-1} を考える．各スレッド x_i は，自分以外のスレッドがどのアドレス領域を使用しているか知らないとする．このとき，「どの 2 つの異なるスレッド x_i とスレッド x_j に対しても，スレッド x_i が使用するアドレス集合とスレッド x_j が使用するアドレス集合が共通部分を持たない確率」を最大にするためには，各スレッドがどのようなアドレス割当ての戦略を採用すればよいか？

そして，上記の問題に対する最適な戦略の 1 つは以下であることが証明できる（証明は付録 A.1 で与える）：

最適な戦略（の 1 つ）　各スレッド x_i は，できるかぎりアドレスを連続的に使用する．

上記の意味は，「各スレッドが離散的にランダムにアドレスを使用するよりも，連続的にアドレスを使用する方がスレッド移動時のアドレス衝突確率が小さい」ということである．たとえば，**図 7** (A) のようにアドレスを使用する方が，図 7 (B) のようにアドレスを使用するよりスレッド移動時のアドレス衝突確率が小さい．

以上の観察に基づき，random-address では，各スレッドができるかぎり連続的にアドレスを使用するように，**図 8** に示すアルゴリズムに従って，各スレッドが使用するアドレス領域（$stack_i^p$ と $threadheap_i^p$）を管理する．

6.3　random-address の改善

各スレッドが使用するメモリ量を予測できるならば，さらに random-address を改善し，スレッド移動時のアドレス衝突確率を下げることができる．前節の議論で導いた，「各スレッド x_i はできるかぎりアドレスを連続的に使用する」という戦略においては，各スレッド x_i は任意のアドレスから始めて連続的なアドレス領域を使用することができる．いい換えると，この戦略では，各スレッド x_i が使用するアドレス領域は 1 の整数倍にしかアラインされない．ところが，この戦略に加えて，「各スレッド x_i が使用するアドレス領域は，*align*

```
Z_i : Set of region

when thread i is created:
    Z_i ← ∅
    return

when thread i mmaps size bytes:
    foreach (ptr,length) ∈ Z_i do
        p ← fixed_mmap(ptr + length,size)
        if p != MAP_FAILED then
            Z_i ← Z_i\{(ptr, length)}
                  ∪{(ptr, length + size)}
            reduction(Z_i)
            return
        endif
    endfor
    while 1 do
        addr ← rand(inf, sup)
        p ← fixed_mmap(addr,size)
        if p != MAP_FAILED then
            Z_i ← Z_i ∪ {(p, size)}
            reduction(Z_i)
            return
        endif
    endwhile
    return

when thread i munmaps region (p,size):
    munmap(p,size)
    foreach (ptr,length) ∈ Z_i do
        if ptr == p and p + size == ptr + length then
            Z_i ← Z_i\{(ptr, length)}
        else if ptr == p and p + size < ptr + length then
            Z_i ← Z_i\{(ptr, length)} ∪ {(p + size, length − size)}
        else if ptr < p and p + size == ptr + length then
            Z_i ← Z_i\{(ptr, length)} ∪ {(ptr, length − size)}
        else if ptr < p and p + size < ptr + length then
            Z_i ← Z_i\{(ptr, length)} ∪ {(ptr, p − ptr)}
                  ∪{(p + size, ptr + length − p − size)}
        endif
    return

when thread i is destroyed:
    foreach (ptr,length) ∈ Z_i do
        munmap(ptr,length)
    endfor
    return
```

図 8　random-address における各プロセスのアドレス空間管理のアルゴリズム．(ptr,$length$) は，アドレス ptr から始まる $length$ バイトの連続領域を表す．inf と sup は，それぞれ，$stack_i^p$ と $threadheap_i^p$ を割り当てるために使用することのできるアドレス空間の下限値と上限値を表す．また，fixed_mmap($addr$,$size$) 関数は，「アドレス $addr$ から $size$ バイトが未使用であれば mmap し，使用中であれば MAP_FAILED を返す」関数とする（7.3 節を参照）．reduction(Z_i) は，領域集合 Z_i 内の任意の 2 個以上の連続領域に関して，連続領域としてまとめられるものを 1 個の連続領域にまとめる関数とする

Fig. 8　An algorithm for address space management of each process in random-address.

の整数倍にアラインされなければならない」という制約を導入し，$align$ の値を適切に調整することによって，さらにアドレス衝突確率を下げることができる．この理由は，直観的には，各スレッド x_i が使用するアドレス領域をある程度大きな数の整数倍にアラインさせることによって，2 つのスレッドのアドレス領域のごく一部だけが衝突することに起因するアドレス衝突が起きにくくなるためである．

ここで問題となるのは，$align$ の最適値である．$align$ の値を 1 から増やしていく場合のアドレス衝突確率の変化を定性的に考えると，ある一定の値まではアドレス衝突確率は下がるが，$align$ の値が各スレッド x_i の使用するメモリ量よりも十分大きくなってしまうと，逆にアドレス衝突確率は上がってしまうことが分かる．すなわち，$align$ には，各スレッド x_i が使用するメモリ量に依存した最適値が存在する．証明は省略するが，最も単純な場合として，一般に，「すべてのスレッド x_i が b バイトを使用するならば，$align = b$ のときにアドレス衝突確率が最小になる」ことが導ける．しかし，実際のユーザプログラムにおいては，各スレッドが使用するメモリ量はスレッドごとに異なるうえ，そもそも各スレッドが使用するメモリ量を事前に知ることは難しい．したがって，実際には，各スレッドが使用するメモリ量を予測し，その予測に基づいて経験的に $align$ の値を設定することが必要となる．

6.4　アドレス領域の管理

random-address では，アドレスが衝突した場合には新しいプロセスを生成して，その新しいプロセスの中にスレッドを移動させるが，当然ながら，このとき新しいプロセスの中へのスレッド移動は確実に成功しなければならない．いい換えると，生成された直後のプロセスが使用しているアドレス領域（以下，初期領域と呼ぶ）が，移動したいスレッドが使用しているアドレス領域（以下，移動領域と呼ぶ）と重なっていることは許されない．なぜなら，仮に重なってしまったとすると，その新たに生成されたプロセスが使用できないことになるため，さらにもう 1 個新しいプロセスを生成する必要があり，この作業を繰り返すうちに無限個のプロセスを生成してしまう可能性があるためである．また，将来的に，スレッド移動を拡張して分散チェックポイント/リスタートを導入しようとした場合に，いくつ新しいプロセスを生成しても，生成したプロセスが使用するアドレス領域とリスタートしようとしているスレッドが使用するアドレス領域が重なってしまい，スレッドをリスタートできないという事態も起こりうる．以上をふまえて，本節では，初期領域と移動領域が重ならないことを保証できるようなアドレス領域の管理について考える．

5 章で述べたように，DMI では，アドレス領域全体を $register_i^p$，$stack_i^p$，$threadheap_i^p$，$static^p$，$processheap^p$，dmi の 6 種類のアドレス領域に分類して管理する．そして，5.2 節と 5.4 節の説明より，このうち移動領域に含まれる可能性があるのは，$register_i^p$，$stack_i^p$，$threadheap_i^p$ の 3 つであり，初期領域に含まれる可能性があるのは，$static^p$，$processheap^p$，dmi の 3 つである．したがって，初期領域と移動領域が重ならないようにするためには，$register_i^p$，$stack_i^p$，$threadheap_i^p$ のアドレス領域と $static^p$，$processheap^p$，dmi のアドレス領域が重ならないように管理すればよい．そこで DMI では，利用可能なアドレス領域全体（たとえば 64 ビットアーキテクチャであれば 2^{47} バイト）を前半部分と後半部分の 2 つに分割し，$register_i^p$，$stack_i^p$，$threadheap_i^p$ のアドレス領域に関しては前半部分に割り当て，$processheap^p$，dmi のアドレス領域に関しては後半部分に割り当てるよう，アドレス領域を管理する．特に，前半部分に関しては，図 8 に示すアルゴリズムに従って，各スレッドの使用するアドレス領域ができるかぎり連続的になるようアドレス領域を管理する．$static^p$ に

関しては，静的に確保されるアドレス領域であるため，そもそも DMI の処理系がアドレス領域を制御できる自由度はないが，$static^p$ はどのアドレスに配置されたとしても移動領域と重なることはないので，問題にならない．なぜなら，同一アーキテクチャの実行環境おいて同一の（再配置可能でない）実行バイナリを実行するならば，$static^p$ のアドレス領域の位置はすべてのプロセスで同一になるため，移動領域が $static^p$ に重なることはありえないからである．なお，ここで考慮したアドレス領域以外にも，コードが配置されるアドレス領域などがあるが，それらのアドレス領域が配置される位置に関しても，同一アーキテクチャの実行環境で同一の（再配置可能でない）実行バイナリを実行するならば同一になるため，移動領域と重なることはない．なお，前半部分と後半部分の大きさをどう配分するかに関しては最適化の余地があるが，現在の実装では単純にアドレス空間全体を 2 等分するよう実装している．

7. 実　　装

本章では，6 章で述べたスレッド移動および random-address によるアドレス空間管理を，ユーザレベルで実装する方法を説明する．

7.1 スレッドのチェックポイント/リスタート

DMI では，`ucontext_t` [4] と呼ばれる，Linux においてユーザレベルでコンテキストスイッチを行う機構を利用して，スレッドのチェックポイント/リスタートを実装している．`ucontext_t` では，`makecontext(ucontext_t *ucp,void *func,...)` 関数を呼び出すことで，新しいコンテキスト `ucp` を作成できる．ここで `func` は，コンテキスト `ucp` へのコンテキストスイッチが初めて起きたときに実行される関数である．また，コンテキスト `ucp` が使用するスタック領域も明示的に指定できる．さらに，`swapcontext(ucontext_t *oucp,ucontext_t *ucp)` 関数を呼び出すことで，この関数を呼び出した現在のコンテキストを `oucp` に保存し，別のコンテキスト `ucp` にコンテキストスイッチすることができる．

スレッドのチェックポイント/リスタートの手順を示す：

(1) `DMI_scheduler_create()` 関数が呼び出されると，あるプロセス p でスレッド i が生成される．この時点におけるこのスレッド i のコンテキストを c_0 と名付ける．

(2) コンテキスト c_0 は，`makecontext(c1,DMI_thread,...)` 関数を呼び出すことで，スタック領域を明示的に指定して新しいコンテキスト `c1` を作る．この新しいコンテキストを c_1 と名付ける．

(3) コンテキスト c_0 が，`swapcontext(c0,c1)` 関数を呼び出す．その結果，現在のコンテキスト c_0 が `c0` に保存され，コンテキスト c_1 へのコンテキストスイッチが起きる．そして，ユーザプログラムに定義されている `DMI_thread()` 関数がコンテキスト c_1 で実行される．

(4) やがて，このスレッド i に対するスレッド移動の必要が生じ，コンテキスト c_1 で実行されているユーザプログラムから `DMI_yield()` 関数が呼び出されたとする．このとき，`DMI_yield()` 関数は内部で，(3) で保存した `c0` を指定して `swapcontext(c1,c0)` 関数を呼び出す．その結果，コンテキスト c_1 が `c1` に保存されたあと，コンテキスト c_0 へのコンテキストスイッチが起き，(3) においてコンテキスト c_0 が呼び出した `swapcontext()` 関数が返る．

(5) この時点で，コンテキスト c_1 つまり `DMI_thread()` 関数のコンテキストが使用している $register_i^p$ は `c1` に保存されている．よって，コンテキスト c_0 は，`c1` そのもの（$register_i^p$）と，コンテキスト c_1 のスタック領域（$stack_i^p$），および（別に管理している）スレッド i のヒープ領域（$threadheap_i^p$）の 3 つを，移動先のプロセス q に対して送信する．

(6) $register_i^p$，$stack_i^p$，$threadheap_i^p$ を受信した移動先プロセス q は，それら 3 つの領域を移動元プロセス p と同一のアドレス領域に割当て可能かどうかを調べ，不可能ならば移動元プロセス p に対して失敗通知を返す．可能ならば，それら 3 つの領域をまったく同一のアドレス領域に割り当て，移動元プロセス p に対して成功通知を返す．

(7) 移動先プロセス q は，移動元プロセス p から受信した `c1` をそのまま指定して `swapcontext(c0,c1)` 関数を呼び出すことで，(4) において移動元プロセス p がコンテキスト c_1 で呼び出した `swapcontext()` 関数が移動先プロセス q で返る．その結果，移動元プロセス p とまったく同一のコンテキスト c_1 で，`DMI_yield()` 関数を返すことができ，リスタートが完了する．

(8) 移動元プロセス p は，(6) において移動先プロセス q が返す成功/失敗通知を待機する．成功通知が受信されれば，そのままスレッド i を終了させる．失敗通知が受信された場合，移動先プロセス q が存在するノードに対して新しいプロセス q' を生成したあと，プロセス q' に対して再度スレッド移動を試みる．

7.2 システムコールのハイジャック

7.2.1 ハイジャックの必要性

6.4 節で述べたアドレス空間管理においては，利用可能なアドレス領域全体を前半部分と後半部分の 2 つに分割し，$register_i^p$，$stack_i^p$，$threadheap_i^p$ のアドレス領域は前半部分に割

り当て，$processheap^p$，dmi のアドレス領域は後半部分に割り当てるようなアドレス空間管理を行う必要がある．そのためには，DMI が，各アドレス領域の確保/解放の処理をランタイムにハイジャックして，そのアドレス領域の確保/解放を具体的にどのアドレスに対して行うかを明示的に制御する必要がある．

ここで，各アドレス領域に関して，アドレスの明示的な制御が可能かどうかを確認する．第 1 に，7.1 節で述べた実装では，$register_i^p$ は `ucontext_t` 型の変数として扱われ，$stack_i^p$ は `makecontext()` 関数を呼び出すときに明示的に指定するので，DMI はこれらのアドレス領域を明示的に制御することができる．第 2 に，$threadheap_i^p$ は，定義より，`DMI_thread_mmap()` 関数/`DMI_thread_munmap()` 関数/`DMI_thread_mremap()` 関数という DMI の専用関数を呼び出すことで確保/解放されるアドレス領域であるから，DMI はそのアドレス領域を明示的に制御することができる．第 3 に，$processheap^p$ と dmi は，（`malloc()` 関数/`free()` 関数などを経由して）システムコールの `mmap()` 関数/`mremap()` 関数/`munmap()` 関数によって確保/解放されるアドレス領域である．よって，それらのシステムコールの関数が確保/解放するアドレス領域を明示的に制御するためには，何らかのレイヤにおいてシステムコールをハイジャックする必要がある．具体的には，アドレス領域の確保/解放に関連するシステムコールである，`mmap()` 関数/`mremap()` 関数/`munmap()` 関数/`mprotect()` 関数/`brk()` 関数をハイジャックする．

7.2.2　どのレイヤでハイジャックするべきか

どのレイヤにおいてシステムコールをハイジャックするべきかを，`mmap()` 関数を例にして議論する．以降では，カーネルに定義されているシステムコールの `mmap()` 関数を `sys_mmap()` 関数，libc 共有ライブラリの `mmap()` 関数を `libc_mmap()` 関数，ハイジャック後に実行したい `mmap()` 関数を `hijack_mmap()` 関数と表記する．

Linux カーネル 2.6 においては，C 言語の実行バイナリから `libc_mmap()` 関数が呼び出されたとき以下の動作が起きる[67]：

(1)　実行バイナリから呼び出された `libc_mmap()` 関数が動的リンクされており，かつその `libc_mmap()` 関数の呼び出しが初回ならば，libc 共有ライブラリの `libc_mmap()` 関数のアドレスが検索される．
(2)　`libc_mmap()` 関数のアドレスが求まったあとで，`libc_mmap()` 関数が呼び出される．
(3)　`libc_mmap()` 関数がシステムコールの `sys_mmap()` 関数を呼び出し，カーネルレベルに制御が移る．
(4)　システムコールテーブルが検索され，`sys_mmap()` 関数のアドレスが求められる．
(5)　このプロセスが ptrace のデバッギプロセスになっていれば，デバッガプロセスに対してシグナルを送り，システムコールが発行されたことを通知する．
(6)　`sys_mmap()` 関数の本体が実行される．
(7)　このプロセスが ptrace のデバッギプロセスになっていれば，デバッガプロセスに対してシグナルを送り，システムコールが完了したことを通知する．
(8)　ユーザレベルに制御が戻り，`libc_mmap()` 関数が返る．

上記の手順を観察すると，`sys_mmap()` 関数をハイジャックして，代わりに `hijack_mmap()` 関数を実行させられると思われる場所は (1)，(3)，(4)，(5) の 4 カ所ある．以下ではこの 4 カ所におけるハイジャックの概要と問題点を議論する．

(1) におけるハイジャック　環境変数 LD_PRELOAD を使用することで共有ライブラリの検索順序を変更する手法である．これにより，(1) の手順において，libc 共有ライブラリの `libc_mmap()` 関数が発見される前に自作の共有ライブラリの `hijack_mmap()` 関数を発見させることができ，`libc_mmap()` 関数の代わりに `hijack_mmap()` 関数を実行させることができる．しかし，この手法は `libc_mmap()` 関数が静的リンクされている場合には使えない．たとえば，libc 共有ライブラリが静的リンクされているならば，libc 共有ライブラリの `malloc()` 関数の内部で呼び出される `libc_mmap()` 関数をハイジャックすることはできず，`hijack_mmap()` 関数を実行させることができない．

(3) におけるハイジャック　`libc_mmap()` 関数のコードを書き換えることで，`sys_mmap()` 関数の代わりに `hijack_mmap()` 関数を実行させる手法である．静的に行う手法と動的に行う手法がある．静的に行う手法では，`libc_mmap()` 関数のソースコードを書き換えてコンパイルした改造 libc 共有ライブラリを用意し，DMI を実行するときには，通常の libc 共有ライブラリではなく，改造 libc 共有ライブラリを使用するようにすればよい．しかし，この手法には，libc 共有ライブラリは多様な実行環境に対応して実装されているためにソースコードの変更箇所が広範囲におよぶうえ，各実行環境ごとに改造 libc ライブラリを用意しなければならず移植性が低いという問題がある．そこで，DMI では，実装の容易化と移植性の向上のため，動的に libc 共有ライブラリのコードを書き換える手法を採用する．この手法の詳細と得失は次節で議論する．

(4) におけるハイジャック　カーネルモジュールを使用して，システムコールテーブル内の `sys_mmap()` 関数のアドレスを `hijack_mmap()` 関数のアドレスに書き換えることで，`sys_mmap()` 関数の代わりに `hijack_mmap()` 関数を実行させる手法である．しかし，Linux カーネル 2.6 以降では，セキュリティ上の理由により，システムコールテーブル

の先頭アドレスを示す変数 sys_call_table が extern されなくなったため，カーネルモジュールからシステムコールテーブルを操作することができない．そのため，この手法を使うためには，変数 sys_call_table を extern するようにカーネルを変更する必要があるが，カーネルの変更は移植性を大きく落としてしまう．また，カーネルモジュールの実行には root 権限が必要な点も問題である．

(5) におけるハイジャック　ptrace を使用して該当プロセスをデバッギプロセスとして登録し，デバッギプロセスで発行されるすべてのシステムコールをデバッガプロセスから監視する手法である．デバッガプロセスがデバッギプロセスで発行された sys_mmap() 関数をフックした時点で，デバッガプロセスから PTRACE_POKEUSER を使ってデバッギプロセスのコードを書き換えたり，システムコールの引数レジスタを書き換えたりすることで，デバッギプロセスに hijack_mmap() 関数を実行させることができる．しかし，本来 ptrace はプロセスの監視用に作られており，DMI が利用している pthread を監視するには不十分な点が多いという問題がある．たとえば，デバッガプロセスが ptrace によってシステムコールをフックした時点で，それがどの pthread によって呼び出されたシステムコールなのかを容易に知る手段が存在しない．

以上で検討した手法はいずれも，システムコールをハイジャックできない場合が存在するか，または移植性が低いという点で問題がある．

7.2.3　動的に共有ライブラリのコードを書き換える

以上の観察をふまえて，DMI では，ほぼすべての場合にシステムコールをハイジャックでき，かつ移植性の高い手法として，libc 共有ライブラリのコードを動的に変更する手法を提案する．この手法では，libc_mmap() 関数が呼び出されたときに hijack_mmap() 関数が呼び出されるよう，DMI の実行が開始された直後に libc 共有ライブラリのコードを書き換える．

第 1 に，基本アイディアを説明する．まず，libc_mmap() 関数のコードの先頭に，hijack_mmap() 関数のアドレスへのジャンプ命令を挿入する．これによって，libc_mmap() 関数が呼び出されたとき，すぐに hijack_mmap() 関数へと制御が飛ばされることになる．しかし，これだけでは不十分で，hijack_mmap() 関数が DMI のアドレス空間管理のための一連の処理を行ったあと，実際に OS からアドレス領域を割り当てようとして libc_mmap() 関数を呼び出すと，再度 hijack_mmap() 関数が呼び出され無限に再帰してしまう．つまり，OS からアドレス領域を割り当てる手段がなくなってしまう．そこで，本来の libc_mmap() 関数も呼び出せるようにするため，本来の libc_mmap() 関数を呼び出すためのエントリポイントを true_mmap() 関数という名前で作成しておく．これにより，hijack_mmap() 関数は，true_mmap() 関数を呼び出すことで実際に OS からアドレス領域を確保できるようになる．まとめると，以下の順序で関数が呼び出されるように libc 共有ライブラリのコードを書き換える：

(1)　C 言語の実行バイナリが libc_mmap() 関数を呼び出す．
(2)　libc_mmap() 関数はすぐに hijack_mmap() 関数へジャンプする．
(3)　hijack_mmap() 関数が OS からアドレス領域を確保するときは true_mmap() 関数を呼び出す．
(4)　true_mmap() 関数は本来の libc_mmap() 関数のエントリポイントになっており，この内部で sys_mmap() 関数が呼び出され，OS からのアドレス領域の確保が実現される．

第 2 に，実装について説明する：

(1)　libc_mmap() 関数と hijack_mmap() 関数の先頭アドレスを，それぞれ libc_mmap，hijack_mmap とおく（**図 9** (A)）．
(2)　「$x \geq injection$ であり，かつ libc_mmap から x バイト先のアドレスがちょうど命令境界になっている」条件を満たす x のうち最小の x を x_0 とする．ここで，*injection* は (4) においてこの位置に挿入したいアセンブリコードのバイト数であり，x86-64 アーキテクチャの場合は 12 バイトである．libc_mmap から何バイト目が命令境界になっているかは，objdump コマンドなどを使用して libc 共有ライブラリを逆アセンブルすることで調べる．たとえば，libc-2.3.6 では $x_0 = 12$ であり，libc-2.7 では $x_0 = 14$ である．空いている適当なアドレス領域に $x_0 + injection2$ バイトを確保し，その先頭アドレスを true_mmap とする．ここで，*injection2* は (5) においてこの位置に挿

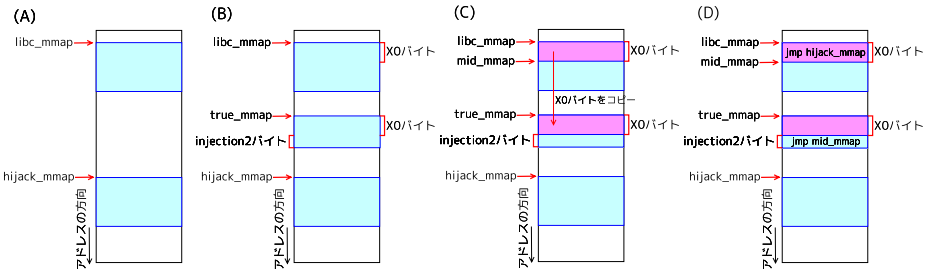

図 9　共有ライブラリのコードを動的に書き換える手順
Fig. 9　How to modify shared library codes dynamically.

入したいアセンブリコードのバイト数であり，x86-64 アーキテクチャの場合 13 バイトである．最終的には，この位置に `true_mmap()` 関数のエントリポイントを作成する（図 9 (B)）．

(3) アドレス領域 [`libc_mmap`, `libc_mmap`+x_0) を，アドレス領域 [`true_mmap`, `true_mmap`+x_0) にコピーする．ここで，アドレス `libc_mmap`+x_0 を `mid_mmap` とおく（図 9 (C)）．

(4) `libc_mmap()` 関数が呼ばれた直後に `hijack_mmap()` 関数に処理を飛ばすため，以下のアセンブリコード（サイズを *injection* バイトとする）をアドレス領域 [`libc_mmap`, `libc_mmap` + *injection*) に挿入する：

```
mov     hijack_mmap, %rax
jmpq    *%rax
```

これにより，`libc_mmap()` 関数が呼び出されると，そのままの引数で `hijack_mmap()` 関数が呼び出されるようになる．

(5) アドレス `true_mmap` の位置に，本来存在していた `libc_mmap()` へのエントリポイントを作るために，以下のアセンブリコード（サイズを *injection2* バイトとする）をアドレス領域 [`true_mmap` + x_0, `true_mmap` + x_0 + *injection2*) に挿入する（図 9 (D)）：

```
mov     mid_mmap, %r11
jmpq    *%r11
```

これにより，`true_mmap()` 関数が呼び出されると，もともとアドレス領域 [`libc_mmap`, `libc_mmap`+x_0) の位置に配置されていたコードが実行されたあと，アドレス `mid_mmap` へとジャンプし，アドレス `libc_mmap` + x_0 からコードが実行されることになる．すなわち，`true_mmap()` 関数を呼び出すことで，本来存在していた `libc_mmap()` 関数を呼び出すことができる．なお，レジスタ `rax` や `r11` を使い分けている理由は，システムコール発行時のコーリングコンベンションに基づき，その時点で使用されていないレジスタを使用するためである．

以上の処理を，`main()` 関数が実行される前に実行することにより，プログラム内で発行されるすべての `libc_mmap()` 関数をハイジャックでき，`hijack_mmap()` 関数の中で $processheap^p$ と *dmi* のアドレス領域を明示的に制御することができる．

なお，この手法では，libc 共有ライブラリをハイジャックしているだけであって，システムコールそのものをハイジャックしているわけではない．したがって，アセンブリコードで直接 `sys_mmap()` 関数を呼び出したり，libc 共有ライブラリの `syscall()` 関数から直接 `sys_mmap()` 関数を呼び出したりする場合など，`libc_mmap()` 関数を経由せずに呼び出される `sys_mmap()` 関数はハイジャックできないという欠点がある．

7.3　指定したアドレス領域の mmap

ここまでの議論では，$register_i^p$，$stack_i^p$，$threadheap_i^p$，$static^p$，$processheap^p$，*dmi* の 6 種類の各アドレス領域の確保/解放の処理をハイジャックする手法を述べた．よって，あとは random-address のアドレス空間管理に基づいてアドレス領域を明示的に制御すればよいが，これを実装するには，当然，「指定したアドレスに安全にアドレス領域を割り当てる」関数が必要である．具体的には，アドレス `addr` とサイズ `size` を指定したとき，「アドレス領域 [`addr`, `addr` + `size`) が使用されていないならばアドレス領域 [`addr`, `addr` + `size`) を割り当て，すでに使用されているならば `MAP_FAILED` を返す」仕様の関数（図 8 における `fixed_mmap()` 関数）が必要である．よって，本節では，カーネルを変更することなくユーザレベルでそのような関数を実装する手法を説明する．

まず，libc 共有ライブラリの仕様[4] によれば，`libc_mmap(void *start,size_t length, int prot,int flags,int fd,off_t offset)` 関数だけでは，意図するアドレスに安全にアドレス領域を割り当てることはできないことを確認する．第 1 の方法として，確保したいアドレス `addr` を第 1 引数 `start` に指定する方法があるが，`libc_mmap()` 関数の仕様によれば，`start` はあくまでもヒントとして使用されるだけであり，仮に指定したアドレス領域 [`start`, `start` + `length`) が使用されていなくても，アドレス領域 [`start`, `start` + `length`) が確保できる保証はない．実際，8.2.1 項の環境ではこれが起きることを確認している．第 2 の方法として，`libc_mmap()` 関数の第四引数の `flags` に `MAP_FIXED` オプションを指定する方法を使えば，つねに指定したアドレス領域 [`start`, `start` + `length`) を確保することができる．しかし，アドレス領域 [`start`, `start` + `length`) がすでに使用されている場合には上書き確保されてしまう仕様であるため，この方法も安全ではない．

以上の動機をふまえて，DMI では，**図 10** に示すアルゴリズムで `fixed_mmap()` 関数を実装している．図 10 では，まず 8 行目で `true_mremap()` 関数を呼び出し，アドレス `start` から始まる 1 ページを 2 ページへと拡張しようとする．このとき，アドレス `start` から始まる 2 ページに関して，(1 ページ目の状態，2 ページ目の状態）の組合せとしては以下の 5 通りの場合が考えられるが，`libc_mremap()` 関数の仕様[4] によれば，この各場合に対して `true_mremap()` 関数は以下の値を返す：

（未使用，未使用） 返り値は `MAP_FAILED`，`errno` は `EFAULT`.
（未使用，使用中） 返り値は `MAP_FAILED`，`errno` は `ENOMEM`.
（使用中，未使用） 返り値は `start`，`errno` は設定されない.

```
00: void* fixed_mmap(void *start, size_t length, int prot, int flags)
01: {
02:   void *ptr;
03:   int ret;
04:   static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
05:
06:   pthread_mutex_lock(&mutex); /* ロック */
07:   errno = 0;
08:   ptr = true_mremap(start, PAGESIZE, PAGESIZE * 2, 0); /* start からの 1 ページを，2 ページへと拡張しようと試みる */
09:   if(ptr == MAP_FAILED && errno == EFAULT) {
10:     ptr = true_mmap(start, PAGESIZE, prot, flags | MAP_FIXED, -1, 0); /* 必ず成功する */
11:     assert(ptr != MAP_FAILED);
12:     ptr = true_mremap(start, PAGESIZE, length, 0); /* start からの 1 ページを，length バイトへと拡張しようと試みる */
13:     if(ptr == MAP_FAILED) { /* 失敗したら */
14:       ret = true_munmap(start, PAGESIZE); /* 後片付け */
15:       assert(ret == 0);
16:     }
17:   } else if(ptr != MAP_FAILED) {  /* 最初の mremap に成功してしまったら */
18:     ret = true_munmap(start + PAGESIZE, PAGESIZE); /* 後片付け */
19:     assert(ret == 0);
20:     ptr = MAP_FAILED; /* fixed_mmap() は失敗 */
21:   }
22:   pthread_mutex_unlock(&mutex);
23:   return ptr;
24: }
```

図 10　fixed_mmap() 関数のアルゴリズム
Fig. 10　An algorithm of fixed_mmap().

（使用中，1 ページ目と同じ属性で使用中）　返り値は MAP_FAILED，errno は ENOMEM.
（使用中，1 ページ目とは異なる属性で使用中）　返り値は MAP_FAILED，errno は ENOMEM.

したがって，9 行目の if 文の条件を満たすのは（未使用，未使用）の場合のみである．つづいて 10 行目では，1 ページ目に対して true_mmap() 関数が MAP_FIXED オプション付きで発行され，1 ページ目のアドレス領域が確実に割り当てられる．いまは 1 ページ目が未使用であることが 9 行目の if 文により保証されているため，10 行目の true_mmap() 関数によって既存のアドレス領域が破壊されることはない．これにより，（使用中，未使用）の状態に変化する．次に 12 行目の true_mremap() 関数により，いま割り当てた 1 ページ目のアドレス領域を length バイトにリサイズすることを試みる．libc_mremap() 関数の仕様[4] によれば，これが成功するのは，アドレス領域 [start, start + length) が使用されていない場合のみである．そしてこれは，いま実現すべき fixed_mmap() 関数の仕様にほかならない．

8. 性能評価

8.1 シミュレーションによる random-address の評価

random-address のアルゴリズムを評価するため，シミュレーションによって，さまざまなパラメータに対するスレッド移動時のアドレス衝突確率を調べた．シミュレーションを用いた理由は，実際にスレッド移動を行って評価すると時間がかかりすぎるうえ，評価できるスレッド数やメモリ量の規模が実際のマシンの資源量に制限されてしまうためである．本シミュレーションでは，乱数として，原始多項式 $x^{521} + x^{32} + 1$ に基づく 64 ビットの M 系列乱数を使用した．

次のような状況を考える．利用可能なアドレス空間の大きさを $2^{address}$ バイトとし，系内に *process* 個のプロセスが存在するとする．また，各プロセスは合計 *memory* バイトのローカルアドレス空間を使用しているとする．いい換えると，各プロセス p に関して，そのプロセス p 内に存在するすべてのスレッド i が使用している $register_i^p$ と $stack_i^p$ と $threadheap_i^p$ のメモリ量の総和が *memory* バイトであるとする．さらに，各プロセスは合計 *memory* バイトのローカルアドレス空間を割り当てるために，大きさが等しい *chunk* 個の離散化されたアドレス領域を使用しているとする．いい換えると，各プロセスがローカルアドレス空間として使用する合計 *memory* バイトのアドレス領域は，アドレス空間上で *chunk* 個の小アドレス領域に分かれており，この各小アドレス領域の大きさはすべて *memory/chunk* バイトであるとする．$chunk = 1$ の場合が，アドレス領域を連続的に使用する場合に相当する．そして，この *chunk* 個の小アドレス領域の先頭アドレスは，*align* の整数倍の値の中からランダムに選ばれるとする．

本シミュレーションでは，以上のような状況において，すべてのプロセスに含まれるすべてのスレッドを，ある 1 個のノード P の中へとスレッド移動させるとき，ノード P の中に何個のプロセスが生成されるかを，さまざまな *address*，*process*，*memory*，*chunk*，*align* の値に対して測定した．以下では，このとき生成されるプロセス数を，*address*，*process*，*memory*，*chunk*，*align* の関数として，$N(address, process, memory, chunk, align)$ と表す．当然，$N(address, process, memory, chunk, align)$ が小さいほどアドレス衝突確率が小さいことを意味し，$N(address, process, memory, chunk, align)$ が大きいほどアドレス衝突確率が大きいことを意味する．なお，すべてのプロセスに含まれるすべてのスレッドをノード P の中へとスレッド移動させるとき，これらのスレッドをどのような順番でノード P へ移動させるかに応じて，ノード P に生成されるプロセス数は変化

しうるが，本シミュレーションではランダムな順序ですべてのスレッドを移動させた．このようなシミュレーションを，($address$，$process$，$memory$，$chunk$，$align$) の各組合せに対して 10 回行い，測定された 10 個の値の最小値，最大値，平均値を算出した．この最小値，最大値，平均値をそれぞれ，$N_{\min}(address, process, memory, chunk, align)$，$N_{\max}(address, process, memory, chunk, align)$，$N_{\mathrm{avg}}(address, process, memory, chunk, align)$ と表す．

図 11，**図 12**，**図 13**，**図 14**，**図 15**，**図 16**，**図 17**，**図 18**，**図 19**，**図 20** に，シミュレーション結果のグラフを示す．これらのすべてのグラフでは $align = 1$ としている．また，各グラフが 1 個の $chunk$ の値に対応しており，各グラフ中の 1 個の折れ線が 1 個の $process$ の値に対応している[⋆1]．各折れ線中の 1 個の点は，$N_{\mathrm{avg}}(address, process, memory, chunk, align)$ をプロットしており，その点に対するエラーバーの下限が $N_{\min}(address, process, memory, chunk, align)$ を，エラーバーの上限が $N_{\max}(address, process, memory, chunk, align)$ をプロットしている．具体例をあげると，図 11 のグラフにおいて最も右下の赤い点は，「『アドレス空間全体が 2^{32} バイトの環境に 4 個のプロセスがあり，各プロセスが 2^{32} バイトを使用している．また，各プロセスはその 2^{32} バイトを，($chunk = 1$ なので) 1 個の連続的なアドレス領域として確保している．さらに，($align = 1$ なので) その連続的なアドレス領域の先頭アドレスはランダムに決まっている』という状況において，これら 4 個のプロセスに存在するすべてのスレッドを，1 個のノードにランダムな順序で詰め込むとき，そのノードに生成されたプロセス数の平均値」を表している．ただし，いまの場合，「2^{32} バイトのアドレス空間から，2^{32} バイトの 1 個の連続的なアドレス領域を確保する方法」はアドレス領域 $[0, 2^{32})$ を確保する方法の 1 通りしか存在せず，ランダム性はない．そして，アドレス領域 $[0, 2^{32})$ を使用している 4 個のプロセスに関して，それらのプロセスに存在するすべてのスレッドを 2^{32} バイトのアドレス空間を持つ 1 個のノードに移動させたとすれば，当然，生成されるプロセス数は必ず 4 個になる．したがって，図 11 のグラフにおいて最も右下の赤い点の値は 4 であり，エラーバーの下限も上限も 4 になっている．なお，6.4 節で述べたように，各プロセス内では各スレッドが使用するアドレス領域が重ならないようなアドレス空間管理が行われるため，各プロセス内に何個のスレッドが存在するかは問題にならない．また，$chunk$ の値は $memory$ の値以下である必要があるため，図 11 から図 20 のグラフでは，$chunk$ の値が大きくなるにつ

⋆1 グラフ中では $process$ を p と略記している．

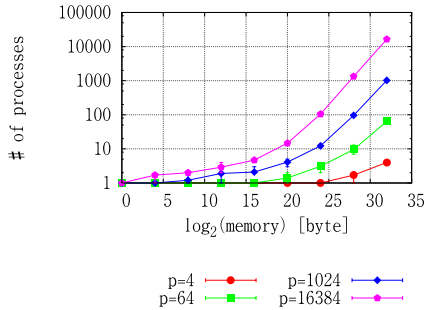

図 11　$N(32, process, memory, 1, 1)$
Fig. 11　$N(32, process, memory, 1, 1)$.

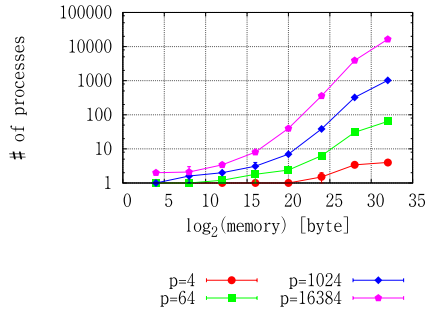

図 12　$N(32, process, memory, 16, 1)$
Fig. 12　$N(32, process, memory, 16, 1)$.

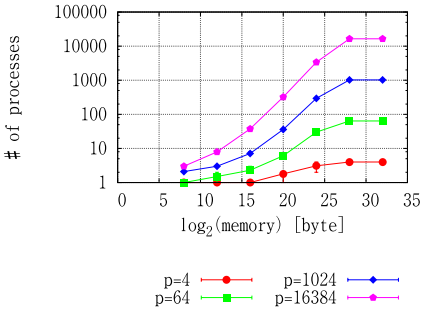

図 13　$N(32, process, memory, 256, 1)$
Fig. 13　$N(32, process, memory, 256, 1)$.

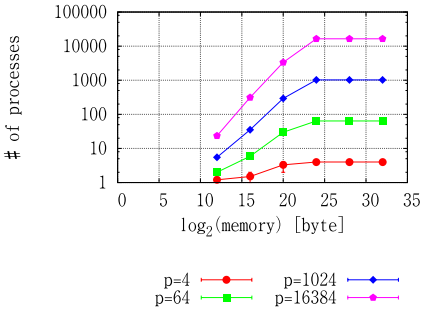

図 14　$N(32, process, memory, 4096, 1)$
Fig. 14　$N(32, process, memory, 4096, 1)$.

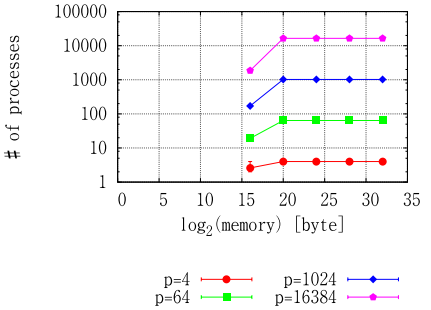

図 15　$N(32, process, memory, 65536, 1)$
Fig. 15　$N(32, process, memory, 65536, 1)$.

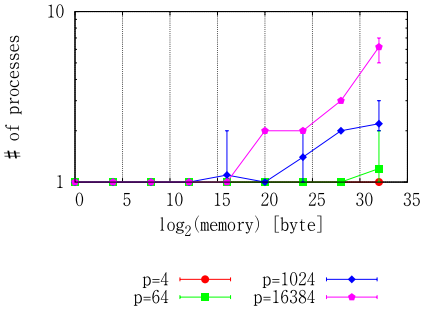

図 16　$N(47, process, memory, 1, 1)$
Fig. 16　$N(47, process, memory, 1, 1)$.

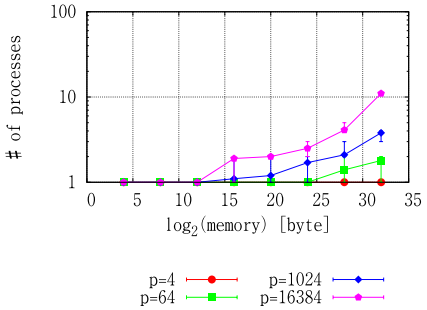

図 17　$N(47, process, memory, 16, 1)$
Fig. 17　$N(47, process, memory, 16, 1)$.

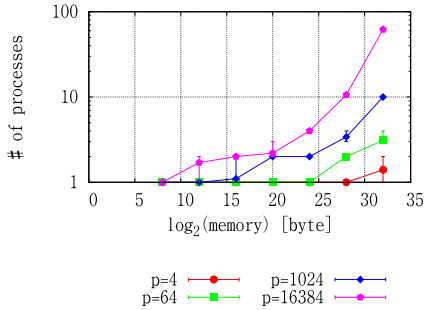

図 18　$N(47, process, memory, 256, 1)$
Fig. 18　$N(47, process, memory, 256, 1)$.

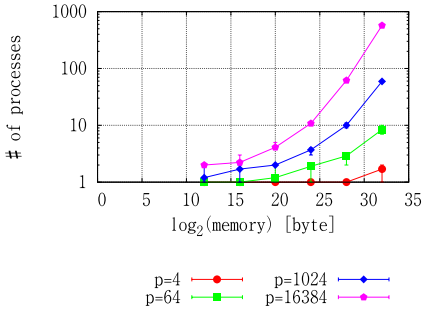

図 19　$N(47, process, memory, 4096, 1)$
Fig. 19　$N(47, process, memory, 4096, 1)$.

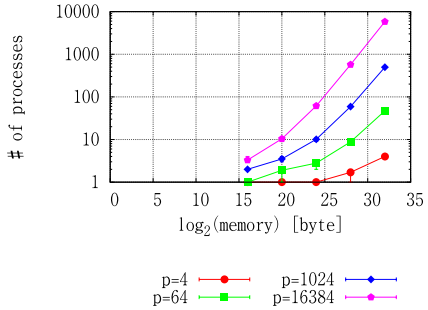

図 20　$N(47, process, memory, 65536, 1)$
Fig. 20　$N(47, process, memory, 65536, 1)$.

れて，折れ線の左側が存在しなくなっている．*addr* として 2^{32} と 2^{47} を使用しているのは，それぞれ，既存の多くの 32 ビットアーキテクチャと 64 ビットアーキテクチャで使用可能なアドレス空間をシミュレートするためである．

第 1 に，このシミュレーション結果より以下の事実が読み取れる：

(1)　*align* と *memory* と *chunk* と *process* を固定したとき，*addr* が増加するほどアドレス衝突確率が小さい．

(2)　*address* と *align* と *chunk* と *process* を固定したとき，*memory* が増加するほどアドレス衝突確率が大きい．

(3)　*address* と *align* と *chunk* と *memory* を固定したとき，*process* が増加するほどアドレス衝突確率が大きい．

(4)　*address* と *align* と *memory* と *process* を固定したとき，*chunk* が増加するほどアドレス衝突確率が大きい．

ここで，(1) と (2) と (3) は定性的に考えて明らかである．一方，(4) は，「各プロセスのアドレス領域が連続的に使用されるとき，アドレス衝突確率が最も小さく，各プロセスの使用するアドレス領域が離散化するに従ってアドレス衝突確率が大きくなる」ことを述べている．すなわち，この結果は，6.2 節で述べた random-address の戦略が確かに最適であることを裏付けている．なお，random-address の最適性の正確な証明は付録 A.1 で行う．

第 2 に，図 16 と図 20 より以下の定量的な事実が読み取れる：

- 2^{47} バイトのアドレス空間において random-address を用いれば，16384 個のプロセスがそれぞれ 2^{32} バイトのローカルアドレス空間を割り当てたとしても，これらのプロセス内のすべてのスレッドをわずか平均 6.2 個のアドレス空間に詰め込むことができる（図 16 における最も右上の点）．つまり，2^{47} バイトのアドレス空間で 16384 個のプロセスを生成する程度の計算規模ではアドレス衝突はほぼ起きないといえ，random-address は今後の大規模な計算環境においても適用可能な手法であることが分かる．
- 2^{47} バイトのアドレス空間において 16384 個のプロセスがそれぞれ 2^{32} バイトのローカルアドレス空間を割り当てるとき，各プロセスがそのローカルアドレス空間を 65536 個の離散的な小アドレス領域として割り当てると，これらのプロセス内のすべてのスレッドを詰め込むのに必要なアドレス空間の数は平均 5828 個にもなる（図 20 における最も右上の点）．この結果より，random-address ではアドレスを連続的に割り当てることにより，アドレス衝突確率を大幅に下げられることが分かる．

第 3 に，6.3 節で述べた，*align* とアドレス衝突確率の関係について調べる．**図 21** には，以下の 2 つのグラフを示す：

- *align* を 2^0 バイトから 2^{32} バイトまで変化させたときの，$N_{\min}(32, 1024, 2^{30}, 1, align)$ と $N_{\max}(32, 1024, 2^{30}, 1, align)$ と $N_{\text{avg}}(32, 1024, 2^{30}, 1, align)$.
- *align* を 2^0 バイトから 2^{32} バイトまで変化させたときの，$N_{\min}(32, 1024, 2^{20}, 1024, align)$ と $N_{\max}(32, 1024, 2^{20}, 1024, align)$ と $N_{\text{avg}}(32, 1024, 2^{20}, 1024, align)$.

図 21 より，$N_{\text{avg}}(32, 1024, 2^{30}, 1, align)$ は $align = 2^{30}$ において，$N_{\text{avg}}(32, 1024, 2^{20}, 1024, align)$ は $align = 2^{10}$ において，アドレス衝突確率が最小になることが分かる．この理由は，$N_{\text{avg}}(32, 1024, 2^{30}, 1, align)$ では各小アドレス領域の大きさが $memory/chunk = 2^{30}$ であり，$N_{\text{avg}}(32, 1024, 2^{20}, 1024, align)$ では $memory/chunk = 2^{10}$ であり，6.3 節で

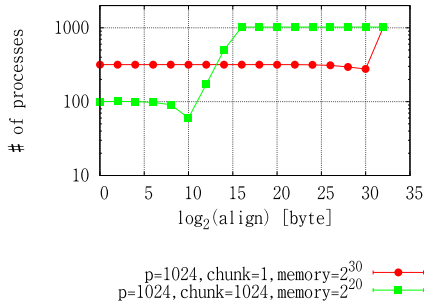

図 21　$N(32, 1024, 2^{30}, 1, align)$, $N(32, 1024, 2^{20}, 1024, align)$
Fig. 21　$N(32, 1024, 2^{30}, 1, align)$, $N(32, 1024, 2^{20}, 1024, align)$.

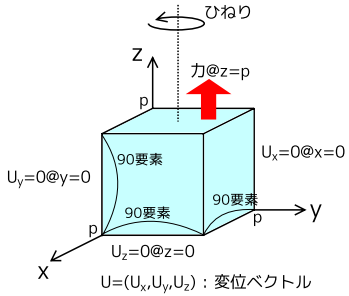

図 22　有限要素法による応力解析
Fig. 22　Stress analysis using a finite element method.

述べたように，すべてのスレッドが $memory/chunk$ バイトを使用するならば，$align = memory/chunk$ のときにアドレス衝突確率が最小になるためである．

8.2　アプリケーションベンチマーク

8.2.1　実 験 環 境

実験環境としては，Intel Xeon E5530 2.40 GHz（4 コア）×2 の CPU，24 GB のメモリ，カーネル 2.6.26-2-amd64 の Linux で構成されるマシン 16 ノードを，10 Gbit 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．コンパイラには gcc 4.3.2，最適化オプションには-O3，MPI には OpenMPI 1.4.2 と mpich2-1.2.1p1 を使用した．以降の実験では，DMI/MPI を n プロセッサで実行する際には，8 個のスレッド/プロセスを $\lfloor n/8 \rfloor$ 台のノードに生成し，残りの $n - 8 \times \lfloor n/8 \rfloor$ 個のスレッド/プロセスを別の 1 個のノードに生成するプロセス構成とした．

8.2.2　有限要素法による応力解析

3 次元立方体物体に対して，**図 22** に示すような境界条件を課したときの応力解析を有限要素法で行った．この有限要素法では，3 次元立方体が 90^3 個の要素に分割されており，各要素に対しては Sequential Gauss Algorithm に基づいて z 軸回りに最大 200 度のひねりが加えられている．この有限要素法は，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[7] の題材として使用された，実際の工学に基づく実用的な並列科学技術計算である．非常に収束させにくい問題であるため各種の高度な工学的手法が必要となる．

アルゴリズムの概要は以下のとおりである．詳細は著者らの資料[64] を参考にされたい：

（1）　立方体物体をプロセッサ数個の領域に分割する．このとき，立方体を単純に直方体領域の集合へと分割するのではなく，領域間オーバラップやフィルインを考慮したとき領域間の負荷が均等化するような非定型な領域分割を行う．また，収束性を改善させるため，修正 RCM オーダリング[40] によって領域内の節点（各要素における 8 個のコーナ点のこと）を並べ替える．

（2）　与えられている節点間の結合関係を連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ として表現する．ここで A は節点間の結合関係を表す疎行列，$\boldsymbol{b}$ は境界条件を表すベクトル，$\boldsymbol{x}$ は求めるべき変位ベクトルである．

（3）　連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ に関して，BiCGSafe 法[28] と呼ばれる反復法を用いて解 $\boldsymbol{x}$ が収束するまで反復計算を行う．各イテレーションの先頭では，収束性を改善させるために，フィルインレベル 3 のブロック不完全 LU 分解による前処理と，領域間オーバラップ 2 による Restricted Additive Schwarz 法[13] に基づく前処理を適用する．

（4）　さらに，各イテレーションの先頭において，`DMI_yield()` 関数を呼び出す．

第 1 に，DMI のプログラマビリティを評価した．スレッド移動に対応した DMI のプログラムは，スレッド移動に非対応の通常の SPMD 型の DMI のプログラムを変更することで実装したが，この際の変更点は，（1）各イテレーションの先頭に `DMI_yield()` 関数を記述すること，（2）`DMI_yield()` 関数をまたいで使用されるヒープ領域の確保/解放を `DMI_thread_malloc()` 関数/`DMI_thread_realloc()` 関数/`DMI_thread_free()` 関数に置き換えること，の 2 点のみだった．すなわち，DMI では，通常の SPMD 型の並列科学技術計算に対してわずかな変更を加えるだけで，計算規模を拡張/縮小可能な並列プログラムが得られることを確認できた．

第 2 に，スレッド移動を行わない場合の DMI の基本的な性能を MPI と比較した．**図 23** と**図 24** に，実行するプロセッサ数を変化させたときの DMI，mpich2，OpenMPI の実行時間とウィークスケーラビリティを示す．mpich2 と OpenMPI のプログラムとしては，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における優勝プログラムを使用した．図 23 と図 24 より，DMI は mpich2 と同等の性能を達成し，OpenMPI よりも高い性能を達成していることが分かる．なお，OpenMPI の性能の低さは，OpenMPI における 1 対 1 通信の send/receive 操作の遅さに起因している．たとえば，65,536 バイトのデータを 2 個のノード間で 10,000 回送受信する実験を行った場合，mpich2 では 4.55 秒を要するが，OpenMPI では 10.7 秒も要する．

第 3 に，DMI における計算規模の拡張/縮小の性能を評価した．DMI において 128 個の

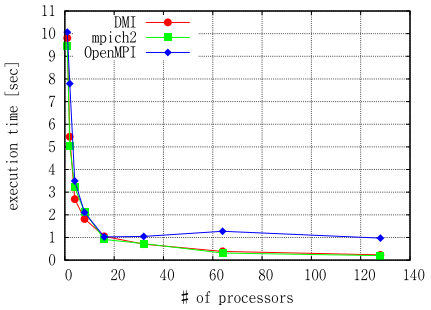

図 23　プロセッサ数を変化させた場合の，有限要素法の実行時間
Fig. 23　The execution time of the finite element method.

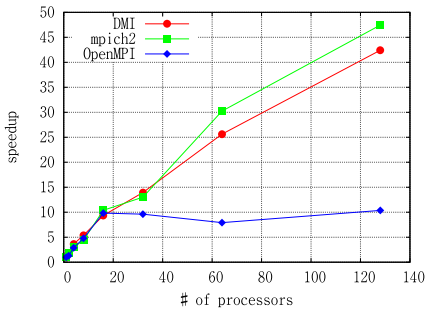

図 24　有限要素法のウィークスケーラビリティ
Fig. 24　The weak scalability of the finite element method.

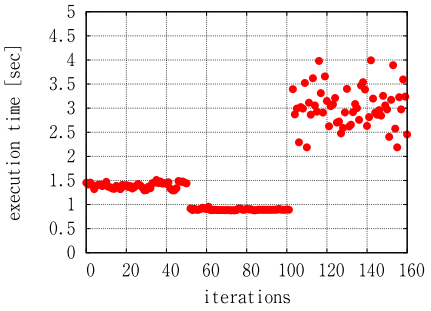

図 25　有限要素法において，計算規模を動的に拡張/縮小した場合における各イテレーションの実行時間
Fig. 25　The execution time of each iteration of the finite element method with dynamic scale-up and scale-down.

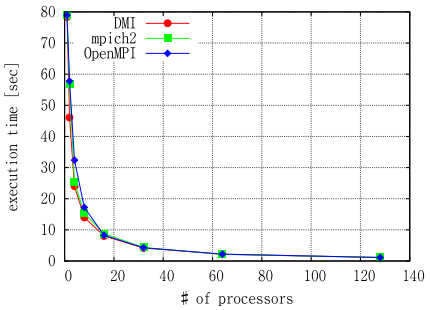

図 26　プロセッサ数を変化させた場合の，N 体問題の実行時間
Fig. 26　The execution time of an N-body problem.

スレッドを生成し，(1) 初期的にはノード 0〜 ノード 7 で実行し（合計 8 ノード，64 プロセッサ），(2) 第 50 イテレーション終了直後にノード 8〜 ノード 15 の 8 ノードを参加させ（合計 16 ノード，128 プロセッサ），(3) 第 101 イテレーション終了直後にノード 0〜 ノード 11 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．このときの各イテレーションの実行時間を**図 25** に示す．この実験では，各イテレーションの実行時間をある程度大きくするため，有限要素法の要素数を 90^3 ではなく 150^3 とした．各スレッドのローカルアドレス空間としては 510 MB〜520 MB が使用され，大域アドレス空間としては 335 MB が使用されていた．図 25 より，利用可能な計算資源の増減に対応して適応的に並列度を増減できていることが読み取れる．合計 4 ノードで実行した場合の実行時間が揺れている理由は，各プロセッサあたり 4 個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．この有限要素法で使用している BiCGSafe 法では，1 イテレーションあたり 22 回の同期が必要であり，この各同期のたびに最も遅いスレッドに律速されるため，スレッドスケジューリングのばらつきの影響が現れやすいものと考えられる．スレッド移動に要した時間としては，(2) における 8 ノードの参加時には，8 ノードの参加処理と 120 スレッドのスレッド移動（=57.6 GB のメモリ移動）が発生して 17.3 秒を要し，(3) における 12 ノードの脱退時には，12 ノードの脱退処理と 120 スレッドのスレッド移動（=57.6 GB のメモリ移動）が発生して 30.9 秒を要した．なお，8 ノードを参加させる際に 120 スレッドものスレッド移動が発生する理由は，現実装ではスレッドスケジューリングを最適化しておらず，単純に，計算規模が変化した時点で存在しているプロセスに対して，スレッド番号が若い方から順に，均等な数ずつスレッドを割り当てているためである．また，この実験では $align = 1$ としたが，多数回の実験を通じてスレッド移動にともなうアドレス衝突は 1 度も発生しなかった．まとめると，DMI では，実用的な並列科学技術計算に対して，通常の SPMD 型の並列プログラムをわずかに変更するだけで，利用可能な計算資源の増減にともなって適応的に並列度を増減させられることを確認できた．

8.2.3　N 体 問 題

N 体問題を題材にして評価を行った．N 体問題のアルゴリズムの概要は以下のとおりである：

(1)　3 次元格子状に並んだ $l_1 \times l_2 \times l_3$ 個の粒子を考え，各粒子に位置と速度の情報を持たせる．各粒子に対して適当な初期位置と初速度を与える．
(2)　n 個のプロセッサで実行する場合，$l_1 \times l_2 \times l_3$ 個の格子状に並んだ粒子を l_1 の方向に n 等分し，i 番目のプロセッサには i 番目の領域を担当させる．つまり，各プロセッサに $l_1 \times l_2 \times l_3/n$ 個の粒子を担当させる．
(3)　以下の処理を反復する：
　(a)　各プロセッサが，そのプロセッサの担当範囲の粒子の位置をすべてのプロセッサに対して送信する Allgather 型の通信を行う．これにより，すべてのプロセッサに全粒子の位置を把握させる．

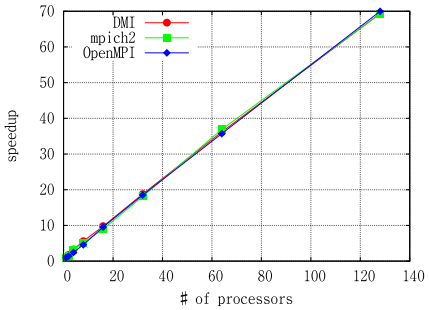

図 27　N 体問題のウィークスケーラビリティ
Fig. 27　The weak scalability of the N-body problem.

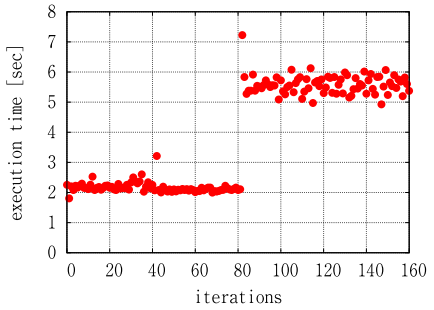

図 28　N 体問題において，計算規模を動的に拡張/縮小した場合における各イテレーションの実行時間
Fig. 28　The execution time of each iteration of the N-body problem with dynamic scale-up and scale-down.

（b）　各プロセッサは，全粒子の位置に基づいて，そのプロセッサの担当範囲の粒子と全粒子との相互作用を計算し，担当範囲の粒子の位置と速度を更新する．

第 1 に，スレッド移動を行わない場合の DMI の基本的な性能を MPI と比較した．**図 26** と**図 27** に，実行するプロセッサ数を変化させたときの DMI，mpich2，OpenMPI の実行時間とウィークスケーラビリティを示す．この実験では $l_1 = l_2 = l_3 = 24$ とした．図 26 と図 27 より，DMI は，mpich2 や OpenMPI と同等のスケーラビリティを達成できていることが分かる．

第 2 に，DMI における計算規模の拡張/縮小の性能を評価した．DMI において 128 個のスレッドを生成し，（1）初期的にはノード 0〜 ノード 7 で実行し，（2）第 41 イテレーション終了直後にノード 8〜 ノード 15 の 8 ノードを参加させ，（3）第 82 イテレーション終了直後にノード 0〜 ノード 11 の 12 ノードを脱退させる，というように利用可能な計算資源を動的に増減させた．このときの各イテレーションの実行時間を**図 28** に示す．この実験では $l_1 = 24$，$l_2 = l_3 = 28$ とした．各スレッドのローカルアドレス空間としては 9.7 MB が使用され，大域アドレス空間としては 441 KB が使用されていた．スレッド移動に要した時間としては，（2）における 8 ノードの参加時には，8 ノードの参加処理と 120 スレッドのスレッド移動（=1.19 GB のメモリ移動）が発生して 2.22 秒を要し，（3）における 12 ノードの脱退時には，12 ノードの脱退処理と 120 スレッドのスレッド移動（=1.19 GB のメモリ移動）が発生して 5.51 秒を要した．また，この実験では $align = 1$ としたが，多数回の実験を通じてスレッド移動にともなうアドレス衝突は 1 度も発生しなかった．

9. 結　　論

9.1 ま と め

本稿では，長時間を要するような大規模な並列科学技術計算を実行できるプラットフォームをクラウドサービスとして効率的に提供するためには，利用可能な計算資源の増減に対応して，並列計算の規模を動的に拡張/縮小できるような仕組みが必要であることを指摘した．また，そのような並列計算を簡単に記述できる並列分散プログラミング処理系として，グローバルビュー型の PGAS モデルに基づいた DMI を実装して評価した．DMI では，プログラマは十分な数のスレッドを生成するだけでよく，あとは処理系が，透過的にそれら大量のスレッドを利用可能な計算資源に対してスケジューリングしてくれる．

本稿の貢献は以下のとおりである：

- DMI の主要な要素技術としてスレッド移動手法について検討し，今後ますます増大する計算規模に対応していくためには，従来のスレッド移動手法では限界に達する可能性があることを指摘したうえで，アドレス空間の大きさに制限されないスレッド移動手法として random-address を提案した．また，random-address の最適性について数学的な証明を与えるとともに，シミュレーションによってその最適性を確認した．random-address は，DMI がプロセスの動的な参加/脱退に対応しているからこそ実現できるスレッド移動手法である．
- random-address の実装にともなって，ユーザレベルでスレッドのチェックポイント/リスタートを行う手法，共有ライブラリのコードを動的に書き換えることでシステムコールをハイジャックする手法を提案した．
- 評価の結果，有限要素法を用いた応力解析という実用的な並列科学技術計算に対して，（1）通常の SPMD 型の並列プログラムに対する変更点がわずかであること，（2）利用可能な計算資源の増減にともなって適応的に並列度を変化させられることを確認した．

著者らの知るかぎり，グローバルビュー型の PGAS モデルに基づいて，ユーザプログラムから透過的に計算規模の拡張/縮小を実現した研究は存在しない．本稿が，アドレス空間の大きさに制限されないスレッド移動手法を新たに提案し，それを DMI の要素技術として適用することで，計算規模の拡張/縮小を透過的に実現する PGAS 処理系を設計・実装・評価している点には新規性があるといえる．

9.2　今後の課題

現時点では単純なスレッドスケジューリングしか行っていないが，DMI のように透過的にスレッドをスケジューリングする処理系においては，スレッドスケジューリングの最適化が性能上きわめて重要である．DMI においてスレッドスケジューリングを最適化するには，ノード間のスレッドの負荷バランス，スレッド移動に要する時間，プロセス数を増やすことによるオーバヘッド，各スレッド間でのデータ共有の度合いなどの要素を総合的に考慮する必要がある．DMI では，DMI の API を通じて大域アドレス空間にアクセスすることになっているため各スレッド間の大域アドレス空間の共有度合いを容易に把握できるほか，メモリ確保/解放関連のシステムコールもハイジャックしているため各スレッドのローカルアドレス空間の使用状況も容易に把握できるなど，スレッドスケジューリングにとって必要な多くの情報を取得できる．したがって，これらの情報に基づいてスレッドスケジューリングを最適化する必要がある．

謝辞　本研究の一部は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する高度にスケーラブルなソフトウェア構成基盤」の助成を得て行われた．

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) Google Docs [Online]. http://docs.google.com/
4) Linux Manpages [Online]. http://linuxmanpages.com/
5) Salesforce.com [Online]. http://www.salesforce.com/
6) Windows Azure [Online]. http://www.microsoft.com/windowsazure/
7) 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/
8) Amza, C., Cox, A.L., Dwarkadas, H., Keleher, P., Lu, H., Yu, W., Rajamony, R. and Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, Vol.29, No.2, pp.18–28 (1996).
9) Antoniu, G., Bouge, L. and Namyst, R.: An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System, *Proc. 11th 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).
10) Antoniu, G. and Perez, C.: Using Preemptive Thread Migration to Load-Balance Data-Parallel Applications, *Proc. 5th International Euro-Par Conference on Parallel Processing*, pp.117–124 (1999).
11) Salama, R.A. and Sameh, A.: *Potential Performance Improvement of Collective Operations in UPC*, John von Neumann Institute for Computing (2007).
12) Buyya, R., Yeo, C.-S., Venugopal, S., Broberg, J. and Brandic, I.: 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 (2008).
13) Cai, X.-C. and Sarkis, M.: A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems, *SIAM Journal on Scientific Computing*, Vol.21, No.2, pp.792–797 (1999).
14) Carlson, W., Sterling, T., Yelick, K. and El-Ghazawi, T.: *UPC Distributed Shared Memory Programming*, Wiley Inter-Science (June 2005).
15) Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data, *ACM Trans. Computer Systems*, Vol.26 (2008).
16) Chaudhary, V. and Jiang, H.: Techniques for Migrating Computations on the Grid, *Engineering the Grid: Status and Perspective*, pp.399–415 (Jan. 2006).
17) Chen, P.-C., Lin, C.-I., Huang, S.-W., Chang, J.-B., Shieh, C.-K. and Liang, T.-Y.: A Performance Study of Virtual Machine Migration vs. Thread Migration for Grid Systems, *International Conference on Advanced Information Networking and Applications* (Mar. 2008).
18) Christopher, B., Clin, C., George, A., Yili, Z., Montse, F., Siddhartha, C. and Nelson, A.J.: Shared memory programming for large scale machines, *ACM SIGPLAN Notices*, Vol.41, No.6, pp.108–117 (2006).
19) Clark, C., Fraser, K., Hand, S., Hansenf, J.G., Julf, E., Limpach, C., Pratt, I. and Warfield, A.: Live migration of virtual machines, *Proc. 2nd conference on Symposium on Networked Systems Design and Implementation*, Vol.2, pp.273–286 (2005).
20) Coarfa, C., Dotsenko, Y., Eckhardt, J. and Mellor-Crummey, J.: Co-Array Fortran Performance and Potential: An NPB Experimental Study, *16th Internatilnal Workshop on Languages and Compilers for Parallel Computing*, Vol.2958, pp.177–193 (2004).
21) Coarfa, C., Dotsenko, Y., Mellor-Crummey, J., Cantonnet, F., El-Ghazawi, T., Mohanty, A., Yao, Y.-Y. and Chavarria-Miranda, D.: An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C, *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.36–47 (2005).
22) Cronk, D., Haines, M. and Mehrotra, P.: Thread Migration in the Presence of Pointers, *Proc. 30th Hawaii International Conference on System Sciences: Software Technology and Architecture*, Vol.1, pp.292–302 (1997).
23) Datta1, K., Bonachea1, D. and Yelick1, K.: Titanium Performance and Poten-

tial: An NPB Experimental Study, *18th International Workshop on Languages and Compilers for Parallel Computing*, Vol.4339, pp.200–214 (2006).
24) Dimitrov, B. and Reg, V.: Arachne: A portable threads system supporting migrant threads on heterogeneous network farms, *IEEE Trans. Parallel and Distributed Systems*, Vol.9, pp.459–469 (1998).
25) Du, C. and Sun, X.-H.: MPI-Mitten: Enabling Migration Technology in MPI, *IEEE International Symposium on Cluster Computing and the Grid*, pp.11–18 (May 2006).
26) Duell, J.: The design and implementation of Berkeley Lab's linuxcheckpoint/restart, Technical report, Ernest Orlando Lawrence Berkeley National Laboratory (Apr. 2005).
27) Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 2.2, Technical report, Message Passing Interface Forum (Sep. 2009).
28) Fujino, S., Fujiwara, M. and Yoshida, M.: BiCGSafe Method Based on Minimization of Associate Residual (in Japanese), *Trans. Japan Society for Computational Engineering and Science*, Vol.8, pp.145–152 (2006).
29) Hara, K. and Taura, K.: A Global Address Space Framework for Irregular Applications (accepted, short paper), *High Performance Distributed Computing* (June 2010).
30) Hilfinger, P., Bonachea, D., Gay, D., Graham, S., Liblit, B., Pike, G. and Yelick, K.: Titanium Language Reference Manual, Technical report, University of California at Berkeley (Aug. 2006).
31) Huang, C., Lawlor, O. and Kale, L.V.: Adaptive MPI, *International workshop on languages and compilers for parallel computing*, Vol.2958, pp.306–322 (2003).
32) Itzkovitz, A., Schuster, A. and Shalev, L.: Thread Migration and its Applications in Distributed Shared Memory Systems, *Journal of Systems and Software*, Vol.42, No.1, pp.71–87 (1998).
33) Jiang, H. and Chaudhary, V.: Compile/Run-Time Support for Thread Migration, *Proc. 16th International Parallel and Distributed Processing Symposium*, pp.58–66 (2002).
34) Jiang, H. and Chaudhary, V.: MigThread: Thread Migration in DSM Systems, *International Conference on Parallel Processing*, p.581 (2002).
35) Jiang, H. and Chaudhary, V.: On Improving Thread Migration: Safety and Performance, *Proc. 9th International Conference on High Performance Computing*, pp.474–484 (2002).
36) Jiang, H. and Chaudhary, V.: Thread Migration/Checkpointing for Type-Unsafe C Programs, *International conference on high performance computing*, Vol.2913, pp.469–479 (2003).
37) Ma, M.J.M., Wang, C.-L. and Lau, F.C.M.: Delta Execution: A preemptive Java thread migration mechanism, *Cluster Computing*, Vol.3, pp.83–94 (2000).
38) Ke, J. and Speight, E.: Tern: Thread Migration in an MPI Runtime Environment, Technical report, Cornell (Nov. 2001).
39) Thitikamol, K. and Keleher, P.: Thread migration and communication minimization in DSM systems, *Proc. IEEE, Special Issue on Distributed Shared Memory*, Vol.87, pp.487–497 (1999).
40) Liu, W.-H. and Sherman, A.H.: Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices, *SIAM Journal on Numerical Analysis*, Vol.13, No.2, pp.198–213 (1976).
41) Johnson, K.L., Kaashoek, M.F. and Wallach, D.A.: CRL: High-Performance All-Software Distributed Shared Memory, *Proc. 15th Symposium on Operating Systems Principles*, Vol.29, No.5, pp.213–228 (1995).
42) Vaquero, L., Rodero-Marino, L., Caceres, J. and Lindner, M.: A Break in the Clouds: Towards a Cloud Definition, *SIGCOMM Computer Communication Review*, pp.137–150 (2009).
43) Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I. and Zaharia, M.: Above the Clouds: A Berkeley View of Cloud Computing, Technical report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory (Feb. 2009).
44) El Maghraoui, K., Szymanski, B.K. and Varela, C.: An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments, *International Conference on Parallel Processing and Applied Mathematics*, Vol.3911, pp.258–271 (2006).
45) Milton, S.: Thread Migration in Distributed Memory Multicomputers, Technical report, Australia National University (1998).
46) Mueller, F.: Distributed Shared-Memory Threads: DSM-Threads, *Workshop on Run-Time Systems for Parallel Programming*, pp.31–40 (Apr. 1997).
47) Mueller, F.: On the Design and Implementation of DSM-Threads, *Conference on Parallel and Distributed Processing Techniques and Applications*, pp.315–324 (June 1997).
48) Nieplocha, J., Krishnan, M., Palmer, B. and Tipparaju, V.: The Global Arrays User's Manual, Technical report, Pacific Northwest National Laboratory (July 2009).
49) Numrich, R.W. and Reid, J.: Co-array Fortran for parallel programming, *ACM SIGPLAN Fortran Forum*, Vol.17, No.2, pp.1–31 (1998).
50) Hines, M.R. and Gopalan, K.: Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning, *Proc. 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*,

pp.51–60 (2009).
51) Roblitz, T. and Mueller, F.: Combining Multi-Threading with Asynchronous Communication: A Case Study with DSM-Threads using Myrinet via BIP and Madeleine, *Myrinet User Group Conference*, pp.131–138 (Sep. 2000).
52) Sankaran, S., Squyres, J.M., Barrett, B., Sahay, V. and Lumsdaine, A.: The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing, *International Journal of High Performance Computing Applications*, Vol.19, pp.479–493 (2005).
53) Sievert, O. and Casanova, H.: A Simple MPI Process Swapping Architecture for Iterative Applications, *International Journal of High Performance Computing Applications*, Vol.18, pp.341–352 (2004).
54) Su, J., Wen, T. and Yelick, K.: Compiler and Runtime Support for Scaling Adaptive Mesh Refinement Computations in Titanium, Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley (June 2006).
55) Su, J. and Yelick, K.: Automatic Support for Irregular Computations in a High-Level Language, *19th IEEE International Parallel and Distributed Processing Symposium*, Vol.1, p.53b (2005).
56) Wang, C., Mueller, F., Engelmann, C. and Scott, S.L.: Proactive process-level live migration in HPC environments, *Proc. 2008 ACM/IEEE Conference on Supercomputing*, pp.1–12 (Nov. 2008).
57) Weissman, B., Gomes, B., Quittek, J.W. and Holtkamp, M.: Efficient Fine-grain Thread Migration with Active Threads, *Proc. 12th International Parallel Processing Symposium on International Parallel Processing Symposium*, p.410 (1998).
58) Numrich, R.W., Reid, J. and Kim, K.: Writing a Multigrid Solver Using Co-array Fortran, *4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems*, Vol.1541, pp.390–399 (1998).
59) Yelick, K., Hilfinger, P., Graham, S., Bonachea, D., Su, J., Kamil, A., Datta, K., Colella, P. and Wen, T.: Parallel Languages and Compilers: Perspective from the Titanium Experience, *Journal of High Performance Computing Applications*, Vol.21, No.3, pp.266–290 (2007).
60) Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P. and Aiken, A.: Titanium: A High-Performance Java Dialect, *ACM 1998 Workshop on Java for High-Performance Network Computing*, Vol.10, No.11–13, pp.825–836 (1998).
61) Zhu, W., Wang, C.-L. and Lau, F.C.M.: Lightweight Transparent Java Thread Migration for Distributed JVM, *International Conference on Parallel Processing*, p.465 (Oct. 2003).
62) Zhu, W., Wang, C.-L. and Lau, F.C.M.: JESSICA2: a distributed Java Virtual Machine with transparent thread migration support, *4th IEEE International Conference on Cluster Computing*, pp.381–388 (2002).
63) Zhu, W. and Zhu, S.W.: JESSICA2: A distributed Java virtual machine with transparent thread migration support, *IEEE International Conference on Cluster Computing* (Sep. 2002).
64) 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).
65) 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).
66) 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 (2010).
67) 高橋浩和，小田逸郎，山幡為佐久：Linux カーネル 2.6 解読室，ソフトバンククリエイティブ (Nov. 2006).
68) 緑川博子，飯塚　肇：ユーザレベル・ソフトウェア分散共有メモリ 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, pp.170–190 (Aug. 2001).

付　　録

A.1　random-address の最適性の証明

A.1.1　証明すべき定理の導出

各スレッドは，自分以外のスレッドがどのアドレスをどれくらい使用しているかに関する知識を持たないとする．このとき，スレッド移動時のアドレス衝突確率を最小化する戦略の 1 つが，各スレッドがアドレスを連続的に使用する戦略であることを証明する．まず，問題を定式化する．m 個のスレッド x_0，x_1，…，x_{m-1} を考え，これらの各スレッド x_i（$0 \le i \le m-1$）が使用するアドレス空間を $A = \{0, 1, \ldots, n-1\}$ とする．このとき，各スレッド x_i は，アドレス集合 $A = \{0, 1, \ldots, n-1\}$ に含まれる n 個のアドレスを何らかの順序で使用することになるが，「各スレッド x_i は置換 σ_i に従って一様にアドレスを使用する」ことを仮定する．ここで，「スレッド x_i が置換 σ_i に従って一様にアドレスを使用する」とは以下の意味である：

定義 1　順列 $\{0, 1, \ldots, n-1\}$ の置換 $\sigma_i = \{\sigma_i(0), \sigma_i(1), \ldots, \sigma_i(n-1)\}$ を考え，さらにこの置換 σ_i に対して以下の集合 $C^{\sigma_i}(A)$ を考える：

$$C^{\sigma_i}(A) = \{\{\sigma_i(\underline{a_i}), \sigma_i(\underline{a_i+1}), \ldots, \sigma_i(\underline{a_i+b_i-1})\} \mid 0 \le a_i \le n-1, 0 \le b_i \le n\}. \quad (1)$$

$C^{\sigma_i}(A)$ はアドレス集合の集合である．このとき，スレッド x_i が使用するアドレス集合が，つねに $C^{\sigma_i}(A)$ の要素集合であるとき，「スレッド x_i は置換 σ_i に従ってアドレスを使用する」と定義する．さらに，スレッド x_i が使用するアドレス集合が，$C^{\sigma_i}(A)$ の各要素集合を等確率でとる[⋆1]とき，「スレッド x_i は置換 σ_i に従って一様にアドレスを使用する」と定義する．

具体例として，$n = 4$ とし，順列 $\{0,1,2,3\}$ の置換 $\sigma_i = \{2,1,3,0\}$ を考える．このとき，$C^{\sigma_i}(A)$ は，

$$C^{\sigma_i}(A) = \{\{\},\{2\},\{1\},\{3\},\{0\},\{2,1\},\{1,3\},\{3,0\},\{0,2\},\\ \{2,1,3\},\{1,3,0\},\{3,0,2\},\{0,2,1\},\{2,1,3,0\}\}$$

となる．よって，「スレッド x_i が置換 σ_i に従ってアドレスを使用する」とは，スレッド x_i が使用するアドレス集合は，つねに，$\{\}$，$\{2\}$，$\{1\}$，$\{3\}$，$\{0\}$，$\{2,1\}$，$\{1,3\}$，$\{3,0\}$，$\{0,2\}$，$\{2,1,3\}$，$\{1,3,0\}$，$\{3,0,2\}$，$\{0,2,1\}$，$\{2,1,3,0\}$ のいずれかであるという意味である．また，「スレッド x_i が置換 σ_i に従って一様にアドレスを使用する」とは，スレッド x_i は，アドレス集合として $\{\}$，$\{2\}$，$\{1\}$，$\{3\}$，$\{0\}$，$\{2,1\}$，$\{1,3\}$，$\{3,0\}$，$\{0,2\}$，$\{2,1,3\}$，$\{1,3,0\}$，$\{3,0,2\}$，$\{0,2,1\}$，$\{2,1,3,0\}$ を等確率で使用するという意味である．

なお，置換 σ_j が置換 σ_i の巡回置換であるとき，「スレッド x_i が置換 σ_i に従って（一様に）アドレスを使用する」ことと「スレッド x_i が置換 σ_j に従って（一様に）アドレスを使用する」ことはまったく等価である．よって，以降の議論では，置換 σ_j が置換 σ_i の巡回置換であるとき，$\sigma_i = \sigma_j$ と表記することにする．

ここで，「各スレッド x_i が置換 σ_i に従って一様にアドレスを使用する」という仮定は，十分に一般的であることを強調しておく．なぜなら，この仮定のもとでは，各スレッド x_i に対して置換 σ_i を適切に選ぶことによって，「各スレッド x_i がアドレス集合 $A = \{0,1,\ldots,n-1\}$ に含まれる n 個のアドレスを任意の順序で使用する」戦略すべてを表現できるからである．また，この仮定における「一様に」という条件は，「自分以外のスレッドがどのアドレスをどれくらい使用しているかに関する知識を持たない」という状況を反映している．以上の議論より，「スレッド移動時のアドレス衝突確率を最小化する戦略の 1 つは，各スレッドがアドレスを連続的に使用する戦略である」ことをいうために証明すべき定理として以下の定理を得る：

定理 1　m 個のスレッド x_0，x_1，…，x_{m-1} を考え，各スレッド x_i は置換 σ_i に従って一様にアドレスを使用するとする．このとき，σ_0，σ_1，…，σ_{m-1} がとりうるすべての組合せ $(\sigma_0,\sigma_1,\ldots,\sigma_{m-1})$ のうち（各 σ_i の選び方は $n!$ 通り存在するから，組合せは全部で $n!^m$ 通り存在する），「どの 2 つの異なるスレッド x_i とスレッド x_j に対しても，スレッド x_i が使用するアドレス集合とスレッド x_j が使用するアドレス集合が共通部分を持たない確率」が最大になるのは，$\sigma_0 = \sigma_1 = \ldots = \sigma_{m-1} = \epsilon$ の場合である．ただしここで ϵ は恒等置換を表す．

さらに，定理 1 を一般化して次の定理を考える：

定理 2　m 個のスレッド x_0，x_1，…，x_{m-1} を考え，各スレッド x_i は置換 σ_i に従って一様にアドレスを使用するとする．このとき，σ_0，σ_1，…，σ_{m-1} がとりうるすべての組合せ $(\sigma_0,\sigma_1,\ldots,\sigma_{m-1})$ のうち，「どの 2 つの異なるスレッド x_i とスレッド x_j に対しても，スレッド x_i が使用するアドレス集合とスレッド x_j が使用するアドレス集合が共通部分を持たない確率」が最大になるのは，$\sigma_0 = \sigma_1 = \ldots = \sigma_{m-1}$ の場合であり，かつその場合に限られる．

明らかに，定理 2 は定理 1 の拡張になっており，定理 2 が証明されれば，定理 1 も証明されたことになる．次節では，定理 2 の証明を行う．

A.1.2　証　　明

定理 2 を証明する．以降，命題や補題をいくつか立てて証明を進めるが，これらの導出関係を**図 29** に示しておく．

まず，以下の命題を考える．これは定理 2 において $m = 2$ とした場合に相当する：

命題 1　スレッド x とスレッド y を考え，スレッド x は置換 σ_x に従って一様にアドレスを使用し，スレッド y は置換 σ_y に従って一様にアドレスを使用するとする．このとき，ス

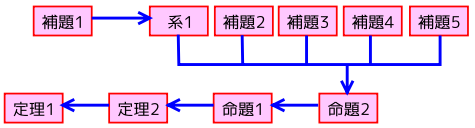

図 29　命題や補題の論理関係．$A \to B$ は，証明において A から B を導くことを意味する
Fig. 29　Logical relationships between propositions and lemmas. $A \to B$ indicates that B is derived from A.

⋆1 具体的な値は重要ではないが，具体的には $1/|C^{\sigma_i}(A)| = 1/(n^2-n+2)$ である．なぜなら，集合 $C^{\sigma_i}(A)$ の定義式 (1) において，$b_i = 0$ の場合にはすべての a_i に対してアドレス集合 $\{\}$ が生成されること，$b_i = n$ の場合にはすべての a_i に対してアドレス集合 $\{\sigma_i(0),\ldots,\sigma_i(n-1)\}$ が生成されること，$1 \le b_i \le n-1$ の場合には各 a_i に対して n 個の異なるアドレス集合 $\{\sigma_i(a_i),\sigma_i(a_i+1),\ldots,\sigma_i(a_i+b_i-1)\}$ が生成されることから，結局，$|C^{\sigma_i}(A)| = 1+1+(n-1)n = n^2-n+2$ となるからである．

レッド x が使用するアドレス集合とスレッド y が使用するアドレス集合が共通部分を持つ確率が最小になるのは，$\sigma_x = \sigma_y$ の場合であり，かつその場合に限られる．

命題 1 の証明に入る前に，以降の議論で使用する記号をいくつか導入する：

- アドレス集合 A の冪集合を $P(A)$ と表す．スレッド x が使用しているアドレス集合を S_x，スレッド y が使用しているアドレス集合を S_y とすれば，明らかに $S_x \in P(A)$，$S_y \in P(A)$ が成り立つ．さらに，$S_x \in C^{\sigma_x}(A)$，$S_y \in C^{\sigma_y}(A)$ も成り立つ．
- アドレス集合 A の冪集合 $P(A)$ の中で，大きさが i であるようなアドレス集合の集合を $P_i(A)$ と表す．すなわち，集合 $P_i(A)$ の任意の要素は大きさ i の集合であり，$|P_i(A)| = {}_nC_i$，$P_i(A) \cap P_j(A) = \emptyset$ $(i \neq j)$，$P(A) = \bigsqcup_{i=0}^{n} P_i(A)$ が成り立つ．同様に，アドレス集合の集合 $C^{\sigma_y}(A)$ の中で，大きさが i であるようなアドレス集合の集合を $C_i^{\sigma_y}(A)$ と表す．
- 任意のアドレス集合の集合 C と任意のアドレス集合 S_0，S_1，... に関して，集合 C に属するすべてのアドレス集合 $S \in C$ のうち，$(S_0 \cap S \neq \emptyset) \wedge (S_1 \cap S \neq \emptyset) \wedge \ldots$ を満足する S の個数を，$M(C; S_0, S_1, \ldots)$ と表す．たとえば，$M(C^{\sigma_y}(A); S_x)$ は，$C^{\sigma_y}(A)$ に属するすべてのアドレス集合 $S_y \in C^{\sigma_y}(A)$ のうち，$S_y \cap S_x \neq \emptyset$ を満足するような S_y の個数を表す．以上のような定義から，明らかに，任意のアドレス集合 S_i，S_j に対して，
$$M(C; \ldots, S_i, \ldots, S_j, \ldots) = M(C; \ldots, S_j, \ldots, S_i, \ldots)$$
が成り立つ．さらに，任意のアドレス集合 S_i，S_j に対して，$S_i \cup S_j = S_i + S_j - S_i \cap S_j$ が成り立つので，
$$M(C; \ldots, S_i \cup S_j, \ldots) = M(C; \ldots, S_i, \ldots) + M(C; \ldots, S_j, \ldots) - M(C; \ldots, S_i, S_j, \ldots) \quad (2)$$
が成り立つ．
- 以降では n を法とする剰余環での議論が多くなるため，任意の整数 i に対して $\mathrm{mod}(i, n)$ を $\underline{i}$ と略記することにする．よって，$0 \leq i < n$ ならば，
$$\underline{i} = i \quad (3)$$
が成り立つ．また，任意の整数 i，j に対して，
$$\underline{\underline{i} + j} = \underline{i + j} \quad (4)$$
が成り立つ．
- 虚数単位 j と整数 i_0，i_1，...，i_{k-1} に対して，複素平面上の k 個の点 $e^{2\pi i_0 j/n}$，$e^{2\pi i_1 j/n}$，...，$e^{2\pi i_{k-1} j/n}$ を考える．偏角を $[0, 2\pi)$ の範囲で考えるとき，ある整数 α（$0 \leq \alpha \leq k-1$）が存在して，$0 \leq \arg(e^{2\pi i_{\mathrm{mod}(\alpha,k)} j/n}) \leq \arg(e^{2\pi i_{\mathrm{mod}(\alpha+1,k)} j/n}) \leq \ldots \leq \arg(e^{2\pi i_{\mathrm{mod}(\alpha+k-1,k)} j/n}) < 2\pi$ の関係が成り立つとき，$i_0 \preceq i_1 \preceq \ldots \preceq i_{k-1} \preceq$ と表す．特に，整数 α'（$0 \leq \alpha' \leq k-1$）に対して，$\arg(e^{2\pi i_{\mathrm{mod}(\alpha',k)} j/n}) < \arg(e^{2\pi i_{\mathrm{mod}(\alpha'+1,k)} j/n})$ が成り立つとき，$i_0 \preceq i_1 \preceq \ldots \preceq i_{\alpha'} \prec i_{\alpha'+1} \preceq \ldots \preceq i_{k-1} \preceq$ と表す．図形的にいえば，$i_0 \preceq i_1 \prec i_2 \prec i_3 \preceq$ は，4 個の点 $e^{2\pi i_0 j/n}$，$e^{2\pi i_1 j/n}$，$e^{2\pi i_2 j/n}$，$e^{2\pi i_3 j/n}$ がこの順序で円周上に反時計回りに配置されており，かつ，$e^{2\pi i_1 j/n}$ と $e^{2\pi i_2 j/n}$ および $e^{2\pi i_2 j/n}$ と $e^{2\pi i_3 j/n}$ は異なる点であることを意味する（**図 30** (A)）．

以上の定義より，任意の整数 i_0，i_1，...，i_{k-1} に対して，
$$i_0 \preceq i_1 \preceq \ldots \preceq i_{k-1} \preceq$$
$$\iff \underline{i_1 - i_0} + \underline{i_2 - i_1} + \ldots + \underline{i_{k-1} - i_{k-2}} + \underline{i_0 - i_{k-1}} = n \quad (5)$$
が成り立つ．また，$i_0 \preceq i_1 \preceq i_2 \preceq$ ならば，
$$\underline{i_1 - i_0} + \underline{i_2 - i_1} = \underline{i_2 - i_0} \quad (6)$$
が成り立つ．

ここで，大きさが b_x の任意のアドレス集合 $S \in P_{b_x}(A)$ は，任意の置換 σ_y に対して，$i_0 \prec i_1 \prec \ldots \prec i_{b_x - 1} \prec$ なる整数 i_0，i_1，...，i_{b_x-1} を用いて，
$$S = \{\sigma_y(i_0), \sigma_y(i_1), \ldots, \sigma_y(i_{b_x-1})\}$$
と表現できることに注意する．たとえば，$n = 8$，$b_x = 4$ として，$S = \{1, 2, 3, 4\}$ は，置換 $\sigma_y = \{5, 2, 1, 0, 7, 3, 6, 4\}$ を用いて，$S = \{\sigma_y(1), \sigma_y(2), \sigma_y(5), \sigma_y(7)\}$ と表現できる．よって，以降では証明の直観的な理解を助けるため，アドレス集合の様子を図 30 のような円周上の点集合として図示することにする．

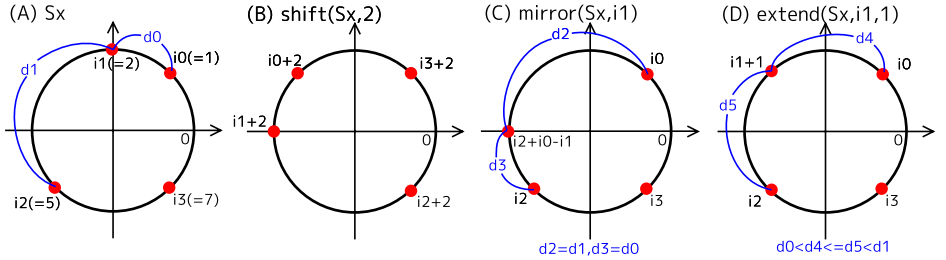

図 30　アドレス集合 S_x と各写像の具体例．(A) S_x，(B) $shift(S_x, s)$，(C) $mirror(S_x, i_\alpha)$，(D) $extend(S_x, i_\alpha, s)$

Fig. 30　Examples of mappings for an address set S_x.

いま導入した記号を用いて，命題 1 を定式化し，より証明しやすい形式の命題 2 にいい換える．

命題 1 においては，スレッド x とスレッド y は，それぞれ置換 σ_x と置換 σ_y に従って「一様に」アドレスを使用することを仮定しているので，「$\sigma_x = \sigma_y$ のときに，スレッド x が使用するアドレス集合とスレッド y が使用するアドレス集合が共通部分を持つ確率が最小になる」という命題 1 の題意は，「$\sigma_x = \sigma_y$ のときに，スレッド x が使用するアドレス集合 $S_x \in C^{\sigma_x}(A)$ とスレッド y が使用するアドレス集合 $S_y \in C^{\sigma_y}(A)$ のすべての組合せ (S_x, S_y)（全部で $|C^{\sigma_x}(A)| \times |C^{\sigma_y}(A)|$ 個ある）の中で，$S_x \cap S_y \neq \emptyset$ を満たすような組合せ (S_x, S_y) の個数が最小になる」ということと等価である．そして，これをさらにいい換えると，「置換 σ_y が与えられたとき，すべての置換 σ_x の中で，『スレッド x が使用するアドレス集合 $S_x \in C^{\sigma_x}(A)$ とスレッド y が使用するアドレス集合 $S_y \in C^{\sigma_y}(A)$ のすべての組合せ (S_x, S_y)（全部で $|C^{\sigma_x}(A)| \times |C^{\sigma_y}(A)|$ 個ある）の中で，$S_x \cap S_y \neq \emptyset$ を満たすような組合せ (S_x, S_y) の個数が最小になる』ような置換 σ_x とは，置換 σ_y である」ということと等価である．したがって，命題 1 をいい換えると，「$P(A)$ に属するすべてのアドレス集合 $S_x \in P(A)$ の中で，『すべてのアドレス集合 $S_y \in C^{\sigma_y}(A)$ のうち，$S_y \cap S_x \neq \emptyset$ を満足するような S_y の個数』が最小になるような S_x の集合は $C^{\sigma_y}(A)$ である」といい換えることができる．さらに，これを S_x の大きさによって分解していい換えると，「$0 \leq b_x \leq n$ とする．$P_{b_x}(A)$ に属するすべてのアドレス集合 $S_x \in P_{b_x}(A)$ の中で，『すべてのアドレス集合 $S_y \in C^{\sigma_y}(A)$ のうち，$S_y \cap S_x \neq \emptyset$ を満足するような S_y の個数』が最小になるような S_x の集合は $C^{\sigma_y}_{b_x}(A)$ である」といい換えることができる．さらに，これを先ほど導入した記号を用いていい換えると，「$0 \leq b_x \leq n$ とする．$P_{b_x}(A)$ に属するすべてのアドレス集合 $S_x \in P_{b_x}(A)$ の中で，$M(C^{\sigma_y}(A); S_x)$ が最小になるような S_x の集合は $C^{\sigma_y}_{b_x}(A)$ である」といい換えることができる．

以上の議論により，命題 1 と等価な命題 2 が得られる：

命題 2　$0 \leq b_x \leq n$ とする．$P_{b_x}(A)$ に属するすべてのアドレス集合 $S_x \in P_{b_x}(A)$ のうち，$M(C^{\sigma_y}(A); S_x)$ を最小にする S_x の集合は $C^{\sigma_y}_{b_x}(A)$ である．

命題 2 を分かりやすく書き下すと，b_x の値に応じて次のようになる．$P_{b_x}(A)$ に属するすべてのアドレス集合 $S_x \in P_{b_x}(A)$ のうち，$M(C^{\sigma_y}(A); S_x)$ を最小化する S_x は，

- $b_x = 0$ のとき，$\{\}$ の 1 通りのみである．
- $b_x = n$ のとき，$\{0, 1, \ldots, n-1\}$ の 1 通りのみである．
- $1 \leq b_x \leq n-1$ のとき，以下の T^0，T^1，…，T^{n-1} の合計 n 通りのみである：

$$
\begin{aligned}
T^0 &= \{\sigma_y(0), \sigma_y(1), \ldots, \sigma_y(b_x - 2), \sigma_y(b_x - 1)\},\\
T^1 &= \{\sigma_y(1), \sigma_y(2), \ldots, \sigma_y(b_x - 1), \sigma_y(b_x)\},\\
&\vdots\\
T^{n-1} &= \{\sigma_y(n - b_x + 1), \sigma_y(n - b_x + 2), \ldots, \sigma_y(n-1), \sigma_y(0)\}.
\end{aligned}
$$

まず $b_x = 0$ のときには，$P_0(A) = \{\{\}\}$ であるから，命題 2 は明らかである．$b_x = 1$ のときには，$P_1(A) = \{\{0\}, \{1\}, \ldots, \{n-1\}\}$ であるから，対称性より命題 2 は明らかである．また $b_x = n$ のときには，$P_n(A) = \{\{0, 1, \ldots, n-1\}\}$ であるから，命題 2 は明らかである．したがって，以降の議論においては，$2 \leq b_x \leq n-1$ の場合に命題 2 が成り立つことを証明する．命題 2 の証明が終わるまでの間，$2 \leq b_x \leq n-1$ を仮定して議論する．

さて，$S_x \in P_{b_x}(A)$ なる任意の S_x は，$i_0 \prec i_1 \prec \ldots \prec i_{b_x-1} \prec$ なる整数 i_0，i_1，…，i_{b_x-1} を用いて，

$$S_x = \{\sigma_y(i_0), \sigma_y(i_1), \ldots, \sigma_y(i_{b_x-1})\}$$

と表すことができる（図 30 (A)）．なお，以降の議論では i の添字は b_x を法とする剰余環上で計算するものとする．つまり，$i_{\mathrm{mod}(j, b_x)}$ を i_j と略記する．

ここで，任意の $S_x \in P_{b_x}(A)$ に対して，3 つの写像 *shift*，*mirror*，*extend* を以下のように定義する：

shift　任意の整数 s に対して，

$$shift(S_x, s) = \{\sigma_y(\underline{i_0 + s}), \sigma_y(\underline{i_1 + s}), \ldots, \sigma_y(\underline{i_{b_x-1} + s})\}.$$

mirror　$\sigma_y(i_\alpha) \in S_x$ を満たす任意の i_α に対して，

$$mirror(S_x, i_\alpha) = S_x \backslash \{\sigma_y(i_\alpha)\} \cup \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}.$$

extend　$(\sigma_y(i_\alpha) \in S_x) \wedge (i_\alpha \prec \underline{i_\alpha + s} \prec i_{\alpha+1} \prec) \wedge (\underline{(i_\alpha + s) - i_{\alpha-1}} \leq \underline{i_{\alpha+1} - (i_\alpha + s)})$ を満たす任意の整数 s と i_α に対して，

$$extend(S_x, i_\alpha, s) = S_x \backslash \{\sigma_y(i_\alpha)\} \cup \{\sigma_y(\underline{i_\alpha + s})\}.$$

なお，いまは $2 \leq b_x \leq n-1$ を仮定しているので，$i_{\alpha-1} = i_{\alpha+1}$ の可能性はあるが（$b_x = 2$ のとき），$i_{\alpha-1} \neq i_\alpha$ かつ $i_\alpha \neq i_{\alpha+1}$ が成り立つことに注意する．

$n = 8$, $b_x = 4$ とした場合の，3 つの写像 *shift*，*mirror*，*extend* の例を，それぞれ図 30 (B), (C), (D) に示す．直観的には，図 30 に示すような円周上の距離で考えたとき，$shift(S_x, s)$ の図形的意味は「S_x 全体を距離 s だけ移動させる」，$mirror(S_x, i_\alpha)$ の図形的意味は「点 i_α を，点 $i_{\alpha-1}$ と点 $i_{\alpha+1}$ の中点に関して対称な位置に移動させる」，$extend(S_x, i_\alpha, s)$ の図形的意味は「点 i_α を距離 s だけ移動させる．ただし，このとき円周上で $i_{\alpha-1}$，i_α，$\underline{i_\alpha + s}$，

$i_{\alpha+1}$ の順に反時計回りに点が並んでおり，かつ，$\underline{i_\alpha + s}$ と $i_{\alpha+1}$ の距離は $i_{\alpha-1}$ と $\underline{i_\alpha + s}$ の距離以上になっていなければならない」という意味である．

ここで，3 つの写像 *shift*，*mirror*，*extend* に関して，以下の 3 つの補題を考え，証明する：

補題 1　任意のアドレス集合 S_x^0，S_x^1，$\ldots \in P_{b_x}(A)$ と任意の整数 s に対して，

$$M(C^{\sigma_y}(A); S_x^0, S_x^1, \ldots) = M(C^{\sigma_y}(A); \mathit{shift}(S_x^0, s), \mathit{shift}(S_x^1, s), \ldots)$$

が成り立つ．

補題 2　任意のアドレス集合 $S_x \in P_{b_x}(A)$ と $\sigma_y(i_\alpha) \in S_x$ を満たす任意の i_α に対して，

$$M(C^{\sigma_y}(A); S_x) = M(C^{\sigma_y}(A); \mathit{mirror}(S_x, i_\alpha))$$

が成り立つ．

補題 3　任意のアドレス集合 $S_x \in P_{b_x}(A)$，および $(\sigma_y(i_\alpha) \in S_x) \wedge (i_\alpha \prec \underline{i_\alpha + s} \prec i_{\alpha+1} \prec) \wedge (\underline{(i_\alpha + s) - i_{\alpha-1}} \leq \underline{i_{\alpha+1} - (i_\alpha + s)})$ を満たす任意の整数 s と整数 i_α に関して，

$$M(C^{\sigma_y}(A); S_x) < M(C^{\sigma_y}(A); \mathit{extend}(S_x, i_\alpha, s))$$

が成り立つ．

補題 1 を示す．まず，$C^{\sigma_y}(A)$ の定義は，式 (1) より，

$$C^{\sigma_y}(A) = \{\{\sigma_y(\underline{a_y}), \sigma_y(\underline{a_y+1}), \ldots, \sigma_y(\underline{a_y+b_y-1})\} \mid 0 \leq a_y \leq n-1, 0 \leq b_y \leq n\}$$

である．ここで，任意の $S_y = \{\sigma_y(\underline{a_y}), \sigma_y(\underline{a_y+1}), \ldots, \sigma_y(\underline{a_y+b_y-1})\} \in C^{\sigma_y}(A)$ $(0 \leq a_y \leq n-1,\ 0 \leq b_y \leq n)$ に対して，

$$\begin{aligned}
&(S_x^0 \cap \{\sigma_y(\underline{a_y}), \sigma_y(\underline{a_y+1}), \ldots, \sigma_y(\underline{a_y+b_y-1})\} \neq \emptyset) \wedge \\
&(S_x^1 \cap \{\sigma_y(\underline{a_y}), \sigma_y(\underline{a_y+1}), \ldots, \sigma_y(\underline{a_y+b_y-1})\} \neq \emptyset) \wedge \ldots \\
\Longleftrightarrow\ &(\mathit{shift}(S_x^0, s) \cap \{\sigma_y(\underline{a_y+s}), \sigma_y(\underline{a_y+1+s}), \ldots, \sigma_y(\underline{a_y+b_y-1+s})\} \neq \emptyset) \wedge \\
&(\mathit{shift}(S_x^1, s) \cap \{\sigma_y(\underline{a_y+s}), \sigma_y(\underline{a_y+1+s}), \ldots, \sigma_y(\underline{a_y+b_y-1+s})\} \neq \emptyset) \wedge \ldots
\end{aligned}$$

が成り立つ．すなわち，アドレス集合 S_x^0，S_x^1，$\ldots$ のすべてがアドレス集合 $S_y \in C^{\sigma_y}(A)$ と共通部分を持つならば，そのような S_x^0，S_x^1，$\ldots$ に対して，アドレス集合 $\mathit{shift}(S_x^0, s)$，$\mathit{shift}(S_x^1, s)$，$\ldots$ のすべてがアドレス集合 S_y' と共通部分を持つようなアドレス集合 $S_y' \in C^{\sigma_y}(A)$ がちょうど 1 つ存在する．したがって，「すべての $S_y \in C^{\sigma_y}(A)$ のうち，$(S_x^0 \cap S_y \neq \emptyset) \wedge (S_x^1 \cap S_y \neq \emptyset) \wedge \ldots$ を満たす S_y の個数」と，「すべての $S_y \in C^{\sigma_y}(A)$ のうち，$(\mathit{shift}(S_x^0, s) \cap S_y \neq \emptyset) \wedge (\mathit{shift}(S_x^1, s) \cap S_y \neq \emptyset) \wedge \ldots$ を満たす S_y の個数」は等しい．よって，

$$M(C^{\sigma_y}(A); S_x^0, S_x^1, \ldots) = M(C^{\sigma_y}(A); \mathit{shift}(S_x^0, s), \mathit{shift}(S_x^1, s), \ldots)$$

が成り立つ．以上より，補題 1 が示された．　∎

系 1　任意のアドレス集合 $S_x \in P_{b_x}(A)$ と任意の整数 s に対して，

$$M(C^{\sigma_y}(A); S_x) = M(C^{\sigma_y}(A); \mathit{shift}(S_x, s)).$$

補題 1 より明らかである．　∎

補題 2 を示す．左辺と右辺をそれぞれ計算し，両者が一致することを示す．

まず，式 (2) を用いて補題 2 の左辺を計算すると，

$$\begin{aligned}
M(C^{\sigma_y}(A); S_x) &= M(C^{\sigma_y}(A); (S_x \backslash \{\sigma_y(i_\alpha)\}) \cup \{\sigma_y(i_\alpha)\}) \\
&= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) \\
&\quad - M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_\alpha)\}) \qquad (7)
\end{aligned}$$

となる．上式の第 1 項，第 2 項，第 3 項をそれぞれ D_1，D_2，D_3 とおく．ここで，$D_3 = M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_\alpha)\})$ は，「すべての $S_y \in C^{\sigma_y}(A)$ のうち，$((S_x \backslash \{\sigma_y(i_\alpha)\}) \cap S_y \neq \emptyset) \wedge (\{\sigma_y(i_\alpha)\} \cap S_y \neq \emptyset)$ を満たす S_y の個数」を意味しているが，これは，「すべての $S_y \in C^{\sigma_y}(A)$ のうち，$(\{\sigma_y(i_{\alpha-1}), \sigma_y(i_{\alpha+1})\} \cap S_y \neq \emptyset) \wedge (\{\sigma_y(i_\alpha)\} \cap S_y \neq \emptyset)$ を満たす S_y の個数」に等しい．なぜなら，$i_0 \prec i_1 \prec \ldots \prec i_{\alpha-1} \prec i_\alpha \prec i_{\alpha+1} \prec \ldots \prec i_{b_x-1} \prec$ なので，$((S_x \backslash \{\sigma_y(i_\alpha)\}) \cap S_y \neq \emptyset) \wedge (\{\sigma_y(i_\alpha)\} \cap S_y \neq \emptyset)$ を満たすような S_y は，必ず $\sigma_y(i_{\alpha-1})$ または $\sigma_y(i_{\alpha+1})$ を含むからである．すなわち，

$$\begin{aligned}
D_3 &= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_\alpha)\} \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1}), \sigma_y(i_{\alpha+1})\}, \{\sigma_y(i_\alpha)\})
\end{aligned}$$

が成り立つ．さらに D_3 の計算を進めると，

$$\begin{aligned}
D_3 &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1}), \sigma_y(i_{\alpha+1})\}, \{\sigma_y(i_\alpha)\}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\} \cup \{\sigma_y(i_{\alpha+1})\}, \{\sigma_y(i_\alpha)\}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\
&\quad - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \quad (\because \text{式 (2)}) \qquad (8)
\end{aligned}$$

が得られる．上式の第 1 項，第 2 項，第 3 項をそれぞれ D_4，D_5，D_6 とおく．以上をまとめると，

$$M(C^{\sigma_y}(A); S_x) = D_1 + D_2 - (D_4 + D_5 - D_6) \qquad (9)$$

となる．

同様にして，式 (2) を用いて補題 2 の右辺を計算すると，

$$\begin{aligned}
&M(C^{\sigma_y}(A); \mathit{mirror}(S_x, i_\alpha)) \\
=\ &M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\} \cup \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\})
\end{aligned}$$

$$
\begin{aligned}
&= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}) \\
&\quad - M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\})
\end{aligned}
$$

となるので，この第 1 項，第 2 項，第 3 項をそれぞれ D'_1，D'_2，D'_3 とおく．先ほどと同様にして D'_3 を計算すると，

$$
\begin{aligned}
D'_3 &= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1}), \sigma_y(i_{\alpha+1})\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\} \cup \{\sigma_y(i_{\alpha+1})\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}) \\
&\quad + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}, \{\sigma_y(i_{\alpha+1})\}) \\
&\quad - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}, \{\sigma_y(i_{\alpha+1})\}) \quad (\because \text{式 (2)})
\end{aligned}
$$

が得られる．上式の第 1 項，第 2 項，第 3 項をそれぞれ D'_4，D'_5，D'_6 とおく．以上をまとめると，

$$
M(C^{\sigma_y}(A); mirror(S_x, i_\alpha)) = D'_1 + D'_2 - (D'_4 + D'_5 - D'_6) \tag{10}
$$

となる．

D_1，D_2，D_4，D_5 と D'_1，D'_2，D'_4，D'_5 の大小を比較すると，

$$
D_1 = M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) = D'_1, \tag{11}
$$

$$
\begin{aligned}
D_2 &= M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) \\
&= M(C^{\sigma_y}(A); shift(\{\sigma_y(i_\alpha)\}, \underline{i_{\alpha-1} + i_{\alpha+1} - 2i_\alpha})) \quad (\because \text{補題 1}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}) \\
&= D'_2,
\end{aligned} \tag{12}
$$

$$
\begin{aligned}
D_4 &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) \\
&= M(C^{\sigma_y}(A); shift(\{\sigma_y(i_{\alpha-1})\}, \underline{i_{\alpha+1} - i_\alpha}), shift(\{\sigma_y(i_\alpha)\}, \underline{i_{\alpha+1} - i_\alpha})) \quad (\because \text{補題 1}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}, \{\sigma_y(i_{\alpha+1})\}) \\
&= D'_5,
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
D_5 &= M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\
&= M(C^{\sigma_y}(A); shift(\{\sigma_y(i_\alpha)\}, \underline{i_{\alpha-1} - i_\alpha}), shift(\{\sigma_y(i_{\alpha+1})\}, \underline{i_{\alpha-1} - i_\alpha})) \quad (\because \text{補題 1}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}) \\
&= D'_4
\end{aligned} \tag{14}
$$

となる．次に，D_6 と D'_6 の大小を考える．まず，

$$
\underline{(i_{\alpha-1} + i_{\alpha+1} - i_\alpha) - i_{\alpha-1}} + \underline{i_{\alpha+1} - (i_{\alpha-1} + i_{\alpha+1} - i_\alpha)} + \underline{i_{\alpha-1} - i_{\alpha+1}}
$$

$$
\begin{aligned}
&= \underline{i_{\alpha+1} - i_\alpha} + \underline{i_\alpha - i_{\alpha-1}} + \underline{i_{\alpha-1} - i_{\alpha+1}} \quad (\because \text{式 (4)}) \\
&= n \quad (\because \text{仮定より } i_{\alpha-1} \preceq i_\alpha \preceq i_{\alpha+1} \preceq \text{と式 (5)})
\end{aligned}
$$

であるから，式 (5) より，$i_{\alpha-1} \preceq \underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha} \preceq i_{\alpha+1} \preceq$ が成り立つ．このことと，仮定より $i_{\alpha-1} \preceq i_\alpha \preceq i_{\alpha+1} \preceq$ であることを考慮すると，ある S_y が $\sigma_y(i_{\alpha-1})$ と $\sigma_y(i_\alpha)$ と $\sigma_y(i_{\alpha+1})$ の 3 要素を含むことと，S_y が $\sigma_y(i_{\alpha-1})$ と $\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})$ と $\sigma_y(i_{\alpha+1})$ の 3 要素を含むことは同値である．したがって，「すべての $S_y \in C^{\sigma_y}(A)$ のうち，$(\{\sigma_y(i_{\alpha-1})\} \cap S_y \neq \emptyset) \wedge (\{\sigma_y(i_\alpha)\} \cap S_y \neq \emptyset) \wedge (\{\sigma_y(i_{\alpha+1})\} \cap S_y \neq \emptyset)$ を満たす S_y の個数」は，「すべての $S_y \in C^{\sigma_y}(A)$ のうち，$(\{\sigma_y(i_{\alpha-1})\} \cap S_y \neq \emptyset) \wedge (\{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\} \cap S_y \neq \emptyset) \wedge (\{\sigma_y(i_{\alpha+1})\} \cap S_y \neq \emptyset)$ を満たす S_y の個数」に等しい．よって，

$$
\begin{aligned}
D_6 &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}, \{\sigma_y(i_{\alpha+1})\}) \\
&= D'_6
\end{aligned} \tag{15}
$$

となる．

式 (9)，(10)，(11)，(12)，(13)，(14)，(15) より，

$$
M(C^{\sigma_y}(A); S_x) = M(C^{\sigma_y}(A); mirror(S_x, i_\alpha))
$$

が成り立つ．以上より，補題 2 が示された．　■

補題 3 を示す．まず，補題 3 の左辺を計算すると，式 (7)，(8) より，

$$
\begin{aligned}
&M(C^{\sigma_y}(A); S_x) \\
&= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) \\
&\quad - M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_\alpha)\}) \\
&= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) \\
&\quad - (M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\
&\quad - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}))
\end{aligned} \tag{16}
$$

となる．上式の第 1 項，第 2 項，第 3 項，第 4 項，第 5 項を，それぞれ D_1，D_2，D_3，D_4，D_5 とおく．

また，補題 2 のときの議論と同様にして，補題 3 の右辺を計算すると，

$$
\begin{aligned}
&M(C^{\sigma_y}(A); extend(S_x, i_\alpha, s)) \\
&= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\} \cup \{\sigma_y(\underline{i_\alpha + s})\}) \\
&= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}) \\
&\quad - M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(\underline{i_\alpha + s})\}) \quad (\because \text{式 (2)}) \\
&= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\})
\end{aligned}
$$

$$
\begin{aligned}
& -M(C^{\sigma_y}(A);\{\sigma_y(i_{\alpha-1}),\sigma_y(i_{\alpha+1})\},\{\sigma_y(\underline{i_\alpha+s})\}) \\
& \quad (\because i_{\alpha-1}\preceq \underline{i_\alpha+s}\preceq i_{\alpha+1}\preceq \text{かつ } i_{\alpha-1}\preceq i_\alpha\preceq i_{\alpha+1}\preceq) \\
= \;& M(C^{\sigma_y}(A);S_x\backslash\{\sigma_y(i_\alpha)\})+M(C^{\sigma_y}(A);\{\sigma_y(\underline{i_\alpha+s})\}) \\
& -M(C^{\sigma_y}(A);\{\sigma_y(i_{\alpha-1})\}\cup\{\sigma_y(i_{\alpha+1})\},\{\sigma_y(\underline{i_\alpha+s})\}) \\
= \;& M(C^{\sigma_y}(A);S_x\backslash\{\sigma_y(i_\alpha)\})+M(C^{\sigma_y}(A);\{\sigma_y(\underline{i_\alpha+s})\}) \\
& -(M(C^{\sigma_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(\underline{i_\alpha+s})\}) \\
& +M(C^{\sigma_y}(A);\{\sigma_y(\underline{i_\alpha+s})\},\{\sigma_y(i_{\alpha+1})\}) \\
& -M(C^{\sigma_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(\underline{i_\alpha+s})\},\{\sigma_y(i_{\alpha+1})\})) \quad (\because \text{式 (2)}) \qquad (17)
\end{aligned}
$$

となる．上式の第 1 項，第 2 項，第 3 項，第 4 項，第 5 項を，それぞれ D'_1，D'_2，D'_3，D'_4，D'_5 とおく．

以降では，D_1，D_2，D_3，D_4，D_5 と D'_1，D'_2，D'_3，D'_4，D'_5 の大小を比較する．まず，D_1，D_2 と D'_1，D'_2 を比較すると，

$$
\begin{aligned}
D_1 &= M(C^{\sigma_y}(A);S_x\backslash\{\sigma_y(i_\alpha)\})=D'_1, \qquad (18)\\
D_2 &= M(C^{\sigma_y}(A);\{\sigma_y(i_\alpha)\})=M(C^{\sigma_y}(A);\mathit{shift}(\{\sigma_y(i_\alpha)\},s)) \\
&= M(C^{\sigma_y}(A);\{\sigma_y(\underline{i_\alpha+s})\})=D'_2, \qquad (19)
\end{aligned}
$$

が成り立つ．また，$i_{\alpha-1}\preceq \underline{i_\alpha+s}\preceq i_{\alpha+1}\preceq$ かつ $i_{\alpha-1}\preceq i_\alpha\preceq i_{\alpha+1}\preceq$ であることから，式 (15) と同様にして，

$$
\begin{aligned}
D_5 &= M(C^{\sigma_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(i_\alpha)\},\{\sigma_y(i_{\alpha+1})\}) \qquad (20)\\
&= M(C^{\sigma_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(\underline{i_\alpha+s})\},\{\sigma_y(i_{\alpha+1})\})=D'_5
\end{aligned}
$$

が成り立つ．

次に，D_3+D_4 と $D'_3+D'_4$ を比較する．いま，$b_y\neq b'_y$ ならば $C^{\sigma_y}_{b_y}(A)\cap C^{\sigma_y}_{b'_y}(A)=\emptyset$ であり，$C^{\sigma_y}(A)=\bigsqcup_{b_y=0}^{n}C^{\sigma_y}_{b_y}(A)$ が成り立つから，任意の集合 S_x に対して，

$$
M(C^{\sigma_y}(A);S_x)=\sum_{b_y=0}^{n}M(C^{\sigma_y}_{b_y}(A);S_x)
$$

が成り立つことに着目する．したがって，

$$
\begin{aligned}
& D_3+D_4 \\
= \;& M(C^{\sigma_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(i_\alpha)\})+M(C^{\sigma_y}(A);\{\sigma_y(i_\alpha)\},\{\sigma_y(i_{\alpha+1})\}) \\
= \;& \sum_{b_y=0}^{n}M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(i_\alpha)\})+\sum_{b_y=0}^{n}M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_\alpha)\},\{\sigma_y(i_{\alpha+1})\}) \\
= \;& \sum_{b_y=0}^{n}\Big(M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(i_\alpha)\})+M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_\alpha)\},\{\sigma_y(i_{\alpha+1})\})\Big), \\
& D'_3+D'_4 \\
= \;& M(C^{\sigma_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(\underline{i_\alpha+s})\})+M(C^{\sigma_y}(A);\{\sigma_y(\underline{i_\alpha+s})\},\{\sigma_y(i_{\alpha+1})\}) \\
= \;& \sum_{b_y=0}^{n}M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(\underline{i_\alpha+s})\}) \\
& +\sum_{b_y=0}^{n}M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(\underline{i_\alpha+s})\},\{\sigma_y(i_{\alpha+1})\}) \\
= \;& \sum_{b_y=0}^{n}\Big(M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(\underline{i_\alpha+s})\}) \\
& +M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(\underline{i_\alpha+s})\},\{\sigma_y(i_{\alpha+1})\})\Big)
\end{aligned}
$$

と展開できる．ここで，

$$
\begin{aligned}
d_3(b_y) &= M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(i_\alpha)\}),\\
d_4(b_y) &= M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_\alpha)\},\{\sigma_y(i_{\alpha+1})\}),\\
d'_3(b_y) &= M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(i_{\alpha-1})\},\{\sigma_y(\underline{i_\alpha+s})\}),\\
d'_4(b_y) &= M(C^{\sigma_y}_{b_y}(A);\{\sigma_y(\underline{i_\alpha+s})\},\{\sigma_y(i_{\alpha+1})\})
\end{aligned}
$$

とおくと，D_3+D_4 と $D'_3+D'_4$ は，

$$
D_3+D_4=\sum_{b_y=0}^{n}(d_3(b_y)+d_4(b_y)), \qquad (21)
$$

$$
D'_3+D'_4=\sum_{b_y=0}^{n}(d'_3(b_y)+d'_4(b_y)) \qquad (22)
$$

と表すことができる．よって，以下では，各 b_y（$0\le b_y\le n$）の値に応じて，$d_3(b_y)+d_4(b_y)$ と $d'_3(b_y)+d'_4(b_y)$ がどのような値をとるか調べる．

まず，$d_3(b_y)$ について考える．

(i)　$0\le b_y\le \underline{i_\alpha-i_{\alpha-1}}$ のとき．アドレス集合 $S_y=\{\sigma_y(\underline{a_y}),\sigma_y(\underline{a_y+1}),\ldots,$

$\sigma_y(\underline{a_y+b_y-1})\} \in C^{\sigma_y}_{b_y}(A)$ $(0 \le a_y \le n-1)$ が，$\sigma_y(i_{\alpha-1})$ と $\sigma_y(i_\alpha)$ の両方の要素を含むことはありえない．よって，$d_3(b_y)=0$ である．

(ii)　$\underline{i_\alpha - i_{\alpha-1}} < b_y \le n-1$ のとき．アドレス集合 $S_y = \{\sigma_y(\underline{a_y}), \sigma_y(\underline{a_y+1}), \ldots, \sigma_y(\underline{a_y+b_y-1})\} \in C^{\sigma_y}_{b_y}(A)$ $(0 \le a_y \le n-1)$ が，$\sigma_y(i_{\alpha-1})$ と $\sigma_y(i_\alpha)$ の両方の要素を含むのは，a_y が，$\underline{i_\alpha - b_y + 1}$，$\underline{i_\alpha - b_y + 2}$，…，$\underline{i_{\alpha-1}-1}$，$i_{\alpha-1}$ を満たす場合である．よって，

$$d_3(b_y) = \underline{i_{\alpha-1} - (i_\alpha - b_y + 1) + 1} = \underline{b_y - (i_\alpha - i_{\alpha-1})}$$

である．ここで，$0 \le b_y - \underline{i_\alpha - i_{\alpha-1}} < n$ であることに注意すると，

$$d_3(b_y) = \underline{b_y - \underline{(i_\alpha - i_{\alpha-1})}} = b_y - \underline{i_\alpha - i_{\alpha-1}} \quad (\because \text{式 (3) (4)})$$

となる．

(iii)　$b_y = n$ のとき．明らかに $d_3(b_y)=1$ である．

同様にして，$d_4(b_y)$，$d'_3(b_y)$，$d'_4(b_y)$ についても計算し，結果をまとめると以下のようになる：

$$d_3(b_y) = \begin{cases} 0 & \text{if } 0 \le b_y \le \underline{i_\alpha - i_{\alpha-1}} \\ b_y - (\underline{i_\alpha - i_{\alpha-1}}) & \text{if } \underline{i_\alpha - i_{\alpha-1}} < b_y \le n-1 \\ 1 & \text{if } b_y = n \end{cases} \tag{23}$$

$$d_4(b_y) = \begin{cases} 0 & \text{if } 0 \le b_y \le \underline{i_{\alpha+1} - i_\alpha} \\ b_y - (\underline{i_{\alpha+1} - i_\alpha}) & \text{if } \underline{i_{\alpha+1} - i_\alpha} < b_y \le n-1 \\ 1 & \text{if } b_y = n \end{cases} \tag{24}$$

$$d'_3(b_y) = \begin{cases} 0 & \text{if } 0 \le b_y \le \underline{(i_\alpha + s) - i_{\alpha-1}} \\ b_y - (\underline{(i_\alpha + s) - i_{\alpha-1}}) & \text{if } \underline{(i_\alpha + s) - i_{\alpha-1}} < b_y \le n-1 \\ 1 & \text{if } b_y = n \end{cases} \tag{25}$$

$$d'_4(b_y) = \begin{cases} 0 & \text{if } 0 \le b_y \le \underline{i_{\alpha+1} - (i_\alpha + s)} \\ b_y - (\underline{i_{\alpha+1} - (i_\alpha + s)}) & \text{if } \underline{i_{\alpha+1} - (i_\alpha + s)} < b_y \le n-1 \\ 1 & \text{if } b_y = n \end{cases} \tag{26}$$

となる．

ここで，式 (23)，(24)，(25)，(26) において，場合分けの境界値になっている b_y の値たち 0，$\underline{i_\alpha - i_{\alpha-1}}$，$\underline{i_{\alpha+1} - i_\alpha}$，$\underline{(i_\alpha + s) - i_{\alpha-1}}$，$\underline{i_{\alpha+1} - (i_\alpha + s)}$，$n$ に関して，その大小関係を調べると，

$$\begin{aligned} 0 &< \underline{i_\alpha - i_{\alpha-1}} \quad (\because \text{仮定より } i_{\alpha-1} \prec i_\alpha \prec) \\ &< \underline{(i_\alpha + s) - i_{\alpha-1}} \quad (\because \text{仮定より } i_{\alpha-1} \prec i_\alpha \prec \underline{i_\alpha + s} \prec) \\ &\le \underline{i_{\alpha+1} - (i_\alpha + s)} \quad (\because \text{仮定そのまま}) \\ &< \underline{i_{\alpha+1} - i_\alpha} \quad (\because \text{仮定より } i_\alpha \prec \underline{i_\alpha + s} \prec i_{\alpha+1} \prec) \\ &< n \quad (\because \text{明らか}) \end{aligned} \tag{27}$$

が成り立つ．図 30 (D) を見ると，この大小関係が図形的に理解できる．

以上をふまえて，$d_3(b_y)+d_4(b_y)$ と $d'_3(b_y)+d'_4(b_y)$ の大小を，b_y の境界値に応じてまとめると以下のようになる．

(i)　$0 \le b_y \le \underline{i_\alpha - i_{\alpha-1}}$ のとき．

$$(d_3(b_y)+d_4(b_y)) - (d'_3(b_y)+d'_4(b_y)) = (0+0)-(0+0) = 0 \tag{28}$$

である．

(ii)　$\underline{i_\alpha - i_{\alpha-1}} < b_y \le \underline{(i_\alpha + s) - i_{\alpha-1}}$ のとき．

$$(d_3(b_y)+d_4(b_y)) - (d'_3(b_y)+d'_4(b_y)) = ((b_y - (\underline{i_\alpha - i_{\alpha-1}})) + 0) - (0+0) > 0 \tag{29}$$

である．

(iii)　$\underline{(i_\alpha + s) - i_{\alpha-1}} < b_y \le \underline{i_{\alpha+1} - (i_\alpha + s)}$ のとき．

$$\begin{aligned} &(d_3(b_y)+d_4(b_y)) - (d'_3(b_y)+d'_4(b_y)) \\ &= ((b_y - (\underline{i_\alpha - i_{\alpha-1}})) + 0) - ((b_y - (\underline{(i_\alpha + s) - i_{\alpha-1}})) + 0) \\ &= (\underline{(i_\alpha + s) - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \\ &= (\underline{(i_\alpha + s) - i_\alpha}) + (\underline{i_\alpha - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \quad (\because i_{\alpha-1} \preceq i_\alpha \preceq \underline{i_\alpha + s} \preceq \text{ と式 (6)}) \\ &= \underline{(i_\alpha + s) - i_\alpha} \\ &= \underline{s} \\ &> 0 \end{aligned} \tag{30}$$

である．

(iv)　$\underline{i_{\alpha+1} - (i_\alpha + s)} < b_y \le \underline{i_{\alpha+1} - i_\alpha}$ のとき．

$$\begin{aligned} &(d_3(b_y)+d_4(b_y)) - (d'_3(b_y)+d'_4(b_y)) \\ &= ((b_y - (\underline{i_\alpha - i_{\alpha-1}})) + 0) - ((b_y - (\underline{(i_\alpha + s) - i_{\alpha-1}})) + (b_y - (\underline{i_{\alpha+1} - (i_\alpha + s)}))) \\ &= -b_y + (\underline{i_{\alpha+1} - (i_\alpha + s)}) + (\underline{(i_\alpha + s) - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \\ &= -b_y + (\underline{i_{\alpha+1} - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \quad (\because i_{\alpha-1} \preceq \underline{i_\alpha + s} \preceq i_{\alpha+1} \preceq \text{ と式 (6)}) \\ &= -b_y + (\underline{i_{\alpha+1} - i_\alpha}) + (\underline{i_\alpha - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \quad (\because i_{\alpha-1} \preceq i_\alpha \preceq i_{\alpha+1} \preceq \text{ と式 (6)}) \end{aligned}$$

$$= -b_y + (\underline{i_{\alpha+1} - i_\alpha}) \geq 0 \quad (31)$$

である．

（v）$\underline{i_{\alpha+1} - i_\alpha} < b_y \leq n-1$ のとき．

$$\begin{aligned}
&(d_3(b_y) + d_4(b_y)) - (d'_3(b_y) + d'_4(b_y)) \\
=\ &((b_y - (\underline{i_\alpha - i_{\alpha-1}})) + (b_y - (\underline{i_{\alpha+1} - i_\alpha}))) \\
&-((b_y - (\underline{(i_\alpha + s) - i_{\alpha-1}})) + (b_y - (\underline{i_{\alpha+1} - (i_\alpha + s)}))) \\
=\ &((\underline{i_{\alpha+1} - (i_\alpha + s)}) + (\underline{(i_\alpha + s) - i_{\alpha-1}})) - ((\underline{i_{\alpha+1} - i_\alpha}) + (\underline{i_\alpha - i_{\alpha-1}})) \\
=\ &(\underline{i_{\alpha+1} - i_{\alpha-1}}) - (\underline{i_{\alpha+1} - i_{\alpha-1}}) \quad (\because i_{\alpha-1} \preceq i_\alpha \preceq \underline{i_\alpha + s} \preceq i_{\alpha+1} \preceq \text{と式 (6)}) \\
=\ &0 \quad (32)
\end{aligned}$$

である．

（vi）$b_y = n$ のとき．

$$(d_3(b_y) + d_4(b_y)) - (d'_3(b_y) + d'_4(b_y)) = (1+1) - (1+1) = 0 \quad (33)$$

である．

以上の式 (21)，(22)，(28)，(29)，(30)，(31)，(32)，(33) より，

$$D_3 + D_4 = \sum_{b_y=0}^{n} (d_3(b_y) + d_4(b_y)) > \sum_{b_y=0}^{n} (d'_3(b_y) + d'_4(b_y)) = D'_3 + D'_4 \quad (34)$$

が成り立つ．なお，式 (34) における大小関係が $\geq$ ではなく $>$ になるのは，式 (29) において $>$ が入っており，かつ，式 (27) より，式 (29) を満たす b_y が少なくとも 1 個存在するためである[⋆1]．

式 (16)，(17)，(18)，(19)，(21)，(34) より，

$$\begin{aligned}
M(C^{\sigma_y}(A); S_x) &= D_1 + D_2 - (D_3 + D_4 - D_5) \\
&< D'_1 + D'_2 - (D'_3 + D'_4 - D'_5) \\
&= M(C^{\sigma_y}(A); \mathit{extend}(S_x, i_\alpha, s))
\end{aligned}$$

が成り立つ．以上より，補題 3 が示された．　■

以上によって，補題 1，補題 2，補題 3 が示された．次に，以下の補題を考えて証明する：

補題 4　$T^0 = \{\sigma_y(0), \sigma_y(1), \ldots, \sigma_y(b_x - 1)\}$ とする．このとき，アドレス集合 T^0 に対して，*shift* または *mirror* または *extend* の写像を有限回適用することによって，$P_{b_x}(A)$ に属する任意のアドレス集合 $S_x \in P_{b_x}(A)$ を構成することができる．

補題 4 を示すために，いくつか準備を行う．$S_x \in P_{b_x}(A)$ なる任意の S_x は，$i_0 \prec i_1 \prec \ldots \prec i_{b_x-1}$ なる整数 i_0，i_1，…，i_{b_x-1} を用いて，

$$S_x = \{\sigma_y(i_0), \sigma_y(i_1), \ldots, \sigma_y(i_{b_x-1})\}$$

と表すことができる（図 30 (A)）．このとき，すべての j（$0 \leq j \leq b_x - 1$）に関して，$\underline{i_k - i_{k-1}} \geq \underline{i_j - i_{j-1}}$ を満たすような k $(0 \leq k \leq b_x - 1)$ が少なくとも 1 個存在するので，そのような k のうちの 1 個を α とおく．すなわち，α は，

$$\forall j\ (0 \leq j \leq b_x - 1) : \underline{i_\alpha - i_{\alpha-1}} \geq \underline{i_j - i_{j-1}} \quad (35)$$

を満たす．図形的には，点の間の距離が最大になるような 2 点を i_α と $i_{\alpha-1}$ とおいている．

さらに，以下の操作 O を考える：

$v \leftarrow b_x - 1$
$T \leftarrow T^0$
$k \leftarrow 0$
while $k < b_x - 1$ **do**
　$T \leftarrow \mathit{mirror}(T, k)$　　/* step1 */
　$v \leftarrow \underline{v+1}$　　/* step2 */
　if $\underline{i_{\alpha+k+1} - i_{\alpha+k} - 1} \neq 0$ **then**
　　$T \leftarrow \mathit{extend}(T, v, \underline{i_{\alpha+k+1} - i_{\alpha+k} - 1})$　　/* step3 */
　endif
　$v \leftarrow \underline{v + i_{\alpha+k+1} - i_{\alpha+k} - 1}$　　/* step4 */
　$k \leftarrow k + 1$
endwhile
$T \leftarrow \mathit{shift}(T, i_\alpha - (b_x - 1))$　　/* step5 */

操作 O の図形的意味を確認するために，たとえば，$n = 4$，$b_x = 3$ とし，$T_0 = \{\sigma_y(0), \sigma_y(1), \sigma_y(2)\}$ に対して操作 O を適用することで，$S_x = \{\sigma_y(1), \sigma_y(3), \sigma_y(7)\}$ を得るまでの手続きを**図 31** に示す．いまの場合，$i_0 = 1$，$i_1 = 3$，$i_2 = 7$ に関して，$\underline{i_0 - i_2} \geq \underline{i_2 - i_1}$，$\underline{i_0 - i_2} \geq \underline{i_1 - i_0}$ であるから，$i_\alpha = i_0$ であることに注意したい．

明らかに，操作 O は，*shift* または *mirror* または *extend* の写像を有限回適用することで終了する．したがって，補題 4 を示すためには，操作 O が終了したとき T が S_x に一致することを示せばよい．そこで，以下の補題を考える：

補題 5　操作 O における k 回目のループの先頭において，以下のループ不変条件が成立

⋆1 式 (30) でも $>$ が入っているが，式 (27) より，式 (30) を満たす b_y は存在しない可能性がある．

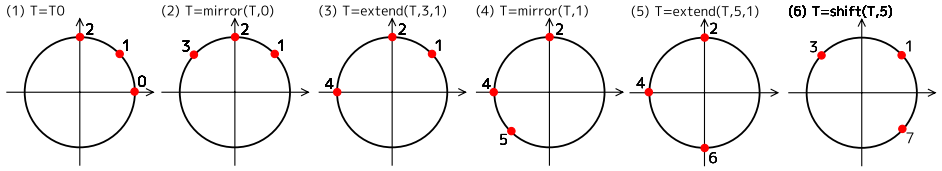

図 31　$T^0 = \{\sigma_y(0), \sigma_y(1), \sigma_y(2)\}$ に対して操作 O を適用することで，$S_x = \{\sigma_y(1), \sigma_y(3), \sigma_y(7)\}$ を得るまでの手続き

Fig. 31　How to derive $S_x = \{\sigma_y(1), \sigma_y(3), \sigma_y(7)\}$ from $T^0 = \{\sigma_y(0), \sigma_y(1), \sigma_y(2)\}$ by applying an operation O.

する：

$$v = \underline{b_x - 1 + i_{\alpha+k} - i_\alpha}, \tag{36}$$

$$T = \{\sigma_y(k), \sigma_y(k+1), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + i_{\alpha+1} - i_\alpha}), \ldots, \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha})\}. \tag{37}$$

補題 5 を数学的帰納法で示す．まず，$k = 0$ の場合には，

$v = b_x - 1$,

$T = T^0 = \{\sigma_y(0), \sigma_y(1), \ldots, \sigma_y(b_x - 1)\}$

であるから，明らかにループ不変条件 (36)，(37) が成り立つ．

次に，k の場合にループ不変条件 (36)，(37) が成り立つことを仮定して，$k+1$ の場合にもループ不変条件 (36)，(37) が成り立つことをいう．

第 1 に，式 (36) について考える．k 回目のループの先頭において，$v = \underline{b_x - 1 + i_{\alpha+k} - i_\alpha}$ が成り立つとすれば，step2 の操作によって，v は，

$$v = \underline{\underline{b_x - 1 + i_{\alpha+k} - i_\alpha} + 1} = \underline{b_x + i_{\alpha+k} - i_\alpha} \quad (\because \text{式 (4)}) \tag{38}$$

に変化する．さらに step4 の操作によって，v は，

$$v = \underline{\underline{b_x + i_{\alpha+k} - i_\alpha} + i_{\alpha+k+1} - i_{\alpha+k} - 1} = \underline{b_x - 1 + i_{\alpha+k+1} - i_\alpha} \quad (\because \text{式 (4)})$$

に変化し，これが $k+1$ 回目のループの先頭で成り立つ．したがって，式 (36) がループ不変条件であることが示された．

第 2 に，式 (37) について考える．k 回目のループの先頭において，

$$T = \{\sigma_y(k), \sigma_y(k+1), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + i_{\alpha+1} - i_\alpha}), \ldots, \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha})\}$$

が成り立つことを仮定する．step1 の操作によって，T は，

$$\begin{aligned} T &= \{\sigma_y(k+1), \sigma_y(k+2), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + i_{\alpha+1} - i_\alpha}), \ldots, \\ &\quad \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha}), \sigma_y(\underline{\underline{b_x - 1 + i_{\alpha+k} - i_\alpha} + (k+1) - k})\} \\ &= \{\sigma_y(k+1), \sigma_y(k+2), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + i_{\alpha+1} - i_\alpha}), \ldots, \\ &\quad \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha}), \sigma_y(\underline{b_x + i_{\alpha+k} - i_\alpha})\} \end{aligned} \tag{39}$$

に変化する．次にこの T に対して step3 の操作を適用することを考えるが，step3 の操作を適用する前に，この時点で写像 *extend* を適用できるための条件が成立していることを確認しなければならない．写像 *extend* の定義により，

$T = \{\sigma_y(\tilde{i}_0), \sigma_y(\tilde{i}_1), \ldots, \sigma_y(\tilde{i}_{b_x-1})\}$ $(\tilde{i}_0 \prec \tilde{i}_1 \prec \ldots \prec \tilde{i}_{b_x-1} \prec)$

に対して，写像 $extend(T, \tilde{i}_\alpha, s)$ を適用するためには，

$$\sigma_y(\tilde{i}_\alpha) \in T, \tag{40}$$

$$\tilde{i}_\alpha \prec \underline{\tilde{i}_\alpha + s} \prec \tilde{i}_{\alpha+1} \prec, \tag{41}$$

$$\underline{(\tilde{i}_\alpha + s) - \tilde{i}_{\alpha-1}} \leq \underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} \tag{42}$$

の 3 条件が満足されなければならない．そこで，式 (39) に対して step3 の操作を適用する時点で確かに式 (40)，(41)，(42) が満足されていることを確認する．

式 (39) に対して step3 の操作を適用する時点では，式 (38) より $v = \underline{b_x + i_{\alpha+k} - i_\alpha}$ になっているから，この時点における T，$\tilde{i}_{\alpha-1}$，$\tilde{i}_\alpha$，$\tilde{i}_{\alpha+1}$，s の値は，それぞれ，

$$\begin{aligned} T &= \{\sigma_y(k+1), \sigma_y(k+2), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + i_{\alpha+1} - i_\alpha}), \ldots, \\ &\quad \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha}), \sigma_y(\underline{b_x + i_{\alpha+k} - i_\alpha})\}, \\ \tilde{i}_{\alpha-1} &= \underline{b_x - 1 + i_{\alpha+k} - i_\alpha}, \\ \tilde{i}_\alpha &= v = \underline{b_x + i_{\alpha+k} - i_\alpha}, \\ \tilde{i}_{\alpha+1} &= k+1, \\ s &= \underline{i_{\alpha+k+1} - i_{\alpha+k} - 1} \end{aligned}$$

になっていることに注意する．

(I)　式 (40) が成り立つことを確認する．$\sigma_y(v) \in T$ が成り立つので，式 (40) は成り立つ．

(II)　式 (41) が成り立つことを確認する．そのためにまず，$\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)}$ と $\underline{\tilde{i}_{\alpha+1} - \tilde{i}_\alpha}$ を計算する．

$\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)}$ については，

$$\begin{aligned}\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} &= \underline{(k+1) - ((b_x + i_{\alpha+k} - i_\alpha) + (i_{\alpha+k+1} - i_{\alpha+k} - 1))} \\ &= \underline{(k+1) - (b_x - 1) - (i_{\alpha+k+1} - i_{\alpha+k}) + (i_{\alpha+k} - i_\alpha)} \quad (\because \text{式 (4)}) \\ &= \underline{(k+1) - (b_x - 1) - (i_{\alpha+k+1} - i_\alpha)} \\ &\qquad (\because i_\alpha \preceq i_{\alpha+k} \preceq i_{\alpha+k+1} \preceq \text{と式 (6)}) \\ &= \underline{n + (k+1) - (b_x - 1) - (i_{\alpha+k+1} - i_\alpha)} \qquad (43)\end{aligned}$$

が得られる．ここで式 (43) がとりうる値の範囲を考えると，

$$\begin{aligned} n &> n + (k+1) - (b_x - 1) - (i_{\alpha+k+1} - i_\alpha) \quad (\because k \leq b_x - 2 \text{ および } i_{\alpha+k+1} \neq i_\alpha) \\ &\geq n - (i_{\alpha+b_x-1} - i_{\alpha+k+1}) - (i_{\alpha+k+1} - i_\alpha) \\ &\qquad (\because i_{\alpha+k+1} \prec i_{\alpha+k+2} \prec \ldots \prec i_{\alpha+b_x-1} \prec \text{より} \\ &\qquad i_{\alpha+b_x-1} - i_{\alpha+k+1} \geq (b_x - 1) - (k+1)) \\ &= n - (i_{\alpha+b_x-1} - i_\alpha) \quad (\because i_\alpha \preceq i_{\alpha+k+1} \preceq i_{\alpha+b_x-1} \preceq \text{と式 (6)}) \\ &= n - (i_{\alpha-1} - i_\alpha) \quad (\because \mathrm{mod}(\alpha + b_x - 1, b_x) = \mathrm{mod}(\alpha - 1, b_x)) \\ &= i_\alpha - i_{\alpha-1} \quad (\because i_{\alpha-1} \preceq i_\alpha \preceq \text{と式 (5)}) \\ &> 0 \quad (\because i_\alpha \neq i_{\alpha-1}) \qquad (44)\end{aligned}$$

が成り立つ．よって，式 (3)，(43)，(44) より，

$$\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} = n + (k+1) - (b_x - 1) - (i_{\alpha+k+1} - i_\alpha) \qquad (45)$$

が成り立つ．

$\underline{\tilde{i}_{\alpha+1} - \tilde{i}_\alpha}$ については，

$$\begin{aligned}\underline{\tilde{i}_{\alpha+1} - \tilde{i}_\alpha} &= \underline{(k+1) - (b_x + i_{\alpha+k} - i_\alpha)} \\ &= \underline{k - (b_x - 1) - (i_{\alpha+k} - i_\alpha)} \quad (\because \text{式 (4)}) \\ &= \underline{n + k - (b_x - 1) - (i_{\alpha+k} - i_\alpha)} \qquad (46)\end{aligned}$$

となる．この式 (46) は，式 (43) における $k+1$ を k に置き換えたものであるから，式 (44) を導いたときの議論と同様にして，

$$n > n + k - (b_x - 1) - (i_{\alpha+k} - i_\alpha) > 0 \qquad (47)$$

がいえる．よって，式 (3)，(46)，(47) より，

$$\underline{\tilde{i}_{\alpha+1} - \tilde{i}_\alpha} = n + k - (b_x - 1) - (i_{\alpha+k} - i_\alpha) \qquad (48)$$

が成り立つ．

以上の結果をもとにして，$\underline{(\tilde{i}_\alpha + s) - \tilde{i}_\alpha} + \underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} + \underline{\tilde{i}_\alpha - \tilde{i}_{\alpha+1}}$ を計算すると，

$$\begin{aligned} &\underline{(\tilde{i}_\alpha + s) - \tilde{i}_\alpha} + \underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} + \underline{\tilde{i}_\alpha - \tilde{i}_{\alpha+1}} \\ =\ & \underline{s} + \underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} + (n - \underline{\tilde{i}_{\alpha+1}} - \tilde{i}_\alpha) \quad (\because \text{式 (4)},\ i_\alpha \preceq i_{\alpha+1} \preceq \text{と式 (5)}) \\ =\ & (\underline{i_{\alpha+k+1} - i_{\alpha+k} - 1}) + (n + (k+1) - (b_x - 1) - (\underline{i_{\alpha+k+1} - i_\alpha})) \\ & + (n - (n + k - (b_x - 1) - (\underline{i_{\alpha+k} - i_\alpha}))) \quad (\because \text{式 (45) (48)}) \\ =\ & ((\underline{i_{\alpha+k+1} - i_{\alpha+k} - 1}) + 1) + (n - (\underline{i_{\alpha+k+1} - i_\alpha})) + (\underline{i_{\alpha+k} - i_\alpha}) \\ =\ & ((\underline{i_{\alpha+k+1} - i_{\alpha+k}}) - 1 + 1) + (\underline{i_\alpha - i_{\alpha+k+1}}) + (\underline{i_{\alpha+k} - i_\alpha}) \\ & (\because i_{\alpha+k+1} \neq i_{\alpha+k},\ i_\alpha \preceq i_{\alpha+k+1} \preceq \text{と式 (5)}) \\ =\ & n \quad (\because i_\alpha \preceq i_{\alpha+k} \preceq i_{\alpha+k+1} \preceq \text{と式 (5)}) \end{aligned}$$

となる．したがって，式 (5) より，$\tilde{i}_\alpha \preceq \underline{\tilde{i}_\alpha + s} \preceq \tilde{i}_{\alpha+1} \preceq$ が成り立つ．さらに，式 (44)，(45) より $\tilde{i}_{\alpha+1} \neq \underline{\tilde{i}_\alpha + s}$ であり，式 (47)，(48) より $\tilde{i}_{\alpha+1} \neq \tilde{i}_\alpha$ である．また，操作 O の定義より step3 を適用できるのは $s = \underline{i_{\alpha+k+1} - i_{\alpha+k} - 1} \neq 0$ のときであるから，$\tilde{i}_\alpha \neq \underline{\tilde{i}_\alpha + s}$ である．結局，$\tilde{i}_\alpha \prec \underline{\tilde{i}_\alpha + s} \prec \tilde{i}_{\alpha+1} \prec$ が成り立つ．つまり，式 (41) が成り立つことが確認できた．

（Ⅲ） 式 (42) が成り立つことを確認する．ここで，式 (42) の左辺である $\underline{(\tilde{i}_\alpha + s) - \tilde{i}_{\alpha-1}}$ を計算すると，

$$\begin{aligned}\underline{(\tilde{i}_\alpha + s) - \tilde{i}_{\alpha-1}} &= \underline{((b_x + i_{\alpha+k} - i_\alpha) + (i_{\alpha+k+1} - i_{\alpha+k} - 1)) - (b_x - 1 + i_{\alpha+k} - i_\alpha)} \\ &= \underline{i_{\alpha+k+1} - i_{\alpha+k}} \quad (\because \text{式 (4)}) \qquad (49)\end{aligned}$$

が得られる．一方で，右辺の $\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)}$ に関しては，式 (44)，(45) より，

$$\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} = n + (k+1) - (b_x - 1) - (\underline{i_{\alpha+k+1} - i_\alpha}) \geq \underline{i_\alpha - i_{\alpha-1}} \qquad (50)$$

が成り立つ．よって，式 (35)，(49)，(50) より，

$$\underline{(\tilde{i}_\alpha + s) - \tilde{i}_{\alpha-1}} \leq \underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)}$$

が成り立つ．つまり，式 (42) が成り立つことが確認できた．

以上の（Ⅰ），（Ⅱ），（Ⅲ）より，式 (39) に対して step3 の操作を適用する時点で，確かに式 (40)，(41)，(42) が満足されていることを確認できた．

そこで，式 (39) に対して step3 の操作を適用すると，T は，

$$\begin{aligned}T &= \{\sigma_y(k+1),\sigma_y(k+2),\ldots,\sigma_y(b_x-1),\sigma_y(\underline{b_x-1+(i_{\alpha+1}-i_\alpha)}),\ldots,\\ &\quad \sigma_y(\underline{b_x-1+i_{\alpha+k}-i_\alpha}),\sigma_y(\underline{b_x+i_{\alpha+k}-i_\alpha}+\underline{i_{\alpha+k+1}-i_{\alpha+k}-1})\}\\ &= \{\sigma_y(k+1),\sigma_y(k+2),\ldots,\sigma_y(b_x-1),\sigma_y(\underline{b_x-1+(i_{\alpha+1}-i_\alpha)}),\ldots,\\ &\quad \sigma_y(\underline{b_x-1+i_{\alpha+k}-i_\alpha}),\sigma_y(\underline{b_x-1+i_{\alpha+k+1}-i_\alpha})\}\quad(\because \text{式 (4)})\end{aligned}$$

に変化し，これが $k+1$ 回目のループの先頭で成り立つ．以上により，式 (37) がループ不変条件であることが示された．つまり，補題 5 が示された．　■

補題 4 を示す．補題 5 より，操作 O におけるループを抜けた直後の T は，ループ不変条件において $k=b_x-1$ としたもの，すなわち，

$$T=\{\sigma_y(b_x-1),\sigma_y(\underline{b_x-1+i_{\alpha+1}-i_\alpha}),\ldots,\sigma_y(\underline{b_x-1+i_{\alpha+b_x-1}-i_\alpha})\}$$

となっている．よって，この T に対して step5 の操作を適用すると，T は，

$$\begin{aligned}T &= \{\sigma_y(\underline{b_x-1+i_\alpha-(b_x-1)}),\sigma_y(\underline{b_x-1+i_{\alpha+1}-i_\alpha}+i_\alpha-(b_x-1)),\ldots,\\ &\quad \sigma_y(\underline{b_x-1+i_{\alpha+b_x-1}-i_\alpha}+i_\alpha-(b_x-1))\}\\ &= \{\sigma_y(\underline{i_\alpha}),\sigma_y(\underline{i_{\alpha+1}}),\ldots,\sigma_y(\underline{i_{\alpha+b_x-1}})\}\quad(\because \text{式 (4)})\\ &= \{\sigma_y(i_0),\sigma_y(i_1),\ldots,\sigma_y(i_{b_x-1})\}\\ &= S_x\end{aligned}$$

に変化する．以上によって，操作 O が終了したとき T が S_x に一致することが示された．つまり，補題 4 が示された．　■

以上の系 1，補題 2，補題 3，補題 4 を用いて，命題 2 を示す．系 1，補題 2 より，任意のアドレス集合 $S_x\in P_{b_x}(A)$ に対して写像 *shift* または写像 *mirror* を 1 回以上の任意回適用したアドレス集合を S'_x とすると，$M(C^{\sigma_y}(A);S'_x)=M(C^{\sigma_y}(A);S_x)$ が成り立つ．また，補題 3 より，任意のアドレス集合 $S_x\in P_{b_x}(A)$ に対して写像 *extend* を 1 回以上の任意回適用したアドレス集合を S'_x とすると，$M(C^{\sigma_y}(A);S'_x)>M(C^{\sigma_y}(A);S_x)$ が成り立つ．さらに，補題 4 より，任意のアドレス集合 $S_x\in P_{b_x}(A)$ は，T^0 に対して，写像 *shift* または写像 *mirror* または写像 *extend* を有限回適用することによって得られる．これらの事実を総合すると，$M(C^{\sigma_y}(A);S_x)$ を最小化する S_x とは，T^0 に対して，写像 *shift* または写像 *mirror* を有限回適用して得られるアドレス集合のみであるといえる．そのようなアドレス集合とは，具体的には，

$$\begin{aligned}T^0 &= \{\sigma_y(0),\sigma_y(1),\ldots,\sigma_y(b_x-2),\sigma_y(b_x-1)\},\\ T^1 &= \{\sigma_y(1),\sigma_y(2),\ldots,\sigma_y(b_x-1),\sigma_y(b_x)\},\\ &\vdots\\ T^{n-1} &= \{\sigma_y(n-b_x+1),\sigma_y(n-b_x+2),\ldots,\sigma_y(n-1),\sigma_y(0)\}.\end{aligned}$$

のことであり，これは $C^{\sigma_y}_{b_x}(A)$ にほかならない．以上によって，$P_{b_x}(A)$ に属するすべてのアドレス集合 $S_x\in P_{b_x}(A)$ のうち，$M(C^{\sigma_y}(A);S_x)$ を最小化する S_x の集合は $C^{\sigma_y}_{b_x}(A)$ であることが示された．つまり，命題 2 が示された．　■

命題 2 が示されたので，それのいい換えである命題 1 も成り立つ．よって，命題 1 が示された．　■

命題 1 が成り立つことを用いて，定理 2 を証明する．m 個のスレッド x_0，x_1，…，x_{m-1} を考え，各スレッド x_i は置換 σ_i に従って一様にアドレスを使用するとする．また，スレッド x_{u_0}，x_{u_1}，…，$x_{u_{k-1}}$ に関して，「どの 2 つの異なるスレッド x_i とスレッド x_j $(x_i,x_j\in\{x_{u_0},x_{u_1},\ldots,x_{u_{k-1}}\})$ に対しても，スレッド x_i が使用するアドレス集合とスレッド x_j が使用するアドレス集合が共通部分を持たない事象」を $E(x_{u_0},x_{u_1},\ldots,x_{u_{k-1}})$ と表す．また，事象 $E(x_{u_0},x_{u_1},\ldots,x_{u_{k-1}})$ が起きる確率を $p(E(x_{u_0},x_{u_1},\ldots,x_{u_{k-1}}))$ と表す．たとえば，$p(E(x_0,x_1))$ は，スレッド x_0 が使用するアドレス集合とスレッド x_1 が使用するアドレス集合が共通部分を持たない確率を意味する．$p(E(x_0,x_1,x_2))=p(E(x_0,x_1)\wedge E(x_1,x_2)\wedge E(x_2,x_0))$ などが成り立つ．

このとき，定理 2 が成り立つことを数学的帰納法で示す．

まず，$m=1$ の場合には明らかに定理 2 は成り立つ．また，$m=2$ の場合には，命題 1 より定理 2 は成り立つ．

次に，m（$m\geq 2$）のときに定理 2 が成り立つことを仮定して，$m+1$ のときにも定理 2 が成り立つことをいう．$p(E(x_0,x_1,\ldots,x_{m-1},x_m))$ を，条件付き確率を用いて分解すると，

$$\begin{aligned}&p(E(x_0,x_1,\ldots,x_{m-1},x_m))\\ =\ &p(E(x_0,x_1,\ldots,x_{m-1}))p(E(x_0,x_1,\ldots,x_{m-1})|&(51)\\ &(E(x_m,x_0)\wedge E(x_m,x_1)\wedge\ldots\wedge E(x_m,x_{m-1})))&(52)\end{aligned}$$

となる．いま，各スレッド x_i の置換の選び方は独立であることを用いて式 (52) を計算すると，

$$\begin{aligned}&p(E(x_0,x_1,\ldots,x_{m-1},x_m))\\ &= p(E(x_0,x_1,\ldots,x_{m-1}))p(E(x_m,x_0)\wedge E(x_m,x_1)\wedge\ldots\wedge E(x_m,x_{m-1}))\\ &= p(E(x_0,x_1,\ldots,x_{m-1}))p(E(x_m,x_0))p(E(x_m,x_1))\ldots p(E(x_m,x_{m-1}))&(53)\end{aligned}$$

となる．式 (53) において，$p(E(x_0,x_1,\ldots,x_{m-1}))$ が最大になるのは，数学的帰納法の仮定

より $\sigma_0 = \sigma_1 = \ldots = \sigma_{m-1}$ のときにかぎられる．また，式 (53) における各 $p(E(x_m, x_i))$ $(0 \leq i \leq m-1)$ が最大になるのは，命題 1 より $\sigma_m = \sigma_i$ のときにかぎられる．すなわち，式 (53) が最大になるのは，$\sigma_0 = \sigma_1 = \ldots = \sigma_{m-1} = \sigma_m$ のときにかぎられる．したがって，定理 2 が成り立つ．

以上より，定理 2 が成り立つことが証明できた．　■

A.1.3　アドレス衝突確率の定量的な評価

以上の証明の過程より，置換 σ_y と，2 つのアドレス集合 S_x^0 と S_x^1 が与えられたとき，「S_x^0 と S_x^1 とでは，どちらがどれくらい，置換 σ_y に従って一様に使用されるアドレス集合とアドレスが衝突しやすいのか」を定量的に計算することができる．

スレッド x とスレッド y を考え，スレッド y は置換 σ_y に従って一様にアドレス集合を使用しているとし，スレッド x は b_x 個のアドレスを割り当てようとしているとする．このとき，スレッド x が，b_x 個のアドレスを割り当てるためにアドレス集合 $S_x^0 \in P_{b_x}(A)$ を使用する場合とアドレス集合 $S_x^1 \in P_{b_x}(A)$ を使用する場合とでは，前者の方が後者より，$M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); S_x^1)$ だけ，スレッド y が使用するアドレス集合とアドレスが衝突しやすい[⋆1]．以下では，$M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); S_x^1)$ の値を計算する．

まず，任意のアドレス集合 $S_x = \{\sigma_y(i_0), \sigma_y(i_1), \ldots, \sigma_y(i_{b_x-1})\}$ $(i_0 \prec i_1 \prec \ldots \prec i_{b_x-1} \prec)$ に対して，$M(C^{\sigma_y}(A); extend(S_x, i_\alpha, s)) - M(C^{\sigma_y}(A); S_x)$ の値を計算すると，式 (16)，(17)，(18)，(19)，(21)，(21)，(22)，(28)，(29)，(30)，(31)，(32)，(33) より，

$$\begin{aligned} & M(C^{\sigma_y}(A); extend(S_x, i_\alpha, s)) - M(C^{\sigma_y}(A); S_x) \\ = & \sum_{b_y=\underline{i_\alpha - i_{\alpha-1}+1}}^{\underline{(i_\alpha+s)-i_{\alpha-1}}} (b_y - (\underline{i_\alpha - i_{\alpha-1}})) + \sum_{b_y=\underline{(i_\alpha+s)-i_{\alpha-1}+1}}^{\underline{i_{\alpha+1}-(i_\alpha+s)}} (\underline{s}) \\ & + \sum_{b_y=\underline{i_{\alpha+1}-(i_\alpha+s)+1}}^{\underline{i_{\alpha+1}-i_\alpha}} (-b_y + (\underline{i_{\alpha+1} - i_\alpha})) \end{aligned} \tag{54}$$

となる．

ここで，アドレス集合 T^0 に対して操作 O を適用することによってアドレス集合 S_x^0 と S_x^1 を構成することを考える．すると，$M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); S_x^1)$ は，

$$\begin{aligned} & M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); S_x^1) \\ = & (M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); T^0)) - (M(C^{\sigma_y}(A); S_x^1) - M(C^{\sigma_y}(A); T^0)) \end{aligned} \tag{55}$$

と表せる．式 (55) において，$M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); T^0)$ は，T^0 から操作 O によって S_x^0 を構成するときに，操作 O の step3 を適用するたびに式 (54) を計算し，その総和を求めることで得られる．同様に，$M(C^{\sigma_y}(A); S_x^1) - M(C^{\sigma_y}(A); T^0)$ は，T^0 から操作 O によって S_x^1 を構成するときに，操作 O の step3 を適用するたびに式 (54) を計算し，その総和を求めることで得られる．このように，$M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); S_x^1)$ は単純な式にはならないが，実際の数値を代入することで計算可能である．

(平成 22 年 7 月 5 日受付)
(平成 22 年 11 月 18 日採録)

原　健太朗（学生会員）
1986 年生．2009 年東京大学学士（工学）．同年より東京大学大学院情報理工学系研究科電子情報学専攻在学中．

中島　　潤（学生会員）
1986 年生．2009 年東京大学学士（工学）．同年より東京大学大学院情報理工学系研究科電子情報学専攻在学中．

田浦健次朗（正会員）
1969 年生．1997 年東京大学大学院理学博士（情報科学専攻）．1996 年より東京大学大学院理学系研究科情報科学専攻助手．2001 年より東京大学大学院情報理工学系研究科電子情報学専攻講師．2002 年より同助教授．2007 年より同准教授．日本ソフトウェア科学会，ACM，IEEE-CS 各会員．

⋆1 ただし，$M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); S_x^1)$ の次元は確率ではない．