修士論文

再構成可能な高性能並列計算のための PGAS プログラミング処理系

A PGAS Programming Framework for Reconfigurable and High-Performance Parallel Computations

指導教員 田浦 健次朗 准教授

東京大学 情報理工学系研究科
電子情報学専攻

氏名 48-096419 原 健太朗

提出日 2011 年 2 月 9 日

概要

高性能マルチコアプロセッサの低価格化，ネットワークの高バンド幅化，メモリやディスクの大容量化などのハードウェア計算環境の技術革新にともなって，並列分散アプリケーションの適用領域や利用機会は飛躍的に拡大している．気象予測，金融計算，衝突解析，地震シミュレーション，波動シミュレーション，遺伝子解析，デバイス設計などの，産業界の各種応用分野における並列分散アプリケーションの利用機会の増大はめざましく，それら並列分散アプリケーションの実行を支える並列分散プログラミング処理系に求められる要請も多様化している．なかでも，本研究では，(1) 非定型な並列計算を性能を落とすことなく簡単に記述できること，(2) 再構成可能な並列計算を簡単に記述できることを目標として，PGAS モデルに基づく並列分散プログラミング処理系 DMI (Distributed Memory Interface) を提案して実装し，評価する．

本研究の第 1 の目標は，非定型な並列計算を性能を落とすことなく簡単に記述できるようにすることである．一般に，実用的な有限要素法や粒子法などの並列科学技術計算や大規模な Web グラフ解析では，非定型な領域分割やデータ通信が必要となる．MPI などのメッセージパッシングモデルに基づく処理系を利用すれば，これらの非定型な並列計算を高性能に記述することは可能だが，プログラマビリティに問題がある．一方で，UPC，X10，Chapel などのグローバルアドレス空間モデルに基づく処理系を利用すれば，容易なプログラミングコストでさまざまな並列計算を記述することが可能になる．しかし，既存の処理系では非定型な並列計算に対する API が不十分なうえに，リモートなデータアクセスをあまりに透過的に記述できてしまうため，どこでどのような通信が内部的に発生するのかをユーザプログラム側から把握しにくく，期待する性能を得ることは難しい．そこで，DMI では，グローバルアドレス空間を提供することで，非定型な並列計算を見通しよく記述できるようにしつつも，グローバルアドレス空間に対するアクセスが内部的に不必要な通信を発生させることがないよう，ユーザプログラム側からわかりやすくかつ強力に性能を最適化できるような API を設計する．DMI では，これらの API を使うことで，ユーザプログラム側から複数のアクセスを明示的に集約したり，内部的に起きる通信を簡単に制御したり，アクセスローカリティを強力に最適化したりすることが簡単にできる．

本研究の第 2 の目標は，再構成可能な並列計算を簡単に記述できるようにすることである．一般に，計算環境が大規模化しそれを利用する並列計算が増加すると，多数の並列計算間で多数の計算資源を効率よく柔軟にスケジューリングする必要が生じる．すなわち，計算資源の利用率を最大化させ，並列計算の結果をいち早くユーザに届けるためには，長時間を要しうる 1 個の並列計算を最初から最後まで固定的な計算資源を占有して実行させるのではなく，周囲の並列計算全体の負荷状況をふまえて，各並

列計算に割り当てる計算資源を動的に増減させることが重要になる．そして，このような柔軟なスケジューリングを行うためには，当然，並列計算それ自体が計算規模を自由に拡張/縮小できるように記述されていなければならない．そこで，DMI では，ノードの動的な参加/脱退を越えてグローバルアドレス空間のコヒーレンシが維持されるようなプロトコルを設計したうえで，再構成可能な並列計算を簡単に記述するためのプログラミングモデルを 3 種類提案し，それらの性能とプログラマビリティを比較検討する．とくに，プログラマビリティの観点からは，「プログラマは並列計算の再構成を意識することなく，単に十分な数のスレッドを生成するだけでよい．すると，処理系がそれら大量のスレッドを，各時点で利用可能な計算資源に対して動的にマッピングすることで，並列計算の再構成を透過的に実現してくれる」というプログラミングモデルが重要であることを指摘し，そのための要素技術である透過的なスレッド移動について新規的な手法を提案する．

評価の結果，第 1 の目標に関しては，実用的な有限要素法による応力解析や大規模な Web グラフ解析などの非定型な並列計算に対して，DMI は，MPI よりも容易なプログラミングコストで，MPI と同程度もしくは優れた性能を達成することを確認できた．第 2 の目標に関しては，並列計算の再構成に対応していない DMI のプログラムに対してわずか 1 行を追加するだけで再構成可能な並列計算を記述でき，利用可能なノード数の増減に対応して効果的に並列度を増減させられることを確認できた．

目次

図目次

表目次

第 1 章

序論

1.1 背景と目的

高性能マルチコアプロセッサの低価格化，ネットワークの高バンド幅化，メモリやディスクの大容量化などのハードウェア計算環境の技術革新にともなって，並列分散アプリケーションの適用領域や利用機会は飛躍的に拡大している．気象予測，金融計算，衝突解析，地震シミュレーション，波動シミュレーション，遺伝子解析，デバイス設計などの，産業界の各種応用分野における並列分散アプリケーションの発展はめざましい．たとえば，スーパーコンピュータ IBM Blue Gene の 294912 個のプロセッサを用いた 2.64 億個の粒子の流体解析 [71]，スーパーコンピュータ Cray-XT5 Jagular の 95256 個のプロセッサを用いた電子フォノン散乱のシミュレーション [122]，66 億本のリンクを持った Web グラフのページランク計算 [47] など，大規模な並列分散アプリケーションがさまざまな分野で成果をあげている．また，これらハイエンドな計算環境でなくとも，汎用アーキテクチャで構成されたワークステーションを組み合わせたコモディティクラスタ環境が大学や大企業などを中心に普及しており，数十プロセッサ〜数百プロセッサ程度の計算資源が手に入りやすい時代になっている．さらに近年では，Amazon EC2[1] や Windows Azure[12]，Google App Engine[2] など，計算資源を従量制課金のサービスとして提供するクラウドコンピューティングサービス [21, 123, 124, 190, 34] も台頭してきており，小規模な企業や個人であっても，大規模な投資を行うことなく手軽に大規模な計算環境を利用できるようになっている．たとえば Amazon EC2 では，わずか 0.085 ドルで 1 台の Linux OS を 1 時間利用することができ，8.5 ドルで 100 台の Linux OS を 1 時間利用することもできる．このように，並列分散アプリケーションの適用領域と利用機会は確実に増大している．

一般に，これらの並列分散アプリケーションは，何らかの並列分散プログラミング処理系を基盤として開発されており，並列分散アプリケーションの性能や機能は，基盤として用いられる並列分散プログラミング処理系の性能や機能に支えられている．そのため，並列分散プログラミング処理系に対しては，さまざまな並列分散アプリケーションに対する性能とプログラマビリティがもっとも基本的な要請として課せられるほか，機能面での要請もますます多様化している．なかでも，本研究では，既存の並列分散プログラミング処理系ではまだ十分に達成されていない重要な要請として，以下の 2 つの要請に

着眼する：

要請 I　非定型な並列計算を性能を落とすことなく簡単に記述できること
要請 II　再構成可能な並列計算を簡単に記述できること

ここで，並列計算が再構成可能であるとは，並列計算の計算規模を動的に自由に拡張したり縮小したりできることを意味する．本研究では，これらの要請を満たす，高性能並列科学技術計算のための並列分散プログラミング処理系として，**DMI（Distributed Memory Interface）**を提案して実装し，評価する．

以降では，上記の 2 つの要請をより詳細化したうえで，要請を満足させるために本研究が採用するアプローチについて概観する．

1.2　要請 I：非定型な並列計算に対する性能とプログラマビリティ

1.2.1　非定型な並列計算に対する要請

本研究の第 1 の目標は，非定型な並列計算を性能を落とすことなく簡単に記述できるようにすることである．高性能並列科学技術計算のためのベンチマークには，定型的な並列計算しか扱っていないものが多い．たとえば，NAS Parallel Benchmark[51] の FT や MG，Himeno Benchmark[3] などで扱われるのは，直方体状の領域を直方体状の小領域に領域分割するような定型的な並列計算である．また，世界で最速のコンピュータシステムをランキングする Top500[9] の性能評価指標となっている LINPACK[20] で扱われるのは，定型的な密行列計算である．しかし，現実世界における有限要素法や粒子法，大規模な Web グラフ解析などでは，より複雑な形状を持った物体を複雑に領域分割したり，疎行列を扱ったりするような非定型な並列計算が要求される．そして，処理系によっては多次元配列などを使うことで簡単に処理を記述できてしまう定型的な並列計算と比較すると，非定型な並列計算は，プログラムを記述するのが難しく性能を引き出すのが難しい．したがって，非定型な並列計算を性能を落とすことなくいかに簡単に記述させられるかは，並列分散プログラミング処理系にとっての 1 つの重要な要請である．

1.2.2　性能とプログラマビリティのバランス

詳細は **2.1** 節で議論するが，一般に，並列分散プログラミング処理系を設計するうえでは，性能とプログラマビリティはトレードオフの関係にあり，このバランスをどのようにとるかが重要な選択になる．たとえば，MPI[63, 25] などのメッセージパッシングモデルに基づく処理系を利用すれば，データの分散配置や通信をユーザプログラム側からすべて明示的に指示できるため，非定型な並列計算であっても高性能に記述することはできるが，プログラマビリティは低い．一方で，Chapel[37, 26] や各種分散共有メモリ [111, 101, 15, 139, 120, 135, 136, 160, 204] などのグローバルアドレス空間モデルに基づく処理系を利用すれば，グローバルアドレス空間を read/write するだけで非定型なメモリアクセスも簡単に記述できるため，プログラマビリティは高い．しかし，簡単に記述したプログラムで期待する性能を得ることは難しく，さらに，リモートなデータへのアクセスをあまりに透過的に記述できてしま

うため，どこでどのような通信が内部的に発生するのかをユーザプログラム側から把握しにくいため，性能を最適化するのが難しい．そこで，本節では，性能とプログラマビリティのバランスをどのようにとるかに関する本研究の立場を明確化させる．

まず，並列プログラミングにおけるプログラム開発は以下のフェーズに分けられる：

プログラミング　性能はそれほど意識することなく，とりあえず正しい結果を出力するプログラムを記述し，デバッグする．

性能最適化　性能を最適化する．

実行　実行して結果を得る．

ここで，並列プログラミングにおける最終的な目的，いい換えると並列分散プログラミング処理系が達成すべき目的は，**実行時間を最短化することではなく，プログラミングを開始してから結果を得るまでの「全体」の時間を最短化する**ことである．プログラミング時間が 10 時間で実行時間が 1 秒であるような処理系よりも，プログラミング時間が 3 時間で実行時間が 6 時間であるような処理系の方が望ましい．ところが，ここで注意すべきことは，有限要素法や Web グラフ解析など本研究が対象にしようとしている並列計算では，作成されたプログラムが使い捨てにされることは少なく，多くの場合には，1 回のプログラミングあたり多数回の実行が行われるという点である．有限要素法であれ Web グラフ解析であれ，いったん並列プログラムを作成すれば，それがさまざまな入力データに対して何度も再利用され実行されることになる．したがって，並列プログラム開発の場合には，プログラミング時間よりも実行時間が重要になる．そして，実行時間を短縮するためには性能最適化が必須になるため，強力な性能最適化が可能であることも重要になる．以上を要約すると，並列プログラム開発においては，プログラマビリティよりも，**強力な性能最適化が可能であることと，性能がよく実行時間が短いことが第一義的な要請として課せられる**．そもそも，逐次プログラムではなくあえて並列プログラムを記述する理由は，プログラムの実行が多数回繰り返されることを想定して，「全体」の時間を最短化するためには実行時間を短縮することが最重要であるという動機づけが存在しているからにほかならない．現在，プログラマビリティは低いものの強力な性能最適化が可能で実際に性能の出る MPI が並列プログラミングにおけるデファクトスタンダードになっていることの背景には，以上のような事情が関係している．

このように，いくら性能最適化と性能が重要であるにせよ，当然ながら，プログラマビリティも高い方がよい．以上の観察に基づき，DMI の設計方針は，とくに非定型な並列計算に対して，**見通しのよい強力な性能最適化によってメッセージパッシングモデルと同等の性能を引き出せるという条件下で，できるかぎりプログラマビリティを高める**ことである（図 1.1）．設計の具体的内容とその根拠については第 **2** 章および第 **3** 章で詳しく述べるが，概要は次のとおりである．

第 1 に，プログラミングモデルとしては，プログラマビリティを高めるために**グローバルビュー型のグローバルアドレス空間モデル**を採用する．UPC[43, 62, 48, 46]，Global Arrays[144, 146, 143]，Chapel[37, 26]，X10[41]，XcalableMP[72] などの既存のグローバルビュー型のグローバルアドレス空間モデルに基づく処理系では，非定型な並列計算のための API が不十分であるのに対して，DMI では，非定型な並列計算のための API として **read-write-set** を新たに提案する．read-write-set を用いる

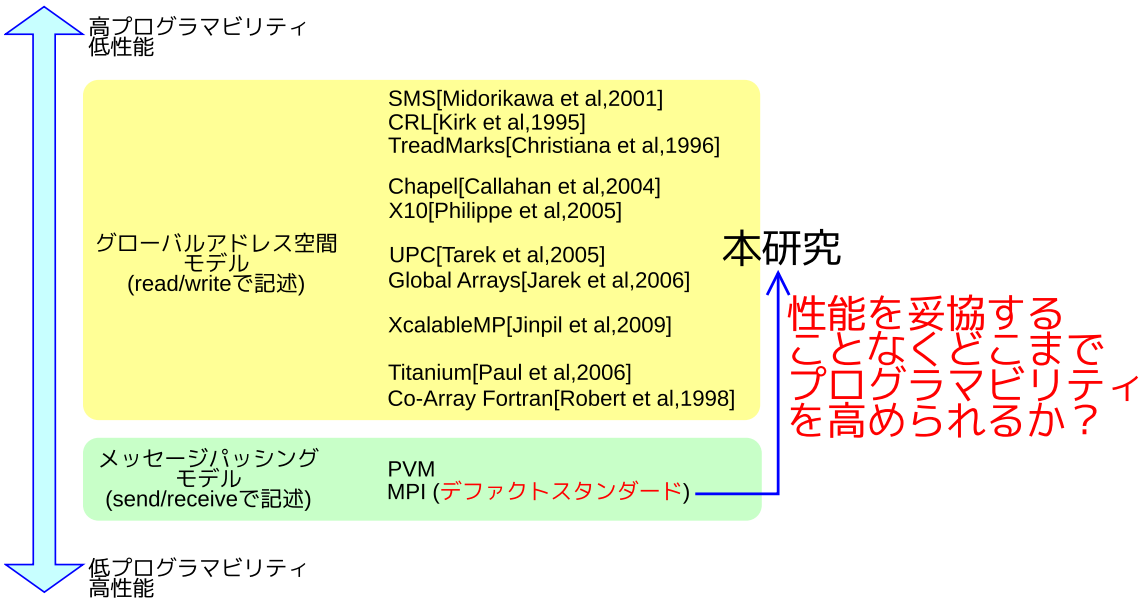

図 1.1 DMI の設計方針.

ことで，プログラマはグローバルビュー型のグローバルアドレス空間モデルに基づいて非定型なメモリアクセスを簡単に記述しつつも，内部的にはメッセージパッシングモデルと同様の無駄のない通信を発生させることができる．

第 2 に，既存のグローバルビュー型のグローバルアドレス空間モデルに基づく処理系では，内部的に発生する通信をユーザプログラムから把握しにくく性能最適化が困難だったのに対して，DMI では，性能を見通しよく強力に最適化できるような API を設計する．たとえば，ユーザプログラム側で複数のアクセスを明示的に集約したり，アクセスローカリティを強力に最適化したり，新しい read-modify-write 命令を定義したりできる．したがって，DMI の API は，透過的にリモートなデータにアクセスできる Chapel や X10，分散共有メモリなどと比較するとプログラマビリティが低く，プログラミングに要する時間は長くなってしまう．しかし，DMI では，これらの API を利用することで性能最適化に要する時間と実行時間を短縮するのが容易なので，多数回の実行を行うのであれば，結果的に，「全体」の時間を短縮できるような設計になっているという点を強調したい．

1.3 要請 II：並列計算の再構成

1.3.1 並列計算の再構成に対する要請

本研究の第 2 の目標は，再構成可能な並列計算 [186, 42, 128] を簡単に記述できるようにすることである．一般に，多数の計算資源を多数の利用者で共有利用する環境で長時間を要する並列計算を実行する場合，並列計算の再構成に対する要請が出てくる．

たとえば，T2K[7] や TSUBAME-2.0[11] などのスーパーコンピュータシステムで採用されている

ジョブスケジューラ [33] を考える．ジョブスケジューラの目的は，**計算資源の利用率を最大化**させ，投入されるジョブの結果をできるかぎり早く利用者に返すことであるが，TORQUE[10] などの一般的なジョブスケジューラではこの目的が十分に達成されているとはいえない．TORQUE などのジョブスケジューラでは，1000 ノードを要求するジョブ A を時刻 t_0 に投入した場合，1000 ノードが利用可能になった時点 $t_1(\geq t_0)$ でジョブ A がディスパッチされ，そのジョブ A は最初から最後まで 1000 ノードを利用して実行される．よって，たとえば，時刻 t_0 で 700 ノードが利用可能であったとしても，1000 ノードが利用可能になる時刻 t_1 まではジョブ A はディスパッチされない．しかし，仮に，時刻 t_0 で 700 ノードを利用してジョブ A をディスパッチしておき，時刻 t_1 でジョブ A の計算規模を 1000 ノードに拡張することができるとすれば，計算資源の利用率を高め，ジョブ A の結果をより早く利用者に返すことができる．さらに，たとえば，時刻 $t_2(> t_1)$ にジョブ A よりも優先度の高いジョブ B が，200 ノードを要求して投入されたとし，時刻 t_2 には利用可能なノードが存在していなかったとする．この場合，仮に，ジョブ A の計算規模を 1000 ノードから 800 ノードに縮小することができるとすれば，ここで空いた 200 ノードにジョブ B をディスパッチすることが可能になる．このように，ジョブ（並列計算）の計算規模を動的に再構成することができれば，柔軟なジョブスケジューリングが可能になり，計算資源の利用率を高めることができる．そして，計算資源の利用率の向上は単位時間あたりの実行ジョブ数の向上を意味するため，ジョブの結果をできるかぎり早く利用者に返すという本来の目的に貢献できることになる．さらに，故障したノードの修理などにより，あるノードをシステムから切り離したい場合，そのノードのうえで実行されている並列計算が再構成できるならば，並列計算全体を中断させることなく，並列計算からそのノードを切り離すことができる．今後，並列計算が大規模化し，1 個の並列計算の実行に要する時間が長くなればなるほど，再構成によって並列計算の途中でも計算資源をスケジューリングできることの恩恵はますます大きくなる．

以上の議論は，スーパーコンピュータにかぎらず，多数の（しかし有限の）計算資源を多数の利用者で共有利用するようなコンピューティング形態一般に対して成り立つ．たとえば，近年着目されているクラウドコンピューティングやユーティリティコンピューティング [21, 124] は，プロバイダの管理する多数の計算資源が，従量制課金のサービスとして多数の利用者に対して提供されるコンピューティング形態である．よって，これらのサービスとして何らかの並列計算を提供しようとする場合には，再構成による柔軟なスケジューリングに対する要請が出てくる可能性がある．

当然，並列計算を再構成するためには，並列計算それ自体が動的に再構成できるように記述されていなければならない．単純なマスタワーカ型の並列計算であれば，ワーカ数を動的に調節できるようにすることで再構成に対応させることは難しくないが，本研究が対象とするような有限要素法や Web グラフ解析などの高性能並列科学技術計算を再構成可能なように記述するのは明らかに難しい．そこで，DMI では，再構成可能な高性能並列科学技術計算を簡単に記述できるようにするための仕組みやプログラミングモデルを設計して実装する．

1.3.2 再構成可能なグローバルアドレス空間のコヒーレンシプロトコル

DMI では，まず，グローバルアドレス空間モデルにおける再構成を可能にするために，**ノードが自由なタイミングで非同期的に参加/脱退できるようなグローバルアドレス空間のコヒーレンシプロトコ**

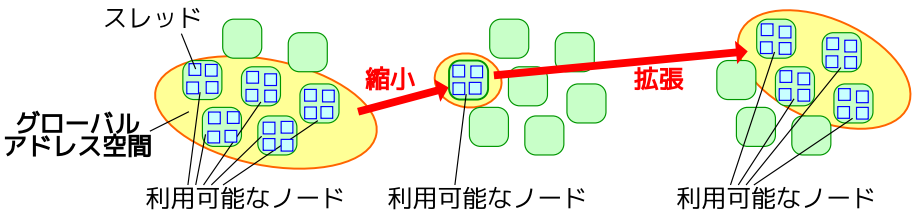

図 1.2　スレッド増減による並列計算の再構成.

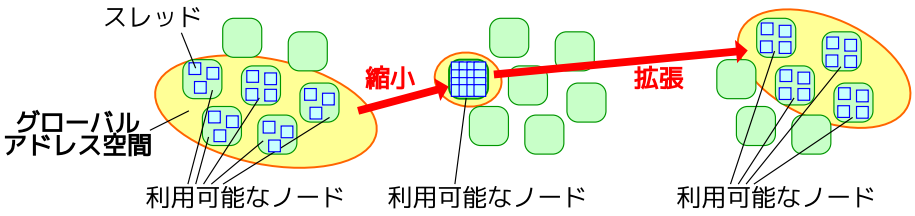

図 1.3　透過的なスレッド移動による並列計算の再構成.

ルを実現する．ここで，ノードの参加/脱退が非同期的であるとは，あるノードが参加/脱退する際に実行中のノードの同期を必要としないという意味である．いい換えると，実行中のノードたちが，それぞれ独立にグローバルアドレス空間に対して read/write などを行っていたとしても，それと並行してノードの参加/脱退を実現できるという意味である．著者の知るかぎり，ノードの非同期的な参加/脱退に対応可能なグローバルアドレス空間を実現した研究は DMI がはじめてである．

1.3.3　並列計算の再構成のためのプログラミングモデル

DMI では，次に，並列計算の再構成を簡単に記述するためのプログラミングモデルを 3 種類提案し，その性能とプログラマビリティを比較検討する．一般に，並列計算の再構成を実現する場合，大きく分類して 2 とおりの実現方法が考えられる．第 1 の方法は，図 1.2 に示すように，ノードの参加/脱退にともなってスレッドを生成/破棄し，つねに 1 プロセッサ上に 1 スレッドが割り当てられるように，スレッド数を増減させる方法である [57, 54, 128, 69, 173, 55, 126]．第 2 の方法は，1 プロセッサ上に複数スレッドが割り当てられてもよいことにし，プログラマが最初に十分な数のスレッドを生成しておけば，あとは処理系が，スレッド移動を通じて，それら大量のスレッドを各時点で利用可能なノードにマッピングする方法である [80, 81].

なお，スレッド移動を議論する場合，C 言語におけるスレッド，Java におけるスレッド，高級言語におけるユーザレベルスレッドなど，どのレベルのスレッドを対象にするかに応じて議論するべき問題は変わってくる．本研究では，一般に高性能並列科学技術計算は Fortran または C 言語で記述される場合が多いことをふまえて，C 言語におけるスレッドについて議論する．

1.3.3.1　スレッド増減に基づく方法

まず，第 1 の方法について考える．スレッド数の増減によって計算規模の拡張/縮小を実現することの利点は，つねに 1 プロセッサに 1 スレッドが割り当てられるため，無駄なオーバヘッドが生じず，実

行時の性能がよいことである．

一方で，欠点は，プログラマビリティの低さである．再構成にともなうスレッドの生成/破棄などの面倒な処理はプログラミングモデルのなかでうまく隠蔽できるとしても，「各スレッドの担当範囲の再計算」のためのコードと「データのチェックポイント/リストア」のためのコードは，どうしてもプログラマに明示的に記述してもらわざるをえない．たとえば，3 次元物体をスレッド数個の領域に分割するような，SPMD 型の並列反復計算を考える．1000 スレッドで実行を開始する場合，まず 3 次元物体を 1000 個に領域分割したあと，各スレッド i に領域 i を担当させて反復計算を実行する．このとき，各スレッド i（$0 \leq i < 1000$）は，領域 i 内の各点の現在の値をメモリ上に保持しながら反復計算を進める．しばらくして再構成が発生し，スレッド数が 200 スレッドに減ったとする．すると，少なくとも，再構成後には，3 次元物体が 200 個に領域分割されていて，各スレッド i（$0 \leq i < 200$）が領域 i 内の各点の現在の値をメモリ上に保持しながら反復計算を進めているような状態になっていなければならない．そして，これを実現するためには，再構成時に，領域を再分割するという処理（＝「各スレッドの担当範囲の再計算」）と，再構成前に領域 i（$0 \leq i < 1000$）内の各点の現在の値をチェックポイントし，再構成後に領域 i（$0 \leq i < 200$）内の各点の現在の値をリストアする処理（＝「データのチェックポイント/リストア」）が必要である．これらの処理の具体的内容はアプリケーションに依存するものであり，DMI のような処理系が自動的に面倒を見ることができる処理ではないため，プログラマに明示的に記述してもらわざるをえない．また，**10.6** 節で評価するように，複雑なアプリケーションでは，どのデータをチェックポイントしてリストアするべきかがそもそも自明でないという問題もある．

要約すると，スレッド数の増減によって計算規模の拡張/縮小を実現する方法は，**実行時の性能はよいがプログラマビリティは低い**．本稿では，この方法に基づくプログラミングモデルを **rescale** と呼ぶ．

1.3.3.2 透過的なスレッド移動に基づく方法

次に，第 2 の方法について考える．この方法では，プログラマは，並列計算の再構成を意識することなく単に十分な数のスレッドを生成するだけでよい．すると処理系が，透過的なスレッド移動を通じて並列計算を再構成してくれる．たとえば，プログラマが 10000 スレッドを生成した場合，処理系は，1000 プロセッサが利用可能なときには各プロセッサに 10 スレッドを割り当て，やがて 100 プロセッサしか利用できなくなれば，スレッド移動を通じて，各プロセッサに 100 スレッドが割り当てられるようにする．この方法の利点は，その透過性により，並列計算の再構成のためにプログラムの修正がほとんど必要なく，プログラマビリティが高いことである．

一方で，欠点は，性能の低さである．どのようなプロセッサ数で実行されたとしてもプロセッサ間の負荷バランスがとれるようにするためには，十分に多くのスレッドを生成しておく必要がある．しかし，これは著しい性能低下を引き起こす [126]．その第 1 の理由としては，並列度を過剰に増やすことによって，並列化のためのオーバヘッドが増えてしまうためである．たとえば，領域分割型の並列科学技術計算では，領域分割数を増やすほど袖領域の総面積が増え，通信量・計算量ともに増えてしまう．第 2 の理由は，1 プロセッサに複数のスレッドを割り当てることに起因する性能劣化である．図 1.4 には，NAS Parallel Benchmark[51] の EP（クラス C），CG（クラス C），MG（クラス C），FT（クラス C），IS（クラス C）に関して，1 プロセッサあたり複数の MPI プロセスを生成した場合の実行時間

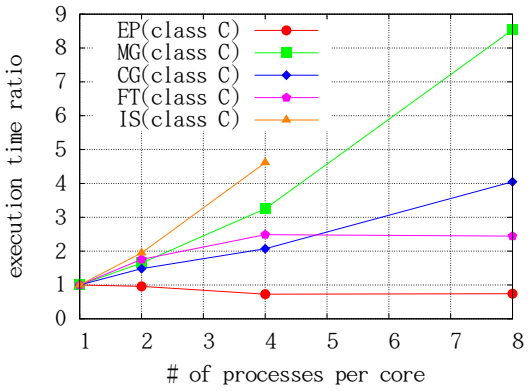

図 1.4　1 プロセッサ上で n 個のプロセスを実行した場合の性能劣化（$n = 8$ の場合の IS の実行時間はグラフ外にあり 167.7 である）.

を示す．なお，実験環境は，Intel Xeon E5530 2.40GHz （4 プロセッサ，ハイパースレッディングにより論理的に 8 プロセッサ）×2 の CPU から構成されるノードを 16 個接続した合計 128 プロセッサのクラスタ環境である．たとえば，横軸が 8 の点は，1 プロセッサあたり 8 個の MPI プロセスが生成されており，16 個のノード全体としては $128 \times 8 = 1024$ 個の MPI プロセスが生成されていることを意味する．また，縦軸の実行時間は，各アプリケーションごとに，1 プロセッサあたり 1 個の MPI プロセスを割り当てる場合の実行時間が 1 になるように正規化している．図 1.4 より，1 プロセッサあたり 8 個の MPI プロセスを割り当てる場合には，1 プロセッサあたり 1 個の MPI プロセスを割り当てる場合と比較して，CG では 8.5 倍，IS では 167.7 倍も遅いことがわかる．

この性能劣化は，次のような理由により，並列計算の再構成を考えるうえで深刻なものである．一般に，再構成が有用となるような並列計算は長時間を要する並列計算であり，そのような並列計算は反復計算の形態をとる場合が多い．第 **10** 章で評価する有限要素法や Web グラフのページランク計算なども並列反復計算の形態をとる．そして，一般に並列反復計算では同期が多く含まれ，同期のたびに，実行速度がもっとも遅いスレッドによって律速されることになるため，あるスレッドの性能低下がアプリケーション全体の性能低下に大きく影響してしまう．たとえば，有限要素法で用いる図 6.37 の BiCGSafe 法では，1 イテレーションあたり 22 回の同期が含まれる．このように，並列計算の再構成を考えるうえでは，1 プロセッサあたり複数のスレッドを割り当てることによる性能劣化はとくに重要である．

要約すると，透過的なスレッド移動によって計算規模の拡張/縮小を実現する方法は，プログラマビリティは高いが性能が低い．ただし，慎重に観察すると，プログラマビリティの高さについても自明ではないことがわかる．正確な事情は **2.2.4** 節および **2.2.5** 節で述べるが，スレッドを粒度として移動を行う場合，各スレッドがアクセスできるメモリ領域に関してプログラミング制約を課す必要が出てくる．たとえば，プロセス p のなかのスレッド t がプロセス p のグローバル変数 g を利用しているとすると，スレッド t を別のプロセス q に移動させるときに，グローバル変数 g の値を移動させても移動させ

なくても，スレッドの安全な実行を継続できなくなる．なぜなら，グローバル変数 g を移動させなければ，g を使用しているスレッド t にとって不都合が生じるし，グローバル変数 g を移動させれば，プロセス q にすでに存在しているグローバル変数 g の値を書きつぶしてしまうことになるため，もとからプロセス q に存在して g を使用していたスレッドにとって不都合が生じる．したがって，少なくともグローバル変数の使用は禁じる必要があり，よってグローバル変数を内部で使う可能性のあるすべてのライブラリ関数の使用も禁じる必要が出てくる．

以上の議論をまとめると，透過的なスレッド移動によって計算規模の拡張/縮小を実現する方法は，**スレッドの増減が必要ないという点ではプログラマビリティは高いが，いくつかのプログラミング制約が加わるという点ではプログラマビリティは低い**．また，1 プロセッサあたり複数のスレッドが割り当てられる可能性があるため，**性能は低い**．本稿では，この方法に基づくプログラミングモデルを **thread-move** と呼ぶ．

さらに，本研究では，thread-move においてプログラミング制約のせいでプログラマビリティが損なわれてしまう点を問題視し，このプログラミング制約を撤廃することを試みる．そもそもこのプログラミング制約は，スレッドを粒度として自由な移動を行うためには各スレッドが使用するアドレス空間が独立している必要があるにもかかわらず，スレッドどうしはアドレス空間を共有しているという矛盾に起因するものである．そう考えると，各スレッドをスレッドではなくプロセスとして実装すればアドレス空間を独立させることができるため，プログラミング制約を撤廃できるように思われる．しかし，一般に，プロセス間のデータ共有はスレッド間のデータ共有よりもオーバヘッドの大きい方法を必要とするため，プロセスとして実装した場合，ノード内のデータ共有のオーバヘッドが大きくなりさらなる性能低下を招いてしまうおそれがある．要するに，スレッドとして実装してもプロセスとして実装しても問題が起きる．そこで，本研究では，これらの要件を整理したうえで，**スレッドとプロセスの「中間」の機能を持つ新たなカーネルプリミティブ**として **half-process** を提案する．half-process を利用することで，スレッド間のデータ共有と同様のデータ共有が可能になると同時に，thread-move におけるプログラミング制約も撤廃することができ，結果的に，**真に透過的なスレッド移動**を実現できるようになる．なお，「真に透過的である」とは，「スレッド移動を実現するための余計なプログラミング制約が存在しない」という意味である．本稿では，この方法に基づくプログラミングモデルを **half-process-move** と呼ぶ[⋆1]．

以上で提案した 3 種類のプログラミングモデルの特徴をまとめると以下のようになる：

rescale　スレッドの増減による方法．つねに 1 プロセッサあたり 1 スレッドが割り当てられるため性能はよいが，プログラマはデータのチェックポイント/リストアを記述する必要があるため，プログラマビリティは低い．

thread-move　透過的なスレッド移動による方法．1 プロセッサに複数スレッドが割り当てられることによる性能低下が起きる．プログラマが再構成を意識しなくてもよいという点ではプログ

[⋆1] half-process-move では，スレッドではなく half-process を使っているため，「真に透過的なスレッド移動」と表現するよりも「half-process 移動」と表現する方が適切である．しかし，プログラマの視点から見ればスレッド移動の一種に見えるため，実装を問題にする場合をのぞいては，「真に透過的なスレッド移動」と表現することにする．

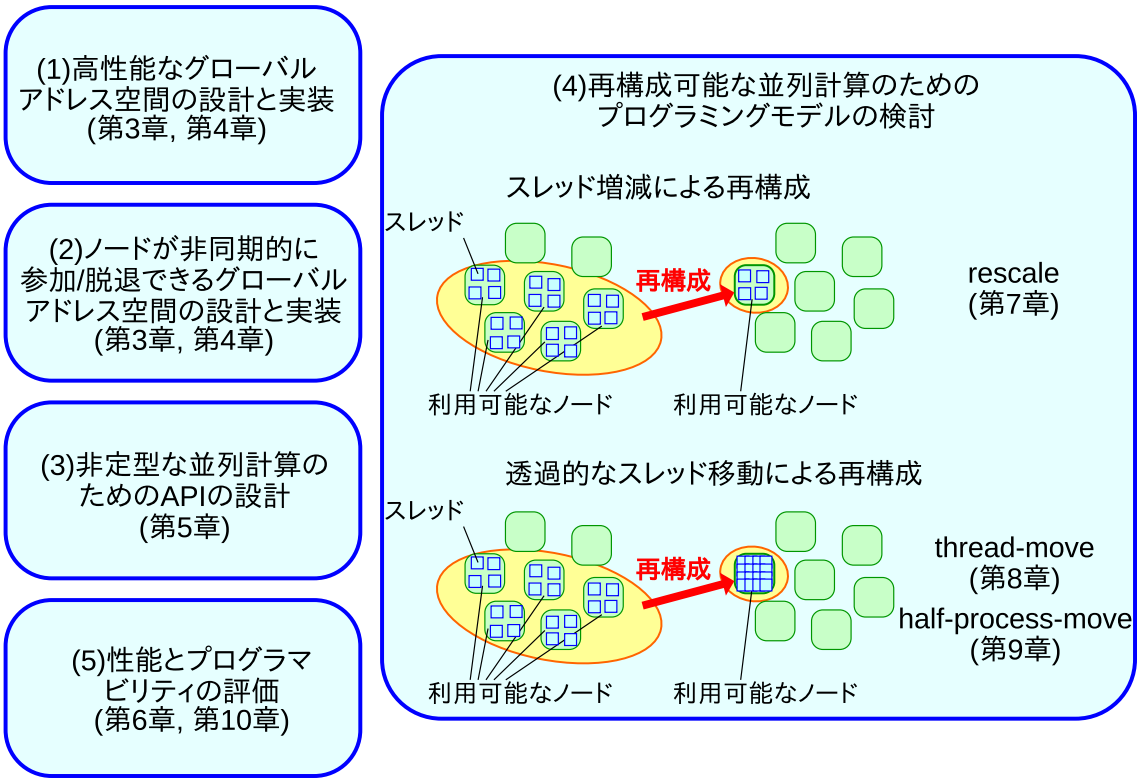

図 1.5　本研究の全体像.

ラマビリティは高いが，いくつかのプログラミング制約が存在する．

half-process-move　真に透過的なスレッド移動による方法．1 プロセッサに複数スレッドが割り当てられることによる性能低下が起きるが，プログラマビリティは高い．ただし，カーネルの改造を必要とする．

本研究では，ここで述べた rescale，thread-move，half-process-move の 3 種類のプログラミングモデルを設計して実装し，有限要素法による応力解析や大規模な Web グラフ解析などの実用的な並列計算を題材にして，プログラマビリティ（＝再構成を行わない通常のプログラムに対して何行の変更を加える必要があるか）と性能（＝ 1 プロセッサ上に複数スレッドを割り当てることがどの程度の性能劣化につながるか）について比較検討する．

1.4　本研究の全体像

本研究の全体像は以下のとおりである（図 1.5）：

(1) 高性能なグローバルアドレス空間を設計して実装する．グローバルビュー型のグローバルアドレス空間を提供しつつも，内部的にどのような通信が起きるのかがユーザプログラムから把握しや

すく，通信を簡単に制御できる API を提供することで，強力で見通しのよい性能最適化を可能にする．

(2) 再構成可能なグローバルアドレス空間を設計して実装する．再構成にともなうアクセスローカリティの変化に対してデータ分散を簡単に適応させるための API を提供する．また，ノードが自由なタイミングで非同期的に参加/脱退できるようなグローバルアドレス空間のコヒーレンシプロトコルを実装する．

(3) 非定型な並列計算を性能を落とすことなく簡単に記述できる API として，read-write-set を設計して実装する．

(4) 再構成可能な並列計算を簡単に記述できるようにするために，rescale，thread-move，half-process-move の 3 種類のプログラミングモデルを設計して実装する．

(5) 有限要素法による応力解析や大規模な Web グラフのページランク計算など，実用的で非定型なアプリケーションを題材として，DMI の基本的な性能とプログラマビリティを評価する．また，再構成可能な並列計算のための 3 種類のプログラミングモデルについて，性能とプログラマビリティを比較検討する．

既存研究と比較して，上記の 5 つの目標すべてが DMI にとって新規的なものである．とくに，著者の知るかぎり，ノードが自由なタイミングで非同期的に参加/脱退できるようなグローバルアドレス空間のコヒーレンシプロトコルを実現したのは DMI がはじめてである．したがって，グローバルアドレス空間モデルに基づいて再構成可能な並列計算を実現し，再構成のための複数のプログラミングモデルの性能とプログラマビリティを緻密に比較検討しているのも DMI がはじめてである．

1.5 本稿の構成

1.5.1 本稿の構成

本稿の構成は以下のとおりである：

第 2 章 関連研究を述べる．既存のさまざまな並列分散プログラミングモデルと処理系の得失を議論し，なぜ，DMI が並列分散プログラミングモデルとしてグローバルビュー型のグローバルアドレス空間モデルを選択したのかを明確化させる．また，並列計算の再構成に応用可能な並列分散技術を幅広い視野からとり上げ，そのいずれもが非定型な高性能並列科学技術計算の再構成を実現するには適していないことを指摘する．

第 3 章 再構成可能かつ高性能なグローバルアドレス空間の設計について述べる．とくに，強力で見通しのよい性能最適化のための API を提案する．

第 4 章 グローバルアドレス空間の実装について述べる．とくに，プロセスが非同期的に参加/脱退できるようなグローバルアドレス空間のコヒーレンシプロトコルを提案する．また，このような複雑なコヒーレンシプロトコルを見通しよく実装するための手法について述べる．

第 5 章 非定型な並列計算を性能を落とすことなく簡単に記述できる API として，read-write-set を設計して実装する．

第 6 章 主に再構成をともなわない場合の，DMI におけるグローバルアドレス空間の基本的な性能とプログラマビリティを評価する．各種マイクロベンチマーク，基本的なアプリケーションのほかに，応用的なアプリケーションとして，有限要素法による応力解析，大規模な Web グラフのページランク計算と最短路計算をとり上げる．

第 7 章 rescale のプログラミングモデルを設計して実装する．

第 8 章 thread-move のプログラミングモデルを設計して実装する．透過的なスレッド移動におけるプログラミング制約を厳密に述べる．また，既存研究におけるスレッド移動の手法では，計算規模が CPU のアドレス空間全体のサイズに制限されてしまう可能性を指摘し，計算規模がアドレス空間全体のサイズに制限されない新たなスレッド移動の手法を提案する．

第 9 章 half-process-move のプログラミングモデルを設計して実装する．新しいカーネルプリミティブとして，部分的にアドレス空間を共有するプロセスを実現する half-process を提案する．half-process を用いて真に透過的なスレッド移動を実現すると同時に，half-process のより広い応用可能性についても述べる．

第 10 章 rescale，thread-move，half-process-move の 3 種類のプログラミングモデルに関して，再構成可能な並列計算に対する性能とプログラマビリティを比較して評価する．

第 11 章 本稿をまとめ，今後の課題について述べる．

1.5.2 本稿における表記

本稿においては，プロセッサとは CPU の 1 個のコアのことを意味する．文脈上，1 プロセッサに 1 スレッド/1 プロセス/1 half-process を割り当てることが明らかな場合には，「スレッド」/「プロセス」/「half-process」と「プロセッサ」を同一の意味で使用する．また，1 ノード上に 1 プロセスを割り当てることが明らかな場合には，「ノード」と「プロセス」を同一の意味で使用する．また，「スレッドまたはプロセスまたは half-process」のことを「インスタンス」と表記することがある．

第 2 章

関連研究

本章では，第 1 に，さまざまな並列分散プログラミングモデルと処理系をとり上げ，DMI が並列分散プログラミングモデルとしてグローバルビュー型のグローバルアドレス空間モデルを選択した理由を明確にする．第 2 に，並列計算の再構成に応用可能な並列分散技術を幅広い視野からとり上げ，そのいずれもが非定型な高性能並列科学技術計算の再構成を実現するには適していないことを指摘する．

2.1 並列分散プログラミングモデル

2.1.1 並列分散プログラミングモデルの比較指標

並列分散プログラミングモデルを設計するうえでは，性能とプログラマビリティのバランスをどのようにとるかが重要である．**1.2.2** 節で述べたように，DMI では，プログラミングを開始してから多数回の実行を繰り返して結果を得るまでの「全体」の時間を最短化するためには，(1) 強力な性能最適化が可能であることと，(2) 性能がよく実行時間が短いことが第一義的に重要であるという立場をとっている．したがって，DMI の設計方針は，**見通しのよい強力な性能最適化によってメッセージパッシングモデルと同等の性能を引き出せるという条件下で，できるかぎりプログラマビリティを高める**ことである．

本節では，並列分散プログラミングモデルとしてもっとも代表的な，メッセージパッシングモデル，ローカルビュー型のグローバルアドレス空間モデル，グローバルビュー型のグローバルアドレス空間モデルの 3 種類をとり上げ，以下の観点を中心にして特徴を比較する：

- 性能のよさ
- 性能最適化の自由度と見通しのよさ
- 非定型な並列計算に対するプログラマビリティ
- 再構成可能な並列計算に対するプログラマビリティ

これらの比較をふまえて，**2.1.4** 節で，DMI がグローバルビュー型のグローバルアドレス空間モデルを選択する理由を明らかにする．

ここで，非定型な並列計算に対するプログラマビリティを議論するための題材として，非定型なグラ

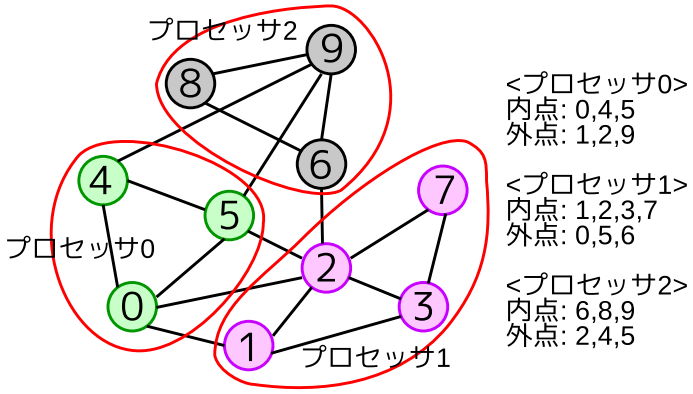

図 2.1　非定型なグラフ分割の例.

フ計算を考える．図 2.1 のように，10 個の節点と 17 個の無向エッジからなるグラフを考え，グラフ全体を 3 個のプロセッサで領域分割するとする．一般に，各プロセッサの視点で見たとき，そのプロセッサの担当領域に含まれている節点のことを内点と呼び，内点とエッジで直接結ばれている節点のうち内点ではない節点のことを外点と呼ぶ．たとえば，プロセッサ 0 の内点は節点 0，節点 4，節点 5 であり，プロセッサ 0 の外点は節点 1，節点 2，節点 9 である．

さて，グラフ計算として定式化できる計算には，有限要素法，ページランク計算，最短路計算などさまざまなものがあるが，いずれのグラフ計算でも，各節点には値が関連づけられており，「各節点 i の持つ値を，その節点 i とエッジで接続されている節点たちの値に基づいて更新する」ことによって計算が進むという点は共通している．たとえば，節点 5 の値を更新するためには，節点 0，節点 2，節点 4，節点 9 の値が必要になる．したがって，領域分割に基づく並列グラフ計算ではおおよそ以下の処理が繰り返されることになる：

(1) 各プロセッサが，外点の値を取得する．
(2) 各プロセッサが，内点の値と（1）で取得した外点の値に基づいて，内点の値を更新する．
(3)（多くの場合）すべてのプロセッサが同期する．

以上をふまえて，次節以降では，それぞれのプログラミングモデルで上記の（1）と（2）の処理がどのように記述できるかを観察することで，非定型な並列計算に対するプログラマビリティについて議論する．

2.1.2　メッセージパッシングモデル

2.1.2.1　概要

メッセージパッシングモデルでは，系内の各プロセッサに対して一意なランク（名前）が与えられ，ユーザプログラムに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を採用する代表的な処理系には MPI[63, 25, 181, 164, 127, 169, 80, 81, 57, 102, 195, 162, 62] があり，高性能な並列プログラミングにおけるデファクトスタンダードになって

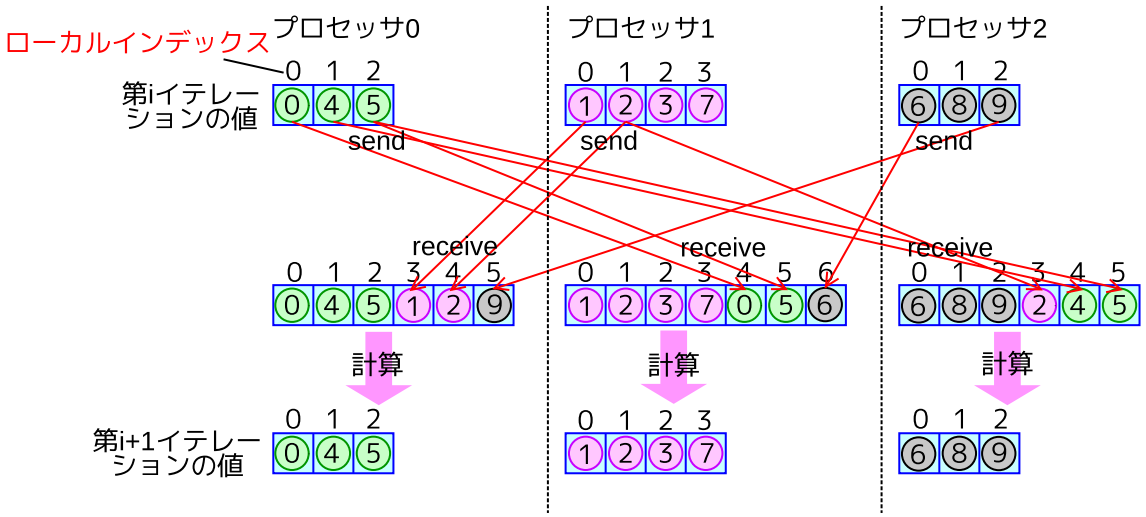

図 2.2　メッセージパッシングモデルで記述した非定型なグラフ計算.

いる.

2.1.2.2　性能

メッセージパッシングモデルの第 1 の利点は，**性能のよさ**である．point-to-point な send/receive に関しては，ユーザプログラムに記述された send/receive 操作がハードウェアで実際に発生する send/receive 操作にそのまま対応しているため，ユーザプログラムにとって本質的に必要とされている通信以外は発生せず，通信に無駄が生じない．また，MPI では，Gather や Broadcast などの集合通信がサポートされており，MPI の処理系によって最適化されたトポロジ上での効率的な集合通信を利用できる [195, 162]．第 2 の利点は，**性能最適化のわかりやすさ**である．point-to-point な send/receive に関しては，ユーザプログラムに記述された操作がそのまま内部的に起きるため，何を記述したときに内部的にどのような通信が起きるのかがプログラマにとって理解しやすい．とくに，意図しない大量の通信が内部的に引き起こされて，著しい性能劣化を招いてしまうようなことはほとんどない．また，データをどのように配置するか，どのプロセッサとどのプロセッサがいつ通信するかなどをユーザプログラム側で明示的に指示できるため，性能最適化の見通しがよい.

2.1.2.3　非定型な並列計算に対するプログラマビリティ

メッセージパッシングモデルの第 1 の欠点は，データの配置および通信をすべて明示しなければならないことに起因する**プログラマビリティの低さ**である．これは，とくに非定型な並列計算を記述しようとする場合に顕著となる．たとえば，図 2.1 のグラフ計算をメッセージパッシングモデルで記述する場合，図 2.2 のように，各プロセッサがローカルアドレス空間を持ち，ローカルアドレス空間のデータを送受信することによってグラフ計算を記述することになる．ここで，各ローカルアドレス空間のインデックスのことをローカルインデックス，節点番号のことをグローバルインデックスと呼ぶことにする．メッセージパッシングモデルのプログラマビリティの低さは，アルゴリズム自体はグローバルイン

デックスに基づいて設計しているにもかかわらず，ユーザプログラムはローカルインデックスに基づいて記述しなければならないという点に起因している．たとえば，「プロセッサ 0 が節点 9 のデータを取得する」というアルゴリズムを実現するためには，「プロセッサ 2 がローカルインデックス 2 のデータをプロセッサ 0 に対して送信し，それを受信したプロセッサ 0 はそのデータをローカルインデックス 5 のデータとして格納する」ということを記述する必要がある．さらに，プロセッサ 0 が外点の値をすべて取得したあとで担当領域の内点の値を計算する場合にも，グローバルインデックスに基づいて計算することができない．たとえば，プロセッサ 0 が節点 4 のデータを更新するためには，節点 0，節点 5，節点 9 のデータが必要になるが，プロセッサ 0 のローカルアドレス空間では，節点 0，節点 5，節点 9 のデータが，それぞれローカルインデックス 0，5，9 のデータとして格納されているわけではなく，実際にはローカルインデックス 0，2，5 のデータとして格納されている．よって，「節点 4 のデータを，節点 0，節点 5，節点 9 のデータに基づいて更新する」というアルゴリズムを実現するためには，「ローカルインデックス 1 のデータを，ローカルインデックス 0，2，5 のデータに基づいて更新する」ということを記述しなければならない．明らかに，このようなローカルインデックスによるプログラミングは，非常に煩雑でバグを引き起こしやすい．要約すると，非定型な並列計算に対するメッセージパッシングモデルにおけるプログラマビリティ上の問題点は，以下の 2 点に集約できる：

問題点 I　グローバルアドレス空間上のデータをローカルアドレス空間に取得する場合に，グローバルインデックスによるアクセスができないこと

問題点 II　ローカルアドレス空間を使って計算を行うため，計算時にもグローバルインデックスによって計算を進められないこと

2.1.2.4　再構成可能な並列計算に対するプログラマビリティ

第 2 の欠点として，メッセージパッシングモデルは**再構成可能な並列計算に適していない**．なぜなら，メッセージパッシングモデルでは，「どのプロセッサにデータを send するのか」と「どのプロセッサからデータを receive するのか」をプログラマが明示的に記述する必要があるため，プログラマはデータの所在（＝「どのプロセッサがどのデータを持っているのか」）を明示的に管理しなければならないが，再構成が起きる状況ではデータの所在がつねに変化しうるからである．たとえば，再構成にともなってプロセッサが参加する場合には，新しく参加してきたプロセッサに計算負荷を分け与えるために，データの移動が必要となる．また，再構成にともなってプロセッサが脱退する場合には，そのプロセッサが所有しているデータが失われないよう，脱退前にどこか他のプロセッサに対してデータを追い出す必要もある．このように，再構成にともなってデータの所在が複雑に変化する状況で，プログラマがその所在の変化を明示的に管理しつつデータの送受信を記述するのは非常に困難である．この困難さは，メッセージパッシングモデルにおいては，**データの所在を管理するのが処理系ではなくプログラマである**という点に起因している．

2.1.2.5　その他の特徴

第 3 の欠点は，メッセージパッシングモデルは双方向通信型のプログラミングモデルであるため，バグを招きやすい点である．双方向通信型のプログラミングモデルにおいては，あるプロセッサ p が別の

プロセッサ q の持つデータ x を取得するためには，プロセッサ p 側で receive を呼び出すだけでは機能せず，プロセッサ q 側でそれに対応する send を呼び出さなければならない．よって，send と receive の対応を誤って意図しないデータを送受信してしまうことに起因するバグや，send と receive の依存関係を誤ることに起因するデッドロックが非常に起きやすい．また，そもそもアルゴリズム上興味があるのは「プロセッサ p が」プロセッサ q のデータ x を取得するということだけであって，「プロセッサ q が」何を行うかには興味がないにもかかわらず，プロセッサ q がデータ x を send するという操作を記述しなければならないという点も，プログラマビリティ上の欠点である．

第 4 の欠点として，双方向通信型のプログラミングモデルでは**非同期的なアルゴリズムを記述しにくい**．ここで，非同期的なアルゴリズムとは，プロセッサ p がプロセッサ q の持つデータ x を取得するときに，プロセッサ p がプロセッサ q と同期を行うことなくプロセッサ q のデータ x を取得するようなアルゴリズムのことをいう．たとえば，図 2.1 のグラフにおいて，プロセッサ 2 が節点 9 の値を更新したことを保証することなく，プロセッサ 0 が節点 9 の値を取得するようなアルゴリズムである．このような非同期的なアルゴリズムは，一般に，「取得できる値が決定的であることには興味がなく，とにかくその時点で最新の値を取得できればよい」ような場面で用いられる．たとえば，プロセッサ p がデータ x を持っていて，各時点で見つかっているデータ x の最良値を基準にして枝刈りを行う並列探索では，各プロセッサはプロセッサ p からデータ x の最新の値を非同期的に取得するのが理想的である．また，**6.6.4** 節では，非同期的なアルゴリズムによってグラフの最短路計算を行うアルゴリズムを述べる．このような非同期的なアルゴリズムでは，データを供給する側の協力（同期）を得ることなくデータを取得できるような単方向通信 [142, 145] が求められており，通信に同期が必要となる双方向通信で記述することはできない．なお，MPI は本来双方向通信型のメッセージパッシングモデルに基づく処理系であるが，MPI-2[63] からは MPI_Get() 関数，MPI_Put() 関数などの単方向通信のための API もサポートされている．

2.1.3 ローカルビュー型のグローバルアドレス空間モデル

2.1.3.1 概要

グローバルアドレス空間モデルでは，物理的には分散したプロセッサ（分散メモリ環境）上に仮想的なグローバルアドレス空間が提供されており，グローバルアドレス空間上のアドレスに対する read/write 操作（あるいは get/put 操作[*1] ）によって，別のプロセッサ上に存在するデータにアクセスすることができる [22, 37, 41, 72, 144, 146, 143, 149, 45, 188, 77]．とくに，性能上の理由から，データがローカルに存在するかリモートに存在するかをプログラマが区別して read/write できるような設計になっているグローバルアドレス空間モデルは，PGAS（Partitioned Global Address Space）モデルと呼ばれることが多い．グローバルアドレス空間モデルは，**ローカルビュー型のグローバルアドレス空間モデル**と**グローバルビュー型のグローバルアドレス空間モデル**に大きく分類することができる．両者の違いは，グローバルアドレス空間上のデータにアクセスするときに，そのデータを保持するプロセッサを明

[*1] 本稿では，データを一般的に読む/書く操作のことを read/write と表記する．read/write のなかでも，とくに，PGAS モデルにおいて，（ローカルではなく）リモートに存在するデータを取得する/ リモートに存在するデータを更新するという意味で read/write することを強調するとき，それを get/put と表記する．

示的に指示する必要があるかないかである．たとえば，ローカルビュー型のグローバルアドレス空間モデルでは，各プロセッサにデータ x が存在することができ，「『プロセッサ p の』データ x を read/write する」と記述するのに対して，グローバルビュー型のグローバルアドレス空間モデルでは，データ x はグローバルに 1 個しか存在できず，単に「データ x を read/write する」と記述する．別の例として，グローバルアドレス空間にサイズ 100 の配列 a を作って 12 番目の要素にアクセスしたい場合，グローバルビュー型のグローバルアドレス空間モデルでは，グローバルな配列 a を宣言したあと「a[12]」と記述できるのに対して，ローカルビュー型のグローバルアドレス空間モデルでは，たとえば 10 個の各プロセッサ p_0，p_1，...，p_9 にサイズ 10 の配列 a を宣言したうえで，「プロセッサ p_1 の a[2]」と記述する必要がある．

ローカルビュー型のグローバルアドレス空間モデルは，メッセージパッシングモデルにおける send/receive による双方向通信を，read/write による単方向通信に置き換えただけにすぎない．よって，メッセージパッシングモデルの特徴を多く引き継いでいる．ローカルビュー型のグローバルアドレス空間モデルに基づく処理系としては，Co-Array Fortran[149, 45, 185, 46]，Titanium[188, 171, 77, 50, 172]，XcalableMP[72] がある．

2.1.3.2 性能

メッセージパッシングモデルと同様に，ローカルビュー型のグローバルアドレス空間モデルの利点は，**性能のよさ**と**性能最適化のわかりやすさ**である．ユーザプログラムに記述された操作（プロセッサを指定した read/write）が内部的に発生する通信（send/receive あるいは RDMA （Remote Direct Memory Access）[177, 113]）にそのまま対応するため，通信に無駄が生じない．また，内部的にどのような通信が起きるのかがプログラマにとって把握しやすく，性能最適化の見通しがよい．さらに，Infiniband や Myrinet で採用されている単方向通信の RDMA を利用して read/write を実装することで，相手側のプロセッサを邪魔することなく通信を実現することができ，メッセージパッシングモデルにおける send/receive よりも性能が出ることもある．XcalableMP は，ローカルビュー型とグローバルビュー型の両方をサポートしているが，性能を追求する場合にはローカルビュー型で記述することが推奨されている [72].

2.1.3.3 非定型な並列計算に対するプログラマビリティ

ローカルビュー型のグローバルアドレス空間モデルはメッセージパッシングモデルにおける send/receive が read/write に変わっただけであるため，**プログラマビリティは低い**．図 2.1 のグラフ計算をローカルビュー型のグローバルアドレス空間モデルで記述すると図 2.3 のようになる．このように，ローカルインデックスによってプログラムを記述する必要があり，ローカルビュー型のグローバルアドレス空間モデルでは，前述の問題点 I も問題点 II も解決されていない．

2.1.3.4 再構成可能な並列計算に対するプログラマビリティ

メッセージパッシングモデルが再構成可能な並列計算に適さない理由は，データの所在を管理するのが処理系ではなくプログラマであるという点に起因していた．そして，ローカルビュー型のグローバルアドレス空間モデルにおいても，「どのプロセッサからデータを read するのか」と「どのプロセッサにデータを write するのか」をプログラマが明示的に記述する必要があるため，プログラマはデータの所

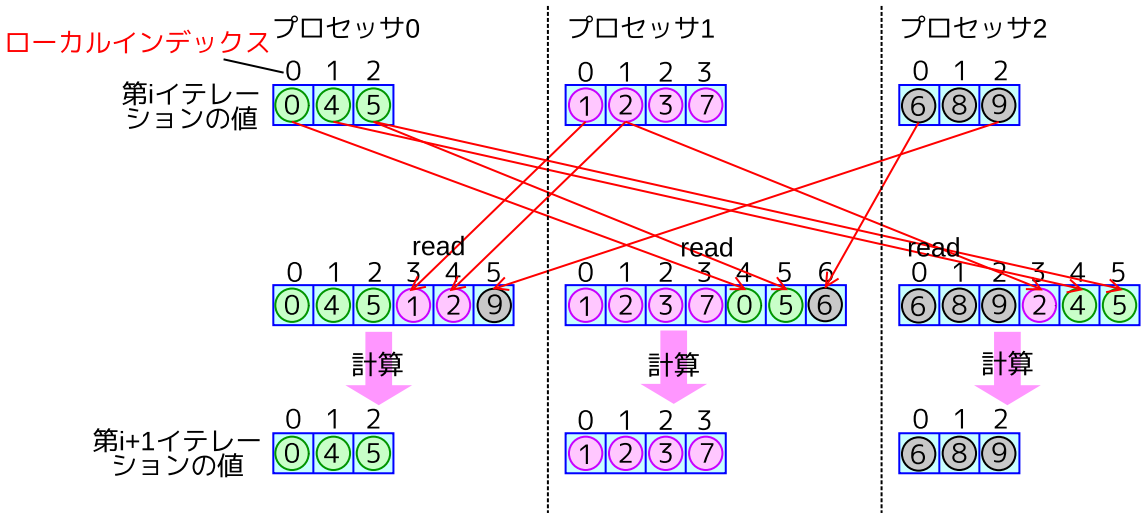

図 2.3　ローカルビュー型のグローバルアドレス空間モデルで記述した非定型なグラフ計算.

在（＝「どのプロセッサがどのデータを持っているのか」）を明示的に管理しなければならない．よって，ローカルビュー型のグローバルアドレス空間モデルも**並列計算の再構成には適していない**.

2.1.3.5　その他の特徴

一方で，ローカルビュー型のグローバルアドレス空間モデルは単方向通信に基づくプログラミングモデルであるため，メッセージパッシングにおける send/receive の対応の誤りに起因するバグやデッドロックは発生しない．また，**非同期的なアルゴリズムも自然に記述することができる**.

2.1.4　グローバルビュー型のグローバルアドレス空間モデル

2.1.4.1　概要

グローバルビュー型のグローバルアドレス空間モデルでは，データの所在は処理系によって管理されているため，read/write にあたってデータを所持するプロセッサを指示する必要はなく，共有メモリ環境と同様の read/write によってグローバルアドレス空間上のデータにアクセスすることができる．なかでも，OS のメモリ保護機構を利用することで，通常のメモリアクセスと同様のシンタックスによって透過的にグローバルアドレス空間にアクセスできるようにしたモデルは分散共有メモリモデルと呼ばれることが多い[*2]．グローバルビュー型のグローバルアドレス空間モデルに基づく処理系としては，UPC[43, 62, 48, 46]，Global Arrays[144, 146, 143]，Chapel[37, 26]，X10[41]，XcalableMP[72] などの PGAS 処理系，IVY[111]，Munin[101]，TreadMarks[15]，Midway[139]，Object-DSM[120]，DSM-Threads[135, 136, 160]，SMS[204] などの分散共有メモリ処理系がある．グローバルビュー型の

[*2] なお，PGAS モデルと分散共有メモリモデルの定義の違いは学術的に定まっているわけではなく，しばしば同じ意味で用いられる．本稿では，グローバルアドレス空間モデルのうち，リモートとローカルの違いをプログラマに見せようとしているモデルを PGAS モデル，OS のメモリ保護機構を利用することでグローバルアドレス空間に対する透過的なアクセスを実現しようとしているモデルを分散共有メモリモデルと呼ぶ．よって，ローカルビュー型のグローバルアドレス空間モデルはつねに PGAS モデルということになる．

```
sum = 0;
for (i = 0; i < 100000; i++) {
    sum += a[i];
}
```

図 2.4 「グローバルアドレス空間を自由にアクセスする方法」で記述したコード.

```
sum = 0;
memget(buf, a, 100000);
for (i = 0; i < 100000; i++) {
    sum += buf[i];
}
```

図 2.5 「グローバルアドレス空間を極力アクセスしない方法」で記述したコード.

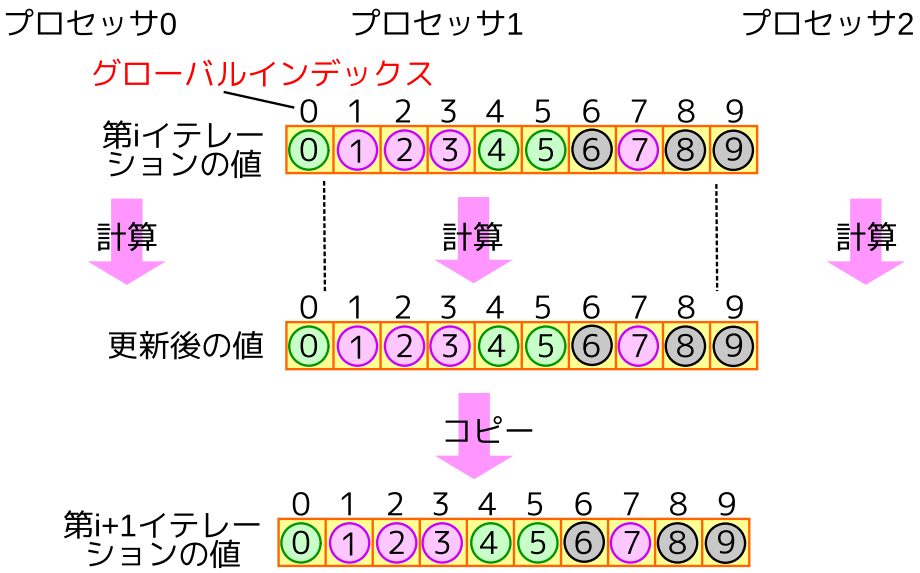

図 2.6 グローバルビュー型のグローバルアドレス空間モデルで記述した非定型なグラフ計算（「グローバルアドレス空間を自由にアクセスする方法」の場合）.

グローバルアドレス空間モデルの利点は，**プログラマビリティの高さ**である．たとえば，分散共有メモリ処理系では，ひとまず性能を度外視するならば，共有メモリ環境上でのマルチスレッドプログラムとほぼ同一のプログラムを分散プログラムとして実行させることができる．また，Java をシンタックス上のベースとしている X10 では，クラスやオブジェクトを利用して分散プログラムを記述することができる．

2.1.4.2 非定型な並列計算に対するプログラマビリティ

一般に，グローバルビュー型のグローバルアドレス空間モデルでプログラムを記述する場合には，性能上の理由から，2 種類の記述方法が存在する．第 1 の記述方法は，グローバルアドレス空間を共有メモリだと見なし，グローバルアドレス空間を自由自在に read/write するように記述する方法である．たとえば，UPC において，グローバルアドレス空間上の配列 a の 100000 要素の合計を計算するコードは図 2.4 のように記述できる．本稿では，この記述方法を「グローバルアドレス空間を自由にアクセスする方法」と呼ぶ．この記述方法では，図 2.1 のグラフ計算は，図 2.6 に示すようにきわめて簡単に

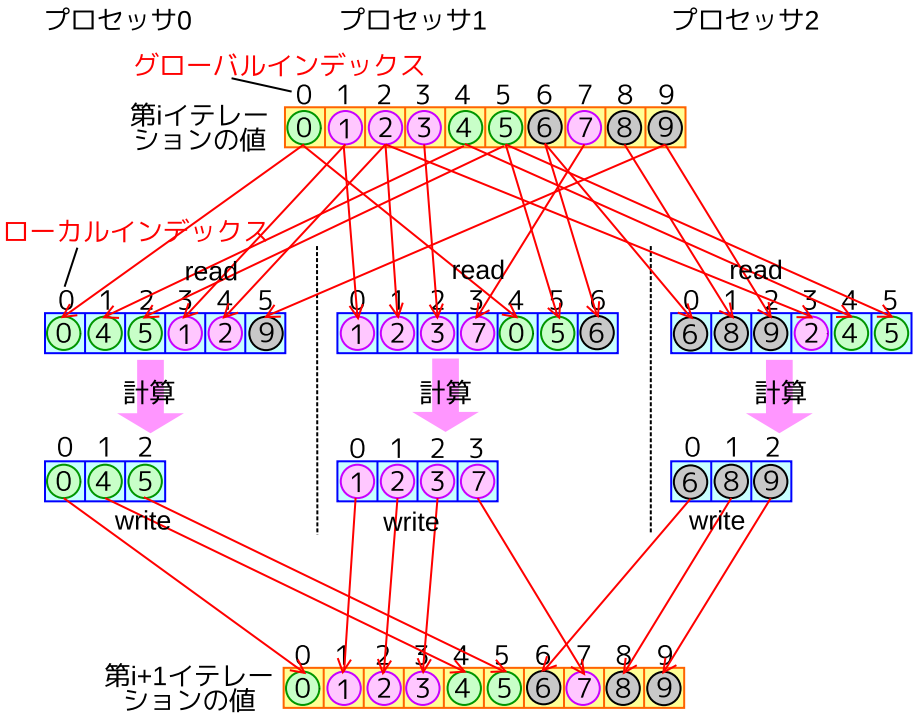

図 2.7　グローバルビュー型のグローバルアドレス空間モデルで記述した非定型なグラフ計算（「グローバルアドレス空間を極力アクセスしない方法」の場合）.

記述することができる．なお，分散共有メモリモデルは，その性質上，必然的に「グローバルアドレス空間を自由にアクセスする方法」になる．この記述方法は問題点 I と問題点 II の両方を解決している．

第 2 の記述方法は，グローバルアドレス空間へのアクセスは内部的な通信を引き起こしうることを意識し，グローバルアドレス空間へのアクセスを極力避ける記述方法である．この方法では，グローバルアドレス空間上のできるかぎり大きい範囲のデータをいったんローカルアドレス空間に読み込み，できるかぎりローカルアドレス空間上で計算を行い，ローカルアドレス空間上の計算結果を一気にグローバルアドレス空間に書き戻すような記述を行う．たとえば，UPC において，グローバルアドレス空間上の配列 a の 100000 要素の合計を計算するコードは図 2.5 のように記述できる．本稿では，この記述方法を「グローバルアドレス空間を極力アクセスしない方法」と呼ぶ．この記述方法では，図 2.1 のグラフ計算は，図 2.7 に示すように記述できる．この記述方法は問題点 I を解決しているが，ローカルアドレス空間が必要になるという点で問題点 II は解決していない．

2.1.4.3　性能

まず，「グローバルアドレス空間を自由にアクセスする方法」について考える．この記述方法の第 1 の欠点は，**性能が低い**点である．たとえば，図 2.4 のコードにおいて，グローバルアドレス空間上の配列 a の実体がリモートに存在する状況を考える．このとき，第 1 の実装方法として，もっとも単純に，「$a[i]$ へのアクセスがあるたびにリモートから 4 バイトのデータを get する」というように処理系が実

装されている場合を考える．この場合，グローバルアドレス空間へのアクセスが起きるたびに内部的な通信が引き起こされるため，著しい性能低下が起きてしまう．

そこで，第 2 の実装方法として，分散共有メモリ処理系のように，OS のページサイズ（多くの場合 4 KB）を単位としたコヒーレンシ管理を行う場合を考える．この場合には，4 KB をアクセスするたびにページフォルトが 1 回発生してリモートからのページ転送が起き，その時点からの 4 KB 分のアクセスはローカルにヒットすることになる．よって，図 2.4 のような連続アクセスに対しては第 1 の実装方法よりは性能がよいと思われるが，離散的なアクセスに対する性能は著しく低下してしまう．なぜなら，4 バイトのデータ $a[i]$ にアクセスするたびに 4 KB のページ転送が発生してしまうためである．これを防ぐためには，OS のページサイズよりも小さい粒度でコヒーレンシ管理を行う方法 [165] が考えられる．しかし一方では，連続的なアクセスに対する性能を高めるうえでは 4 KB というコヒーレンシ粒度でさえ小さすぎるという問題もある．このように，どのようなコヒーレンシ粒度が適切であるかは各アプリケーションの各局面に応じて大きく変化するため，処理系によってあらかじめ決められたコヒーレンシ粒度を用いるだけでは，アプリケーション全体の性能を十分に高めることは難しい．

そこで，第 3 の実装方法として，inspector/executor[171, 172, 14] を利用する場合を考える．繰り返し実行されるあるループ文 L に関して，「ループ文 L の i（$i \geq 2$）回目の実行でアクセスされるグローバルアドレスたちおよびその順序は，ループ文 L の 1 回目の実行でアクセスされるグローバルアドレスたちおよびその順序とつねに等しい」という性質が成り立つことが静的に保証できているとする．いい換えると，ループ文 L の本体が必要とするグローバルアドレスたちとその順序は不変であることを仮定できるとする．このとき，inspector/executor では，ループ文 L を 1 回目に実行するときに，ループ文 L のなかでアクセスされるグローバルアドレスたちとその順序をすべて記録する．そして，ループ文 L を 2 回目以降に実行する場合には，ループ文 L の実行直前に，1 回目の実行で記録したグローバルアドレスたちの値を一括でローカルアドレス空間に get してしまうことで，ループ文 L の本体を実行している最中のグローバルアドレス空間へのアクセスはすべてローカルアドレス空間にヒットするようにする．すなわち，inspector/executor を使うことで，2 回目以降のループ文 L の実行では，（通信隠蔽などの高度な最適化をのぞけば）もっとも効率的な通信を内部的に起こすことができる．しかし，この原理からわかるように，inspector/executor は，1 回しか実行されないループ文や，ループ文の本体でアクセスされるグローバルアドレスたちが動的に変化する可能性があるループ文には適用できないため，実際のアプリケーションでは適用範囲が限定されてしまうという欠点がある．

以上のように，プログラムが「グローバルアドレス空間を自由にアクセスする方法」で記述されている場合には，それを性能よく実行するのは難しい．これは，一般的には次のような理由によるものだと考えられる．まず，理想的な性能を達成するためには，グローバルアドレス空間へのアクセスをできるかぎり集約して，できるかぎりローカルアドレス空間上で計算を進めることが重要になる．そのためには，当然，処理系が，グローバルアドレス空間へのアクセスをどのように集約すればよいかをわかっている必要がある．ところが，「グローバルアドレス空間を自由にアクセスする方法」で記述されているコードだけからは，それがわからない場合が多い．その結果，現実的には，適当なコヒーレンシ粒度のもとでコヒーレンシ管理を行ったり，都合のよい条件が保証できるループ文だけを最適化したりする程

度のことしかできない．このような理由により，図 2.4 に示すような単純な例だけではなく，非定型なアクセスをともないうる一般のアプリケーションに関して，「グローバルアドレス空間を自由にアクセスする方法」で記述されたコードを性能よく実行することは非常に難しいと考えられる．

「グローバルアドレス空間を自由にアクセスする方法」の第 2 の欠点は，**性能最適化の施しにくさ**である．この記述方法では，何を記述したときにどの程度の通信が内部的に起きるのかを非常に把握しにくい．通常のローカルなメモリアクセスと同様のシンタックスでリモートに存在するデータにアクセスできてしまうため，そのあまりの透過性ゆえに，意識しないうちに大量のリモートアクセスを記述してしまう場合も多い．

実際に，著者の大学院で 2010 年度に行われた並列分散プログラミングの講義において，UPC，X10，Chapel を使って並列粒子法を学生に記述させたところ，「あまりに性能が悪いが原因がわからない」「どこで通信が起きているのかを把握できない」という意見が多数寄せられた [6]．たとえば，著者が用いた UPC の実装では，グローバルアドレス空間を read/write するたびに内部的な通信が発生するため，図 2.4 のようなコードを記述してしまうと，100000 回の通信が起きることになり著しい性能劣化が起きる．別の例として，著者が用いた Chapel の実装では，サイズ n の配列 a と b を用意し，配列 a が存在するノードとは別のノードで $b = a$ を実行すると著しい性能低下が起きる．この原因は，リモートな配列のコピーが，内部的には n 個の各要素のコピーとして実装されており，ノード間で n 回の通信が発生してしまうためである．かわりに，a と b を配列ではなくタプルとして宣言するようにすると，内部的にはノード間で 1 回の通信しか起きなくなり，性能が 100 倍改善する [6]．

当然，ここで紹介した事例は実装依存の例にすぎないが，一般に，「グローバルアドレス空間を自由にアクセスする方法」で記述する場合には，何を記述したときにどの程度の通信が起きるのかを把握しにくく性能最適化が難しいというのは事実である．まして，グローバルアドレス空間を read/write するたびに内部的な通信が発生することをできるかぎり避けるために，処理系がコヒーレンシ管理や緩和型のメモリコンシステンシモデル [204, 79, 15, 189] などを実装しているとすれば，ますます内部的な通信を把握しにくくなる．

以上をふまえると，グローバルビュー型のグローバルアドレス空間モデルにおいて，性能および強力な性能最適化を求めるのならば，「グローバルアドレス空間を自由にアクセスする方法」は適切ではない．そこで，次に，「グローバルアドレス空間を極力アクセスしない方法」について考える．

「グローバルアドレス空間を極力アクセスしない方法」は，グローバルアドレス空間への read/write をできるかぎり集約して，できるかぎりローカルアドレス空間上で計算を進めることを意図した記述方法である．したがって，グローバルアドレス空間への read/write をメッセージパッシングモデルと同等の性能で実現できるような read/write の API さえ用意されていれば，原理的には，メッセージパッシングモデルと同等の性能を達成できると考えられる．いい換えると，メッセージパッシングモデルで記述した場合に起きる通信と同等の通信が起きるように，グローバルアドレス空間への read/write が引き起こす内部的な通信を強力に制御できるような read/write の API が用意されていれば，グローバルビュー型のグローバルアドレス空間モデルでもメッセージパッシングモデルと同等の性能を達成できると考えられる．ところが，UPC，Global Arrays，X10，Chapel，XcalableMP などの既存の PGAS

処理系では，そのような強力な read/write の API は提供されていない．たとえば，図 2.1 のグラフ計算を図 2.7 のように記述した場合に，内部的な通信としては図 2.2 と同様の通信が起きるようにしたいわけであるが，既存の処理系の API ではそのような記述はできない．また，グローバルアドレス空間への read/write が引き起こす内部的な通信を自由に制御するという観点から考えると，以下のような操作を明示的に実現できる API も提供されていることが望ましい：

(1) グローバルアドレス空間上で連続するデータの read/write を一括で行う．
(2) グローバルアドレス空間上のさまざまな位置に散らばったデータの read/write を必要最小限の通信回数で一括で行う．
(3) グローバルアドレス空間上のデータの所在を動的に変更する．
(4) グローバルアドレス空間上のデータを一時的にキャッシュする．

しかし，著者の知るかぎり，get/put 操作を基本とする既存の PGAS 処理系で，(2)(3)(4) のための API を提供しているものは存在しない．要約すると，**read/write が内部的に引き起こす通信をわかりやすく強力に制御できるような API を設計しさえすれば，グローバルビュー型のグローバルアドレス空間モデルでもメッセージパッシングモデルに匹敵する性能と性能最適化を達成できると考えられるにもかかわらず，実際にそのような API を設計している処理系は存在しない**のが現状である．そこで，DMI では，そのような API を設計し，メッセージパッシングモデルよりもプログラマビリティの高いグローバルビュー型のグローバルアドレス空間モデルを提供しつつも，MPI と同等の性能を達成できるような処理系を実現させる．

2.1.4.4 再構成に対するプログラマビリティ

グローバルビュー型のグローバルアドレス空間モデルは，**再構成可能な並列計算を記述するのに適している**．なぜなら，グローバルアドレス空間モデルにおいては，データ通信の媒体がメモリアドレスとして抽象化されているため，プログラマは，そのときどのようなプロセッサが並列計算に参加しているかとは関係なく，単に意図したメモリアドレスを read/write するだけでデータを通信できるためである．たとえば，あるデータ x がメモリアドレス 0x12340000 に格納されている場合，再構成にともなってプロセッサがどのように参加/脱退したとしても，データ x はメモリアドレス 0x12340000 に存在し続けるので，プログラマの視点ではつねに 0x12340000 を read/write するだけでよい．すなわち，メッセージパッシングモデルやローカルビュー型のグローバルアドレス空間モデルとは異なり，プログラマは，プロセッサの参加/脱退にともなうデータの移動などを考慮する必要がない．このように，グローバルビュー型のグローバルアドレス空間モデルでプログラマが再構成を意識することなくデータ通信を簡単に記述できる理由は，データの所在管理がプログラマではなく処理系によって行われているからである．

2.1.4.5 その他の特徴

グローバルビュー型のグローバルアドレス空間モデルも，ローカルビュー型のグローバルアドレス空間モデルと同様に，単方向通信に基づくプログラミングモデルであるため，双方向通信における send/receive の対応の誤りに起因するバグやデッドロックは発生しない．また，**非同期的なアルゴリズ**

ムも自然に記述できる.

2.2 再構成可能な並列計算のための処理系

本研究の第 2 の目標は，主に高性能並列科学技術計算を対象として，再構成可能な並列計算を簡単に記述できるようにすることである．本節では，仮想マシンを粒度とした再構成，MapReduce による再構成，スレッド移動による再構成など，並列計算の再構成に自然に適用可能だと考えられる並列分散技術を幅広くとり上げる．そして，高性能並列科学技術計算の再構成を実現するためには，スレッド移動またはプロセス移動に基づく再構成が適切であることを指摘する．そのうえで，**2.2.4** 節と **2.2.5** 節で，スレッド移動またはプロセス移動の既存手法が抱える問題点について指摘する．

2.2.1 仮想マシンを粒度とした再構成

Amazon EC2[1]，Windows Azure[12] などの IaaS（Infrastructure as a Service）型のクラウドコンピューティングサービス [21, 123, 124, 190, 34] では，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．よって，仮想マシンのうえで並列計算を実行し，必要に応じて仮想マシンの台数を増減させることによって，並列計算の再構成を実現することができる．たとえば，ニュースサイトや SNS などの Web サーバであれば，負荷が小さい平常時には 2 台の仮想マシンで Web サーバ機能を負荷分散して運用し，負荷が上昇した場合には新たに 18 台の仮想マシンを追加して，合計 20 台の仮想マシンで Web サーバ機能を負荷分散して運用するなど，必要に応じて並列計算の規模を自由に拡張/縮小させることができる．このように，仮想マシンを粒度として再構成を行うことの利点は，利用者に対して仮想マシンという汎用的な実行環境が提供されているため，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が幅広いという点である．

一方で，第 1 の欠点は，仮想マシンの起動/停止には数分を要するため，並列計算の再構成の要求に対する応答性が悪い点である．また，仮想マシンを移動させる場合にも，いま興味のある並列計算が消費しているデータ以外に仮想マシンの OS 自体が消費しているデータも転送しなければならないため，プロセス移動やスレッド移動などによって並列計算だけを移動させる場合と比較すると，より長い移動時間が必要となる．ライブマイグレーション [157, 44] や転送データの圧縮などの技術によって仮想マシンの移動時間を削減することは可能だが，いずれにせよ，仮想マシンが使用しているデータ全体を移動させる必要があることには変わりない．本研究が対象とするような高性能並列科学技術計算においては，その並列科学技術計算を実行しているプロセス/スレッドが起動/停止/移動されれば十分な場合が多く，OS の実行環境すべてが起動/停止/移動されることは要請されていないため，仮想マシンを粒度とした再構成は不必要に重すぎる場合が多い．

第 2 の欠点は，仮想マシンでは I/O 処理が仮想化されているため，物理マシンと比較すると，高性能並列科学技術計算に要求されるようなファイル I/O やネットワーク I/O の性能が低い点である．通常，ゲストの仮想マシンで発行された I/O 処理は，いったん仮想マシンモニタあるいは特権ドメインによってフックされ，ハードウェアに対するアクセス権限などが検査されたのちに発行される．よって，各 I/O 処理ごとに仮想マシンと仮想マシンモニタとのコンテキストスイッチが必要であり，無視できな

いオーバヘッドをともなう [84, 83]．4 台の Xen 上で mpich を用いて NAS Parallel Benchmark による性能評価を行ったところ，4 台の物理マシン上で実行する場合と比較して，IS と CG の性能がそれぞれ 12% と 17% 低かったという報告 [84] がある．また，Xen の仮想マシンスケジューラがゲストの仮想マシンをスケジューリングする間隔が 30 ミリ秒であるため，各仮想マシンにパケットが届けられるまでに最大 30 ミリ秒の遅延が生じ，これが TCP のスロースタートフェーズにおける輻輳ウィンドウの増加速度を大きく下げてしまうという報告 [98] もある．このように，現状のデフォルトの仮想マシンは，通信性能に敏感な高性能並列科学技術計算を行う環境としては適していない．ただし，近年では，仮想マシンのネットワーク I/O の性能を改善するため，仮想マシンモニタをバイパスしてゲストの仮想マシンから直接ネットワーク I/O を発行する技術 [84] などが提案されており，物理マシンにおけるネットワーク I/O との性能差は埋まりつつある．また，同一ノード内の仮想マシン間の通信を共有メモリ経由で実現する方法も研究されている [191].

まとめると，仮想マシンを粒度とした再構成は，Web サーバの運用など，OS を単位とした計算規模の拡張/縮小が要請されるような並列計算に対しては適している．しかし，高性能並列科学技術計算のように，プロセス/スレッドを単位として計算規模が拡張/縮小されれば十分であり，プロセス/スレッドどうしの密な通信性能が重要視されるような並列計算には適していないといえる．

2.2.2 プロセスを粒度とした再構成

2.2.2.1 Google App Engine

プロセスを粒度として並列計算の再構成を実現できるクラウドコンピューティングサービスとして，Google App Engine[2] がある．Google App Engine では，利用者が Java もしくは Python で記述した Web アプリケーションを登録しておくと，その 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，Web アプリケーションの処理プロセス数が透過的に増減され，利用者が何の意識を払わずとも負荷分散が図られる．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よさである．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であるとされている．この応答性のよさは，仮想マシンと比較するとプロセスは軽量であり，生成/破棄などのとり扱いを高速に実現でき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通信するような並列計算は想定されていない点である．Google App Engine では，プロセス間のソケット通信などは許可されておらず，プロセス間のデータ共有は，BigTable[40] と呼ばれるデータベースや Memcached[5] と呼ばれる分散メモリオブジェクトキャッシュを通じて実現する．しかし，BigTable はデータの永続的なストレージとしての使用を想定したものであり，Memcached は，BigTable へのアクセス回数を減らすために，過去のリクエストを処理した結果を一時的にメモリ上にキャッシュするための使用を想定したシステムにすぎない．よって，このようなデータ共有の手段だけを使って，高性能並列科学技術計算に要求されるような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．

Google App Engine の第 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 では，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という制約

```
map(Key k, Value v):
  output(<k,v>)

reduce(Key k, Value v[0..r − 1]):
  s = 0
  foreach i in 0..r − 1 do
    s += v[i]
  done
  output(<k,s>)
```

図 2.8 MapReduce を使って文書中の単語数をカウントするプログラム.

```
SATIN int fib(int n) {
  if (n < 2) {
    return n;
  }
  int x = SPAWN fib(n − 1);
  int y = SPAWN fib(n − 2);
  SYNC;
  return x + y;
}
```

図 2.9 Satin を使ってフィボナッチ数列を計算するプログラム.

がある．これは，短時間単位で計算資源を細かくスケジューリングすることによって，各利用者からの計算規模の拡張/縮小の要求に対する応答性を高めるために設けられている制約だと考えられる．しかし，有限要素法や粒子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，Google App Engine も，高性能並列科学技術計算の再構成に適した処理系とはいえない.

2.2.2.2 MapReduce

MapReduce[47, 115, 179, 53] は，2004 年に Google によって提案された分散処理のためのプログラミングモデルである．本来は分散処理の容易化をねらって設計されたプログラミングモデルであるが，並列計算の再構成にも自然と応用させることができる．MapReduce では，プログラマは，map 関数と reduce 関数の 2 つの関数だけを記述すればよい．MapReduce の入力は多数の key-value ペアである．map 関数は，1 個の key-value ペアを入力として，新しい key-value ペア $< k_1, v_1 >$，$< k_2, v_2 >$，$< k_3, v_3 >$，... を出力する関数として定義する．reduce 関数は，map 関数が出力したすべての key-value ペアのうち，ある同一のキー k を持つソート済みのペアたち $< k, u_1 >$，$< k, u_2 >$，$< k, u_3 >$，... （$u_1 \prec u_2 \prec u_3 \prec \ldots$）を入力として，何らかの結果を出力する関数として定義する．要するに，reduce 関数には，map 関数によって生成されたたくさんの key-value ペアについて，同じキーを持つソート済みの key-value ペアたちをどのように統合するべきかを定義する．たとえば，たくさんの文書があるとき，各単語がすべての文書中に合計何回出現するかを計算するプログラムは，図 2.8 のように記述できる．ここで，入力として与える key-value ペアは，各単語 w と各文書 f に対する $<$ 単語 w，文書 f 中の単語 w の出現数 $>$ とする.

MapReduce では，このような map 関数と reduce 関数だけを記述しておくと，あとは処理系が以下の処理を自動的に行ってくれる：

(1) 入力の各 key-value ペアに対して map 関数を呼び出す．その結果，大量の key-value ペアたちが生成される．map 関数の呼び出しは，入力の各 key-value ペアに対して完全に独立に行われる.
(2) それら大量の key-value ペアたちを key，value の優先度順でソートする.
(3) 同じキーを持つソート済みの key-value ペアたちを入力として，reduce 関数を呼び出す．reduce

関数の呼び出しは，各キーに対して完全に独立に行われる．

MapReduce では，map 関数と reduce 関数を複数組み合わせることで，転置インデックス作成，文書間の共通出現単語の抽出，ページランク計算などのグラフ計算など，多様な並列計算を記述することができる [53, 47, 179].

MapReduce では，プログラミングモデルの性質上，すべての map 関数と reduce 関数は独立に実行可能であるため，各 map 関数と各 reduce 関数は，（入力データのローカリティを考慮しつつ）その時点で空いているノードが適当に選択されて実行されることになる．また，ノードの故障などが原因で，ある map 関数や reduce 関数が途中でクラッシュしてしまった場合にはその関数は再実行される．したがって，MapReduce では，並列計算の実行中にノードを動的に参加/脱退させることで，簡単に計算規模を再構成させることができる．

このように，簡単なプログラミングによって並列計算の再構成を実現できるという点で MapReduce は理想的であるが，高性能並列科学技術計算に対して十分な性能を達成することは難しい．その理由は，MPI などの並列分散プログラミング処理系であれば本質的に通信する必要のあるデータのみを送受信すればよいのに対して，MapReduce の場合には，**本質的には通信する必要のないデータまでいったん key-value ペアの形で出力し，それら大量の key-value ペアをノード間で送受信する必要がある**からである．たとえば，仮にある map 関数 m で出力するデータの大部分がある reduce 関数 r の入力になることがアルゴリズム上わかっているとしても，map 関数 m と reduce 関数 r を同一のノードで実行することを指示して無駄なデータの送受信を省略するようなことは記述できないし，map 関数 m から reduce 関数 r に対して直接データを送信するようなことも記述できない．このように，MapReduce では，map 関数が実行されるノードと reduce 関数が実行されるノードの関係をプログラマは知ることも指示することもできず，map 関数のすべての結果を key-value ペアの形でいったん出力しなければならない．そして，それら大量の key-value ペアは，処理系によってソートされたあと，各 reduce 関数を実行するノードへと送信されることになるため，アルゴリズム上は本質的には必要のない通信が大量に発生してしまう．以上のような理由により，MapReduce は密な通信を必要とする高性能並列科学技術には適していない．なお，map 関数と reduce 関数を実行するノードの関係をプログラマが指示できるようにすることによって，無駄な key-value ペアの生成を省略するなどのプログラミングモデルの拡張は可能だが [61]，どのように拡張すれば，再構成可能な並列計算に対するプログラマビリティを失うことなくプログラミングモデルを拡張できるのかは自明ではない．なぜなら，そもそも，MapReduce において再構成可能な並列計算を簡単に記述できる理由は，map 関数と reduce 関数を実行するノードの関係をプログラマにいっさい指示させないことによって，処理系側が map 関数と reduce 関数のスケジューリングを自由に決定できているという点にあるからである．

2.2.2.3 Satin

Satin[147, 73] は，Java のシンタックスを拡張したタスク並列型の並列分散プログラミング処理系であり，分割統治法型の並列計算を簡単に記述し，かつ高性能に実行することができる．Satin では，関数に spawn 修飾子を付けることでその関数をタスクとして非同期に実行させることができ，sync 修飾子によって生成したタスクの完了を待機することができる．たとえば，Satin でフィボナッチ数列を計

算するプログラムを図 2.9 に示す．Satin では，各プロセッサが持つタスクキューを使って，生成された大量のタスクを Lazy Task Creation[134] の仕組みによって管理することで，タスクの生成/破棄を少ないオーバヘッドで実現している．また，タスクキューが空になったプロセッサは，Cluster-Aware Random Stealing[147] と呼ばれるワークスティーリングによって，他のプロセッサのタスクキューからタスクを奪う．これにより，良好なデータローカリティを保ちつつ，すべてのプロセッサ間での負荷バランスが図られる．

Satin では，動的にプロセッサを参加/脱退させることができる．プロセッサが参加する場合には，単純に，ワークスティーリングの要領で，実行中の適当なプロセッサからタスクを奪ってくるだけでよい．プロセッサ i が脱退する場合には，もっとも単純には，「その時点でプロセッサ i が所持しているタスクから生成されたすべてのタスク」を他のプロセッサのタスクキューに挿入すればよい．これにより，「その時点でプロセッサ i が所持しているタスクから生成されたすべてのタスク」が再計算されることになるため[*3]，プロセッサ i が脱退しても，並列計算全体の実行を正しく継続することができる．ただし，すでに計算済みのタスクの結果を再計算するのは無駄であるため，Satin では，計算済みのタスクの結果を再利用できるようにするためのより高度なプロトコルが実装されている [73].

以上のように，Satin では，分割統治法型で記述された高性能並列科学技術計算を良好なデータローカリティと負荷バランスを保ちつつ実行できるうえ，並列計算を再構成することもできる．マージソート，SAT ソルバ，レイトレーシングなど，分割統治法型で自然に記述できる並列科学技術計算は幅広い．しかし，たとえば **6.6.1** 節で述べるような非定型な領域分割をともなう有限要素法などは，分割統治法型で記述するよりも SPMD 型で記述する方が自然であり，かならずしも分割統治法型で並列科学技術計算を記述することがアルゴリズム上自然とはいえない場合もある．また，Satin ではグローバルアドレス空間が提供されているわけではないため，タスク間でデータを共有する手段も限定されている．これに対して，DMI では，グローバルアドレス空間を提供すると同時に，プログラミングモデルを限定することはせず，**より汎用的に**，プロセスが非同期的に参加/脱退できるグローバルアドレス空間を設計して実装する．

2.2.2.4 Phoenix

メッセージパッシングをベースとして，ノードの動的な参加/脱退をサポートした並列分散プログラミング処理系に Phoenix[175, 202] がある．Phoenix では，想定するノード数より十分に大きい定数 L に対して，仮想ノード名空間 $[0, L)$ を考え，各ノードにこの部分集合を重複なく割り当てる．つまり，任意の仮想ノード名 $i \in [0, L)$ がちょうど 1 個のノードに保持されるよう，各ノードに対して仮想ノー

[*3] プロセッサ i が所持しているタスクだけではなく，「プロセッサ i が所持しているタスクから生成されたすべてのタスク」を再計算する必要があるのは，次の理由による．プロセッサ i が所持しているタスク t がタスク t' を生成し，いまタスク t はタスク t' の終了を待機しているとする．また，ワークスティーリングによって，タスク t' はプロセッサ j によって所持されているとする．このとき，プロセッサ j は，タスク t' が終了した場合，プロセッサ i が脱退していなければ，タスク t を所持しているプロセッサ i に対してタスク t' の終了を通知すればよい．これによって，タスク t' の終了を待機していたタスク t の実行が再開される．ところが，プロセッサ i が脱退してしまうと，タスク t がどのプロセッサによって所持されているかが不明になってしまうため，プロセッサ j はどのプロセッサに対してタスク t' の終了を通知すればよいかを判断できなくなる．このような問題を単純に防ぐためには，タスク t だけでなく，タスク t から生成されたタスク t' も再計算することにすればよい．

ド名集合の割り当てを行う．そして，ノード p が参加する場合には，すでに実行中のノードが持つ仮想ノード名集合の一部をノード p に分け与える．ノード p が脱退する場合には，ノード p が持つ仮想ノード名集合を他のノードに対して委譲する．これにより，1 個の並列計算を通じて，実行中のノード全体でつねに仮想ノード名空間が重複なく包まれるように管理する．この管理のもとでは，ノードの参加/脱退を局所的な変更操作のみで実現できるうえ，仮想ノード名を用いたメッセージの送受信を行えば，ノードの参加/脱退が生じてもメッセージの損失が起こることはない．しかし，Phoenix では，ある時点でどの仮想ノードがどのノードによって保持されているのかがプログラマの視点から把握しにくく，データローカリティを意識したプログラムを記述するのが難しいという問題がある．また，Phoenix はメッセージパッシングモデルをベースとしているため，グローバルアドレス空間モデルをベースとしている DMI よりもプログラマビリティが低い．

2.2.2.5 MPI のチェックポイント/リスタート

MPI のチェックポイント/リスタートを用いることで，MPI プロセスをノード間で自由に移動させることができる．一般に，プロセスを粒度とした MPI のチェックポイント/リスタートは，プロセスのチェックポイント/リスタートを実現するミドルウェアに対して MPI 特有の通信をチェックポイント/リスタートするための機能を付け加えることで実現される．たとえば，研究 [164] 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[59] を基盤として，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てリスタート時にそれらの MPI メッセージを復旧させる機能を付け加えることで，MPI プロセスのチェックポイント/リスタートを実現している．また，MPI-Mitten[57] では，ヘテロジニアスな環境のプロセスのチェックポイント/リスタートを実現するミドルウェアである HPCM[58] を基盤として，プロセス移動を越えて MPI のコミュニケータや集合通信をサポートするためのアルゴリズムが提案されている．

本来，MPI のチェックポイント/リスタート [164, 181, 57] は，主として，耐故障な並列計算や並列計算の動的負荷分散を目的とした技術である．第 1 に，耐故障性に関しては，定期的に MPI プロセスをチェックポイントすることにより，並列計算がクラッシュした場合にはもっとも直近にチェックポイントした状態から並列計算をリスタートさせることができる．また，さまざまなハードウェア情報からノードの故障が予測される場合には，実際に故障が起きる前にそのノード上の MPI プロセスを別のノードへと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することもできる [181]．第 2 に，並列計算の動的負荷分散に関しては，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，利用可能な計算資源やその性能が動的に変化する計算環境において，その動的な変化に追随して MPI プロセスを移動させることによって，計算環境の変化に対して並列計算を効果的に適応させることが可能になる．このように，MPI のチェックポイント/リスタートは耐故障な並列計算や並列計算の動的負荷分散への応用が広く研究されているが，高性能並列科学技術計算の再構成にも自然に応用させることができる．すなわち，初期的に十分な数の MPI プロセスを生成しておき，ノードの参加/脱退にともなってそれらの MPI プロセスを適宜移動させることで，並列計算を自由に再構成することができる [80, 81]．すなわち，透過的なプロセス移動に基づく並列計算の再構成に応用させることができる．

しかし，これらの MPI のチェックポイント/リスタートは，いずれもメッセージパッシングモデルである MPI を前提としたものである．これに対して，DMI では，よりプログラマビリティの高いプログラミングモデルであるグローバルアドレス空間モデルを対象にして，並列計算の再構成を実現する．メッセージパッシングモデルでは，原理的には，チェックポイント/リスタート時に on the fly なメッセージを回収して復旧させればチェックポイント/リスタートを実現できるが，グローバルアドレス空間モデルの場合にはそれだけでは不十分であり，より高度な仕組みが必要になることを強調したい．詳しくは第 **3** 章で述べるが，たとえば，プロセスのチェックポイント/リスタートを越えて，read/write に対するコンシステンシを保証するための仕組みが必要であるし，あるプロセスでグローバルアドレスに対する read/write フォルトが発生したとき，そのデータがその時点でどのプロセスによって保持されているのかを追跡するための仕組みなども必要である．

2.2.2.6 MPI のプロセス増減による再構成

SRS[173], DyRecT[69], DRMS[103], PCM[128, 126] などの処理系では，MPI で記述された SPMD 型の反復計算を対象にして，動的なプロセスの増減を記述することができる．これらの処理系では，再構成前にチェックポイントするべきデータを登録する API と，再構成を行う API と，再構成後にデータをリストアするための API が提供されており，これらの API を通常の MPI のプログラムに対して挿入することで再構成可能な MPI プログラムを記述できる．たとえば，SRS では，第 1 に，すべてのプロセスが，SRS_Register(“foo”, a, $size$, BLOCK,...) 関数を呼び出すことで，再構成時に，各プロセス上の $size$ バイトのローカルな配列 a を各ブロックとしてブロック分散されているデータを，“foo” という名前でチェックポイントするよう登録することができる．第 2 に，SRS_Check_Stop() 関数を呼び出すことで，実際に再構成を起こすことができる．このとき，SRS_Register() 関数で登録したデータのチェックポイントが行われる．そして，再構成後に，新しいプロセス集合に属するすべてのプロセスが SRS_Read(“foo”, b, BLOCK,...) 関数を呼び出すことで，“foo” という名前でチェックポイントされているデータを新しいプロセス集合でブロック分散しなおして，各プロセスのローカルな配列 b にリストアすることができる．要するに，これらの処理系では，チェックポイントされるべきデータが格納されているポインタ，そのデータ分散，データのリストア先となるポインタさえ指定しておけば，再構成時に必要なデータの再分散を処理系が透過的に実現してくれる．しかし，これらの処理系は MPI を前提としたものであるうえに，適用可能な並列計算が SPMD 型の同期的な反復計算に限定されている．これに対して DMI では，グローバルアドレス空間モデルに基づき，ノードが非同期的に参加/脱退するような，より幅広い並列計算を記述できるようにする．

2.2.3 スレッドを粒度とした再構成

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し，MPI におけるスレッド移動を実現した処理系として，Tern[102] や Adaptive MPI[80, 81] などがある．Adaptive MPI は，Adaptive Mesh Refinement[171] などの静的な負荷分散が難しい並列計算を対象にして，ノード間でスレッドを移動させることで動的な負荷分散を図ろうとする処理系である．Adaptive MPI では，プログラマはプロセッサ数よりも十分に多いスレッドを生成するだけでよい．すると，あとは処理系が各ノードの負荷バランスやスレッド間のデータ共有の度合いを判断して，これら大量のスレッドを透過的にノード間で

スケジューリングし，動的な負荷分散を実現してくれる．これらの処理系は主として並列計算の動的負荷分散を対象にしたものであるが，透過的なスレッド移動に基づく並列計算の再構成にも自然と応用させることができる．しかし，これらの研究も MPI を前提としたものであり，DMI のように，グローバルアドレス空間モデルに基づいて並列計算の再構成を実現するものではない．

2.2.4 スレッド移動の既存手法とその問題点

ここまで，並列計算の再構成を実現する粒度の違いに応じて，仮想マシンを粒度とした再構成，プロセスを粒度とした再構成，スレッドを粒度とした再構成についての関連研究を述べてきた．とくに，高性能並列科学技術計算を再構成するという目的では，プロセス/スレッドを粒度とした再構成が適していることを指摘し，実際に，Satin や MPI のチェックポイント/リスタートでは，（グローバルアドレス空間は提供されてはいないものの）プロセス/スレッドを粒度として高性能並列科学技術計算を再構成できることを述べた．この事情は DMI においても同様で，DMI でも，プロセス/スレッドを粒度としてプロセス移動/スレッド移動を通じて並列計算の再構成を実現する．したがって，本節と次節では，それぞれ，スレッド移動とプロセス移動に関する既存手法を観察し，その問題点を指摘する．

2.2.4.1 スレッド移動時におけるポインタの無効化

スレッド移動 [18, 19, 105, 49, 92, 90, 91, 89, 56, 183, 42, 85, 132, 193, 95, 192, 194, 114] 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領域を（とくに別のノードの）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．既存のスレッド移動の手法には大きな問題点が 2 点ある．

第 1 の問題点は，スレッド移動時のポインタのとり扱いである．スレッド移動にあたっては，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，この各スレッド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，スレッドの正しい実行を保証できなくなる．もちろん，移動元プロセスと移動先プロセスとで，まったく同一のアドレスにスレッドのメモリ領域を配置することができればポインタが無効化することはないが，スレッド移動時に，スレッドが移動元プロセスで使用していたアドレスが移動先プロセスにおいて空いている保証はない．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に合わせて完全に正しく更新する手法 [49, 92, 90, 91, 89] 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．よって，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，前者の方法には，プログラミングの負担を増大させるという問題があり，後者の方法には，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不可能であるという問題がある．

第 2 の解決策は，iso-address[18, 19, 132] と呼ばれる方法である．iso-address では，CPU で利用可

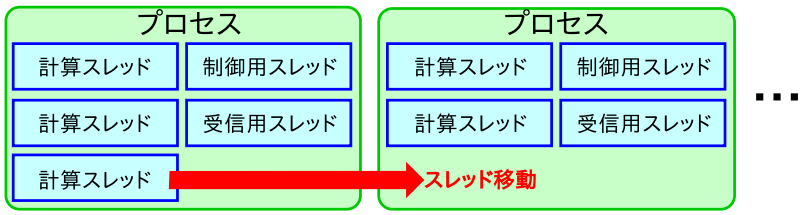

図 2.10　マルチスレッド型の処理系におけるスレッド構成の例.

能な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．たとえば，2^{32} のアドレス空間を 1024 個の小アドレス空間に均等に分割し，スレッド i（$0 \leq i < 1024$）は i 番目の小アドレス空間を使う，というように静的に決めておく．これにより，あるスレッドが使用しているアドレス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．いい換えると，スレッド移動時に，移動先プロセスにおいて，そのスレッドが移動元プロセスで使用しているアドレスが使用されていることはありえない．よって，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のアドレスにメモリ領域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 [89, 42].

しかし，iso-address には，**計算規模が CPU のアドレス空間全体のサイズ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サイズを 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領域のサイズを 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 $s = 4$ MB であり，各スレッドが $s = 2$ 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 [105, 183, 85] 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 $s = 64$ GB，$s = 512$ GB ならば $n = 1024$ 個であり，これらの数字は 2011 年 2 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，今後ますます拡大する計算規模に対応していくためには，iso-address では不十分であり，計算規模がアドレス空間全体のサイズに制限されないスレッド移動の手法が要請されているといえる．当然，今後のハードウェアの進化にともなって $w = 2^{47}$ という数字自体が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サイズに制限されないスレッド移動の手法が存在することには価値がある．そこで，DMI では，計算規模がアドレス空間全体のサイズに制限されないスレッド移動の手法として random-address を提案する．random-address によるスレッド移動については第 **8** 章で述べる．

2.2.4.2　プログラミング制約の存在

スレッド移動における第 2 の問題点は，安全なスレッド移動を実現するためには，各スレッドからアクセスできるメモリ領域に関してプログラミング制約を設ける必要がある点である．一般に，スレッド

移動を用いる処理系は，図 2.10 に示すように，各プロセスのなかに複数のスレッドが存在するマルチスレッド型の構成をとる [121, 97]．そして，並列計算の動的負荷分散や再構成などの目的のために，各スレッドをさまざまなプロセス間で移動させることになるが，このとき，各スレッドが，そのスレッドからアクセス可能なメモリ領域を自由に使用してしまっているとするといくつかの問題が生じる．

たとえば，プロセス p のなかにスレッド 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るとすると，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．このとき，スレッド j の移動にともなってグローバル変数 g もプロセス q に移動させるべきかさせないべきかが問題となるが，いずれの場合にも，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．なぜなら，グローバル変数 g を移動させないとすれば，スレッド j はプロセス q に移動したあとでグローバル変数 g を参照できなくなるし，一方で，グローバル変数 g を移動させるとすれば，プロセス q にもグローバル変数 g がすでに存在して別の値を持っていた場合に，プロセス q のグローバル変数 g の値を書きつぶしてしまうことになるため，もとからプロセス q で走っていたスレッド k の実行を正しく継続できなくなるためである．このように，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，各スレッドが使用できるメモリ領域についてプログラミング制約を加える必要がある．

このように，マルチスレッド型の処理系において，各スレッドを粒度とした移動を実現するためには，各スレッドが使用できるメモリ領域が何らか制約されること自体は避けられない．すなわち，「真に透過的なスレッド移動」を実現することはできない．本稿では，**8.3.4** 節で DMI のスレッド移動におけるプログラミング制約について厳密に議論したうえで，**8.3.5** 節で，スレッド移動の既存研究におけるプログラミング制約との比較を行う．

2.2.5 プロセス移動の既存手法とその問題点

そもそも，スレッド移動を用いるとなぜプログラミング制約が必要になるかといえば，**各スレッドを粒度としてスレッド移動を行うという行為は各スレッドが使用するアドレス空間が独立していることを要求しているにもかかわらず，マルチスレッド型の構成で実装するという行為は複数のスレッドがアドレス空間を共有することを要求しているという点で，矛盾が生じているから**である．そう考えると，各インスタンスをスレッドではなくプロセスとして実装すれば，各インスタンスが使用するアドレス空間を独立させることができるため，プログラミング制約を完全に撤廃することができ，真に透過的なインスタンス移動を実現できるように思われる．プロセス移動 [163, 170, 180, 38, 107, 66] は，BLCR[59] や Libckpt[150] などのプロセスをチェックポイント/リスタートするためのライブラリを利用することで実現できる．ところが，これはたしかに事実ではあるが，実際の事情はそう単純ではない．その理由は，一般に，並列分散プログラミング処理系における各インスタンスをプロセスとして実装すると，スレッドとして実装する場合と比較して，**インスタンス間のデータ共有のオーバヘッドが大きい**という問題と，**処理系の開発者にとってのプログラマビリティが悪い**という問題が起きるからである．ここで，

処理系の開発者とは，ユーザプログラムを記述するプログラマのことではなく，MPI や DMI のような並列分散プログラミング処理系自体を開発する開発者のことを意味する．以下では，この 2 つの問題について詳しく分析する．

第 1 に，インスタンス間のデータ共有のオーバヘッドの問題について考える．一般に，並列分散プログラミング処理系においては，同一ノード内のインスタンス間でさまざまなデータを共有する必要がある．たとえば，図 2.10 に示すように，計算スレッドのほかに，他のプロセスからのメッセージを受信する受信用スレッド，計算スレッドの挙動を監視しスレッド移動を制御する制御用スレッドが存在するようなマルチスレッド型の処理系を考えると，受信用スレッドが受信したデータを計算スレッドのメモリ領域に書き込んだり，制御用スレッドが計算スレッドに対してスレッド移動を指示したりする必要がある．処理系として実現する機能が複雑になればなるほど，インスタンス間での複雑なデータ共有が必要とされる．このとき，各インスタンスがスレッドとして実装されていれば，アドレス空間が共有されているため，データ共有のオーバヘッドは小さい．たとえば，スレッド t が別のスレッド t' のアドレス空間にデータを write するためには，意図するアドレスに単に write すればよいからである．これに対して，各インスタンスがプロセスとして実装されている場合，アドレス空間が共有されていないため，ソケット，パイプ，プロセス間共有メモリなどを利用してデータ共有を実現する必要がある．これら 3 種類の手段のなかではプロセス間共有メモリによるデータ共有がもっとも高速な場合が多いが，プロセス間共有メモリを利用する場合でも，余分なメモリコピーが 1 回必要になる．たとえば，プロセス p がデータ d をプロセス p' のアドレス空間に write するためには，(1) プロセス p がいったんデータ d をプロセス間共有メモリに write したあと，(2) プロセス p とプロセス p' で同期をとり，(3) プロセス p' がプロセス間共有メモリからプロセス p' のアドレス空間にデータ d をコピーする，という作業が必要となる．詳しくは **9.2** 節で述べるが，この余分なメモリコピーを防ぐために，カーネルに修正を加えて，プロセス p から別のプロセス p' のアドレス空間を直接 read/write するためのシステムコールを作り出す方法 [70, 108, 94] もある．しかし，この方法はデータのサイズが大きい場合には効果的であるものの，データのサイズが小さい場合には，システムコールを呼び出す際のコンテキストスイッチのオーバヘッドが無視できなくなる．以上のように，プロセス間でのデータ共有は，スレッド間でのデータ共有と比較してオーバヘッドが大きいという問題がある．

第 2 に，並列分散プログラミング処理系の開発者にとってのプログラマビリティの問題について考える．各インスタンスがスレッドとして実装されていれば，処理系の開発者は，1 個の共有されたアドレス空間のうえで，意図するアドレスを read/write するだけでスレッド間のデータ共有を実現できる．たとえば，スレッド t がデータ d をスレッド t' のアドレス空間に write するためには，単純にそのアドレスに write すればよい．これに対して，各インスタンスがプロセスとして実装されている場合，プロセスどうしがお互いのアドレス空間を自由にアクセスすることはできないため，前述のように，プロセス間共有メモリを介したメモリコピーを記述しなければならない．また，詳しくは **9.2** 節で述べるが，プロセス間共有メモリを動的に拡張/縮小させることはプロセス間共有メモリのセマンティクス上記述しにくいため，プロセス p がプロセス p' とデータ共有するために使用するプロセス間共有メモリのサイズ s は静的に固定されることが多い．この場合，プロセス p がプロセス p' に対してサイズ s 以上の

データを write する際には，たとえば，そのデータをサイズ s ごとに区切って，プロセス間共有メモリを介してパイプライン方式で送るような記述が必要になる．以上のように，プロセス間でのデータ共有は，スレッド間でのデータ共有と比較するとプログラミングが複雑化する．いい換えると，インスタンス間でアドレス空間が共有されている方が，インスタンス間でアドレス空間が独立しているよりも，処理系を実装するのが容易である．この影響は，処理系が複雑化し，インスタンス間での複雑なデータ共有が要求されるほど顕著になる．たとえば，メッセージパッシングモデルの処理系であれば，基本的にはユーザプログラムから指示された通信をそのまま実現すればよいだけであるため，プロセス間共有メモリを介した通信を記述するのもそれほど困難なわけではなく，既存の MPI の処理系では実際に行われている [39, 108]．しかし，DMI の場合には，高機能なグローバルアドレス空間上のデータのコヒーレンシを管理するために，第 **4** 章で説明するような非常に複雑なインスタンス間でのデータ共有が必要であり，これをプロセス間共有メモリを介して記述するのは難しい．

以上の議論をまとめると以下のようになる：

- 真に透過的なインスタンス移動を実現するためには，各インスタンスはプロセスとして実装される必要がある．
- しかし，インスタンス間でのデータ共有のオーバヘッドを小さくしたり，（処理系の開発者にとっての）データ共有のプログラマビリティを高めるためには，各インスタンスはスレッドとして実装される必要がある．

いい換えると，真に透過的なインスタンス移動を実現するためには，インスタンスどうしでアドレス空間を共有したい局面と共有したくない局面が混在する．そこで，本研究では，スレッドとプロセスの「中間」の機能を持つ新たなカーネルプリミティブとして half-process を提案し，上記の 2 つの矛盾する要求を同時に解決する．

2.3 要約：既存研究との相違点

本章では，第 1 に，性能のよさ，性能最適化の自由度と見通しのよさ，非定型な並列計算に対するプログラマビリティ，再構成可能な並列計算に対するプログラマビリティという 4 つの観点から，メッセージパッシングモデル，ローカルビュー型のグローバルアドレス空間モデル，グローバルビュー型のグローバルアドレス空間モデルを比較し，とくに以下の 2 点を明らかにした：

- メッセージパッシングモデルよりもグローバルビュー型のグローバルアドレス空間モデルの方が，非定型な並列計算に対するプログラマビリティも再構成可能な並列計算に対するプログラマビリティも高い．
- read/write が内部的に引き起こす通信をわかりやすく強力に制御できるような API を設計しさえすれば，グローバルビュー型のグローバルアドレス空間モデルでもメッセージパッシングモデルに匹敵する性能と性能最適化を達成できると考えられる．ただし，著者の知るかぎり，実際にそのような API を設計している処理系は存在しない．

上記の 2 点を根拠として，DMI では，並列分散プログラミングモデルとしてグローバルビュー型のグローバルアドレス空間モデルを採用し，read/write が内部的に引き起こす通信をわかりやすく強力に最適化できるような API を設計する．

本章では，第 2 に，高性能並列科学技術計算の再構成を実現するためには，スレッド移動またはプロセス移動に基づく再構成が適切であることを指摘した．スレッド移動またはプロセス移動に基づいて再構成を実現する既存研究は存在するが，DMI のように，グローバルアドレス空間モデルに基づいて再構成を実現した例は存在しない．また，スレッド移動の既存手法には，計算規模が CPU のアドレス空間全体のサイズに制限されてしまうという問題と，プログラミング制約が存在するという問題がある．そこで，DMI では，前者の問題を random-address と呼ばれる手法で解決し，後者の問題を half-process と呼ばれるカーネルプリミティブを導入することで解決する．

第 3 章

高性能かつ再構成可能なグローバルアドレス空間の設計

PGAS 処理系においてもっとも基本となるのはグローバルアドレス空間の設計と実装である．本章では，DMI の処理系の全体像を俯瞰したうえで，再構成をともないうる高性能な並列計算をサポートするためのグローバルアドレス空間の設計について，関連する処理系との比較を行いつつ説明する．

3.1 全体像

DMI は，グローバルビュー型の PGAS モデルに基づくマルチスレッド型の並列分散プログラミング処理系である [198, 76, 199, 196]．DMI のシステム構成を図 3.1 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．

第 1 に，DMI は**キャッシュコヒーレントなグローバルアドレス空間**を提供する．DMI における各

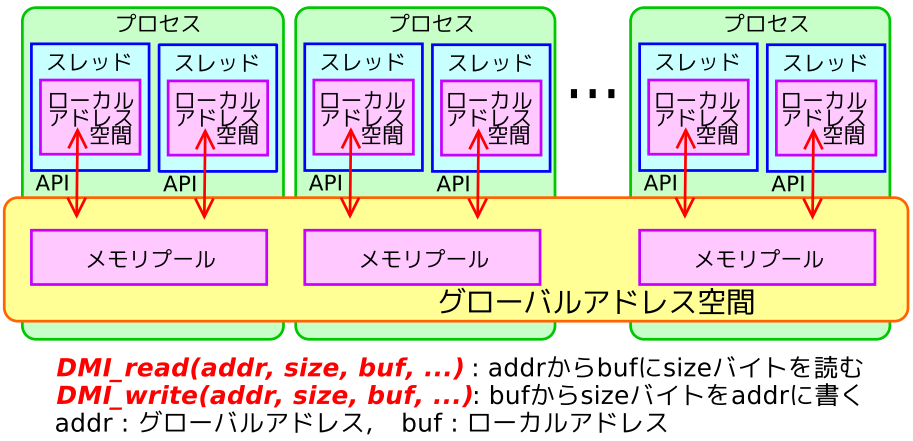

図 3.1 DMI のシステム構成．

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サイズは各プロセスを生成するときに明示的に指定できる．すると，DMI はこれらのメモリプールをメモリ資源として，ページテーブルやキャッシュ管理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，分散環境上にグローバルアドレス空間を構築する．これにより，各スレッドは，グローバル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スレッドが属する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，その時点で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．Co-Array Fortran[149, 45, 185, 46] や Titanium[188, 171, 77, 50, 172]，UPC[43, 62, 48, 46]，X10[41]，Chapel[37, 26] などの大部分の PGAS 処理系では get/put 操作しかサポートされていないが，DMI では，データのアクセスローカリティに応じてグローバルアドレス空間のデータを各プロセスにキャッシュすることができる．当然，このキャッシュのコヒーレンシは DMI によって自動的に維持される．

第 2 に，図 3.1 に示すように，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，内部的には物理的な共有メモリ経由で実現される．すなわち，DMI は，ノード間とノード内の階層的な並列性を活用したハイブリッドプログラミング [153, 178, 78, 24] を透過的に実現している．

第 3 に，多数のプロセスを利用することで大容量のグローバルアドレス空間を構築し，DMI を**遠隔スワップシステム**[203, 205, 141, 158, 140, 184, 182, 151] として動作させることもできる．遠隔スワップシステムとは，近年のネットワーク性能の向上により，ローカルなディスクスワップへのアクセス時間よりもネットワーク経由での他ノードのメモリへのアクセス時間の方が高速になっていることを背景として，記憶階層においてローカルなメモリとディスクの中間に位置する新たな記憶階層として他ノードのメモリを組み込もうとする技術である．既存の遠隔スワップシステムを大きく分類すると，Teramem[205] や DLM[203]，Nswap[141] のように逐次プログラムのみに対して大容量のメモリを提供するシステムと，Cashmere-VLM[60] や JIAJIA[79] のように並列プログラムに対して大容量の分散共有メモリを提供するシステムに分類できるが，DMI は後者の部類に属する．遠隔スワップシステムによる大容量のメモリの実現は，モデル検査 [88, 87, 67, 65, 68] や Web グラフ解析 [47, 129] などのように巨大なグラフ探索問題に帰着するような各種のユーザプログラムをはじめとして，解ける問題の規模が利用可能なメモリ量によって支配されるようなユーザプログラムを性能よく実行するうえで特に重要である．DMI では，リモートページングを繰り返すうちに各プロセスの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のメモリプールに対するページアウトが行われる．

第 4 に，DMI のグローバルアドレス空間のコヒーレンシは，非同期的なプロセス（ノード）の参加/脱退を越えて維持されるよう設計されており，1 つの並列計算の途中で自由にプロセスを参加/脱退させ，並列計算を再構成させることができる．

第 5 に，DMI は C 言語の静的ライブラリとして実装されており，第 **9** 章で述べる一部のカーネル拡

張機能をのぞいては，コンパイラや OS には一切手を加えていないため移植性が高い．DMI の処理系は約 27000 行からなる C 言語で実装されており，83 個の API を提供している．次節以降で論じるように，具体的な API としては，メモリ確保/解放および read/write のための基本的な API の他に，排他制御のための API，ユーザ定義の read-modify-write 命令を作り出す API，非同期な read/write のための API，グローバル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，非定型なグラフ計算における無駄のない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，多様な並列科学技術計算の性能を明示的かつ強力に最適化することができる．第 **6** 章で述べるように，合計 13000 行以上の DMI のプログラムに関して動作検証と性能評価を行っている．

3.2 グローバルアドレス空間の確保/解放と read/write

DMI では，直観的なメモリコンシステンシモデルを採用しつつも，多様なメモリアクセス特性に対して，プログラマが明示的で強力な最適化を見通しよく適用できるような API を設計する．以下で述べる API の設計の根拠については **3.2.6** 節で詳しく述べるが，DMI の API は，以下の 3 点を強く意識して設計されている：

- 再構成をともないうる多様なメモリアクセス特性に対して，**内部的に起きる通信を明示的に簡単に制御する**ことができる．
- 多様なメモリアクセス特性に対して，内部的に起きる通信が無駄に細分化されることがないよう，**通信を明示的に集約させる**ことができる．
- プログラマから見て**内部的にどのような通信が起きるのかがわかりやすく**，最適化の見通しが立てやすい．

3.2.1 グローバルアドレス空間の確保/解放

図 3.1 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グローバル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，（malloc() 関数/realloc() 関数/free() 関数などが内部的に呼び出す）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．一方で，グローバルアドレス空間は 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．

DMI では，CRL[118] や HIVE[23] などの region-based な分散共有メモリ処理系 [112] と同様に，ユーザプログラムの振る舞いに合致した任意のコヒーレンシ粒度 [148, 16, 165] を指定してグローバルアドレス空間を確保することができる．DMI では，コヒーレ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 $page\_num$ 個から構成されるグローバルアドレス空間を確保できる．これにより，プログラマは，任意のページサイズに基づくブロックサイクリックなデータ分散を自然に指示することができる．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実行したい場合には，各行列ブロックのサイズをページサイズに指定してグローバルアドレス空間を割り当てればよい．このように，**DMI ではコヒーレンシ粒度を明示的に調節することで内部的に発生する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る**ことができる．これにより，OS のメモリ保護機構を利用する page-based な分散共有メモリ処理系 [135, 136, 118, 15, 79] ではコヒーレンシ粒度が OS のページサイズ（多くの場合 4 KB）の整数倍に制限されてしまうのに対して，DMI では，ページフォルトの回数を大幅に抑制することができ，通信が無駄に細分化されることがないため，オーバヘッドの少ない効率的な通信を実現できる．再構成可能な並列計算を記述する場合には，実行されるスレッド数を事前に予測することは難しいため，どのようなスレッド数で実行されてもフォルスシェアリングによる深刻な性能低下が起きないようにするために，ある程度小さいページサイズを選択することが重要である．なお，後述するように，DMI では選択的キャッシュ read/write と呼ばれる仕組みによって，各ページの所属プロセスを動的に自由に変更できるため，ブロックサイクリックなデータ分散だけではなく，より柔軟で動的なデータ分散も簡単に実現することができる．

3.2.2 グローバルアドレス空間に対する read/write とコンシステンシモデル

グローバルアドレス空間に対して read/write するためには，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，グローバル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* バイトをグローバル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() 関数に関する **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．

なお，DMI_read() 関数/DMI_write() 関数を複数のページにまたがって呼び出せるという機能はプログラマビリティ上重要である．たとえば，UPC では memget() 関数/memput() 関数を利用することで，連続したグローバルアドレス領域のデータをローカルアドレス空間に read/write することができるが，UPC の memget() 関数/memput() 関数は，複数のページ（UPC の用語ではブロック）にまたがって read/write することは許されない．よって，プログラマは，各グローバルアドレス空間のページサイズをつねに意識し，複数のページにまたがる read/write を行う場合には，各ページごとに独立に memget() 関数/memput() 関数を記述するよう注意する必要があり，プログラミングが非常に面倒になる．

DMI が，Sequential Consistency という，もっとも強いコンシステンシモデルを採用しているのは

以下の理由による．従来の分散共有メモリ処理系では，内部的に起こすことのできる通信の自由度を高めて性能を向上させるために，Eager Release Consistency の Munin[101]，Lazy Release Consistency の TreadMarks，Entry Consistency の Midway[139] など，コンシステンシモデルの緩和が積極的に試されてきた [204]．しかし，コンシステンシモデルの緩和はプログラマビリティとトレードオフの関係にあり，緩和型のコンシステンシモデルでは，read/write したときにどの値が read/write されるのかの動作をつかみにくく，グローバルアドレス空間モデルとしてのプログラマビリティが損なわれる [104, 168]．そこで DMI では，プログラミングのわかりやすさを優先させるために，コンシステンシモデルとしては直観的に理解しやすい Sequential Consistency を採用し，そのかわりに，ページサイズを任意に指定したり read/write を非同期化したりできるような機能など（後述）を提供することで，プログラマがそれらをうまく組み合わせれば，コンシステンシモデルの強さに由来する性能劣化を明示的に緩和できるようにしている．このような設計により，DMI では，初期的には直観的に理解しやすい Sequential Consistency のもとでプログラムを記述し，性能改善が必要な部分だけインクリメンタルに最適化していくような，見通しのよいプログラミングが可能になっている．

3.2.3 選択的キャッシュ read/write

3.2.3.1 基本アイディア

グローバルアドレス空間に対するアクセス性能を高めるためには，ページフォルトの回数を削減することが重要である．そのためには，ページの分散配置を，ユーザプログラムの実際のアクセスローカリティに合致させることが重要である．そこで DMI では，ユーザプログラムの実際のアクセスローカリティに対応してページの分散配置を最適化するための手段として**選択的キャッシュ read/write** を提供している．プログラマは選択的キャッシュ read/write を利用することで，各 read/write に関して，その read/write がページフォルトを起こした場合に，内部的なページの分散配置をどのように変更するべきかを明示的に制御できる．

説明を具体化させるため，まず，DMI におけるキャッシュ管理と read/write フォルトの関係について整理する．DMI では，各ページごとにオーナーと呼ばれるプロセスが存在し，そのページの最新状態とコヒーレンシの管理を担当している．すべての read/write フォルトはいったんオーナーに通知され，オーナーが適切な処理を行うことによって解決される．オーナーは固定されているわけではなく動的に変化しうる．DMI において read フォルトが発生する条件は，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る．write フォルトが発生する条件は，そのプロセスがオーナーでないか，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，あるプロセスが read/write フォルトが発生した場合には，オーナーにページの最新状態を転送したり，あるいはオーナーからページの最新状態を転送してもらうことによって read/write フォルトが解決されることになるが，このとき選択的キャッシュ read/write を利用することで，(1) どのようにページを転送するのか，(2) 転送されたページをどのようにキャッシュするのか，(3) オーナーをどのように移動させるのか，を明示的に制御することができる．

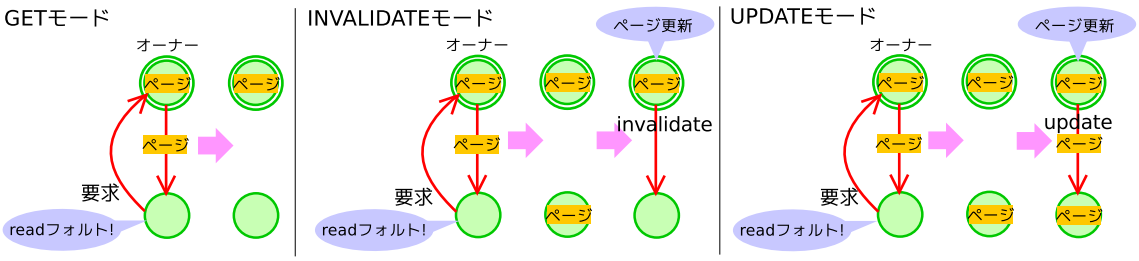

図 3.2 選択的キャッシュ read の挙動.

3.2.3.2 選択的キャッシュ read

read に関しては，DMI_read(int64_t *addr*, int64_t *size*, void **buf*, int *mode*) 関数の引数 *mode* に，この DMI_read() 関数が read フォルトを引き起こした場合の挙動として以下の 3 とおりを指定できる（図 3.2）：

INVALIDATE モード オーナー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（INVALIDATE 型キャッシュ）．このキャッシュは，そのページが次に更新された際にオーナーによって無効化される．

UPDATE モード オーナー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（UPDATE 型キャッシュ）．このキャッシュは，ページが更新されるたびにオーナーによってその更新が反映され，常に最新状態に保たれる．

GET モード 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ーから取得する．メモリプールには何もキャッシュしない．PGAS 処理系における get 操作に相当する．

上記の説明のように，DMI では，write によってページが更新される場合には，INVALIDATE 型キャッシュの無効化と UPDATE 型キャッシュの更新をオーナーが行うことで，キャッシュのコヒーレンシを維持する．よって，キャッシュを行うことで read フォルトを回避できるようにはなるものの，キャッシュの数が増えれば write にともなうオーバヘッドが増大する．したがって，実践的な使い分けとしては，各ページに対するアクセス特性に応じて，(1) 近い将来にそのページを read しかつ write の性能が read の性能よりも重要な場合には INVALIDATE モードを，(2) 近い将来にそのページを read しかつ read の性能が write の性能よりも重要な場合にはには UPDATE モードを，(3) 近い将来にそのページを read しないか，またはページのごく一部のみを read したいのならば GET モードを指定するのがよい．

3.2.3.3 選択的キャッシュ write

write に関しては，DMI_write(int64_t *addr*, int64_t *size*, void **buf*, int *mode*) 関数の引数 *mode* に，この DMI_write() 関数が write フォルトを引き起こした場合の挙動として次の 2 とおりを指定できる（図 3.3）：

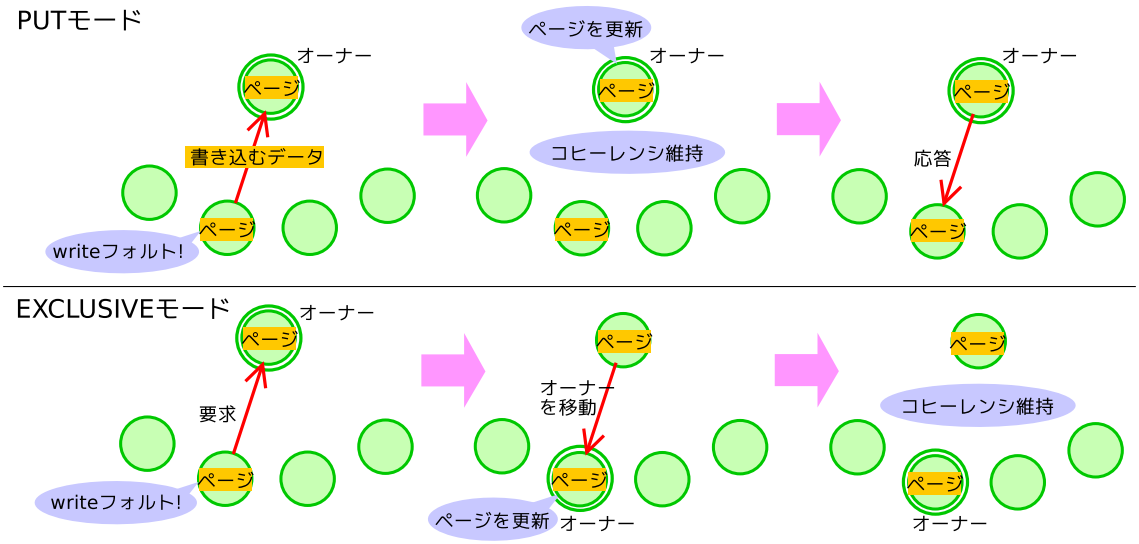

図 3.3 選択的キャッシュ write の挙動.

EXCLUSIVE モード まず現在のオーナーからオーナー権を奪って自分がオーナーになったあとで，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ーを移動する．

PUT モード write すべきデータをオーナーに対して送信し，オーナーに write してもらう．つまり，オーナーを移動しない．PGAS 処理系における put 操作に相当する．

オーナーの移動はページの最新状態の転送をともなう可能性があるうえに，オーナーが頻繁に移動してしまうと，read/write フォルトが発生したときに DMI がオーナーの所在を追跡するためのオーバヘッドが増大する．よって，むやみにオーナーを移動するのは性能上望ましくない．

ここで，EXCLUSIVE モードと PUT モードのどちらが効率的かを比較する．m 個のプロセスが合計 a 回の write を行うような処理を考え，簡単のため read はいっさい行われないものとして，a 回の write すべてを EXCLUSIVE モードで行う場合と，a 回の write すべてを PUT モードで行う場合を比較する．また，実行開始時のオーナーはプロセス v であるとし，i $(1 \leq i \leq a)$ 回目の write を行うプロセスを $f(i)$ で表す．

まず，a 回すべての write が PUT モードで行われる場合を考える．この場合，実行開始から終了までオーナーはプロセス v に固定されるため，実行開始時にすべてのプロセスが正しいオーナー v を知っていると仮定すれば，オーナー追跡グラフの形状はつねにオーナー v を根とする flat tree になる．よって，オーナー以外の各プロセスで生じる write はつねに write フォルトを引き起こすが，それらの write フォルトはつねに 1 ホップでオーナーに通知できる．したがって，write フォルトをオーナーに通知するために飛び交うメッセージ数は，k 回の write のうち $f(i) \neq v$ なる i の個数を k_1 とすれば，$O(k_1)$ になる．次に，k 回すべての write が EXCLUSIVE モードで行われる場合を考える．この場合には，

実行開始から終了までオーナーは動的に移動し続ける．よって，各プロセスで生じた write は，その時点でそのプロセスがオーナーでないかぎり write フォルトを引き起こす．一般に，動的に移動するオーナーの所在を特定する方法としては probable owner[111] と呼ばれる手法が代表的であり，probable owner では，任意のプロセスからオーナーまで平均 $O(\log m)$ ホップでメッセージを届けることができる [111, 138] [*1] ．したがって，write フォルトをオーナーに通知するために飛び交うメッセージ数は，write フォルトが発生するのは連続する 2 回の write を行うプロセスが異なる場合であることに注意すると，$f(i) \neq f(i+1)$ なる i の個数を k_2 とすれば，$O(k_2 \log m)$ となる．

このように，単純な計算モデルのもとでのメッセージ数で比較するならば，$k_1 \leq k_2 \log m$ であれば PUT モードの方が有利であり，$k_1 > k_2 \log m$ であれば EXCLUSIVE モードの方が有利といえるが，k_1 と $k_2 \log m$ の大小関係はユーザプログラムの write ローカリティによって決まる．したがって，実践的な使い分けとしては，各ページに対する write ローカリティに応じて，(1) 特定の 1 つのプロセスだけがそのページに対する write ローカリティを持つ場合には，そのプロセスが write するときに EXCLUSIVE モードを，その他のプロセスが write するときに PUT モードを指定するのがよく，(2) それ以外の場合にはすべてのプロセスが PUT モードを指定するのがよい．

3.2.3.4 選択的キャッシュ read/write の新規性

第 1 に，選択的キャッシュ read/write は，invalidate 型と update 型のハイブリッド型のキャッシュプロトコルを各 read/write の粒度で実現できると同時に get/put 操作も実現可能であり，**アクセスローカリティを非常に柔軟に最適化できる**という点で新規的である．既存研究を眺めると，Co-Array Fortran[149, 45, 185, 46] や UPC[43, 62, 48, 46]，Titanium[188, 171, 77, 50, 172]，X10[41]，Chapel[37, 26] などの多くの PGAS 処理系では，get/put 操作がサポートされているだけであり，データのキャッシュやデータ分散の動的な変更はサポートされていない．一方で，Treadmarks[15] や JIAJIA[79]，DSM-Threads[135, 136, 160]，SMS[204] などの多くの分散共有メモリ処理系では，データのキャッシュやページのオーナー権の変更はサポートされているが，そのキャッシュを invalidate 型で管理するか update 型で管理するかはプロトコル単位で固定されている [136, 112, 118, 15, 79]．さらに，OS のメモリ保護機構を利用してコヒーレンシ管理を実現する page-based な分散共有メモリ処理系 [135, 136, 118, 15, 79] では，そもそも原理的に get 操作を実現できない．なぜなら，これらの処理系では，（OS の意味での）read フォルトが発生してシグナルハンドラで SIGSEGV をフックしたあと，シグナルハンドラから返る前にはそのページのアクセス権限を read 可能に設定する必要があるためである．read 可能に設定しないかぎり，SIGSEGV が永久に発生し続けてしまう．ところが，ここで read 可能に設定するということは，いまわれようとしている read だけではなく，以降で行われるそのページに対するすべての read を許可するということを意味し，これはつまり，ページをキャッシュするということにほかならない．このように，OS のメモリ保護機構を利用するかぎり，いま起きている 1 回の read フォルトだけを解決する方法がないため，ページをキャッシュせざるをえず，get 操作を実現することはできない．

[*1] **4.2.1.1** 節で述べるように，DMI でも probable owner を用いているが，実装上の都合で平均ホップ数を $O(\log m)$ までには抑えられていない．

第 2 に，選択的キャッシュ read/write は，再構成をともなう並列計算に関して，動的に変化しうるアクセスローカリティに適応してデータ分散を動的に適応させるための手段として新規的である．多くの PGAS 処理系や分散共有メモリ処理系では，グローバルアドレス空間を確保する時点でブロックサイクリックなどのデータ分散を決定する必要があり，そのデータ分散を動的に変更することはできない．しかし，再構成にともなってスレッド数が増減すると，各スレッドから見たアクセスローカリティが変化するため，何らかの方法でデータ分散をアクセスローカリティに動的に適応させるための手段が必要になる．これを行うためのもっとも単純なアプローチは，再構成時に必要なデータの再分散をプログラマに明示的に記述させる方法である．しかし，とくに非定型な並列計算の場合には，データの再分散を記述するのは非常に煩雑である．そのうえ，データの再分散を行うためには，一般に，その時点ですべてのスレッドを同期させる必要があることをふまえると [69, 126, 173, 128, 103]，データの再分散を記述させるアプローチでは，DMI が目指しているような，非同期的にプロセスが自由に参加/脱退するような並列計算には適用できない．これに対して，選択的キャッシュ read/write では，**各 read/write に対してその read/write が持つアクセスローカリティを指示するだけで，実際のアクセスローカリティにしたがってデータ分散が動的に適応される**．煩雑なデータの再分散を記述する必要もなければ，スレッドを何らか同期させる必要もない．具体的には，write に対して EXCLUSIVE モードを指示しておけば，その write を発行したスレッドがその時点でどのプロセス上で実行されていようとも，ページのオーナー権がそのスレッドが属するプロセスのもとに移動してきて，それ以降で行われる write は write フォルトを起こすことなく完了できるようになる．同様に，read に対して INVALIDATE モードを指示しておけば，その時点でそのスレッドがどのプロセス上で実行されていようとも，read したデータが INVALIDATE 型キャッシュとしてキャッシュされ，それ以降で行われる read は read フォルトを起こすことなく完了できるようになる．このように，選択的キャッシュ read/write は，動的に変化しうるアクセスローカリティに適応してデータ分散を動的に適応させるための非常に明快な手段であるといえる．

3.2.4 非同期 read/write

DMI_mmap() 関数/DMI_munmap() 関数/DMI_read() 関数/DMI_write() 関数など，内部的に通信をともないうる多くの API を非同期に実行することができる．とくに，非同期な DMI_read() 関数/DMI_write() 関数を利用することで，プリフェッチやポストストアを実現したり，DMI_read() 関数/DMI_write() 関数が保証している Sequential Consistency を自由に緩和させることができる．

3.2.5 離散アクセスのグルーピング

とくに非定型な並列計算においては，コヒーレンシ粒度（ページサイズ）をどのように設定したとしても，多数のページにわたって離散的なメモリアクセスを行う必要が生じる．その場合，各メモリアクセスごとに逐一通信を発生させていたのでは性能が著しく劣化するため，通信をできるかぎり集約して発行するための仕組みが重要となる．そこで DMI では，ページサイズに関係なく，必要なデータだけを必要最小限の通信回数でまとめて read/write するための API として，**離散アクセスのグルーピング**を提供している．

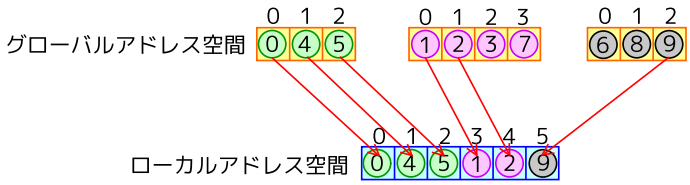

図 3.4 離散アクセスのグルーピング.

擬似的な API として表現すると，離散アクセスのグルーピングは以下の 4 つの API で実現される：

g=**group_init**(A,S) 離散アクセスの対象となるグローバルアドレス領域たちを定義する．ここで，グローバルアドレス領域たちの個数を n とおくと，A はサイズ n のグローバルアドレスの順序集合，S はサイズ n のバイト数の順序集合であり，各 i（$0 \leq i < n$）に対して，i 番目のアドレス領域の先頭アドレスを $A[i]$，i 番目のアドレス領域のバイト数を $S[i]$ として指定する．各アドレス領域が複数のページにまたがっていても問題ない．結果として，これらのグローバルアドレス領域たちを表すハンドラ g が返る．すなわち，ハンドラ g はグローバルアドレス空間上のアドレス領域：$[A[0],A[0]+S[0]) \cup [A[1],A[1]+S[1]) \cup \ldots \cup [A[n-1],A[n-1]+S[n-1])$ を定義している．

group_read(g, *buf*, O, *mode*) ハンドラ g に定義されているグローバルアドレス領域たちを read し，その結果をローカルアドレス空間上のバッファ *buf* に格納する．ここで，O はサイズ n のローカルアドレスの順序集合であり，各 i（$0 \leq i < n$）に対して，i 番目のアドレス領域を read した結果としての $S[i]$ バイトが，$buf[O[i]]$ からはじまる $S[i]$ バイトとして格納される．*mode* には，INVALIDATE モード，UPDATE モード，GET モードを指定できる．

group_write(g, *buf*, O, *mode*) ローカルアドレス空間上のバッファ *buf* から，ハンドラ g に定義されているグローバルアドレス領域たちに対して write する．ここで，O はサイズ n のローカルアドレスの順序集合であり，各 i（$0 \leq i < n$）に対して，$buf[O[i]]$ からはじまる $S[i]$ バイトが，i 番目のアドレス領域の値として write される．*mode* には，EXCLUSIVE モード，PUT モードを指定できる．

group_destroy(g) ハンドラ g を破棄する．

要するに，離散的なアドレス領域を group_init() 関数で定義したあと，ハンドラを介して group_read() 関数/group_write() 関数を発行することで離散的な read/write を実現する．重要なことは，group_init() 関数が発行された段階で n 個のグローバルアドレス領域たちがページごとに整理されており，group_read() 関数/group_write() 関数は内部的には各ページに対する通信を集約して行われるという点である．たとえば，図 2.1 のグラフを，図 3.4 に示すような 3 つのグローバルアドレス空間として表現しているとする．このとき，プロセッサ 0 上のスレッドが節点 0，節点 4，節点 5，節点 1，節点 2，節点 9 の値を read するためには，グローバルアドレスたち&$g0$[0]，&$g0$[1]，&$g0$[2]，&$g1$[0]，&$g1$[1]，&$g2$[2] を group_read() することになるが，DMI はこれら 6 個のグローバルアド

レス領域たちに対して個別に read が発行されるわけではなく，グローバルアドレス領域たちがページごとに集約され，&*g0*[0] と&*g0*[1] と&*g0*[2] に対する read，&*g1*[0] と&*g1*[1] に対する read，&*g2*[2] に対する read の 3 個の read だけが発行される．このように，x 個のページにまたがる n 個のグローバルアドレス領域たちを group_read() 関数/group_write() 関数する場合には，n の値に関係なく x 個の通信しか発生せず，**必要なデータだけを必要最小限の通信回数で read/write する**ことができる．通常の DMI_read() 関数/DMI_write() 関数もこの離散アクセスのグルーピングの特別な場合であり，離散アクセスのグルーピングは，グローバルアドレス空間に対するもっとも汎用的な read/write の手段を提供しているといえる．

3.2.6 議論：API の設計思想

本説では，以上で述べた DMI の API の設計思想について議論する．**3.2** 節の冒頭で述べたように，DMI の API は，一貫して以下の 3 点を意識して設計している：

設計指針 1 再構成をともないうる多様なメモリアクセス特性に対して，**内部的に起きる通信を明示的に簡単に制御する**ことができる．

設計指針 2 多様なメモリアクセス特性に対して，内部的に起きる通信が無駄に細分化されることがないよう，**通信を明示的に集約させる**ことができる．

設計指針 3 プログラマから見て**内部的にどのような通信が起きるのかがわかりやすく**，最適化の見通しが立てやすい．

設計指針 1 に関しては，DMI_mmap() 関数において適切なコヒーレンシ粒度を設定したうえで，各 read/write に対して適切な選択的キャッシュ read/write を指示することによって，内部的に起きる通信を明示的に簡単に制御することができる．

設計指針 2 に関しては，離散的なアクセスに対しては離散アクセスのグルーピングによって通信を明示的に集約させることができる．連続的なアクセスに対しては，コヒーレンシ粒度を必要十分に大きくとることと，DMI_read() 関数/DMI_write() 関数をできるかぎり大きなアドレス領域に対して発行することによって通信を明示的に集約させることができる．

設計指針 3 に関しては，(1) DMI の API を記述した箇所でしか通信が発生しないという点，(2) 選択的キャッシュ read/write の仕組みさえ理解していれば，内部的にどのような通信が起きるかが明確であるという点において，DMI では，意図しない通信が内部的に発生することを回避するのが容易で，最適化の見通しを立てやすい．第 1 に，DMI が Sequential Consistency を採用しており，**3.2.3.1** 節で述べたように Single Writer/Multiple Reader 型のプロトコルを採用している理由は，内部的な通信をわかりやすくするためである．一般には，複数のプロセスによる read/write の独立性を高めるためには，より緩和されたコンシステンシモデルのもとで Multiple Writer/Multiple Reader 型のプロトコルを採用するのが望ましいが，Multiple Writer/Multiple Reader 型のプロトコルでは，write されたデータの差分や更新順序などを管理する必要があるため，プロトコル全体が煩雑化し，それにともなうオーバヘッドも増大する [79]．そして何より，内部的にどのような通信が発生しているのかを非常に把握しにくくなるという問題がある．第 2 に，**2.1.4.3** 節で述べたように，通常の変数参照や配列参照に

よってグローバルアドレス空間にアクセス可能な分散共有メモリ処理系や，高抽象度なシンタックスを多数備えた X10 や Chapel などの高生産並列言語においては，たしかにユーザプログラムの記述自体は非常に簡潔ではあるが，ユーザプログラム中のどの場所でどのような通信が発生するかが非常にわかりにくく，不本意な性能劣化を招きやすく最適化も難しいという問題がある [6]．これに対して，DMI では，内部的に発生する通信が明確かつ制御可能であり，「DMI の API を呼び出す回数を最小化する」というわかりやすい目標に沿ってユーザプログラムを記述するだけで，意図しない性能劣化を防げる．

次に，DMI の API 設計の欠点について考える．第 1 の欠点は，プログラミングの複雑さである．DMI では，グローバルアドレス空間上のデータを直接操作することはできず，API 呼び出しによっていったんローカルアドレス空間にデータを読み込んだうえで操作する必要がある．したがって，DMI のプログラムは必然的に，(1) 必要なデータをグローバルアドレス空間からローカルアドレス空間に読み込んだあと，(2) ローカルアドレス空間上で計算を進め，(3) 計算の結果として必要なデータをローカルアドレス空間からグローバルアドレス空間に書き出す，という形態となる．これは，DMI がグローバルビューのグローバルアドレス空間を提供しているとはいえ，プログラム全体を真のグローバルビューでは記述することはできないことを意味している．これは大きな欠点といえるが，しかし一方で，いま述べたように，真のグローバルビューによって直接グローバルアドレス空間上のデータを自由に触らせないという設計自体が，高性能で最適化の施しやすいプログラムを記述させるための最重要な根拠になっているため，真のグローバルビューによる記述を単純に許可するわけにはいかない．本研究では，DMI のレイヤでは性能を優先してこのプログラミングの複雑さは容認することにし，そのかわり，**11.2** 節で述べるように，トランスレータによって高性能な DMI のプログラムに変換できるような，真のグローバルビューで記述できる高生産言語を将来的に設計することを計画している．

第 2 の欠点は，DMI_read() 関数/DMI_write() 関数にともなうオーバヘッドの大きさである．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 処理系では，ページフォルトが発生しない場合には通常のメモリアクセスと同じオーバヘッドでグローバルアドレス空間にアクセスできるのに対して，DMI では，ページフォルトが発生しない場合でも，ユーザレベルでの検査が必要なうえ，メモリプールからローカルアドレス空間へのメモリコピーが必要になる．しかし，前述のように，DMI の API 呼び出しを最小化することが DMI におけるプログラム開発の基本であるため，このオーバヘッドは性能上の問題にはなりにくいと考えられる．

3.3 非同期的なプロセスの参加/脱退に対応したコヒーレンシプロトコル

プロセスを任意のタイミングで非同期的に参加/脱退させようとする場合，グローバルアドレス空間のコヒーレンシを維持するうえで難しい問題がいくつか生じる．本節では，これらの問題のうち，オーナーの所在をどのように特定するかという問題に焦点を絞り，プロセスを非同期的に参加/脱退させることの難しさについて議論したうえで，実装する必要のあるプロトコルを整理する．実際のプロトコルの実装については **4.2** 節で述べる．

read/write フォルトが発生した場合にはオーナーに通知する必要があるため，任意のプロセスは任意の時点でオーナーの所在を特定できる必要がある．ところが，選択的キャッシュ read/write では，オーナーの所在が動的に変化してしまう．また，プロセスが脱退する際には，脱退前に，そのプロセスがオーナーであるようなすべてのページを他のプロセスに追い出す必要があり，ここでもオーナーの移動が起きる．さらに，各プロセスのメモリプールの使用量が閾値を超えて，ページ置換の対象として，そのプロセスがオーナーであるようなページが選択された場合には，やはりオーナーの移動が起きる．このように，DMI ではオーナーの所在が動的に変化するため，各プロセスが read/write フォルトを起こした場合などに，どのようにオーナーの所在を特定するかは自明な問題ではない．

この問題に対する解決策としては，各ページに対して，オーナーとは別にホームプロセスと呼ばれるプロセスを固定しておく方法がある [15, 154]．この方法では，オーナーの所在が変化するたびにホームプロセスにそれを通知することで，ホームプロセスがつねに正しいオーナーの所在を把握できるようにしておき，各プロセスがオーナーの所在を見失った場合には，（固定されている）ホームプロセスに問い合わせることでオーナーの所在を特定する．しかし，当然ながら，DMI のようにプロセスが動的に参加/脱退する状況では，ホームプロセスのように固定的なプロセスを設置することはできない．参加/脱退にともなってホームプロセスの移動を許そうとすれば，今度はホームプロセスの所在をどのように特定するかという問題が起きてしまう．

さて，ここで思考実験として，プロセスの非同期的な参加/脱退は許さず，プロセスの参加/脱退はつねに同期的にしか起きないという状況を考えてみる．つまり，プロセスの参加/脱退を処理するための何らかの同期的な API が存在していて，実行中のプロセスすべてがその同期的な API を呼び出した時点でのみ，プロセスの参加/脱退が処理されるような状況を考える．この場合には，上記で述べたようなオーナーの移動にともなう問題も含め，参加/脱退にともなうさまざまな問題を簡単に解決できることに注意したい．なぜなら，参加/脱退にともなうさまざまな状況の変化を，すべてのプロセスに対して同期的に反映させられるからである．たとえば，参加/脱退にともなってホームプロセスが移動する場合でも，ホームプロセスが移動したという情報をすべてのプロセスに同期的に反映させることができるため，実行中のプロセスたちが同期的な API から返った時点では，ホームプロセスの所在を特定できないという問題は起きえない．

以上の観察からわかるように，問題の難しさは，DMI がプロセスの非同期的な参加/脱退を許そうとしている点にあり，著者の知るかぎり，プロセスの非同期的な参加/脱退に対応可能なグローバルアドレス空間を実現した研究は DMI がはじめてである．ここで，**参加/脱退が非同期的である**とは，実行中のプロセスの同期を必要とすることなくプロセスの参加/脱退を実現できるという意味である．いい換えると，**実行中のプロセスたちがそれぞれ独立に，グローバルアドレス空間に対する選択的キャッシュ read/write，あるいはメモリプールのなかのページ置換を行っていたとしても，それと並行してプロセスの参加/脱退を実現できる**という意味である．これを実現するためには，以下の 4 種類の操作に対するプロトコルを実装する必要がある：

- 動的に移動しうる各ページのオーナーを特定するためのプロトコル（**4.2.1.1** 節）．
- 選択的キャッシュ read/write を行うとき，各ページのコヒーレンシをどのように維持するかを規

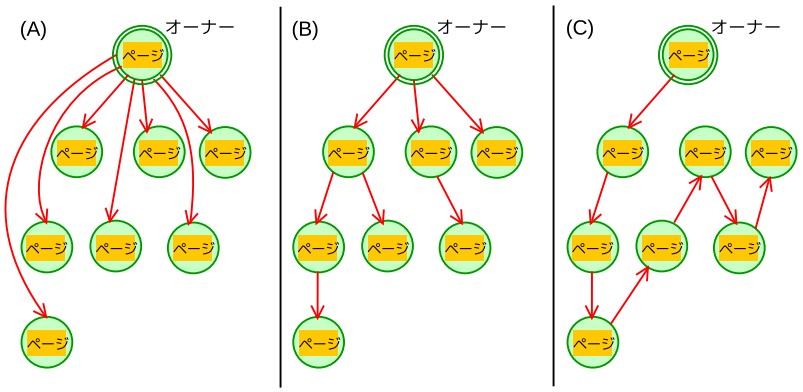

図 3.5 ページ転送の動的負荷分散の基本アイディア．（A）read フォルトに対してオーナーがページを逐次的に転送する場合，（B）ページ転送を動的に木構造化させる場合，（C）ページ転送を数珠つなぎで行う場合．

定するプロトコル（**4.2.4.2** 節，**4.2.4.3** 節）．

- ページ置換およびプロセスの脱退前にはメモリプールのなかのページを追い出す必要があるが，ページを追い出すときのコヒーレンシをどのように維持するかを規定するプロトコル（**4.2.4.4** 節）．
- プロセスが参加/脱退を行うときに，何をどのような順序で行うかを規定するプロトコル（**4.2.5** 節）．

また，これらのプロトコルの実装は非常に複雑であるため，**4.2.3** 節において，このような複雑なプロトコルを見通しよく正しく実装するための方針について議論する．

3.4 データ転送の動的負荷分散

3.4.1 基本アイディア

あるプロセス i で read フォルトが発生した場合，read フォルトがオーナーに通知され，オーナーがプロセス i に対してページの最新状態を転送する．よって，Broadcast のように，多数のプロセスがほぼ同時に同一のページを read する場合，ページ転送の負荷がオーナーへ一極集中し，これが性能上のボトルネックになりうる．これを解決するため，DMI では，ページ転送の負荷が特定のオーナーに集中した場合，そのページをキャッシュしているプロセスを利用して，ページ転送の負荷を動的に負荷分散させる機能を導入している．

たとえば，**6.5.5** 節で述べる横ブロック分割による行列行列積 $AB = C$ のアルゴリズムにおいては，行列 B 全体をすべてのスレッドに Broadcast する必要がある．ここで，プロセス数を m，各プロセス内のスレッド数を t，行列 B のサイズを s，ネットワークバンド幅を w とおき，各スレッドが同時に，グローバルアドレス空間上の行列 B を INVALIDATE モードで read する状況を考える．このとき，合計 mt 個の read フォルトが発生してオーナーに通知されるが，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する構成となっているため，オーナーは mt 回のページ転送を行うわけではなく，合計 m 回のページ転送しか行われない．しかし，これらの m 回のページ転送はオーナーにおいて逐次的に行われるため，合計 sm/w の時間を要する（図 3.5（A））．そこで DMI では，各オーナーに対して要求されるページ転送の負荷をつねに監視し，オーナーのページ転送の負荷がある閾値を超えた場合，**本来であればオーナーが行うべきページ転送を，すでにページをキャッシュしているプロセスに依頼することで，ページ転送の負荷を動的に分散させる**（図 3.5（B））．

3.4.2 アルゴリズム

ページ p に関するページ転送を依頼させるプロセスは，以下のアルゴリズムによって選択する：

ルール 1 ページ p をキャッシュしているプロセス（オーナー自身も含む）のうち，「もっとも過去にページ転送を依頼したプロセス」に対してページ転送を依頼する．

ルール 2 ただし，便宜上，ページ p を転送された直後のプロセスは，ページ転送が完了した時点でページ転送を依頼されたものと見なす．

ルール 1 に関して，「もっとも過去にページ転送を依頼したプロセス」を選択する意味は，「もっとも過去にページ転送を依頼したプロセス」は，いまページ p をキャッシュしているプロセスのなかでは，ページ p のページ転送を前回依頼された時刻がもっとも過去であるため，いまページ転送を依頼したときに，そのページ転送を開始してくれるまでの時間がもっとも短いと期待できるからである．ルール 2 に関して，ページ p を転送された直後のプロセスをその時点でページ転送を依頼されたと見なす理由は次のとおりである．仮にそのように見なさなかったとすると，ページ p を転送された直後のプロセスは，まだ一度もページ転送を依頼されていないことになるため，その時点でページ p をキャッシュしているプロセスのなかでは，「もっとも過去にページ転送を依頼されたプロセス」としてとり扱われることになる（実際にはまだ一度も依頼されていないが）．その結果，オーナーが「もっとも過去にページ転送を依頼されたプロセス」にページ転送を依頼しようとすると，つねに，もっとも直近にページを転送されたプロセスにページ転送を依頼することになり，図 3.5（C）に示すように，全体のページ転送が数珠つなぎのトポロジで行われることになる．これでは，全体のページ転送に要する時間は合計 sm/w となり，オーナーが逐次的に転送する場合と変わらなくなってしまう．

このアルゴリズムは，具体的には図 3.6 に示すように動作する．初期的には，オーナーのみがページの最新状態を保持している．ここで，プロセス 1，2，...，15 から，ほぼ同時にしかし厳密にはこの順序で read フォルトが通知されたとし，かつ，ページは十分に大きく，これら 6 回のページ転送のすべてにおいてオーナーにおけるページ転送の負荷が閾値を超えると仮定する．第 1 に，オーナーがプロセス 1 からの read 要求を処理する場合，ページのキャッシュを持っているプロセスはオーナーのみなので，オーナーが自らプロセス 1 に対してページを転送する（図 3.6（A））．第 2 に，オーナーがプロセス 2 からの read 要求を処理する場合，この時点でページのキャッシュを持っているプロセスはオーナーとプロセス 1 の 2 つであるが，ルール 2 により「もっとも過去にページ転送を依頼されたプロセス」はオーナーであると計算されるので，オーナーが自らプロセス 2 に対してページを転送する（図 3.6（B））．第 3 に，オーナーがプロセス 3 からの read 要求を処理する場合，この時点でページのキャッシュを持って

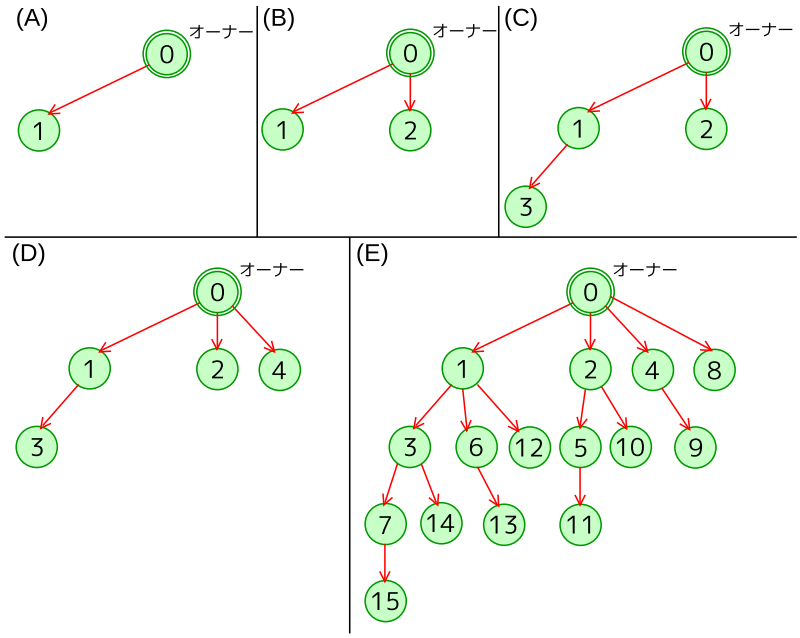

図 3.6 ページ転送の動的負荷分散のアルゴリズムの動作例.

いるプロセスはオーナーとプロセス 1 とプロセス 2 の 3 つであり，「もっとも過去にページ転送を依頼されたプロセス」はプロセス 1 であると計算されるので，オーナーは，プロセス 3 に対するページ転送をプロセス 2 に依頼する（図 3.6（C））．第 4 に，オーナーがプロセス 4 からの read 要求を処理する場合，この時点でページのキャッシュを持っているプロセスはオーナーとプロセス 1 とプロセス 2 とプロセス 3 の 4 つであり，「もっとも過去にページ転送を依頼されたプロセス」はオーナーであると計算されるので，オーナーが自らプロセス 4 に対してページを転送する（図 3.6（D））．同様にしてこのアルゴリズムを繰り返すと，全体のページ転送のトポロジは図 3.6（E）に示すような**二項木を形成する**．二項木による Broadcast は，MPI における Broadcast の効率的な実装の 1 つとして知られており [17]，全体のページ転送に要する時間は合計 $s \log m/w$ である．

3.4.3 議論：利点と欠点

Broadcast などの集合通信におけるデータ転送のトポロジの最適化は，主に MPI を対象として広く研究されている [17, 195, 162]．しかし，プログラマが集合通信関数を記述できるのは，すべてのスレッドの時系列的な挙動が静的に判明していて，それらを SPMD 型のプログラムとして記述できるような同期的なアプリケーションにかぎられている．すなわち，プログラマに集合通信関数を記述させることによって集合通信におけるデータ転送のトポロジを最適化しようとするアプローチは，同期的なアプリケーションには適しているが，非同期的にスレッドが増減を繰り返すような並列計算には適していない．これに対して，DMI におけるページ転送の動的負荷分散では，各スレッドは単に read したいタイミングで read を呼ぶだけでよく，あとは DMI の処理系がページ転送のボトルネックを自動的に検出

し，ページ転送のトポロジを動的に最適化してくれる．とくに，Broadcast のような定型的な集合通信にかぎらず，**非定型に通信が集中するような場合にも効果的に適応できる**．たとえば，ページ p とページ q が同一のオーナーを持つ場合，仮にページ p に対する read 要求はオーナーに集中していなくても，ページ q に対する read 要求が集中してオーナーにおけるページ転送の負荷が閾値を超えているならば，ページ p に対する read 要求も負荷分散の対象となる．

一方で，第 1 の欠点は，プログラマに集合通信関数を記述させることを前提として集合通信を最適化しようとするアプローチと比較すると，効率のよい転送トポロジを形成しにくい点である．ページ転送の動的負荷分散では，「いまから全体としてどのような集合通信が実現されるのか」に関する知識がないうえに，すべてのスレッドの同期を前提にできないため，MPI の集合通信で採用されているような，より効率的なアルゴリズムを採用できない．第 2 の欠点は，ページ転送の依頼の対象になるのはそのページをキャッシュしているプロセスだけであるため，各スレッドは INVALIDATE モードもしくは UPDATE モードで read を発行する必要があり，プログラミング上ややわかりにくい注意が必要になる点である．

3.5 同期

3.5.1 アドレスベースの同期の必要性

並列プログラムを記述するうえでは排他制御などの同期機構は必須である．並列分散プログラミング処理系における同期機構を大きく分類すると，スレッドベースの同期とアドレスベースの同期に分類できる．スレッドベースの同期とは，どのスレッドたちが同期するべきかのスレッド集合を具体的に記述することにより，それらのスレッド集合間で同期が実現されるタイプの同期のことである．たとえば，UPC，Global Arrays，X10 などで提供されているバリア関数や，Co-array Fortran で提供されている，任意のスレッド間で局所的に同期を行うための SYNC_TEAM() 関数などがそれに相当する．一方で，アドレスベースの同期とは，すべてのスレッドで共有されているアドレスを記述することにより，実行時にたまたまその時点で同じアドレスを使用しているスレッド間で同期が実現されるタイプの同期のことである．たとえば，pthread における mutex や条件変数，各種 CPU に実装されている compare-and-swap や fetch-and-store などの read-modify-write[75]，DSM-Threads や SMS における mutex や条件変数がそれに相当する．これらのアドレスベースの同期は，**指定されたアドレスを実行時にたまたま同時に使用しているスレッド間で行われるものであって，具体的にどのスレッド間で同期を行うかに関する記述は必要ない**点に注意する．

さて，DMI が目指しているような，非同期的にプロセスが参加/脱退しうる並列計算においては，全体のスレッド集合が動的に増減するため，ユーザプログラム側でスレッドどうしの挙動を制御するのは容易ではない．よって，何らかのデータ構造を排他制御するにあたって，スレッドベースの同期によってスレッドどうしの挙動を制御するのは記述上難しい場合がある．これに対して，アドレスベースの同期であれば，具体的にどのスレッドどうしが同期するべきかに関する記述は不要であり，単に共有されたアドレスを指定するだけで，その指定されたアドレスを実行時に同時に使用しているスレッド間の同期が実現できる．これらの観察により，DMI にはアドレスベースの同期が必須である．

3.5.2 ユーザ定義の read-modify-write

3.5.2.1 必要性

DMI が mutex や条件変数などのアドレスベースの同期を提供するうえでは，まずもっともプリミティブな API として，**プログラマが read-modify-write を自由に定義できる**API を提供することが必要であることを確認する．

一般に，共有メモリマシン環境ではアドレスベースの同期が提供されているが，もっともプリミティブな命令は read-modify-write である [75]．mutex や条件変数などのより高機能な同期命令や，lock-free や wait-free などの性質を満たすノンブロッキングなデータ構造は，read/write と read-modify-write を組み合わせることで実装される．具体的にどのような read-modify-write が使用できるかは CPU アーキテクチャに依存するが，使用できる read-modify-write が多様で強力であればあるほど，より高機能な同期命令やノンブロッキングなデータ構造を高性能に実装することができる．たとえば，プロセッサ数を N としたとき，read/write だけで排他制御を実現する場合には，1 回の排他制御あたりのリモートキャッシュへのアクセス回数を $O(\log N)$ 未満にするアルゴリズムは知られていない [75] が，read/write と fetch-and-store と compare-and-swap を用いて排他制御を実現する MCS アルゴリズム [133] ではリモートキャッシュへのアクセス回数を $O(1)$ にすることができる．

したがって，プログラマが自由に read-modify-write を定義できる API があれば，多様で高機能な同期命令を高性能に実装するためのよいプリミティブになると考えられる．

3.5.2.2 API と具体例

具体的な API についてやや簡略化して説明する．まず，DMI プログラム中に DMI_rmw(void *$page$, void *in_data, void *out_data, int tag, ...){...}という関数を定義し，この関数内に任意の read-modify-write を記述しておく．ここで，$page$ はこの read-modify-write を適用する対象となるデータ，in_data は read-modify-write への入力データ，out_data は read-modify-write の出力データである．tag は read-modify-write の識別番号であり，DMI_rmw() 関数のなかには，tag の値に応じて複数の read-modify-write を記述することができる．次に，DMI_atomic(int64_t $addr$, int64_t $size$, void *$\_in\_data$, void *$\_out\_data$, int $\_tag$, int $mode$, ...) という関数を呼び出す．ここで，[$addr$, $addr$+$size$) は read-modify-write を適用するグローバルアドレス領域であり，1 個のページに収まっている必要がある．$\_in\_data$ は read-modify-write への入力データ，$\_out\_data$ は read-modify-write の出力データ，$\_tag$ は read-modify-write の識別番号，$mode$ には選択的キャッシュ write における EXCLUSIVE モードまたは PUT モードを指定する．

さて，DMI_rmw() 関数を定義したうえで DMI_atomic() 関数を呼び出すと，DMI_atomic() 関数で指定した入力データ $\_in\_data$ と識別番号 $\_tag$ が，それぞれ DMI_rmw() 関数の引数 in_data と tag に渡される．また，DMI_rmw() 関数の引数 $page$ にはグローバルアドレス $addr$ から $size$ バイト分のデータ本体が渡される．そして，tag の値に応じて，DMI_rmw() 関数がグローバルアドレス空間上のデータ $page$ に対して何らかの read-modify-write を実行したあと，出力データ out_data にデータを格納すると，それが DMI_atomic() 関数の引数 $\_out\_data$ として返る．まとめると，DMI_rmw() 関数には，in_data を入力データとして out_data を出力データとするような，グローバルアドレス空

```
#define FETCH_AND_STORE 1
#define COMPARE_AND_SWAP 2

void fetch_and_store(int64_t addr, int64_t size, void *fetch_data, void *store_data) {
    DMI_atomic(addr, size, store_data, size, fetch_data, size, FETCH_AND_STORE, PUT);
    return;
}

int compare_and_swap(int64_t addr, int64_t size, void *compare_data, void *swap_data) {
    int result;
    void *in_data = malloc(size * 2);
    memcpy(in_data, compare_data, size);
    memcpy(in_data + size, swap_data, size);
    DMI_atomic(addr, size, in_data, size * 2, &result, sizeof(int), COMPARE_AND_SWAP,
PUT);
    free(in_data);
    return result;
}

void DMI_rmw(void *page, int64_t size, void *in_data, int64_t in_size,
                    void *out_data, int64_t out_size, int tag) {
    switch (tag) {
    case FETCH_AND_STORE:
        memcpy(out_data, page, size);
        memcpy(page, in_data, size);
        break;
    case COMPARE_AND_SWAP:
        if (memcmp(page, in_data, size) == 0) {
            memcpy(page, in_data + size, size);
            ((int*)out_data)[0] = 0;
        } else {
            ((int*)out_data)[0] = 1;
        }
        break;
    }
    return;
}
```

図 3.7 ユーザ定義の read-modify-write を使って fetch-and-store と compare-and-swap を実現するプログラム.

間上のデータ *page* に対する任意の read-modify-write を各 *tag* ごとに記述しておき，それを実行するためには DMI_atomic() 関数を呼び出せばよい.

具体例として，fetch-and-store と compare-and-swap の実現例を図 3.7 に示す．また，n 個のスレッドが，各スレッドが保持する整数値 *sub_sum* の総和を Allreduce によって計算するコードを図 3.8 に示す．図 3.8 では，整数値 *sub_sum* の加算と *count* のインクリメントなどの複合的な処理を，read-modify-write を用いてアトミックに実現している．このように，意味的に複数のデータを同一のページに含めておくことで，それらの複数のデータに対するアトミックな操作を実現することができる.

```
#define ALLREDUCE 3

int allreduce(int64_t addr, int sub_sum, int n) {
    int sum, in_data[2];
    int64_t wait_addr = addr + sizeof(int) * 2;
    in_data[0] = sub_sum;
    in_data[1] = n;
    DMI_atomic(addr, sizeof(int) * 2, in_data, sizeof(int) * 2,
                    &sum, sizeof(int), ALLREDUCE, PUT);
    if (sum != WAIT_A_MOMENT) {
        DMI_write(wait_addr, sizeof(int), &sum, PUT);
    } else {
        do { /* (#) */
            DMI_read(wait_addr, sizeof(int), &sum, UPDATE); /* (##) */
        } while (sum == WAIT_A_MOMENT);
    }
    return sum;
}

void DMI_rmw(void *page, int64_t size, void *in_data, int64_t in_size,
                void *out_data, int64_t out_size, int tag) {
    int *sum = &((int*)page)[0];
    int *count = &((int*)page)[1];
    int *sub_sum = &((int*)in_data)[0];
    int *n = &((int*)in_data)[1];
    int *ret = &((int*)out_data)[0];
    switch (tag) {
    case ALLREDUCE:
        *sum += *sub_sum; /* (###) */
        *count += 1;
        if (*count == *n) {
            *ret = *sum;
            *sum = *count = 0;
        } else {
            *ret = WAIT_A_MOMENT;
        }
        break;
    }
    return;
}
```

図 3.8　ユーザ定義の read-modify-write を使って Allreduce を実現するプログラム.

3.5.3　アドレスの変更監視

図 3.8 において，(#) を示した箇所では *wait_addr* の値が *WAIT_A_MOMENT* から総和に変化するのをビジーウェイトで待機している．しかし，きわめて短時間の間に値が変化する保証はないため，ビジーウェイトによる待機は他のスレッドの実行を邪魔し，アプリケーション全体の性能を劣化させてしまう可能性がある（**6.3.3** 節を参照）.

一般に，Allreduce や mutex，条件変数など同期命令を実装する際には，あるアドレスの値が変化す

るのを待機する操作が必須になる．そこで DMI では，この待機操作をビジーウェイトを使うことなく実現する API として，DMI_watch(int64_t *addr*, int64_t *size*, void **buf*, void **compare_buf*) を提供している．DMI_watch() 関数は，グローバルアドレス領域 [*addr*, *addr*+*size*) の値が変更されるたびに，最新の値をローカルアドレス領域 [*compare_buf*, *compare_buf*+*size*) の値と比較し，値が異なった時点で，そのときの値をローカルアドレス領域 [*buf*, *buf*+*size*) に read する．具体的には，図 3.8 における (#) のビジーウェイトは，

compare = WAIT_A_MOMENT;
DMI_watch(*wait_addr*,sizeof(int),&*sum*,&*compare*);

に置き換えることができる．

この API のセマンティクスは，Linux カーネルにおいて，あるアドレスの値が変化するのを効率よく待機するためのシステムコールである futex[4] から着想を得たものである．

3.6 スレッドの生成/破棄に基づく並列性の表現

並列プログラムの並列性を（自動並列化ではなく）明示的に表現する手段としては，大きく分類して，スレッドを生成/破棄するスタイルと SPMD 型のスタイルが存在する．スレッドを生成/破棄するスタイルは，pthread，Cilk[27]，Chapel などでサポートされており，プログラマはスレッドを自由に生成/破棄することで並列性を表現する．一方で，SPMD 型のスタイルは，MPI，UPC，Co-Array Fortran，Titanium など分散メモリ環境を前提とする多くの処理系でサポートされている．SPMD 型のスタイルでは，プログラムが開始した時点ですでに一定数のスレッドが実行されており，プログラムが終了するまでスレッド数は変化しないため，プログラマは，集合通信や集合同期操作などを使いつつ，各スレッドがどのタイミングで何を行うべきかを記述することで並列性を表現する．そのため，SPMD 型のスタイルは，アルゴリズムレベルの並列性が静的であり，すべてのスレッドが時系列的にどのように動作するべきかが静的に明確であるような同期的な並列計算を記述するのには適している．しかし，アルゴリズムレベルの並列性が動的な並列計算や，非同期的なプロセスの参加/脱退にともなって並列性が増減するような並列計算を記述するのには適していない．以上の観察より，DMI では，**非同期的なプロセスの参加/脱退にともなう並列性の増減を，スレッド数の増減として簡単に記述できる**ようにするために，SPMD 型のスタイルではなく，スレッドを生成/破棄するスタイルを採用する．

スレッドを生成/破棄するスタイルにもさまざまなものがあるが, DMI では, 共有メモリ環境上のマルチスレッドプログラミングから飛躍の少ないプログラミングインタフェースを整備することをねらいとして, **pthread との対応性を重視した API** を提供する．具体的には, スレッドの create/join/detach, mutex による排他制御，条件変数による同期など，pthread に対応した API を提供する．

また，プロセスの参加/脱退をユーザプログラムから簡単に制御できるようにするために，プロセスの参加/脱退のイベントをポーリングする API，プロセスの参加/脱退を承認する API，プロセスが属するノードの情報を取得する API，実行中のプロセスの一覧を取得する API などを提供する．これらの具体的な使用例を次節で示す．

3.7 プログラム例と実行例

プロセスが非同期に参加/脱退しながら，排他制御された 1 個のカウンタ変数をインクリメントするプログラムを図 3.9 に示す.

プログラムの実行の流れは以下のようになる：

(1) このプログラムを libdmi.a と静的リンクしてコンパイルすることで実行バイナリ./a.out が生成される.

(2) ノード A でコマンド「./dmirun ./a.out」を実行すると，ノード A 上に 1 個のプロセス 0 が生成され，DMI_main(...) が走り始める（15 行目）.

(3) 別のノード B でコマンド「./dmirun -i ノード 0 のホスト名 ./a.out」を実行すると，ノード B 上にプロセス 1 が生成され，DMI に対してプロセス 1 の参加宣言が通知される．DMI に対して通知されるこれらの参加/脱退宣言は，DMI_poll() 関数でポーリングすることができる．いまの場合，プロセス 1 の参加宣言は，DMI_main() 関数が呼び出している DMI_poll(*proc*) 関数によって捕捉され，プロセス 1 の情報が引数の *proc* に格納される（30 行目）．この *proc* はプロセス 1 に関するさまざまな情報を保持している．具体的には，*proc.hostname* がプロセス 1 が属するノード B のホスト名，*proc.pid* がプロセス 1 の識別番号，*proc.core* がプロセス 1 が属するノード B のプロセッサ数，*proc.memory* がプロセス 1 が提供しているメモリプールの容量，*proc.state* がプロセス 1 の状態（参加を宣言している状態/実行中の状態/脱退を宣言している状態/実行していない状態）を表す．とくに，*proc.pid* は，DMI がプロセス 1 に対して割り振った一意な識別番号であり，この識別番号を指定して DMI_welcome() 関数を呼ぶことでプロセス 1 の参加が完了する（32 行目）．そのあと，プロセス 1 の識別番号を指定して DMI_create() 関数を呼ぶと，プロセス 1 上に任意個のスレッドを生成することができる（36 行目）．これらのスレッドは DMI_thread() 関数から実行を開始する（57 行目）．以降，同様にして，コマンド「./a.out -i すでに実行中のホスト名」を実行することによって任意個のノードの任意個のプロセスの参加を実現できる.

(4) ノード B 上のプロセス 1 を脱退させる場合には，ノード B 上で Ctrl+C を叩くなどしてプロセス 1 に対して SIGINT 割り込みを行う．するとプロセス 1 の脱退宣言が DMI に対して通知され，この脱退宣言がやがて DMI_poll() 関数に拾われる（30 行目）．よって，グローバルアドレス空間を利用して終了通知を書き込み，プロセス 1 上で実行中のスレッドを終了させたあと，それらのスレッドを DMI_join() 関数によって回収する（42 行目）．最後に，プロセス 1 の識別番号を指定して DMI_goodbye() 関数を呼ぶことでプロセス 1 の脱退が完了する（44 行目）.

このように，DMI では，コマンドの実行という外的操作によってプロセスの参加/脱退を宣言したあとで，ユーザプログラムが DMI_welcome() 関数/DMI_goodbye() 関数を呼び出すことでそれらの参加/脱退宣言を内的操作によって承認することで，実際の参加/脱退が実現される仕組みになっている．また，参加するプロセスに対してスレッドを生成したり，脱退するプロセスからスレッドを回収するの

```
01: #include "dmi_api.h"
02: #define PROC_MAX 128
03: #define CORE_MAX 16
04:
05: typedef struct arg_t {
06:     DMI_mutex_t mutex;
07:     int value; /* a counter */
08:     int64_t flag_addr;
09: }arg_t;
10:
11: #define MUTEX(arg_addr) ((int64_t)&(((arg_t*)arg_addr)->mutex))
12: #define VALUE(arg_addr) ((int64_t)&(((arg_t*)arg_addr)->value))
13: #define FLAG_ADDR(arg_addr) ((int64_t)&(((arg_t*)arg_addr)->flag_addr))
14:
15: void DMI_main(int argc, char **argv) {
16:     DMI_proc_t proc;
17:     DMI_thread_t handle[PROC_MAX][CORE_MAX];
18:     int i, value, flag, my_pid;
19:     int64_t arg_addr, flag_addr;
20:
21:     DMI_pid(&my_pid); /* know the process id of this process */
22:     DMI_mmap(&arg_addr, sizeof(arg_t), 1); /* allocate a global address space for storing arguments for a thread */
23:     DMI_mmap(&flag_addr, sizeof(int), PROC_MAX);
          /* allocate a global address space for notifying threads of their termination */
24:     DMI_mutex_init(MUTEX(arg_addr));
25:     value = 0;
26:     DMI_write(VALUE(arg_addr), sizeof(int), &value, EXCLUSIVE);
27:     DMI_write(FLAG_ADDR(arg_addr), sizeof(int64_t), &flag_addr, EXCLUSIVE);
28:
29:     while (1) {
30:         DMI_poll(&proc); /* poll a process which wishes to join or leave */
31:         if (proc.state == DMI_OPEN) { /* if the process wishes to join */
32:             DMI_welcome(proc.pid); /* the process joins */
33:             flag = 0;
34:             DMI_write(flag_addr + proc.pid * sizeof(int), sizeof(int), &flag, EXCLUSIVE);
35:             for (i = 0; i < proc.core; i++) { /* create threads on the process */
36:                 DMI_create(&handle[proc.pid][i], proc.pid, arg_addr);
37:             }
38:         } else if (proc.state == DMI_CLOSE) { /* if the process wishes to leave */
39:             flag = 1;
40:             DMI_write(flag_addr + proc.pid * sizeof(int), sizeof(int), &flag, EXCLUSIVE);
                  /* notify threads on the process of their termination */
41:             for (i = 0; i < proc.core; i++) { /* join the threads */
42:                 DMI_join(handle[proc.pid][i], NULL);
43:             }
44:             DMI_goodbye(proc.pid); /* the process leaves */
45:             if (proc.pid == my_pid) break;
46:         }
47:     }
48:
49:     DMI_read(VALUE(arg_addr), sizeof(int), &value, GET);
50:     printf("The final value is %d\n", value);
51:     DMI_mutex_destroy(MUTEX(arg_addr));
52:     DMI_munmap(flag_addr); /* deallocate the global address space */
53:     DMI_munmap(arg_addr); /* deallocate the global address space */
54:     return;
55: }
56:
57: int64_t DMI_thread(int64_t arg_addr) { /* each thread */
58:     int value, flag, my_pid;
59:     int64_t flag_addr;
60:
61:     DMI_pid(&my_pid);
62:     DMI_read(FLAG_ADDR(arg_addr), sizeof(int64_t), &flag_addr, GET);
63:     while (1) {
64:         DMI_read(flag_addr + my_pid * sizeof(int), sizeof(int), &flag, UPDATE);
65:         if (flag == 1) break; /* terminate this thread if the termination of this thread is notified */
66:         DMI_mutex_lock(MUTEX(arg_addr)); /* lock */
67:         DMI_read(VALUE(arg_addr), sizeof(int), &value, INVALIDATE); /* read the counter */
68:         value++; /* increment the counter */
69:         DMI_write(VALUE(arg_addr), sizeof(int), &value, PUT); /* update the counter */
70:         DMI_mutex_unlock(MUTEX(arg_addr)); /* unlock */
71:     }
72:     return DMI_NULL;
73: }
```

図 3.9 プロセスが非同期的に参加/脱退しながら，排他制御されたカウンタ変数をインクリメントするプログラム．

もユーザプログラムの責任である．このような仕様になっている理由は，再構成可能なアプリケーションとはいえ，外的操作を契機として任意のタイミングでプロセスが参加/脱退したりスレッドが増減したりしてもよいわけではなく，個々のアプリケーションに応じて，プロセスを参加/脱退させてもよいタイミング，スレッドを生成/破棄してもよいタイミング，スレッドの生成/破棄の前後で行う必要がある処理などが存在するのが通常であり，それらはユーザプログラム側で明示的に制御できる必要があるためである．

なお，再構成可能な並列計算だけではなく，開始から終了まで一定のプロセス数で実行するようなSPMD 型の並列計算も容易に実行できるようにするため，多数のノードを一括して参加/脱退させるためのコマンドも用意している．また，GXP[174] などの並列シェルを利用することでも，多数のノードの一括参加/脱退を容易に実現できる．

3.8 各要素技術に対する関連研究

本章で述べた各要素技術について，関連研究および新規性について指摘する：

任意のコヒーレンシ粒度 ページサイズを任意に指定することでコヒーレンシ粒度を調節できる機能は，region-based な分散共有メモリ処理系で採用されている [112, 118, 23] ほか，UPC，Chapel，X10 などの PGAS 処理系においても，ページサイズ（これらの処理系の用語ではブロックサイズ）を任意に指定してブロックサイクリックなデータ分散に基づいたグローバルアドレス空間を確保する機能が採用されている．

選択的キャッシュ read/write **3.2.3.4** 節で述べたように，選択的キャッシュ read/write は DMI にとって新規的である．各 read/write に対してその read/write が持つアクセスローカリティを指示するだけで，get/put 操作および invalidate 型と update 型のハイブリッド型のキャッシュプロトコルを柔軟にかつわかりやすく制御できる処理系は存在しない．また，再構成をともなう並列計算に関して，実際のアクセスローカリティにしたがってデータ分散を動的に適応させることができるという点も新規的である．

離散アクセスのグルーピング 離散アクセスのグルーピングは Global Arrays でもサポートされている．Global Arrays では，任意のブロックサイクリックなデータ分散を指定して n 次元領域を確保することができ，各次元の下限値と上限値を指定することで，その n 次元領域内の任意の m 次元領域に対する get/put 操作を発行することができる．そして，これらの get/put 操作が引き起こす通信は，内部的に宛先プロセスごとに集約されて処理される．しかし，Global Arrays がサポートしているのは n 次元領域内の任意の m 次元領域という定型的な領域に対する get/put 操作だけであり，DMI における離散アクセスのグルーピングのように，任意の非定型な領域に対する離散的なアクセスをグルーピングできるわけではない．

非同期 read/write 非同期 read/write は Global Arrays や DDSS[106] に採用されている．また，Chapel の begin 文や X10 の async 文を利用すると，複数の文を非同期に実行させることができ，非同期な read/write を実現することができる．

ユーザ定義の read-modify-write MPI における MPI_Accumulate() 関数や Global Arrays における NGA_Acc() 関数などの Accumulate 演算を利用すると，アトミックな加算や定数倍などの read-modify-write を実現できる [63, 143]．しかし，これらの Accumulate 演算では処理系によってあらかじめ定義された種類の演算しか行うことができず，ユーザが自由に read-modify-write を定義できるわけではない．

pthread に対応した並列性の表現 共有メモリ環境上のマルチスレッドプログラミングを分散化させる際の敷居を下げることを目指し，pthread と類似のプログラミングインタフェースによって分散プログラムを記述できるようにした分散共有メモリ処理系としては，DSM-Threads[135, 136, 160] がある．しかし，DSM-Threads はプロセスの動的な参加/脱退には対応しておらず，DMI のように，プロセスの動的な参加/脱退にともなうスレッド数の増減を簡単に記述できるようにすることを目的として pthread と類似のプログラミングインタフェースを採用するという視点は含まれていない．また，DSM-Threads では，本節で述べたような，グローバルアドレス空間を高性能化させるための要素技術が採用されているわけでもない．

遠隔スワップシステム 並列プログラムに対して大容量の分散共有メモリを提供する処理系としては，Cashmere-VLM[60] や JIAJIA[79] がある．しかし，これらの遠隔スワップシステムで採用されているプロトコルは，各ページに対して固定的なプロセスを設けることで実現されているため，DMI のように，プロセスを動的に参加/脱退させることで遠隔スワップシステムの総メモリ容量を自由に変化させることはできない．

プロセスの非同期的な参加/脱退が可能なコヒーレンシプロトコル プロセスの非同期的な参加/脱退を許可するプロトコルは，DMI にとってきわめて新規的なものである．研究 [168] では，広域環境においてプロセスが動的に参加/脱退できるような分散共有メモリの設計が論じられている．しかし，この研究では，各クラスタごとに固定的な管理用サーバを 1 個設置しておき，プロセスの参加/脱退はそのプロセスが属するクラスタの管理用サーバの指示に基づいて実現されるプロトコルになっており，管理用サーバ自身を脱退させることはできない．これに対して DMI では，固定的なプロセスを仮定することなく，すべてのプロセスが自由なタイミングで参加/脱退できるような柔軟なプロトコルを実装する．

3.1 節で述べたように，DMI のグローバルアドレス空間は，各プロセスが提供するメモリプールをメモリ資源として利用し，各ページがつねに少なくとも 1 個のメモリプールに含まれるように，ページのコヒーレンシ管理が行われる．このようなシステムアーキテクチャは，ハードウェア上の技術である COMA（Cache Only Memory Architecture）に似ている [74]．COMA は，物理的な共有メモリを設置することなく，各プロセッサのキャッシュだけを利用して共有メモリ機構を実現する技術であり，各ページ（COMA の用語ではアイテム）がつねに少なくとも 1 個のプロセッサのキャッシュには存在するようにページのコヒーレンシが管理される．COMA の代表的な実装としては DDM[74] などがある．しかし，DDM はハードウェア上の技術であるためプロセッサ数はつねに一定であることが前提とされており，そのプロトコルはプロセッサの動的な参加/脱退に対応したものではなく，DMI のグローバルアドレ

ス空間のコヒーレンシプロトコルに応用できるものではない．たとえば DDM では，各ページに対して固定的な帰属キャッシュが決められており，あるキャッシュからページを追い出すときに適切な追い出し先が見つからなければ，そのページの帰属キャッシュへと追い出すことになっている．これに対して，DMI のプロトコルでは，各ページに対して固定的な帰属プロセスを設けることなく，ページ置換や脱退前のページの追い出しを実現する．

3.9 要約

本章では，DMI のグローバルアドレス空間の設計について述べた．

DMI では，第 1 に，高性能なグローバルアドレス空間を実現するために，(1) 内部的に起きる通信を明示的に簡単に制御でき，(2) 内部的に起きる通信を明示的に集約でき，(3) 内部的に起きる通信が予測しやすく最適化しやすい API を提供している．DMI_mmap() 関数において適切なページサイズを設定したうえで，各 DMI_read() 関数/DMI_write() 関数において適切な選択的キャッシュ read/write を選択することが DMI におけるプログラム開発の基本であり，そのほか高性能化のための機能として，非同期 read/write，離散的なアクセスのグルーピング，ユーザ定義のアトミック命令，データ転送の動的負荷分散を導入している．

第 2 に，再構成可能な並列計算を記述できるようにするために，スレッドの生成/破棄によって並列性を表現できるようにし，プロセスが非同期的に参加/脱退できるようなグローバルアドレス空間のコヒーレンシプロトコルを実装している．

第 4 章

再構成可能かつ高性能なグローバルアドレス空間の実装

本章では，グローバルアドレス空間の実装について述べる．とくに，**4.2** 節で述べる，非同期的にプロセスを参加/脱退させられるコヒーレンシプロトコルは新規的なものである．

4.1 プロセスの構成要素

DMI における各プロセスの構成要素は，receiver スレッド，handler スレッド，sweeper スレッドと複数の計算スレッドであり，それぞれ以下の役割を果たす（図 4.1）．本稿では計算スレッドのことを単にスレッドとも呼ぶ：

receiver スレッド receiver スレッドは，他のプロセスから送信されてくるメッセージを受信しては，そのメッセージをメッセージキューに挿入する作業を繰り返す．

handler スレッド handler スレッドは，メッセージキューのなかのメッセージを 1 個ずつ処理する．具体的には，メッセージキューからメッセージをとり出し，そのメッセージを解釈して，かならず有限時間で終了することが保証されるようなローカルな処理を行ったあと，必要であればそのメッセージに対して応答メッセージを送信する，という作業を繰り返す．つまり，handler スレッドは，メッセージキューからメッセージをとり出してから次にメッセージキューを覗くまでの間に，他のプロセスとのメッセージ送受信を介するような，有限時間で終了するかどうかを保証できないような処理は行わない．これにより，プロセス間にまたがるメッセージの依存関係に起因するデッドロックを回避している．また，handler スレッドは 1 本しか存在しないため，すべてのメッセージの処理をシリアライズする役割も持つ．なお，receiver スレッドがメッセージを直接処理するのではなく，いったんメッセージキューに挿入したメッセージを別の handler スレッドが処理する理由は，TCP の受信バッファの飽和によるデッドロックを回避するためである．たとえば，プロセス A とプロセス B を考え，両者とも receiver スレッドが直接メッセージを処理するとする．このとき，プロセス A の receiver ス

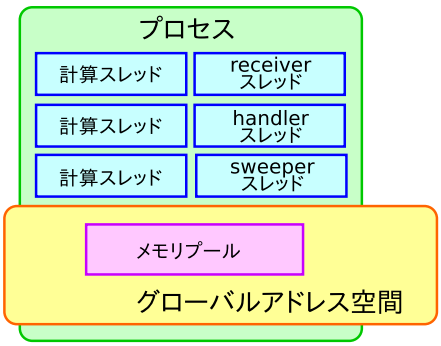

図 4.1 DMI の各プロセスの構成要素.

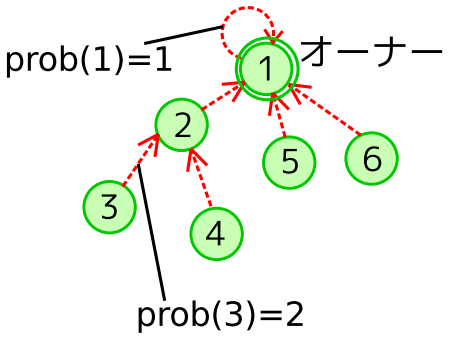

図 4.2 オーナー追跡グラフ.

レッドが何らかのメッセージを受信して処理した結果としてプロセス B に対して巨大なメッセージを送ろうとし，それとほぼ同時に，プロセス B の receiver スレッドも何らかのメッセージを受信して処理した結果としてプロセス A に対して巨大なメッセージを送ろうとした場合，両者の TCP の受信バッファが飽和してしまうためデッドロックが発生する．

sweeper スレッド sweeper スレッドは，そのプロセスのメモリプールの使用量をつねに監視しており，メモリプールの使用量が一定量を超過した場合にページ置換を行う．つまり，sweeper スレッドは通常の OS におけるページスワッパの役割を果たす．

計算スレッド ユーザプログラムが DMI_create() 関数を呼び出すことによって生成されるスレッドで，DMI_thread() 関数から実行を開始する．各プロセスに任意個生成できる．

4.2 再構成可能なグローバルアドレス空間のコヒーレンシプロトコル

本節では，DMI にとって大きな新規性のある，プロセスが非同期的に参加/脱退可能なグローバルアドレス空間のコヒーレンシプロトコルの詳細について述べる．まず，**4.2.1** 節で基本アイディアをまとめたうえで，**4.2.2** 節でデータ構造を定義し，**4.2.3** 節で複雑なプロトコルを見通しよく正しく実装するための手法について議論し，**4.2.4** 節でプロトコルの詳細な実装を述べる．また，プロトコルの厳密なアルゴリズムについては付録の第 A 章に載せる．DMI におけるコヒーレンシ粒度はページであり，各ページに対するプロトコルは独立に動作するため，以下ではある 1 個のページについて議論する．たとえば，プロセス i のデータ x といった場合，それは，ある 1 個のページに関してプロセス i が管理しているデータ x を意味する．

4.2.1 基本アイディア

4.2.1.1 オーナー追跡グラフ

3.3 節で述べたように，プロセスの非同期的な参加/脱退を行うと同時に選択的キャッシュ read/write によってオーナーを移動させる場合にまず問題となるのは，オーナーの所在をどのように特定するかである．DMI では，この問題を probable owner[111] と呼ばれる手法を応用させることで解決している．

この手法では，各プロセス i に対して probable owner と呼ばれるデータを持たせる．プロセス i の probable owner は，プロセス i による「オーナーの予想」であり，プロセス i が「いま現在のオーナーはプロセス j だろう」と予想しているとき，プロセス i の probable owner はプロセス j になっている．よって，各プロセスの probable owner は真のオーナーを参照しているとはかぎらない．しかし，各プロセスの予想ができるかぎり正しく保たれるようにプロトコルを実装することによって，（一時的な過渡状態を覗く）任意の時刻において，すべてのプロセス i による probable owner の参照関係を有向グラフとして描くと，図 4.2 に示すように，参照関係が真のオーナーへと収束するグラフになることを保証できる．本稿では，このグラフをオーナー追跡グラフと呼び，「任意のプロセスから開始して，各プロセスの probable owner をたどっていけば，有限時間内にかならず真のオーナーに到達できる」という性質をオーナー追跡グラフの正しさと呼ぶ．

オーナー追跡グラフを用いれば，任意のプロセス i で read/write フォルトなどが発生してオーナーにメッセージを通知する必要が生じた場合，そのプロセス i から開始して，プロセス i の probable owner，プロセス i の probable owner の probable owner，プロセス i の probable owner の probabler owner の probable owner... というように，各プロセスにおいて probable owner の方向へメッセージをフォワーディングすることによって，いずれは真のオーナーにメッセージを通知することができる[*1] ．

さて，オーナー追跡グラフにおける課題は，どのように各プロセスの probable owner を管理しておけば，オーナー追跡グラフの正しさを保証できるかであるが，おおまかには，以下のルールによって probable owner を更新すればよい：

(1) ページが確保された時点では，すべてのプロセス i の probable owner は正しいオーナーを参照するようにする．
(2) プロセス i がオーナー v からのメッセージを受信するたびに，プロセス i の probable owner をプロセス v に更新する．

(2) のルールは，直観的には，「プロセス i は，プロセス i がもっとも直近にオーナーから受信したメッセージの送信元を，オーナーとして予想している」ことを意味する．probable owner の更新に関する正確なプロトコルは **4.2.4** 節で述べ，それがオーナー追跡グラフの正しさを満足することを **4.2.4.7** で証明する．

4.2.1.2 ページの状態管理

選択的キャッシュ read/write における invalidate 型キャッシュや update 型キャッシュを実装するために，各プロセスにおける各ページの状態は，以下の 4 とおりで管理する：

INVALID 状態 最新ページを持っていない状態．

DOWN_VALID 状態 invalidate 型キャッシュとして最新ページを持っている状態．つまり，次にページの更新があった時点では，オーナーから無効化の要求が送信されて INVALID に遷移

[*1] オーナー追跡グラフに沿ってメッセージをフォワーディングする速度よりもオーナーの移動速度が遅いことを仮定している．

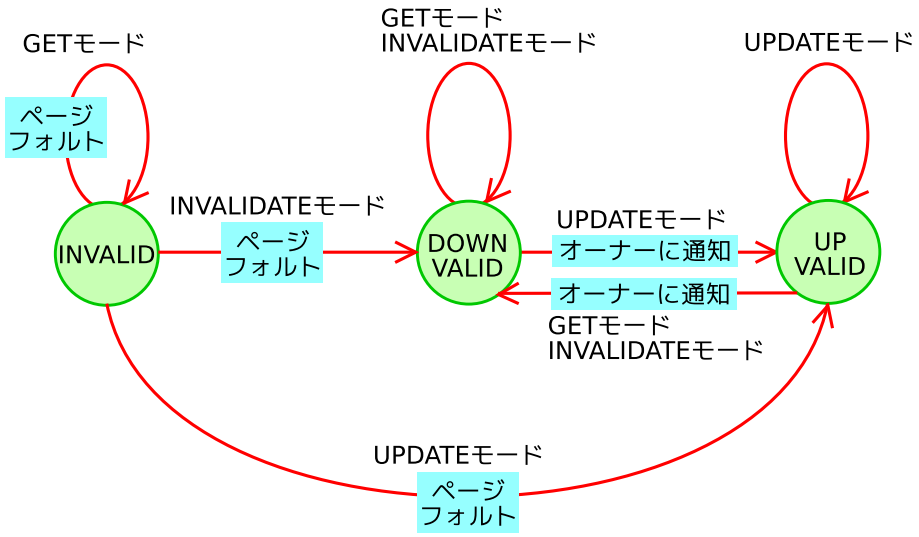

図 4.3 選択的キャッシュ read におけるページの状態遷移.

する.

UP_VALID 状態 update 型キャッシュとして最新ページを持っている状態．つまり，次にページの更新があった時点では，オーナーからページの更新情報が送信されてページが最新に保たれ，UP_VALID であり続ける.

DMI では，オーナーにおいてのみ write が可能であり，キャッシュを持っているときに read が可能な，Single Writer/Multiple Reader 型のプロトコルを実装する．したがって，プロセス i において read フォルトが発生する条件は，プロセス i におけるページが INVALID 状態の場合または選択的キャッシュ read によって要求されたモードとページの現在の状態が一致していない場合である．一方で，write フォルトが発生する条件は，プロセス i がオーナーでないか，または DOWN_VALID 状態もしくは UP_VALID 状態のページを持っているプロセスがプロセス i 以外に存在する場合である.

4.2.1.3 選択的キャッシュ read

選択的キャッシュ read では，アクセスローカリティに応じて INVALIDATE モード，UPDATE モード，GET モードを指示できるが，このときページの状態に対応して，図 4.3 に示すページの状態遷移が起きる．状態遷移の具体例をあげる：

- プロセス i において，ページが INVALID 状態で UPDATE モードの read が発行された場合，オーナーに read フォルトが送信され，オーナーにおいてプロセス i が update 型キャッシュを持つプロセスとして登録される．その後，オーナーからプロセス i に対して最新ページの転送が行われ，プロセス i のページは UP_VALID 状態に遷移する.
- プロセス i において，ページが UP_VALID 状態で GET モードの read が発行された場合，オーナーに状態遷移の要求が送信され，今まで update 型キャッシュを持つプロセスとして登録されて

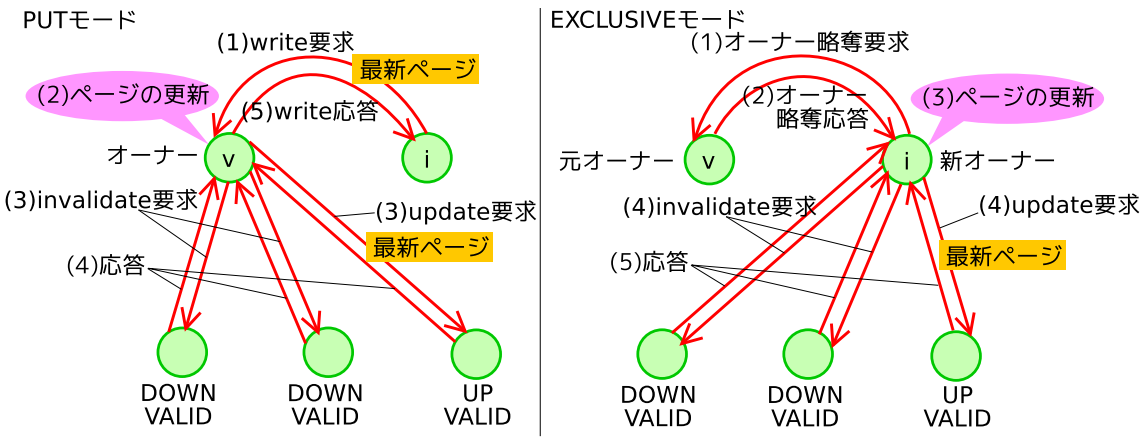

図 4.4 選択的キャッシュ write におけるキャッシュのコヒーレンシ維持（プロセス i で write フォルトが発生するとし，この時点でのオーナーはプロセス v とする）．

いたプロセス i が invalidate 型キャッシュを持つプロセスとして登録しなおされる．その後，オーナーからプロセス i に対してメッセージが送信され，プロセス i のページが DOWN_VALID 状態に遷移する．なお，このとき INVALID 状態ではなく DOWN_VALID 状態に遷移させるのは，無理やり INVALID 状態に遷移させることには利点がないためである．

4.2.1.4 選択的キャッシュ write

選択的キャッシュ write では，アクセスローカリティに応じて EXCLUSIVE モード，PUT モードを指示できるが，このときページの更新とキャッシュのコヒーレンシ維持は図 4.4 に示すようなプロトコルで行われる．オーナーはページを更新したあと，invalidate 型キャッシュを持つプロセスに対して invalidate 要求を送信し，update 型キャッシュを持つプロセスに対して update 要求を送信し，これらすべての invalidate 要求と update 要求に対する応答を回収することで，キャッシュのコヒーレンシを維持する．

4.2.1.5 そのほかの要請

以上が選択的キャッシュ read/write を実現させるプロトコルの基本アイディアである．DMI ではこのほかに，(1) ページ置換を行う場合やプロセスが脱退する直前にページを追い出す場合に必要となるページの追い出しのプロトコル，(2) ユーザ定義の read-modify-write およびアドレスの変更監視を実現するプロトコル，(3) プロセスが参加/脱退する場合にそのプロセスをオーナー追跡グラフに追加/削除するためのプロトコルが必要である．

4.2.2 ページテーブルのデータ構造

4.2.2.1 ページテーブル

DMI のグローバルアドレスは 64 ビット整数で表現される．デフォルトでは，図 4.5 に示すように，先頭 16 ビットでグローバルアドレス空間を識別し，残りの 48 ビットでそのグローバルアドレス空間内のオフセットを指定する．したがって，DMI_mmap() 関数で確保できるグローバルアドレス空間の個

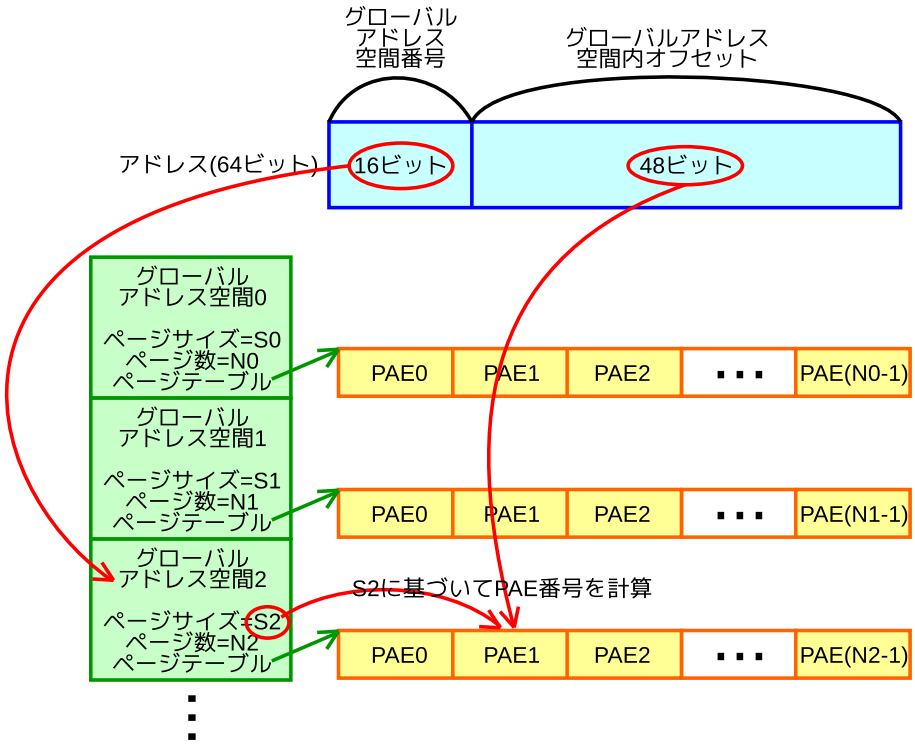

図 4.5 ページテーブルの構造.

数は 2^{16} 個に制限され，1 個のグローバルアドレス空間のサイズは 2^{48} バイトに制限される．各ページごとに，そのページのコヒーレンシ管理のためのページテーブルエントリを設ける．

4.2.2.2 ページテーブルエントリ

各プロセスは各ページに対して，*owner*，*probable*，*buffer*，*state*，*state_array*，*valid*，*seq_array* の 7 種類のデータを管理する．以下では，(ある 1 個のページに関する) プロセス i の *owner* を $i.owner$ などと表記する．各データの意味と性質は以下のとおりである：

owner プロセス i がオーナーであれば $i.owner = \mathrm{TRUE}$，そうでなければ $i.owner = \mathrm{FALSE}$ である．いい換えると，$i.owner = \mathrm{TRUE}$ であることが，プロセス i がオーナーであることの定義である．任意の時刻においてオーナーは系内に高々 1 個しか存在しない．オーナーが移動する場合には，一時的に系内にオーナーが存在しなくなる時刻が存在するが，有限時間内には必ずオーナーが確定する．

probable プロセス i の probable owner を管理する．つまり，プロセス i がプロセス j をオーナーであると予想しているとき，$i.probable = j$ である．すべてのプロセスを通じた *probable* の参照関係がオーナー追跡グラフであり，任意のプロセスで発生したオーナー宛のメッセージは，各プロセス i において $i.probable$ へとフォワーディングされることによってやがて真のオー

ナーにたどり着く．なお，$i.probable = i$ であってもプロセス i がオーナーであるとはかぎらない．

buffer ページのデータ本体を格納するためのバッファである．

state $i.state$ はプロセス i が保持しているページの状態を記録しており，INVALID，DOWN_VALID，UP_VALID のいずれかの値をとる．$i.state$ = DOWN_VALID ならば，$i.buffer$ には最新ページが格納されており，プロセス i は invalidate 型キャッシュを持つプロセスとしてオーナーに登録されている．$i.state$ = UP_VALID ならば，$i.buffer$ には最新ページが格納されており，プロセス i は update 型キャッシュを持つプロセスとしてオーナーに登録されている．$i.state$ = INVALID ならば，プロセス i は最新ページを持っておらず，read すると read フォルトが発生する．

state_array 以上の *owner*，*probable*，*buffer*，*state* の 4 種類のデータはすべてのプロセスが管理するデータであるが，これ以降に述べる *state_array*，*valid*，*seq_array* の 3 種類のデータはオーナーのみが管理するデータである．*state_array* はオーナー v が管理する配列データで，すべてのプロセスにおけるページの状態を記録している．$v.state\_array[i]$ には $i.state$ の値が格納されている．

valid オーナー v のみが管理するデータで，DOWN_VALID 状態または UP_VALID 状態にあるプロセスの個数を管理する．すなわち，$v.valid$ は，$v.state\_array[i]$ = DOWN_VALID または $v.state\_array[i]$ = UP_VALID であるような i の個数に等しい．

seq_array オーナー v のみが管理する配列データで，$v.seq\_array[i]$ には，実行開始から現在にいたるまでにオーナーから各プロセス i に対して送信したメッセージの個数が記録されている．これはプロセス v がオーナーになった時点以降にオーナー v がプロセス i へ送信したメッセージの個数ではなく，プログラムの実行開始以降，オーナーがどのように遷移したかにかかわらず，オーナーとなっているプロセスがプロセス i へ送信したメッセージの個数の総和である．*seq_array* は，**4.2.3** 節で述べるメッセージの順序制御に利用する．

4.2.3 複雑なプロトコルを見通しよく正しく実装する方法

選択的キャッシュ read/write，ページの追い出し，プロセスの参加/脱退などの多様な操作を，各プロトコルが他のプロトコルに及ぼす影響などを考慮しつつ正しく実装するのは非常に難しい．そこで，このような複雑なプロトコルを見通しよく正しく実装するための方針について整理する．

まず，オーナーが移動するという性質が議論を複雑化させていることをふまえ，思考実験として，オーナーはいっさい移動せず，各ページに対してオーナーが固定されているような状況を仮定する．また，オーナーから各プロセスに対してメッセージを送信するための FIFO な通信路が存在することも仮定する．この状況は，いわゆる単純なクライアント・サーバ型のモデルに相当し，いかに複雑な操作であっても正しいプロトコルを実装するのが容易である．なぜなら，各プロセス（クライアントに相当）とオーナー（サーバに相当）は，基本的には以下のような形態の処理を行うだけで，系内で生じるすべての操作をシリアライズして処理できるからである：

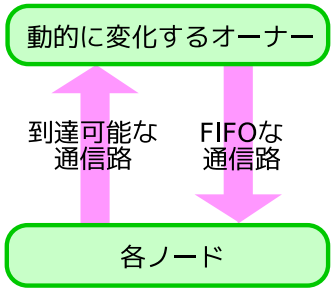

図 4.6 DMI のコヒーレンシプロトコルが保証する，各プロセスとオーナーとの通信経路.

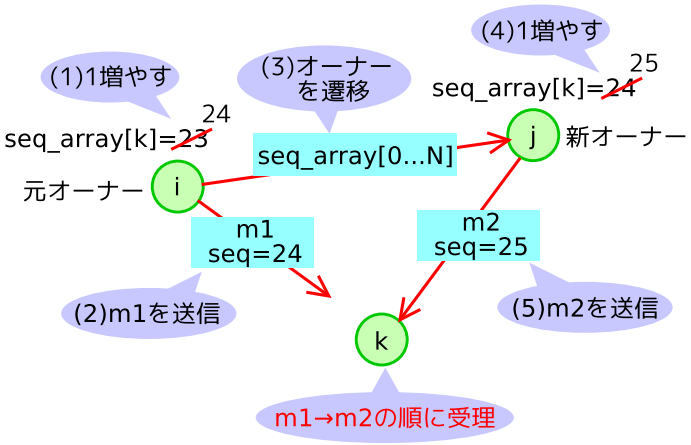

図 4.7 オーナーから各プロセスに対するメッセージの順序制御.

- 各プロセスは，オーナーの助けを必要とする何らかの操作（read/write フォルト，ページの追い出しなど）が必要になった場合，オーナーに対して要求メッセージを送信する．そのあと，オーナーから応答メッセージが返って来るのを待機する．
- オーナーは，自分に届く要求メッセージを 1 個ずつシリアライズして処理し，処理した結果，要求メッセージの送信元に対して応答メッセージを返す．

ここで，なぜ，クライアント・サーバ型のモデルでは，系内で生じるすべての操作をシリアライズして処理できるのかを分析してみると，それは要するに以下の性質が保証されているためである：

- 各プロセスがオーナーの所在をつねに知っていること
- 単独のオーナーがすべての要求メッセージをシリアライズして処理すること
- オーナーがプロセス i に対して発行したメッセージは，オーナーが発行した順序どおりにプロセス i に受信されること

以上の思考実験をふまえると，動的にオーナーが移動する場合でも，上記の 3 つの性質が満足されるようなプロトコルをまず実装しておけば，そのプロトコルのうえに，選択的キャッシュ read/write などの複雑な操作に対するプロトコルを正しく実装するのが容易になることがわかる．したがって，DMI では，まず第 1 ステップとして，以下の 3 条件（図 4.6）を満たすプロトコルを実装し，第 2 ステップとして，そのうえに選択的キャッシュ read/write などの複雑な操作に対するプロトコルを実装する：

条件 1　各プロセスからオーナーに到達できるような通信路が存在すること
条件 2　1 個だけ存在するオーナーがすべての要求をシリアライズして処理すること
条件 3　オーナーの遷移に関係なく，オーナーから各プロセスへの FIFO な通信路が存在すること

条件 2 に関しては，オーナーが移動中であるような一時的な状態をのぞいては，オーナーが 1 個しか

存在しないようにし，かつ，その単独のオーナーが，受信したすべての要求メッセージをシリアライズして処理するように実装すればよい．

条件 3 に関しては，オーナーから各プロセスへのメッセージに順序番号を付与し，その順序番号に基づいて各プロセス側で**メッセージを順序制御**すればよい．具体的には，オーナー v からプロセス i へメッセージを送信する場合には，$v.seq\_array[i]$ の値を 1 増やしたあと，その $v.seq\_array[i]$ の値を順序番号としてメッセージに付与する．そして，各プロセス i は，オーナーから受信したメッセージについては，順序番号の順にメッセージを受信するように順序制御を行う．また，オーナーをプロセス v からプロセス v' に移動させる際には，$v.seq\_array$ をプロセス v からプロセス v' へ丸ごとコピーする．これにより，オーナーの移動にかかわらず，オーナーから各プロセス i に対して送信されるメッセージに付与される順序番号が連続的になることを保証する．たとえば，図 4.7 に示すように，オーナー v がプロセス k に対してメッセージ m_1 を送信したあと，オーナーがプロセス v' に移動し，新しいオーナー v' がプロセス k に対してメッセージ m_2 を送信する状況を考える．このとき，物理的にはメッセージ m_2 が m_1 よりも先に到着する可能性があるが，メッセージ m_2 の順序番号はメッセージ m_1 の順序番号より若いため，プロセス k における順序制御によって，m_1，m_2 の順で到着したかのように扱われる．このように順序制御を行うことで，オーナーの動的な移動にかかわらず，オーナーから各プロセスに対する FIFO な通信路を保証することができる．

条件 1 に関しては，オーナー追跡グラフの正しさが要請されており，任意のプロセスから各プロセスの *probable* をたどることでやがて真のオーナーに到達できるように，各プロセスの *probable* を適切に管理すればよい．しかし，各プロセスの *probable* というデータはすべてのプロセスに「分散された」データであり，各プロセスが自由なタイミングでその値を更新してしまうようでは，オーナー追跡グラフの正しさのような，すべてのプロセスの *probable* 全体に関する何らかの性質を保証することは不可能である．そこで，各プロセスが *probable* の値を更新できるタイミングに制約を設ける．具体的には，各プロセスの *probable* の値に関して，プロセス i が $i.probable$ を参照することは任意の時点で可能だが，プロセス i が $i.probable$ を書き換えることは，その書き換えを指示するような，オーナーからの（順序制御された）メッセージをプロセス i が受信した時点でしかできないという制約を課す．要するに，オーナーによって指示された時点でしか値を書き換えることはできないという制約を課す．この制約により，*probable* というデータは，データの所在としてはすべてのプロセスに分散してはいるものの，データの値としては実質的にオーナーによって一括管理されることになるため，オーナーが値の書き換えを正しく指示しさえすれば，オーナー追跡グラフの正しさを保証することが可能になる．

同様の議論は，*probable* だけではなく，すべてのプロセスにおいて管理されるデータである，*owner*，*buffer*，*state* についても成り立つ．以上の観察より，DMI では，すべてのプロセスにおいて管理されるデータである，*owner*，*probable*，*buffer*，*state* の 4 種類のデータの参照と更新に関して以下の制約を課す：

条件 4　各プロセスは任意の時点で値を参照することはできるが，**値を更新できるのは，その更新を指示するようなオーナーからの順序制御されたメッセージを受信した時点のみ**である

最後に，プロセスの参加/脱退とグローバルアドレス空間の確保/解放について考える．一般に，プロセスの参加/脱退とグローバルアドレス空間の確保/解放の操作を，排他制御を行うことなく同時に進行させることは難しい．たとえば，プロセスを識別するためには各プロセスに対して一意な識別番号を振る必要があるため，プロセスの識別番号を決定するための排他制御が必要である．また，DMI_mmap() 関数で確保されるグローバルアドレス空間を識別するためには，各グローバルアドレス空間に対して一意な識別番号を振る必要がある．さらに，グローバルアドレス空間を確保するためには，すべてのプロセスに対して **4.2.2** 節で述べたページテーブルのデータ構造を準備する必要があるが，この最中にプロセスが新たに自由に参加してしまうと，どのプロセスまでデータ構造を準備し終えたのかの管理などが難しくなる．したがって，DMI では，プロトコルを単純化させるために以下の条件を課す：

条件 5　プロセスの参加/脱退とグローバルアドレス空間の確保/解放は，系全体におけるグローバルロックを利用してシリアライズする．

なお，このグローバルロックによってシリアライズされる操作は，プロセスの参加/脱退およびグローバルアドレス空間の確保/解放のみであり，read/write やページ置換などは完全に独立して実行可能である．

4.2.4　コヒーレンシプロトコルの詳細

4.2.4.1　グローバルアドレス空間の確保

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 $page\_num$ 個から構成されるグローバルアドレス空間を確保できる．DMI_mmap() 関数を呼び出したプロセスをプロセス 0，確保されるページを p_0，p_1，...，p_{l-1}，この時点で参加しているプロセスをプロセス 0，プロセス 1，...，プロセス $m-1$ とすると，以下の手順で各ページのデータ構造が初期化される：

(1) プロセス 0 は，グローバルロックを取得する．
(2) プロセス 0 は，すべてのプロセスに対して，ページたち p_0，p_1，...，p_{l-1} の確保を指示するメッセージを送信する．
(3) そのメッセージを受信した各プロセス i は，ページたちのオーナーがラウンドロビンに割り当てられるよう，ページ p_k のオーナーがプロセス $\mathrm{mod}(k,m)$ になるようにページテーブルエントリを初期化する．具体的には，各ページ p_k に関して，プロセス i は，$i = \mathrm{mod}(k,m)$ ならば，$i.owner$ = TRUE，$i.probable = i$，$i.buffer = \emptyset$，$i.state = DOWN\_VALID$，$i.valid = 1$，$i.state\_array[i] = DOWN\_VALID$，$i \neq j$ なるすべての j（$0 \leq j < m-1$）に対して $i.state\_array[j] = INVALID$，すべての $j(0 \leq j < m-1)$ に対して $i.seq\_array[j] = 0$ に初期化する．一方で，$i \neq \mathrm{mod}(k,m)$ ならば，$i.owner$ = FALSE，$i.probable = \mathrm{mod}(k,m)$，$i.buffer = \emptyset$，$i.state = INVALID$ に初期化する．すなわち，オーナーにおいてのみ DOWN_VALID 状態のページを確保する．
(4) 各プロセスは，要求されたすべてのページに対するページテーブルエントリを確保したあと，プ

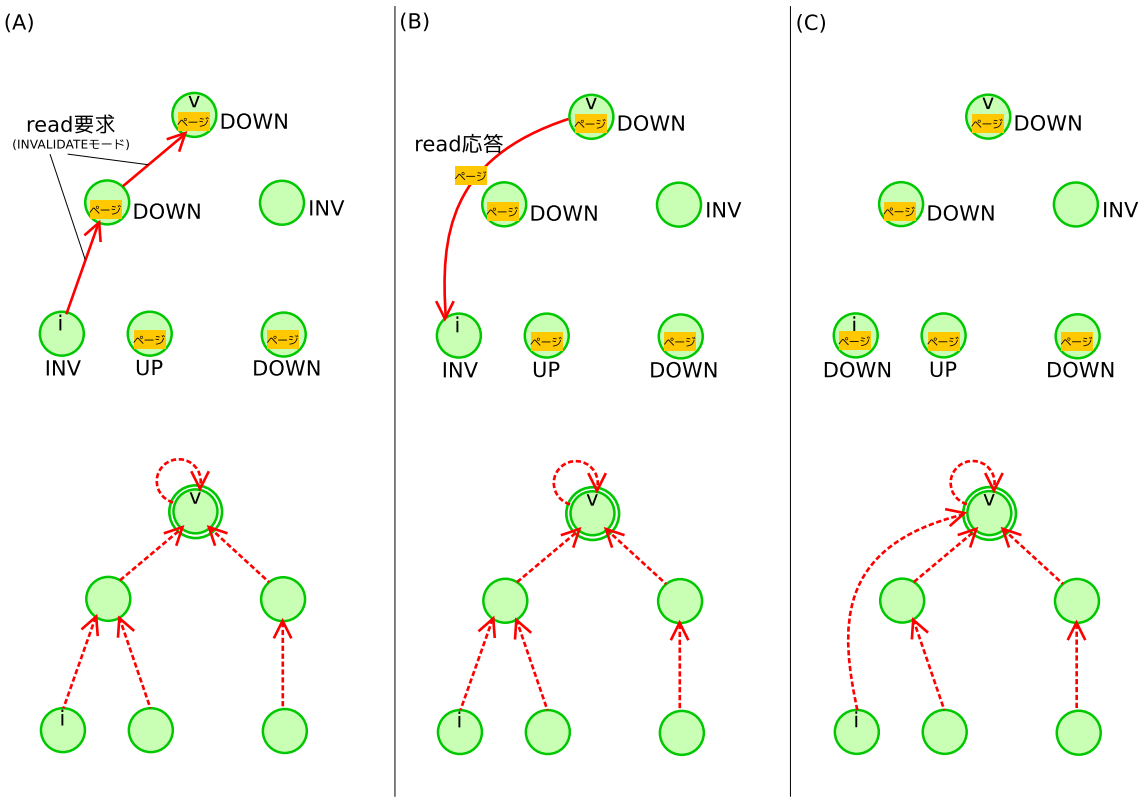

図 4.8 read のプロトコル.

ロセス 0 に対して応答のメッセージを送信する.

(5) プロセス 0 は m 個のすべてのプロセスから応答のメッセージを回収したあと，グローバルロックを解放する.

手順（3）において $i.buffer = \emptyset$ としているが，グローバルアドレス空間を確保する時点ではページのデータ本体である *buffer* にはメモリを割り当てない．はじめてこのページに対してアクセスがあったときに，*buffer* に対してページサイズ分のメモリをオーナーのメモリプールから割り当てる．なお，ラウンドロビンにオーナーを決定する理由は，すべてのプロセスが PUT モードで write を行ったとしても，ページのデータ本体を格納するためのメモリを提供するプロセスが系全体に分散するようにするためである．仮にすべてのページの初期的なオーナーをプロセス 0 にしてしまうとすると，すべてのプロセスが PUT モードで write を行った場合，これらのページのデータ本体がすべてプロセス 0 のメモリプール上に確保されることになり，プロセス 0 のメモリプールが飽和してページ置換が発動される可能性が高くなる.

4.2.4.2 選択的キャッシュ read

以降では，図 4.8，図 4.9，図 4.10，図 4.11 を使いながらプロトコルについて説明する．図 4.8，図 4.9，図 4.10，図 4.11 では，各コマにおいて，下段がオーナー追跡グラフの形状，上段が送受信メッセージやページの状態の様子を表す．また，下段において二重丸で囲まれたプロセスはオーナーを表し，図中の INV は INVALID，DOWN は DOWN_VALID，UP は UP_VALID を意味する．

プロセス i で read が発行された場合，$i.state$ = DOWN_VALID または $i.state$ = UP_VALID であり，かつ $i.state$ と read によって指示されるモードが矛盾していないならば，すぐに $i.buffer$ からデータが読み込まれて read が完了する．それ以外の場合には read フォルトが発生し，read 要求がオーナー宛に送信される（図 4.8（A））．なお，図 4.3 の状態遷移図に示すように，$i.state$ とモードが矛盾しているとは，$i.state$ = DOWN_VALID かつモードが UPDATE モードの場合，または $i.state$ = UP_VALID かつモードが INVALIDATE モードまたは GET モードの場合を指す．

プロセス i の read 要求がオーナー v に受信されたとき，以下の 2 とおりの場合が考えられる：

(I) $v.state\_array[i]$ = INVALID の場合

(1) プロセス i が要求するモードが INVALIDATE モードの場合には，オーナー v は，$v.state\_array[i]$ = DOWN_VALID とし，$v.valid$ を 1 増やす．UPDATE モードの場合には，オーナー v は，$v.state\_array[i]$ = UP_VALID としたうえで，$v.valid$ を 1 増やす．GET モードの場合には何も行わない．

(2) INVALIDATE モードまたは UPDATE モードの場合には，オーナー v は，プロセス i に対して，最新ページと $v.state\_array[i]$ の値を載せた read 応答を，順序番号を付与したうえで送信する（図 4.8（B））．GET モードの場合には，read 要求によって要求されている部分のデータのみを載せた read 応答を，順序番号を付与したうえで送信する．

(3) INVALIDATE モードまたは UPDATE モードの場合には，read 応答を受信したプロセス i は，$i.state$ を $v.state\_array[i]$ に更新し，$i.probable$ を v に更新し，$i.buffer$ に最新ページを格納する（図 4.8（C））．$i.buffer$ を読み込んで read を完了させる．一方で，GET モードの場合には，read 応答を受信したプロセス i は，$i.probable$ を v に更新したあと，read 応答に載っているデータを読み込んで read を完了させる．

(II) それ以外の場合

この状況は，プロセス i の read フォルトが $i.state$ と read が指示するモードの矛盾に起因したものである場合や，過去にプロセス i で発行された read 要求が先に処理されたために，いま着目している read 要求がオーナー v に到達した時点では，すでにプロセス i への最新ページの転送が完了している場合などに生じる．たとえば，DMI では同一プロセス内の複数スレッドがメモリプールを共有する構成になっているため，それらのスレッドが同一のページに対して同時に read フォルトを発生させた場合，それらの read フォルトのうちもっとも速くオーナーに到着したものだけが（I）で示した手順にしたがって処理され，残りの read フォルトは（II）の手順で処理されることになる．

(1) オーナー v は，プロセス i が要求するモードが GET モードまたは INVALIDATE モードであり，

かつ $v.state\_array[i] = \text{UP\_VALID}$ ならば $v.state\_array[i]$ を DOWN_VALID に更新する．一方で，要求するモードが UPDATE モードであり，かつ $v.state\_array[i] = \text{DOWN\_VALID}$ ならば $v.state\_array[i]$ を UP_VALID に更新する．

(2) オーナー v は，プロセス i に対して，$v.state\_array[i]$ の値を載せた read 応答を，順序番号を付与したうえで送信する．

(3) read 応答を受信したプロセス i は，$i.state$ を $v.state\_array[i]$ に更新し，$i.probable$ を v に更新する．$i.buffer$ を読み込んで read を完了させる．

4.2.4.3 選択的キャッシュ write

プロセス i で write が発行されたとき，i がオーナーであり，かつ $i.valid = 1$ ならば，すぐに $i.buffer$ にデータが書き込まれて write が完了する．それ以外の場合には write フォルトが発生し，write によって指示されるモードに応じた処理が行われる．

(I) モードが PUT モードの場合

(1) プロセス i は，書き込むべきデータ d を載せた write 要求をオーナー v に対して送信する（図 4.9（A））．

(2) write 要求を受信したオーナー v は，受信したデータ d に基づいて $v.buffer$ を更新する．

(3) オーナー v は，$v.owner$ を FALSE に更新し，オーナーを一時的に放棄する．プロセス v は，$v.state\_array[j] = \text{DOWN\_VALID}$ を満たすすべてのプロセス $j(\neq v)$ に対して，invalidate 要求を順序番号を付与して送信する．このとき，invalidate 要求を送信する各プロセス j に対して，$v.state\_array[j]$ を INVALID に更新し，$v.valid$ を 1 減らす．$v.state\_array[j] = \text{UP\_VALID}$ を満たすすべてのプロセス $j(\neq v)$ に対して，データ d を載せた update 要求を，順序番号を付与して送信する（図 4.9（B））．

(4) invalidate 要求を受信した各プロセス j は，$j.state$ を INVALID に更新し，$j.probable$ を v に更新したうえで，プロセス v に対して invalidate 応答を送信する（図 4.9（C））．

(5) update 要求を受信した各プロセス j は，受信したデータ d に基づいて $j.buffer$ を更新し，$j.probable$ を v に更新したうえで，プロセス v に対して update 応答を送信する（図 4.9（C））．

(6) プロセス v は，先ほど発行したすべての invalidate 要求と update 要求に対する invalidate 応答と update 応答を回収したあと，$v.owner$ を TRUE に更新し，再度オーナーに戻る．そして，プロセス i に対して，write 応答を順序番号を付与したうえで送信する（図 4.9（D））．なお，先ほど $v.owner$ を FALSE に変えてからここで再度 TRUE に戻すまで，一時的に系内にはオーナーが存在しない状態になる．この期間にプロセス v に届いたメッセージは，プロセス v が再びオーナーに確定するまでの間，$v.probable(= v)$ へとフォワーディングされ続ける．ここで一時的にオーナーを放棄する理由は，いま処理している write 要求の処理が完了するまでの期間は，他の read 要求や write 要求に対応しないようにするためである．

(7) write 応答を受信したプロセス i は，$i.probable$ を v に更新する（図 4.9（E））．write を完了させる．

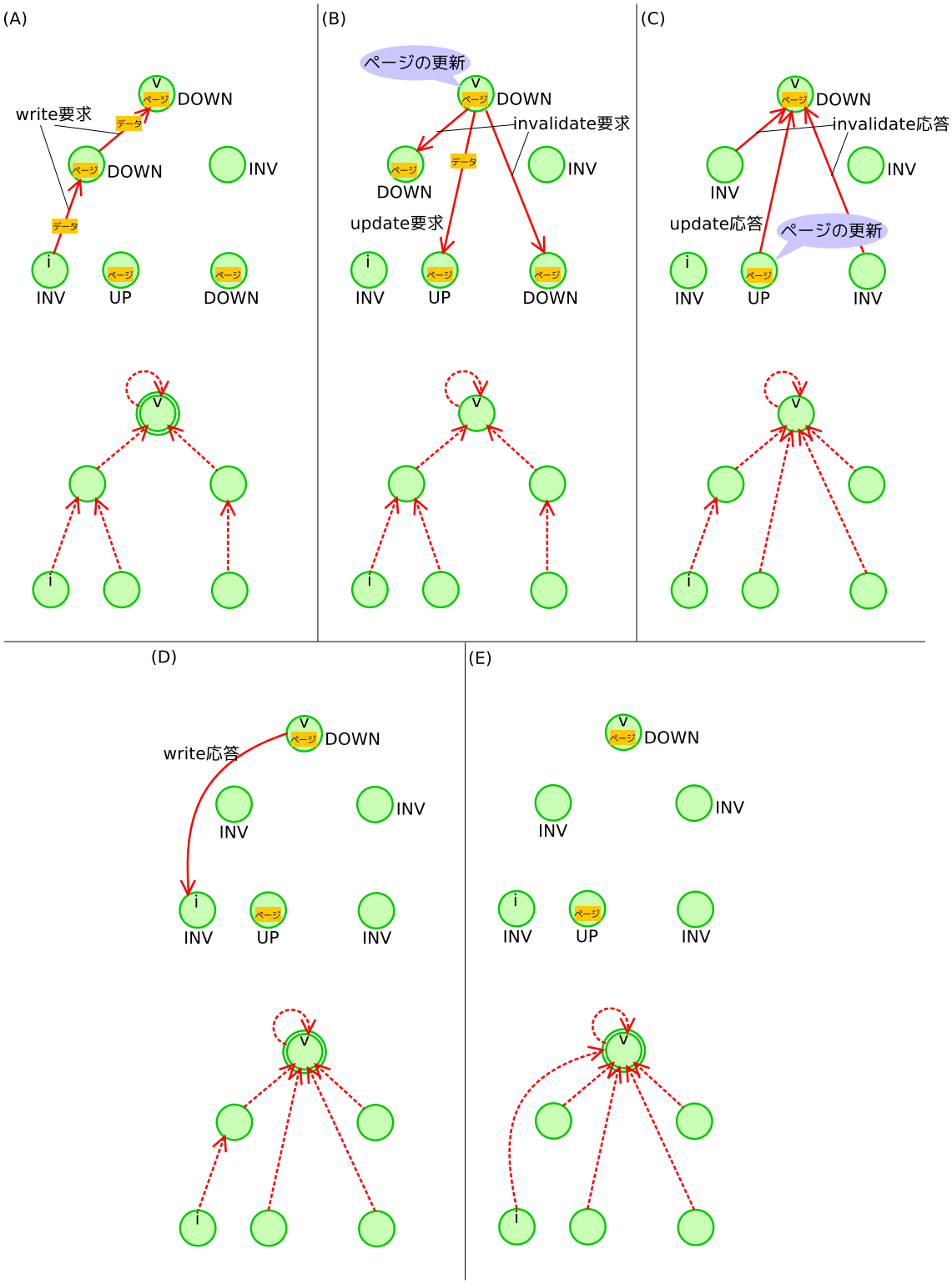

図 4.9 PUT モードの write のプロトコル.

(Ⅱ) モードが EXCLUSIVE モードの場合

プロセス i がオーナーでない場合には以下の（1）〜（7）の手順を行う．プロセス i 自身がオーナーである場合には以下の（6）〜（7）の手順のみを行う：

(1) プロセス i は，オーナー略奪要求をオーナー v に対して送信する（図 4.10（A））．
(2) オーナー略奪要求を受信したオーナー v は，自分自身 v に対して，新オーナーである i の値を載せたオーナー変更要求を，順序番号を付与したうえで送信する（図 4.10（B））．このオーナー変更要求は $v.probable$ を i に更新するためであるが，ここですぐに $v.probable$ を i に更新せず，わざわざ自分自身に対してオーナー変更要求を送信するのは，条件 4 を守るためである．つまり，$probable$ の値は，オーナーからのメッセージを受信したタイミングでしか更新できないためである．実際に，ここでオーナー変更要求を送信することなく，すぐに $v.probable$ を i に更新してしまうとオーナー追跡グラフの正しさを保証できなくなる．
(3) オーナー変更要求を受信したプロセス v は，$v.probable$ を i に更新する（図 4.10（C））．このときオーナー変更要求に対する応答を送信する必要はない．
(4) オーナー v は，$v.owner$ を FALSE に更新し，オーナーを放棄する．プロセス i に対して，配列 $v.state\_array$ と配列 $v.seq\_array$ と $v.valid$ を載せたオーナー略奪応答を，順序番号を付与したうえで送信する．このとき $v.state\_array[i] =$ INVALID ならば，プロセス i は最新ページを持っていないため，$v.state\_array[i]$ を DOWN_VALID に更新し，$v.valid$ を 1 増やしたうえで，オーナー略奪応答に最新ページも載せる（図 4.10（B））．
(5) オーナー略奪応答を受信したプロセス i は，配列 $i.state\_array$ と配列 $i.seq\_array$ と $i.valid$ を，それぞれ，受信した配列 $v.state\_array$ と配列 $v.seq\_array$ と $v.valid$ に更新する．このときオーナー略奪応答に最新ページが載っていれば $i.buffer$ に最新ページを格納する．$i.probable$ を i に更新する（図 4.10（C））．
(6) プロセス i は，書き込むべきデータに基づいて $i.buffer$ を更新する．プロセス i は，$i.owner$ を FALSE に更新し，一時的にオーナーを放棄する．以降は，(I) の EXCLUSIVE モードの場合と同様にして，invalidate 要求と update 要求を送信し，それらに対するすべての応答を回収する（図 4.10（D））．
(7) プロセス i は，これらの invalidate 要求または update 要求に対するすべての invalidate 応答または update 応答を回収したあと，$i.owner$ を TRUE に更新してオーナーになる（図 4.10（E））．

4.2.4.4 ページの追い出し

プロセス i におけるページの追い出しの目標は，$i.state$ を INVALID に変更して，$i.buffer$ のメモリを解放することである．オーナー v では各プロセスのキャッシュの状態を一括管理しているため，プロセス i は，オーナーに通知することなく，勝手に $i.state$ を INVALID に変更したり $i.buffer$ のメモリを解放することは許されない．条件 4 より，$i.state$ や $i.buffer$ の値を更新できるのは，それを指示するオーナーからのメッセージを受信したときにかぎられる．

プロセス i でページの追い出しが発行されたとき，$i.state =$ INVALID ならば，すでに目標は達成さ

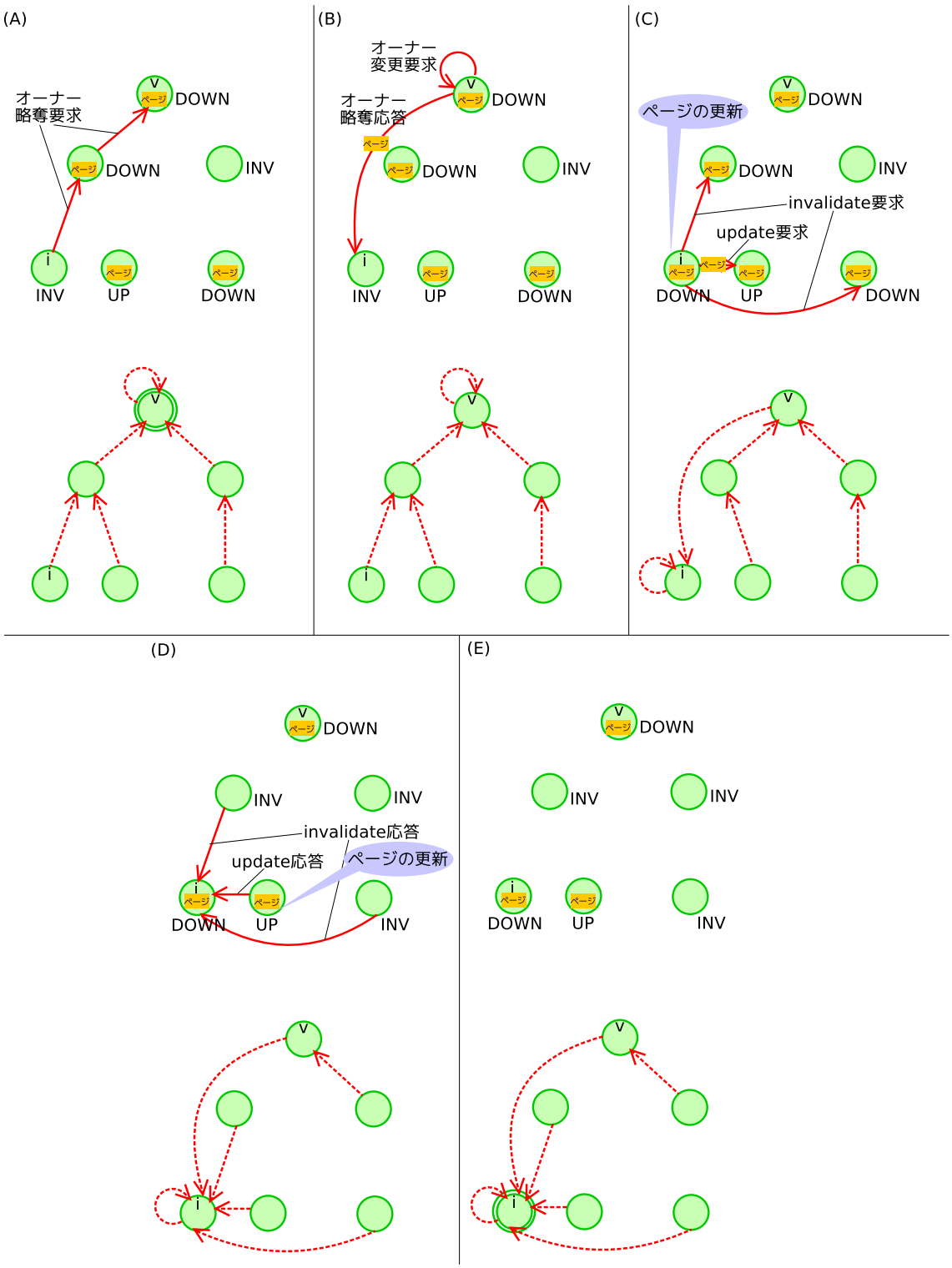

図 4.10 EXCLUSIVE モードの write のプロトコル.

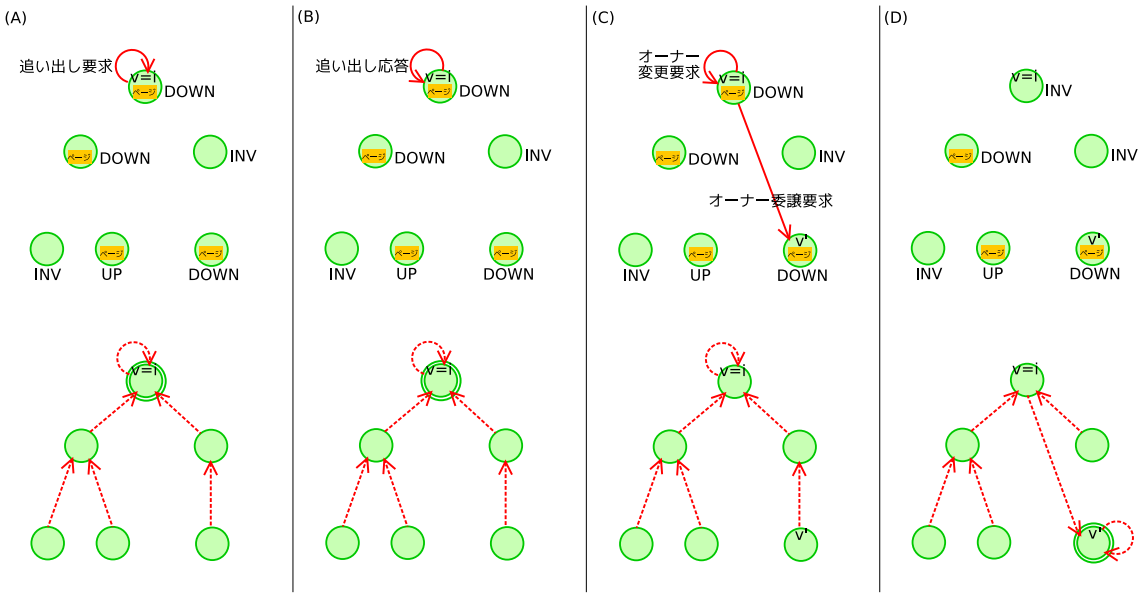

図 4.11 ページの追い出しのプロトコル.

れているため，何の処理も発生せず追い出しが完了する．それ以外の場合には追い出し要求がオーナー宛に送信される（図 4.11（A））.

(1) 追い出し要求を受信したオーナー v は，$v.state\_array[i] = \text{DOWN\_VALID}$ または $v.state\_array[i] = \text{UP\_VALID}$ ならば $v.state\_array[i]$ を INVALID に更新し，$v.valid$ を 1 減らす．オーナー v はプロセス i に対して，追い出し応答を，順序番号を付与したうえで送信する（図 4.11（B））．この時点で $i \neq v$ ならば，(3) 以降は行わない．$i = v$ ならば，この追い出し応答によってプロセス $i(= v)$ の状態を INVALID に変更するだけでは不十分であり，オーナー権を他のプロセスに追い出す必要もあるため，(3) 以降のプロトコルも実行する．

(2) 追い出し応答を受信したプロセス i は，$i.state$ を INVALID に更新し，$i.probable$ を v に更新する（図 4.11（D））.

(3) オーナー v は，新オーナー v' を適当に選択する．オーナー v は，自分自身 v に対して，新オーナーである v' の値を載せたオーナー変更要求を，順序番号を付与したうえで送信する（図 4.11（C））．このオーナー変更要求は $v.probable$ を v' に更新するためであるが，ここですぐに $v.probable$ を v' に更新せず，わざわざ自分自身に対してオーナー変更要求を送信するのは，条件 4 を守るためである．

(4) オーナー変更要求を受信したプロセス v は，$v.probable$ を v' に更新する（図 4.11（D））．このときオーナー変更要求に対する応答を送信する必要はない．

(5) オーナー v は，$v.owner$ を FALSE に更新し，オーナーを放棄する．オーナー v は新オーナー v' に対して，配列 $v.state\_array$ と配列 $v.seq\_array$ と $v.valid$ を載せたオーナー委譲要求を，

順序番号を付与したうえで送信する．このとき $v.state\_array[v'] = \text{INVALID}$ ならば，新オーナー v' は最新ページを持っていないため，$v.state\_array[v']$ を DOWN_VALID に更新し，$v.valid$ を 1 増やしたうえで，オーナー委譲要求に最新ページも載せる（図 4.11（C））．ここで，新オーナー v' としては v 以外のいずれのプロセスを選んでもプロトコル上は問題ないが，オーナー委譲要求における最新ページの転送を省略するためには，すでに最新ページを持っているプロセスのいずれかを新オーナー v' として選ぶのが望ましい．

(6) オーナー委譲要求を受信した新オーナー v' は，配列 $v'.state\_array$ と配列 $v'.seq\_array$ と $v'.valid$ を，それぞれ，受信した配列 $v.state\_array$ と配列 $v.seq\_array$ と $v.valid$ に更新する．このときオーナー委譲要求に最新ページが載っていれば $v'.buffer$ に最新ページを格納する．$v'.owner$ を TRUE に更新してオーナーになり，$v'.probable$ を v' に更新する（図 4.11（D））．このとき，オーナー委譲要求に対する応答を送信する必要はない．

なお，適切な排他制御を行うことで，解放作業中のグローバルアドレス空間のページの追い出しが起きないようにしている．

4.2.4.5 ユーザ定義の read-modify-write

3.5.2 節で述べたユーザ定義の read-modify-write を実現するプロトコルは，選択的キャッシュ write と同様である．選択的キャッシュ write においてデータを write するかわりに，ユーザプログラム内に定義された DMI_function() 関数を呼び出せばよい．

4.2.4.6 アドレスの変更監視

3.5.3 節で述べた DMI_watch() 関数では，あるアドレス a のデータがあるデータ d から別のデータに変更されるまで待機することができる．この操作は以下のプロトコルによって実現できる：

(1) まず，UPDATE モードで read を呼び出す．read した結果，DMI_watch() 関数で指示されたアドレス a のデータがデータ d と異なっていれば，アドレスの変更監視を終了する．異なっていない場合，オーナーから update 要求を受信するまで待機する．ここでは，UPDATE モードで read しているため，そのアドレス a を含むページに更新があるたびにオーナーからの update 要求を受信できることに注意する．

(2) オーナーからの update 要求を受信するたびに，アドレス a のデータがデータ d と異なっているかどうかを検査し，異なっていればアドレスの変更監視を終了する．異なっていなければ，オーナーから次の update 要求を受信するまで再度待機する．

(3) ページ置換によるページの追い出しなどが原因で，アドレス a を含むページに関してオーナーから invalidate 要求を受信することがある．invalidate 要求を受信した場合には，再度 UPDATE モードで read を呼び出したうえで，オーナーから update 要求を受信するまで待機する．

4.2.4.7 オーナー追跡グラフの正しさの証明

以上で述べたプロトコルにおいて，オーナー追跡グラフの正しさが保証されていることを証明する．すなわち，「任意のプロセス x から開始して，各プロセスの probable owner をたどっていけば，有限時間内にかならず真のオーナーに到達できる」という性質がつねに成り立つことを証明する．ここでは，

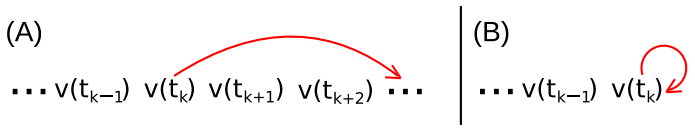

図 4.12 歴代のオーナー系列とオーナー追跡グラフとの関係．（A）$v(t_k)$ より新しいオーナーが存在する場合，（B）$v(t_k)$ より新しいオーナーが存在しない場合．

各プロセス間の通信時間が有限時間であることと，オーナー追跡グラフによって真のオーナーを追跡する速度よりもオーナーの移動速度が遅いことを仮定する．

プロセス x が $x' = x.probable$ へとメッセージを送信し，そのメッセージがプロセス x' に受信された時点におけるプロセス x' の状況を考えると，プロセス x' がオーナーである場合とプロセス x' がオーナーではない場合に場合分けできる．まず，プロセス x' がオーナーである場合には，明らかに，有限時間内にメッセージを真のオーナーに届けられたことになるので，題意は示せている．次に，プロセス x' がオーナーではない場合を考える．プロトコルの性質上，プロセス x' がいずれかのプロセス（いまの場合プロセス x）の *probable* の値になっているということは，プロセス x' がオーナーであった時刻が過去に存在することを意味する．ここで，プログラムの実行が開始してから，オーナーが移動して新しいオーナーが確定した時刻たちを t_0，t_1，t_2，... とおき，時刻 t_i において確定したオーナーを $v(t_i)$ と表すことにする．すると，プログラムの実行開始からの歴代のオーナー系列は，$v(t_0) \to v(t_1) \to v(t_2) \to \ldots$ と表せる．さらに，もっとも直近に x' がオーナーに確定した時刻を t_k とおけば，歴代のオーナー系列は，

$$v(t_0) \to v(t_1) \to \ldots \to v(t_{k-1}) \to v(t_k)(= x') \to v(t_{k+1}) \to \ldots (\forall j > k : v_j \neq x')$$

と表せる．

このとき，（I）$v(t_{k+1})$ が存在しない場合と（II）存在する場合の 2 とおりが存在する．第 1 に，（I）$v(t_{k+1})$ が存在する場合には，$v(t_k)(= x')$ よりも新しいオーナーが存在することになる．プロトコルの性質上，オーナー y がオーナーではなくなるときには，かならず新しいオーナーを $y.probable$ に代入することに注意すると，「過去のいずれかの時点で $v(t_k)(= x')$ がオーナーだった」かつ「現在は $v(t_k)(= x')$ よりも新しいオーナーが存在する」ということは，現在の $v(t_k).probable(= x'.probable)$ は，$v(t_{k+1})$，$v(t_{k+2})$，$v(t_{k+3})$，... のうちのいずれかの値になっている（図 4.12（A））．したがって，プロセス x' がプロセス x から届いたメッセージを $x'.probable$ にフォワーディングすることによって，より新しいオーナーへとメッセージをフォワーディングすることができる．

第 2 に，（II）$v(t_{k+1})$ が存在しない場合には，$v(t_k)(= x')$ がもっとも直近のオーナーであることを意味するが，そもそもいまは $v(t_k)(= x')$ がオーナーではない状況を考えている．すなわち，この状況は「$v(t_k)(= x')$ がもっとも直近のオーナーであるにもかかわらず，$v(t_k)(= x')$ がオーナーではない」状況である．具体的には，オーナーが write フォルトを処理するために一時的に *owner* を FALSE に変えている期間においてのみ，この状況が発生する．そして，write フォルトの場合のプロトコルを観察すると，この状況では $v(t_k).probable(= x'.probable)$ の値は $v(t_k)$ 自身になっており，かならず有限

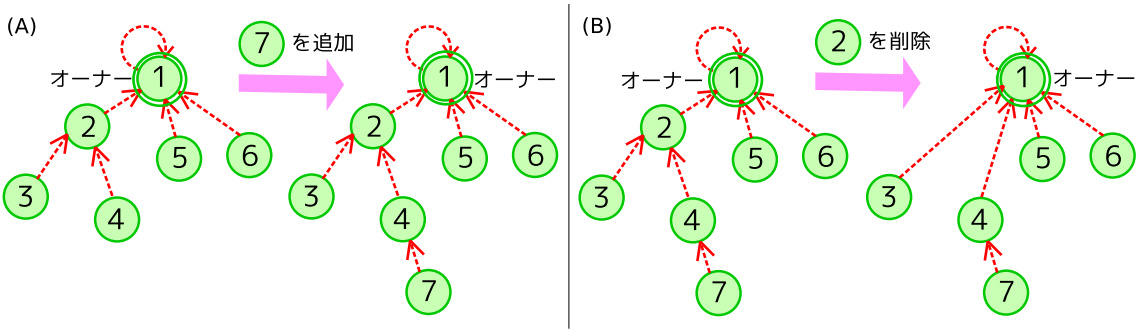

図 4.13 プロセスの参加/脱退にともなうオーナー追跡グラフの再形成.（A）プロセスの参加,（B）プロセスの脱退.

時間後には $v(t_k)(= x')$ がオーナーに確定することがわかる（図 4.12（B））. したがって，プロセス x' がプロセス x から届いたメッセージを $x'.probable(= x')$ にフォワーディングすることによって，有限時間後には，「オーナーに確定した x' にメッセージを届けることができる」か，または「x' にメッセージを届けることができるが，その時点ではすでに x' はオーナーではなくなっている」. 前者の場合には，有限時間でメッセージをオーナーに届けられたことになるので，題意は示せている. 後者の場合には，x' よりも新しいオーナーが存在していることを意味するため，結局（I）の状態に帰着したことになる. そして，先ほどの議論より，（I）の場合には，プロセス x' はプロセス x から届いたメッセージを $x'.probable$ にフォワーディングすることによって，より新しいオーナーへとメッセージをフォワーディングすることができる.

以上の議論をまとめると，プロセス x からのメッセージがプロセス x' に届いた時点でプロセス x' がオーナーではない場合には，プロセス x' がプロセス x から届いたメッセージを $x'.probable$ にフォワーディングすることによって，「有限時間後に x' がオーナーに確定し，メッセージを x' に届けることができる」か，または「より新しいオーナーへとメッセージをフォワーディングすることができる」. したがって，オーナー追跡グラフによって真のオーナーを追跡する速度よりもオーナーの移動速度が遅いことを仮定するならば，メッセージはかならず有限時間後に真のオーナーに到達できることになる. よって，題意は示された.

4.2.5 非同期的なプロセスの参加/脱退

4.2.5.1 プロセスの参加

プロセスの参加は，実行中の任意のプロセス 1 個をブートストラップとして実現される. プロセス i がプロセス j をブートストラップとして参加する手順は以下のとおりである：

(1) プロセス i は，プロセス j に参加要求を送信する.
(2) 参加要求を受理したプロセス j はグローバルロックを取得する.
(3) プロセス j は，プロセス i に対して，すべてのプロセスの情報とすべてのページに対する $j.probable$ を送信する.

(4) プロセス i は，すべてのページを割り当て，すべてのページに関して $i.owner$ を FALSE に，$i.probable$ **を** $j.probable$ **に，**$i.state$ **を INVALID に初期化する**．このように設定するだけで，プロセス i がはじめてページに read/write した時点で自然と read/write フォルトが発生し，前節で述べたプロトコルが動作するようになる．このように，**グローバルアドレス空間に対する参加は，read/write フォルトの仕組みを利用してきわめて自然な形で実現できる**．なお，ここでは $i.probable$ を $j.probable$ に初期化しているが，$i.probable$ には実行中の任意のプロセスを設定したとしてもオーナー追跡グラフの正しさは維持される（図 4.13（A））．
(5) プロセス i は，すべてのプロセスとの接続を確立し，必要な初期化処理を行う．
(6) プロセス i は，グローバルロックを解放する．

4.2.5.2 プロセスの脱退

プロセス i が脱退する手順は以下のとおりである：

(1) プロセス i は，グローバルロックを取得する．
(2) プロセス i は，プロセス i に対するページの追い出しを禁止するよう，すべてのプロセスに対して送信する．
(3) プロセス i は，プロセス i のメモリプールに含まれている**すべてのページの追い出し**を行う．
(4) プロセス i は，すべてのプロセスに対して，すべてのページに関する $i.probable$ を送信する．これを受信した各プロセス k は，$k.probable = i$ **であるようなページすべてに関して，**$k.probable$ **を** $i.probable$ **に更新**する．この更新により，すべてのページのオーナー追跡グラフが，プロセス i を含まないオーナー追跡グラフへと再形成される（図 4.13(B)）．
(5) プロセス i は，すべてのプロセスとの接続を切断し，終了処理を行う．
(6) プロセス i は，まだ実行中の適当なプロセス j に対して脱退要求を送信する．
(7) 脱退要求を受信したプロセス j は，グローバルロックを解放する．

4.3 ページ置換

DMI では，計算スレッドとは独立に走っている sweeper スレッドがつねにメモリプールの使用量を監視しており，一定量を超過した時点で他のプロセスに対してページの追い出しを行う．ページ置換にあたっては，どのページをどのプロセスに対して追い出すべきかが鍵となる．

第 1 に，どのページを追い出すべきかを考える．あるプロセス i からページを追い出すとは，$i.buffer$ のために消費されているメモリを解放することである．よって，**ページサイズが大きく，かつ追い出しのための負荷が小さいページから追い出す**のが効率的である．ここで，**4.2.4.4** 節で述べたページの追い出しのプロトコルを観察すると，各プロセス i のページの状態に応じて，追い出しのための負荷は以下の順に大きくなることがわかる：

(1)（そもそも追い出す必要はないが）INVALID 状態のページ
(2) DOWN_VALID 状態または UP_VALID 状態であり，かつプロセス i がオーナーではないよう

なページ

(3) プロセス i がオーナーであり，かつプロセス i 以外に DOWN_VALID 状態または UP_VALID 状態にあるプロセスが存在するようなページ

(4) プロセス i がオーナーであり，かつプロセス i 以外には DOWN_VALID 状態または UP_VALID 状態にあるプロセスが存在しないようなページ

したがって DMI では，(2) → (3) → (4) の優先度順で，ページサイズの大きい順にページを追い出す．なお，ユーザプログラム側の事情により性能上追い出されたくないページがある場合には，DMI_save(int64_t *addr*, int64_t *size*) 関数を呼び出すことで，グローバルアドレス領域 [*addr*, *addr*+*size*) に属するページを追い出し対象から外すことができる．逆に，指定したグローバルアドレス領域を追い出し対象に含めるための DMI_unsave(int64_t *addr*, int64_t *size*) 関数も存在する．

第 2 に，どのプロセスに対して追い出すべきかを考える．(1) と (2) と (3) の場合には，オーナーとのやりとりの結果として，単にプロセス i のページの状態が INVALID に変化してメモリが解放されるだけなので，そもそも追い出し先のプロセスという概念が存在しない．追い出し先のプロセスが問題になるのは，追い出しにともなって最新ページの転送が必要となる (4) の場合である．仮にすでにメモリプールが飽和状態に近いプロセスに対してページを追い出せば，追い出し先のプロセスでも再度追い出しが発生し，結果的にプロセス間でページ置換がスラッシングを起こす可能性がある．よって，追い出し先としては，できるかぎりメモリプールに空きが多いプロセスを選択することが重要である．そこで，グローバルアドレス空間のコヒーレンシ維持のためにプロセス間をつねに飛び交っているさまざまなメッセージに対して，各メッセージの送信元プロセスのメモリプールの使用量の情報を載せることで，すべてのプロセスがすべてのプロセスのメモリプールの使用量を gossip-based におおまかに把握できるようにし，この情報に基づいて追い出し先のプロセスを選択するのが望ましいと考えられる．ただし，現状の実装では，単純に乱数によって追い出し先のプロセスを決定している．

なお，各プロセスが提供するメモリプールの容量 l は各プロセスの生成時に指定可能であるが，この値 l は，あくまでも sweeper スレッドの動作タイミングを決定するためのパラメータにすぎない．具体的には，sweeper スレッドは，メモリプールの使用量が l を超過した時点で，合計 $0.7l$ 程度のページを追い出そうと試みるが，性能上の理由から，l を超えたからといってユーザプログラムを一時的に停止させるなどの処理は行わない．つまり，使用量がつねに l 以下になることを保証するわけではない．したがって，原理的には，すべてのプロセスを通じたメモリプールの総容量を超えるグローバルアドレス空間を確保して利用することも可能ではある．ただし，その場合にはプロセス間で頻繁なスラッシングが発生して著しい性能低下が起きる．

4.4 データ転送の動的負荷分散

3.4.2 節で述べたように，DMI では，ページ転送のための負荷がオーナーに集中した場合，オーナーは，そのページをキャッシュしているプロセスに対してページ転送を依頼することで，ページ転送の負荷を動的に負荷分散させる．このときオーナーがページ p の転送を依頼するプロセスは，以下のルール

```
01: when a global address space M is allocated:
02:     p.time := 0 for all pages p ∈ M
03:     p.N := {(the owner of the page p)} for all pages p ∈ M
04:     p.stamp[i] := 0 for all processes i, for all pages p ∈ M
05:
06: when the receiver thread of a process v receives a message m:
07:     v.l := v.l+(the estimated amount of the data transfer
        that the message m will request from the process v)
08:     put the message m into a FIFO queue v.q
09:
10: when the handler thread of a process v gets a message m from a FIFO queue v.q:
11:     v.l := v.l−(the estimated amount of the data transfer of the message m)
12:     i := the source process of the message m
13:     p := the page that the message m requests
14:     s := the size of the page p
15:     if the process v is the owner of the page p
          and the page p has to be transfered to the process i then
16:       if s > T_s and v.l > T_l then
17:           select j s.t. p.stamp[j] is minimum for ∀j ∈ p.N
18:           p.stamp[j] := p.time
19:           p.stamp[i] := p.time + 1
20:           p.time := p.time + 2
21:           delegate to the process j a transfer of the page p to the process i
22:       else
23:           p.stamp[i] := p.time
24:           p.time := p.time + 1
25:           transfer the page p to the process i directly
26:       endif
27:       p.N := p.N ∪ {i}
28:     else
29:       handle the message m normally
30:     endif
31:     if the process v is the owner of the page p
          and is going to invalidate the cached page p on the process i then
32:           p.N := p.N\{i}
33:     endif
```

図 4.14 ページ転送の動的負荷分散を実現するアルゴリズム.

によって選択する：

ルール 1 ページ p をキャッシュしているプロセス（オーナー自身も含む）のうち，「もっとも過去にページ転送を依頼したプロセス」に対してページ転送を依頼する.

ルール 2 ただし，便宜上，ページ p を転送された直後のプロセスは，ページ転送が完了した時点でページ転送を依頼されたものと見なす.

上記のルールは，図 4.14 に示すアルゴリズムで実現できる.

4.1 節で述べたように，DMI の各プロセスには receiver スレッドと handler スレッドが存在する．また，各プロセス v には，そのプロセスの負荷を表す変数 $v.l$ と，処理待ちのメッセージを貯めるための FIFO なキュー $v.q$ が存在する．プロセス v に届いたメッセージ m はまず receiver スレッドによっ

て受信され，そのメッセージ m が引き起こすであろうデータ転送量が大まかに見積もられる（7 行目）．たとえば，メッセージ m がプロセス i で発生したページ p の read フォルトに起因するページ要求であるとする．このとき，プロセス v がページ p のオーナーであれば，プロセス v はメッセージ m を処理する際にページ p をプロセス i に転送する必要があるため，メッセージ m が引き起こすであろうデータ転送量の見積りはページ p のページサイズになる．一方で，プロセス v がオーナーでなければ，プロセス v はオーナー追跡グラフに沿ってメッセージ m をフォワーディングするだけでよいので，メッセージ m が引き起こすであろうデータ転送量の見積りは 0 になる．見積りが完了したら，receiver スレッドはその値を $v.l$ にアトミックに加算したあとで（7 行目），メッセージ m を FIFO なキュー $v.q$ に入れる（8 行目）．

これに対して，handler スレッドは，キュー $v.q$ のなかのメッセージを 1 個ずつ FIFO に処理する作業を繰り返す．このとき handler スレッドは，キュー $v.q$ からメッセージ m をとり出したあと，メッセージ m の処理を始める前に，receiver スレッドがメッセージ m をキューに入れる際に見積もったデータ転送量を $v.l$ からアトミックに減算する（11 行目）．このように $v.l$ を管理することで，$v.l$ は，「現在キューに貯まっているメッセージが引き起こすであろうデータ転送量の総和の予測値」になる．したがって，$v.l$ の値の大小によって，プロセス v に要求されているデータ転送の負荷集中の度合いを判断することができる．ただし，$v.l$ の値はあくまでも予測値であって正確なものではない．たとえば，receiver スレッドがメッセージ m をキューに入れる時点と，handler スレッドがメッセージ m をキューからとり出して処理する時点では，オーナーが変化している可能性があり，receiver スレッドが必要だと判断したページ転送が，handler スレッドが処理する時点では不要になっている場合などがある．

プロセス v にデータ転送の負荷が集中した場合，プロセス v はページ転送の動的な負荷分散を試みる．ここで，プロセス v はページ p のオーナーであり，メッセージ m はプロセス i に対してページ p を転送することを要求しているとする．また，ページ p のページサイズを s とする．handler スレッドがメッセージ m をキュー $v.q$ からとり出したとき，あらかじめ設定した閾値 T_s と T_l に対して，$s > T_s$ かつ $v.l > T_l$ となっていれば（16 行目），オーナー v は，プロセス i に対するページ転送を，すでにページ p をキャッシュしているいずれかのプロセスに対して依頼する．ここで，依頼対象のプロセスとして「もっとも過去にページ転送を依頼したプロセス」を選択するために，ページ p のオーナー v は以下の 3 種類のデータを管理する：

$p.N$　ページ p をキャッシュしているプロセスの集合[*2]．

$p.time$　ページ p の現在時刻．

$p.stamp[i]$　オーナーが，もっとも直近にページ p のページ転送をプロセス $i \in p.N$ に対して依頼した時点におけるページ p の時刻．

[*2] **4.2.2** 節で述べたデータ構造に即していえば，$v.state\_array$ のことである．

オーナー v はかならずページ p をメモリプールに保持しているため，$v \in p.N$ が成り立つ．$p.time$ はページ p が確保されたときに 0 に初期化され（2 行目），プロセスに対してページ転送を行うたびに時刻を進める．

$s > T_s$ かつ $v.l > T_l$ が成り立たず，オーナー v がプロセス i に対して直接ページ p を転送する場合には（22 行目），オーナー v は $p.stamp[i]$ を $p.time$ に更新したうえで $p.time$ を 1 進める．これはルール 2 に基づく処理である．一方で，$s > T_s$ かつ $v.l > T_l$ が成り立つ場合には（16 行目），オーナー v は，すべての $j \in p.N$ のうち $p.stamp[j]$ が最小となる j' を選択し，プロセス j' に対してページ転送を依頼する．これはルール 1 に基づく処理で，「もっとも過去にページ転送を依頼したプロセス」を選択することに相当する．このとき $j'.stamp$ を $p.time$ に，$p.stamp[i]$ を $p.time + 1$ に更新したうえで $p.time$ を 2 進める．これはルール 2 に基づく処理である．最後に，オーナー v はプロセス i を $p.N$ に加える（27 行目）．なお，$j' = v$ となる場合もあるが，この場合には，オーナー v がプロセス i に対して直接ページ転送を行うことになる．とくに，ページ p をキャッシュしているプロセスがオーナー v 以外に存在しないならば，$p.N = \{v\}$ となるので，かならず $j' = v$ となる．また，オーナーを移動する場合には，$p.N$，$p.time$，各 $p.stamp[i]$ の値を新しいオーナーに丸ごとコピーする．これにより，オーナーの移動にかかわらず時刻の連続性が保たれる．

なお，DMI のデフォルトでは，$T_s = 16$ KB，$T_l = 512$ KB としている．

4.5 排他制御

DMI では，ユーザ定義の read-modify-write とアドレスの変更監視を提供しているため，原理的には，プログラマはこれらを組み合わせることで多様な同期を実現できる．しかし，実際のユーザプログラムを記述するうえでは，より高抽象度な同期機構を利用できた方が便利なため，pthread と同様のセマンティクスの排他制御（mutex）や条件変数（cond），図 3.8 に示した Allreduce などを API として提供している．なかでも DMI では，既存の PGAS 処理系や分散共有メモリと比較して特徴的な排他制御を実装しているため，本節では排他制御の実装について述べる．

4.5.1 実装方針の検討

まず，分散メモリ環境における排他制御に関する既存研究を観察したうえで，DMI に適した実装方針について議論する．一般に，分散メモリ環境における排他制御のアルゴリズムは，メッセージパッシングベース（send/receive）のアルゴリズムと，グローバルアドレス空間を利用する共有メモリベース（read/write）のアルゴリズムに大きく分類できる．

4.5.1.1 メッセージパッシングベースの排他制御

メッセージパッシングベースの排他制御は，プロセスたちがメッセージを send/receive することによってプロセスどうしのコンセンサスをとって排他制御を実現するもので，permission-based なアルゴリズム，token-based なアルゴリズム，server-based なアルゴリズムの 3 つに大別できる [119, 167]．

第 1 に，permission-based なアルゴリズムでは，クリティカルセクションに突入しようとするプロセス i は適当なプロセス集合に対して排他要求を送信する．排他要求を受信した各プロセス j は，プロセ

ス i がクリティカルセクションに突入することを許可するならばプロセス i に対して排他許可を送信し，（まさにその瞬間にプロセス j もクリティカルセクションに突入しようとしているなどの理由で）許可できないのであればプロセス i の排他要求を保留したり拒否したりする．そして，プロセス i は一定数以上のプロセスたちから排他許可を受けとることができた場合に，クリティカルセクションに突入できる．以上が permission-based なアルゴリズムの基本アイディアであり，アルゴリズムによって，1 回のクリティカルセクションにいたるまでに必要となる排他要求や排他許可などのメッセージ数が異なる．具体的には，プロセス数を m としたとき，各クリティカルセクションあたり，$O(3(m-1))$ のメッセージ数を要する Lamport のアルゴリズム [109]，$O(2(m-1))$ のメッセージ数を要する Ricart Agrawala のアルゴリズム [159]，$O(\sqrt{m}\,)$ のメッセージ数を要する Maekawa のアルゴリズム [125] などがある．

第 2 に，token-based なアルゴリズムでは，系内に token が 1 個だけ存在し，token を取得したプロセスだけがクリティカルセクションに突入できる．よって，クリティカルセクションに突入しようとするプロセスは，まず token を所有するプロセスに対して token 要求を送信する．token 要求を送信したプロセスたちは，token に関連づけられたキューなどのデータ構造で管理されており，各クリティカルセクションが終わるたびに，次に token を譲るべきプロセスが 1 個選択されて，そのプロセスに token が譲られる [137]．具体例としては，各クリティカルセクションあたり，平均メッセージ数が $O(\log m)$ の Naimi らのアルゴリズム [138]，平均メッセージ数は $O(\log m)$ であるがコンテンションが高い場合には $O(1)$ で済む Raymond のアルゴリズム [155]，Raymond のアルゴリズムにおいてコンテンションが低い場合の挙動を改善した Neilsen Mizuno のアルゴリズム [119] などがある．計算量からわかるように，一般に，token-based なアルゴリズムは permission-based なアルゴリズムよりもメッセージ数が少なくて済むが，上記の 3 つの token-based なアルゴリズム間の優劣は，プロセス数やコンテンションの程度に大きく左右されることが指摘されている [167, 96]．分散共有メモリ処理系では，DSM-Threads などが token-based なアルゴリズムで排他制御を実現している．

第 3 に，server-based なアルゴリズムでは，特定の 1 個のプロセスに排他制御のサーバを担当させる．クリティカルセクションに突入しようとするプロセスは，サーバに対して排他要求を送信し，サーバから排他許可を受信できたときにクリティカルセクションに突入できる．server-based なアルゴリズムでは，サーバにおいてすべての排他要求をシリアライズできるため，permission-based なアルゴリズムや token-based なアルゴリズムなどの高度な分散アルゴリズムは必要なく，実装が非常に容易である．具体例としては，SMS などが server-based なアルゴリズムで排他制御を実装している．著者の知るかぎり，UPC，X10，Chapel などの多くの PGAS 処理系や分散共有メモリ処理系では排他制御の実装については言及されていないが，これらの処理系では，単純な server-based なアルゴリズムが実装されているのではないかと思われる．

以上の観察をふまえ，DMI の排他制御を実装するうえでメッセージパッシングベースの排他制御が適切かどうかを議論する．まず，プロセスを参加/脱退させる必要がある DMI では固定的なプロセスを設置できないため，server-based なアルゴリズムは採用できない．よって，permission-based なアルゴリズムまたは token-based なアルゴリズムのいずれかが候補となるが，メッセージ数の計算量の観点からは token-based なアルゴリズムが望ましいと考えられる．ところが，新たに token を導入す

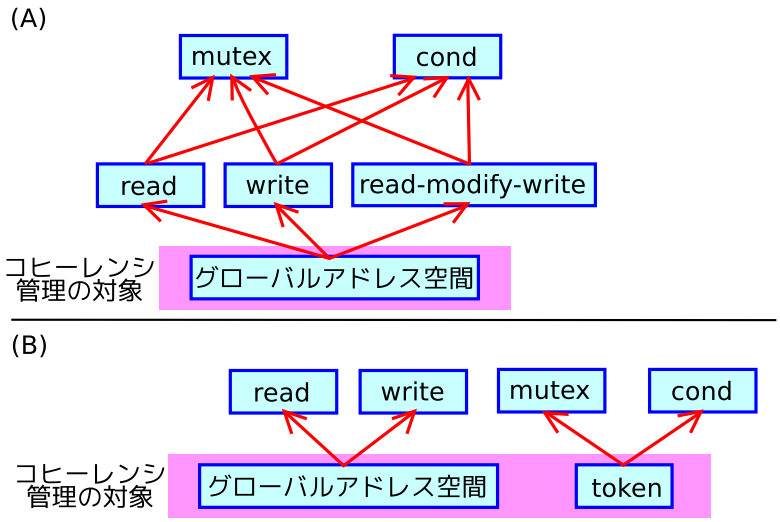

図 4.15 同期プリミティブの階層関係．（A）read/write と read-modify-write を組み合わせて同期を実現する場合，（B）read/write と token を組み合わせて同期を実現する場合．

```
while true do
    Noncritical Section
    Entry Section
    Critical Section
    Exit Section
endwhile
```

図 4.16 共有メモリベースの排他制御における実行モデル．

るということは，グローバルアドレス空間のコヒーレンシプロトコルとは別に，token に対しても別途コヒーレンシプロトコルが必要になることを意味する．たとえば，新たに参加するプロセスに対して token の所在をどのように教えるべきか，脱退するプロセスが token を保持していた場合にその token をどのように追い出すべきかなどのプロトコルを検討する必要があり，**4.2** 節で説明したグローバルアドレス空間のコヒーレンシプロトコルに似た複雑なコヒーレンシプロトコルを，token に対しても正しく実装する必要が生じてしまう．このように，コヒーレンシ管理の対象を増やすことは実装を煩雑化させるため望ましくない．それよりは，すでにコヒーレントに実装してあるグローバルアドレス空間上の read/write，ユーザ定義の read-modify-write，アドレスの変更監視の同期プリミティブを組み合わせることで，共有メモリベースの排他制御を実現する方が，実装が単純で見通しがよい（図 4.15）．

4.5.2 共有メモリベースの排他制御

共有メモリベースの排他制御は，物理的な共有メモリあるいは分散メモリ環境上に作り出されたグローバルアドレス空間における，read/write および read-modify-write を組み合わせることでスレッドどうしのコンセンサスをとって排他制御を実現する．一般に，共有メモリベースの排他制御に関する研究では，図 4.16 の構造を持つコードを各スレッドが独立に実行するような実行モデルを想定し，Entry Section と Exit Section の中身をどのように設計するかが論じられる [82, 187, 75]．具体例として，この構造に沿った MCS アルゴリズムを図 4.17 に示す [75]．

しかし，pthread のロック関数/アンロック関数と同様のセマンティクスを持つ排他制御の API を実装しようとするとき，どうすれば図 4.16 の実行モデルに基づいて提案されている排他制御のアルゴリズムを適用できるかは自明ではない．この理由は，図 4.16 の実行モデルでは，Entry Section と Exit

```
struct node_t {
    int locked;
    struct node_t *next;
};

struct node_t *tail = NULL;
    /* shared to all processes */

void each_process() {
    struct node_t node, *pred, *p = &node;
    while (1) {
        NoncriticalSection();
        /* begin of the EntrySection */
        p->next = NULL;
        pred = fetch_and_store(&tail, p);
        if (pred != NULL) {
            p->locked = 1;
            pred->next = p;
            while (p->locked == 1); /* spin */
        }
        /* end of the EntrySection */
        CriticalSection();
        /* begin of the ExitSection */
        if (p->next == NULL) {
            if (!compare_and_swap(&tail, p, NULL)) {
                while (p->next == NULL); /* spin */
                p->next->locked = 0;
            }
        } else {
            p->next->locked = 0;
        }
        /* end of the ExitSection */
    }
}
```

図 4.17 MCS アルゴリズム.

```
struct vars_t {
    ...; /* necessary variables for
    both EntrySection and ExitSection */
};

struct mutex_t {
    ...; /* necessary variables
    for the mutual exclusion */
    struct vars_t *vars;
};

void lock(struct mutex_t *mutex) {
    struct vars_t *vars;
    vars = malloc(sizeof(struct vars_t));
    EntrySection();
    mutex->vars = vars;
    return;
}

void unlock(struct mutex_t *mutex) {
    struct vars_t *vars;
    vars = mutex->vars;
    ExitSection();
    free(vars);
    return;
}
```

図 4.18 Entry Section/Exit Section を pthread のロック関数/アンロック関数に分離する方法.

Section が同一のスレッドによって実行されることが前提とされているのに対して，pthread ではロック関数とアンロック関数を呼び出すスレッドがかならずしも同一ではないことに起因する．いい換えると，図 4.16 の実行モデルでは，Entry Section と Exit Section で変数を共有できることが前提とされているため，単純に，Entry Section の中身を pthread のロック関数として切り出し，Exit Section の中身を pthread のアンロック関数として切り出すだけでは機能しない．たとえば，図 4.17 の MCS アルゴリズムでは，Entry Section で利用した *node* 変数を Exit Section でも利用できるという事実がアルゴリズムを成立させるうえでの鍵になっているため，Entry Section と Exit Section を単純に別の関数に分離するとアルゴリズムが成立しなくなる．この問題の安直な解決法としては，図 4.18 に示すような方法が考えられる．すなわち，Entry Section と Exit Section とで共有したい変数を Entry Section の直前に malloc して，Entry Section のなかで利用し，Entry Section の直後にその共有変数のアド

```
01: struct mutex_t {
02:     int *head;
03:     int *next;
04:     int *p1;
05:     int *p2;
06: };
07:
08: void init(struct mutex_t *m) {
09:     m->head = NULL;
10:     m->next = NULL;
11:     m->p1 = NULL;
12:     m->p2 = NULL;
13: }
14:
15: void lock(struct mutex_t *m) {
16:     int flag;
17:     int *prev, *curr;
18:
19:     flag = 0;
20:     curr = convert(&flag, m->p1, m->p2);
        /* address conversion */
21:     prev = fetch_and_store(&m->head, curr);
22:     if (prev == NULL) {
23:         m->p1 = curr;
24:     } else {
25:         watch(&flag, 0); /* wait until flag != 0 */
26:     }
27:     m->next = prev;
28: }
29:
30: void unlock(struct mutex_t *m) {
31:     int *curr;
32:
33:     if (m->next == NULL
            || m->next == m->p1) {
34:         if (m->next == m->p1) {
35:             m->p1 = m->p2;
36:         }
37:         if (!compare_and_swap(
                &m->head, m->p1, NULL)) {
38:             m->p2 = m->head;
39:             curr = revert(m->p2);
                    /* address reversion */
40:             *curr = 1;
41:         }
42:     } else {
43:         curr = revert(m->next);
                /* address reversion */
44:         *curr = 1;
45:     }
46: }
47:
48: void destroy(struct mutex_t *m) {
49: }
50:
51: int* convert(int *curr, int *p, int *q) {
52:     int v1, v2, d;
53:
54:     v1 = (intptr_t)p & 0x3;
55:     v2 = (intptr_t)q & 0x3;
56:     d = 0;
57:     if (d == v1 || d == v2) {
58:         d = 1;
59:         if (d == v1 || d == v2) {
60:             d = 2;
61:         }
62:     }
63:     return (int*)((intptr_t)curr + d);
64: }
65:
66: int* revert(int *curr) {
67:     return (int*)((intptr_t)curr
                    - ((intptr_t)curr & 0x3));
68: }
```

図 4.19 Permission Word アルゴリズムを用いた，pthread と同様のセマンティクスを持つ init() 関数/destroy() 関数/lock() 関数/unlock() 関数の実装.

レスを *mutex* 変数のなかに保存しておく．そして，Exit Section の直前に *mutex* 変数からそのアドレスを復帰して，Exit Section のなかで利用し，Exit Section の直後に free する．しかし，この方法ではクリティカルセクションのたびに malloc/free が必要となるため重い．以上の観察をふまえると，pthread のロック関数/アンロック関数と同様のセマンティクスを持つ排他制御の API を性能よく実装するためには，**Entry Section と Exit Section との間で変数を共有する必要がないアルゴリズムが必要**である．

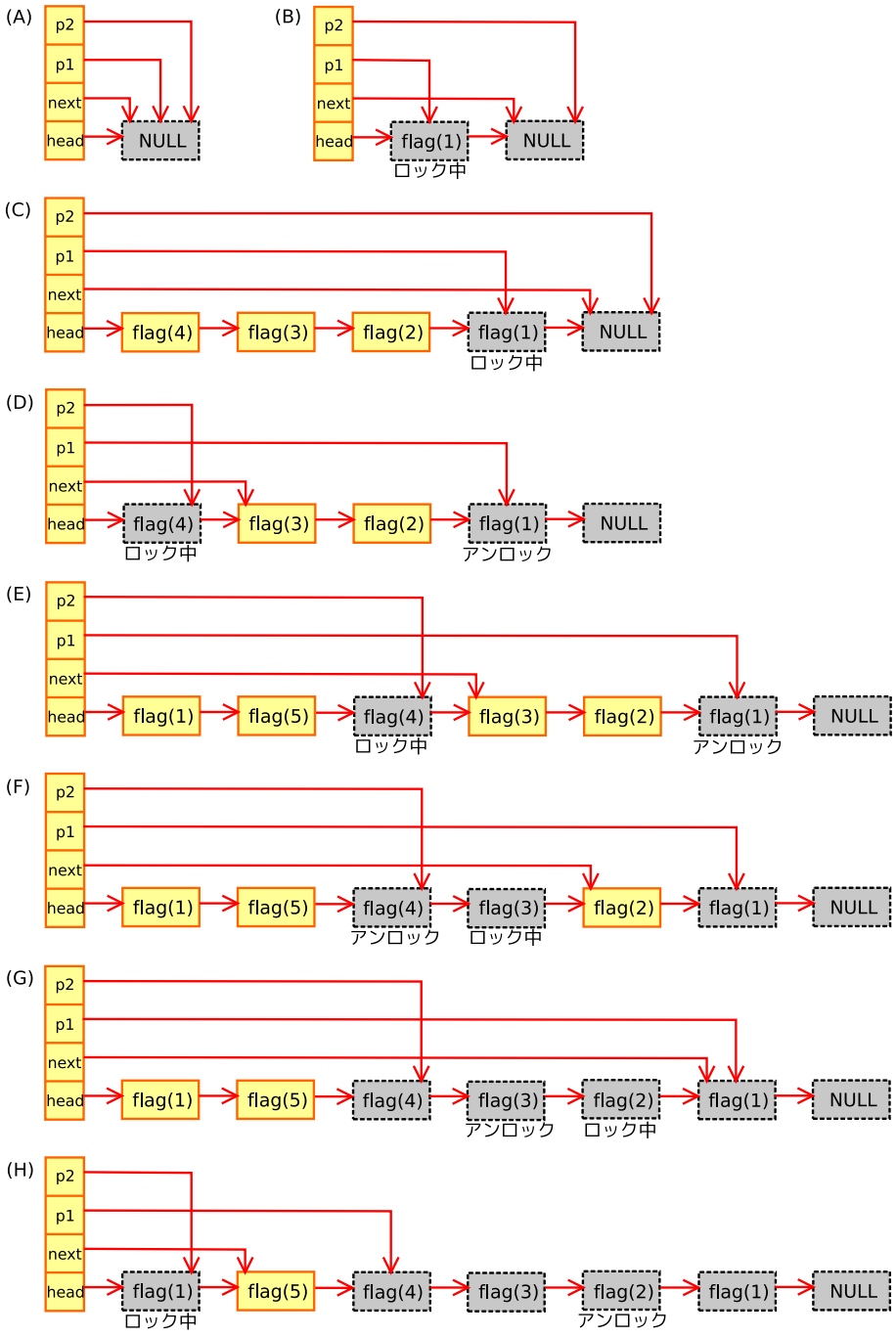

図 4.20　図 4.19 のコードによる Permission Word アルゴリズムの動作.

4.5.3 Permission Word アルゴリズムに基づく実装

DMI では，図 4.19 に示すアルゴリズムにより排他制御を実装している．このアルゴリズムは，Entry Section と Exit Section との間で変数を共有しないという要請のもとで，スレッド間の公平性やリモートキャッシュへのアクセス回数の最小化などを考慮しつつ著者が考案したものであるが，はからずも，既存の **Permission Word アルゴリズム**[82] と本質的には同じであることがのちにわかった．ただし，既存の Permission Word アルゴリズムは図 4.16 の実行モデルにしたがって記述されており，Entry Section と Exit Section との間で変数を共有しないことを意図して提案されているわけではない．Permission Word アルゴリズムは以下の性質を満たす：

- ロック関数のスタック領域上の変数をトリッキーに利用することにより，Entry Section と Exit Section との間で変数を共有する必要がなくなっている．
- read/write，fetch-and-store，compare-and-swap，アドレスの変更監視を組み合わせている．
- 各クリティカルセクションあたり，リモートキャッシュへのアクセス回数は $O(1)$ である．
- Weak Fairness を満たす．

以下では，図 4.20 の図にしたがって図 4.19 のアルゴリズムの動作を説明する．なお，図 4.20 において，*flag*(i) は，スレッド i における lock() 関数内の *flag* 変数を意味する．実線枠で囲まれた *flag*(i) は，まだ lock() 関数が実行中であるため実体が存在している *flag* 変数，点線枠で囲まれた *flag*(i) は，すでに lock() 関数が終了しているため実体が消滅している *flag* 変数を意味する：

(1) 初期状態では，*head*，*next*，*p1*，*p2* はすべて *NULL* である（図 4.20（A））．*head* は，クリティカルセクションへの突入を待機しているスレッドリストの先頭を表す変数である．*next* は，あるスレッドがクリティカルセクションを実行しているとき，そのスレッドがクリティカルセクションを抜けた時点でどのスレッドを起こせばよいかを表す変数である．*p1* と *p2* の意味は後述する．また，20 行目の convert() 関数と 39 行目と 43 行目の revert() 関数についても後述する．

(2) スレッド 1 が lock() 関数を呼び出したとすると，スレッド 1 は 21 行目で *head* に対して fetch-and-store を実行する．このとき，自分の後ろが *NULL* であるため，22 行目の if 文が成立してクリティカルセクションに突入することができ，lock() 関数はすぐに返る（図 4.20（B））．

(3) スレッド 2, スレッド 3, スレッド 4 が lock() 関数を呼び出し，この順に 21 行目の fetch-and-store を実行したとすると，図 4.20（C）の状態になる．

(4) スレッド 1 が unlock() 関数を呼び出したとすると，スレッド 1 は *next* で示されるスレッドを起こそうとするが，いまは *next* が *NULL* なので，自分より後ろに起こすスレッドは存在しないと判断し，*head* 側から起こそうと試みる．いまの場合，*head* の後ろに待ちスレッドが存在するため，37 行目の compare-and-swap は失敗し，38 行目で *p2* にスレッド 4 を入れたうえで（正確には *p2* にスレッド 4 の *flag* のアドレスを入れたうえで），40 行目でスレッド 4 を起こす．これによりスレッド 4 は 25 行目の watch() 関数から返り，27 行目により *next* にスレッド 3 を入れたうえで，lock() 関数から返る（図 4.20（D））．

(5) スレッド 5，スレッド 1 が lock() 関数を呼び出したとすると，21 行目の fetch-and-store により，スレッド 5 とスレッド 1 が *head* に連結される（図 4.20（E））．

(6) スレッド 4 が unlock() 関数を呼び出したとすると，スレッド 4 は，*next* が指しているスレッド 3 を 44 行目で起こす．これによりスレッド 3 は 25 行目の watch() 関数から返り，スレッド 3 は *next* にスレッド 2 を入れたうえで，lock() 関数から返る（図 4.20（F））．

(7) 同様に，スレッド 3 が unlock() 関数を呼び出したとすると，スレッド 3 は，*next* が指しているスレッド 2 を 44 行目で起こす．これによりスレッド 2 は 25 行目の watch() 関数から返り，スレッド 2 は *next* にスレッド 1 を入れたうえで，lock() 関数から返る（図 4.20（G））．

(8) このように，スレッド 4→ スレッド 3→ スレッド 2→ $\cdots$ の順に起こされていくが，この起床処理の連鎖はスレッド 2 までで止める必要がある．そして，この起床処理の連鎖をどこで止めればよいかを表すのが *p1* である．また，この起床処理の連鎖がどこから始まったのかを表すのが *p2* である．スレッド 2 が unlock() 関数を呼び出したとすると，33 行目の条件文で *next* と *p1* が一致するため，自分より後ろに起こすスレッドは存在しないと判断し，起床処理の連鎖を中止して，再度 *head* 側から起こそうと試みる．このとき，35 行目で，それまでの *p2* の値を *p1* に代入する．これは，いま行っている起床処理の連鎖が始まった位置を *p1* に仕込んでいることを意味するが，「いま行っている起床処理の連鎖が始まった位置」＝「次に行われる起床処理の連鎖が止まるべき位置」であるから，いい換えると，それまでの *p2* の値を *p1* に代入することは，次に行われる起床処理の連鎖が止まるべき位置を *p1* に仕込むことにほかならない．さらに，これから新しい起床処理の連鎖を始めるにあたって，*p2* の値も更新しなければならないが，それを行っているのが 38 行目である．つまり，38 行目で，次に行われる起床処理の連鎖がどこから始まるのかを *p2* に記憶している．これにより，次に行われる起床処理の連鎖が終了した時点で，「次の次に行われる起床処理の連鎖がどこで止まればよいか」を *p1* に教えられるようにしておくわけである（図 4.20（H））．改めて *p1* と *p2* の意味をまとめると，ある起床処理の連鎖を行っているとき，その起床処理の連鎖をどこで止めればよいかを表すのが *p1* であり，その起床処理の連鎖をどこから始めたか（＝次の起床処理の連鎖をどこで止めればよいか）を表すのが *p2* である．

最後に，convert() 関数と revert() 関数について説明する．図 4.20（B）において，スレッド 1 がクリティカルセクションに突入した時点では，スレッド 1 の lock() 関数はすでに返っているため，lock() 関数内のスタック領域に確保されていた *flag* の実体は消滅している．ところが，依然として *p1* や *head* は *flag* のアドレスを指している．そして，スレッド 1 が unlock() 関数を呼び出し，起床処理の連鎖が中止され，*head* から起床処理を開始しようとする際に，*head* の先に起こすべきスレッドが存在するかどうかを判定するために，37 行目においてこれらのアドレスが利用されることに注意しておく．さて，ここで，図 4.20（E）のようにスレッド 1 が再度 lock() 関数を呼び出したときに，この lock() 関数内の *flag* が，前回スレッド 1 が lock() 関数を呼び出したときの *flag* と同じアドレスに割り当てられてしまう場合を考える．すると，図 4.20（G）において，スレッド 2 が unlock() 関数内で 37 行目の compare-and-swap を呼び出す際に，本当は *head* の先に起こすべきスレッドが存在するにもかかわらず，*head* と *p1* の値が一致してしまっているがために，*head* の先には起こすべきスレッドが存在しな

いと勘違いしてしまう．このように，Permission Word アルゴリズムでは，すでに実体が存在しなくなった *flag* のアドレスをアルゴリズムに利用しているため，lock() 関数が複数回呼び出されて *flag* のアドレスが再利用されてしまうと不都合が起きる．しかし，lock() 関数が複数回呼び出される場合に *flag* のアドレスが再利用されてしまうのを防ぐことは不可能である．そこで，アドレスが再利用されてしまっても問題が起きないように，*p1*，*p2*，*flag* のアドレスが絶対に一致することがないよう，アドレスの値自体を操作しておく．具体的には，convert() 関数によってアドレスの下位 2 ビットを適宜ずらし，実際にそのアドレスの値を使う際には revert() 関数によって正規のアドレスに還元する．

4.6 要約

本章では，グローバルアドレス空間の実装として，プロセスが自由なタイミングで非同期的に参加/脱退できるグローバルアドレス空間のコヒーレンシプロトコルの実装，ページ置換の実装，データ転送の動的負荷分散の実装，排他制御の実装について述べた．とくに，非同期的にプロセスを参加/脱退させられるコヒーレンシプロトコルは新規的なものである．

第 5 章

非定型なグラフ計算のためのプログラミングインタフェース

本章では，グローバルビュー型のグローバルアドレス空間モデルに基づいて非定型なメモリアクセスを簡単に記述できるようにしつつも，内部的にはメッセージパッシングモデルと同様の無駄のない通信しか発生させないような API として，**read-write-set** を設計して実装する．

5.1 非定型なグラフ計算のモデル化

グローバルアドレス空間に対する非定型なアクセスが必要となる並列計算にはさまざまなものが存在するが，本章では，図 2.1 に示すように，節点全体が複数の領域に分割されており，節点間の結合関係に基づいて節点の値を更新するような並列計算を対象にする．とくに，NAS Parallel Benchmark[51] の FT や MG，Himeno Benchmark[3] などのように，直方体状の領域を直方体状の小領域に領域分割するような定型的な並列計算だけではなく，節点全体のなす形状や節点間の結合関係が複雑な非定型な並列計算を対象にする．有限要素法，マルチグリッド法，ページランク計算，最短路計算など，上記のようにモデル化できる並列計算は，実用的な世界にも数多い．

より正確にモデル化するならば，以下の性質を持つ並列計算を考える：

- 領域全体は節点集合 Z から構成されており，節点間に結合関係が定義されている．
- 節点集合 Z は，n 個の領域に分割されている．各領域 i は内点の集合 $writeset_i$ から構成されている．$i \neq j$ ならば $writeset_i$ と $writeset_j$ に重なりはない．たとえば，図 2.1 の例の場合，領域 0 の $writeset_0$ は $\{0, 4, 5\}$，領域 1 の $writeset_1$ は $\{1, 2, 3, 7\}$，領域 2 の $writeset_2$ は $\{6, 8, 9\}$ である．
- 節点間の結合関係に基づき，各集合 $writeset_i$ に対して，$writeset_i$ に属する節点の値を更新するために必要となる節点の集合 $readset_i$ が決まる．いい換えると，$readset_i$ は領域 i の内点と外点の集合である．たとえば，図 2.1 の例の場合，領域 0 の $readset_0$ は $\{0, 4, 5, 1, 2, 9\}$，領域 1 の $readset_1$ は $\{1, 2, 3, 7, 0, 5, 6\}$，領域 2 の $readset_2$ は $\{6, 8, 9, 2, 4, 5\}$ である．
- 各領域 i の節点の値を更新するためには，まず $readset_i$ に属する節点の値を読み込み，それらの値

に基づいて $writeset_i$ に属する節点の値を更新する．とくに，反復計算によって，各領域 i の節点の値が繰り返し更新される場合を考える．

5.2 設計

5.2.1 基本アイディア

2.1.2.3 節で述べたように，**5.1** 節でモデル化した並列計算をメッセージパッシングモデルで記述する場合の問題点は，節点番号とローカルインデックスとを対応づけるための煩雑な計算が必要になることである．そこで，read-write-set における API の設計目標は，**グローバルインデックスに基づいて並列計算を記述できるようにしつつも，内部的にはメッセージパッシングモデルと同等の通信しか発生させない**ような API を設計することである．いい換えると，プログラマの視点では図 2.7 のように記述できるようにしつつも，内部的には，図 2.2 に示すように**外点の値の通信だけが起きる**ようにする．そのためには，図 2.7 に示すように，単純に 1 個の大きなグローバルアドレス空間を確保して，そのグローバルアドレス空間から節点を read/write するという実装では不十分である．なぜなら，単純に 1 個の大きなグローバルアドレス空間を確保して read/write するだけでは，コヒーレンシ粒度の調節と離散アクセスのグルーピングをどのように駆使したとしても，外点の値以外の通信が起きてしまうことは避けられないためである．そこで，**5.1** 節でモデル化した並列計算に特化した，より洗練された API とその実装が必要となる．

具体的には，主に以下の 4 種類の API を設計する：

rwset_decompose() 各領域 i について，内点の集合 $writeset_i$ をグローバルインデックスによって定義する API.

rwset_build() 各領域 i について，内点と外点の集合 $readset_i$ をグローバルインデックスによって定義する API.

rwset_read() 各領域 i について，$readset_i$ に含まれる節点の値を read する API.

rwset_write() 各領域 i について，$writeset_i$ に含まれる節点の値を write する API.

次節では，これらの API の設計について詳しく述べる．

5.2.2 API

rwset = **rwset_init**(*element_num*, *domain_num*) read-write-set を生成して初期化する．*element_num* は節点数，*domain_num* は領域数である．この関数は，各 read-write-set について最初に 1 回だけ呼ぶ．返り値の *rwset* は，生成された read-write-set のハンドルである．

rwset_destroy(*rwset*) read-write-set *rwset* を破棄する．

rwset_decompose(*rwset*, *i*, $writeset_i$) read-write-set *rwset* に対して，領域 i の内点の順序集合 $writeset_i$ を定義する．$writeset_i$ が単なる集合ではなく順序集合である理由は後述する．すべての領域 i に対して，rwset_decompose(*rwset*, *i*, $writeset_i$) 関数を呼び出すことによって，領域が完全に定義される．

```
main() {
   rwset = rwset_init(10, 3);
   create 3 threads;
   join 3 threads;
   rwset_destroy(rwset);
   return;
}

each_thread(i, /* processor i */
            rwset /* a read-write-set handle */ ) {
   if (i == 0) writeset[0..2] = {0,4,5};
   else if (i == 1) writeset[0..3] = {1,2,3,7};
   else if (i == 2) writeset[0..2] = {6,8,9};
   rwset_decompose(rwset, i, writeset); /* define a writeset */
   barrier();
   if (i == 0) readset[0..5] = {0,4,5,1,2,9};
   else if (i == 1) readset[0..6] = {1,2,3,7,0,5,6};
   else if (i == 2) readset[0..5] = {6,8,9,2,4,5};
   rwset_handle = rwset_build(rwset, i, readset); /* define a readset */

   for (iter = 0; /* until convergence */; iter++) {
      barrier();
      rwset_write(rwset_handle, wbuf); /* write the values of the writeset */
      barrier();
      rwset_read(rwset_handle, rbuf); /* read the values of the readset */
      ...; /* calculation based on connectivity */
   }
}
```

図 5.1 read-write-set を使って図 2.1 に示したグラフ計算を行うプログラム．

$rwset\_handle_i$ = **rwset_build(**$rwset$, i, $readset_i$**)** read-write-set $rwset$ に対して，領域 i の内点と外点の順序集合 $readset_i$ を定義する．返り値の $rwset\_handle_i$ は領域 i のハンドルであり，後述する rwset_read() 関数/rwset_write() 関数を使って内点や外点の値を read/write するときに使う．$rwset\_handle_i$ は，内部的に，$readset_i$ と $writeset_i$ の情報を保持している．なお，rwset_build() 関数を呼ぶ時点では，領域が完全に定義されていなければならない．すなわち，rwset_build() 関数を呼ぶ時点では，すべての領域 i に対して rwset_decompose($rwset$, i, $writeset_i$) 関数がすでに完了していることが保証されている必要がある．これは，たとえば，すべての領域 i に対して rwset_decompose($rwset$, i, $writeset_i$) 関数を呼び出したあと，バリアを行うことで保証できる．

rwset_write($rwset\_handle_i$, $wbuf_i$**)** $wbuf_i$ に格納されている値を，順序集合 $writeset_i$ に含まれる節点の値としてグローバルアドレス空間に書き込む．いい換えると，$wbuf_i$ に格納されている値を，領域 i の内点の値としてグローバルアドレス空間に書き込む．とくに，$wbuf_i$ の j 番目の値が，順序集合 $writeset_i$ の j 番目の節点の値として書き込まれる．すなわち，順序集合 $writeset_i$ における節点の順序が，$wbuf_i$ からグローバルアドレス空間に対して値が書き込

まれる順序を決定する．

rwset_read($rwset\_handle_i$, $rbuf_i$**)** 順序集合 $readset_i$ に含まれる節点の値をグローバルアドレス空間から読み込み，$rbuf_i$ に格納する．いい換えると，領域 i の内点と外点の値をグローバルアドレス空間から読み込み，$rbuf_i$ に格納する．とくに，順序集合 $readset_i$ の j 番目の節点の値が，$rbuf_i$ の j 番目に格納される．すなわち，順序集合 $readset_i$ における節点の順序が，$rbuf_i$ に値が読み込まれる順序を決定する．

read-write-set の API の使用例を図 5.1 に示す．図 5.1 では，簡単のため，$writeset[0]$=0，$writeset[1]$=4，$writeset[2]$=5 を，$writeset[0..2]$={0,4,5}などと略記している．read-write-set では，多くの場合，以下の手順でプログラムを記述する：

(1) 各スレッド i が rwset_decompose() 関数によって領域 i の内点を定義する．
(2) すべてのスレッドが同期する．
(3) 各スレッド i が rwset_build() 関数によって領域 i の内点と外点を定義して，領域 i のハンドルを得る．
(4) 領域 i のハンドルを使って，rwset_write() 関数によって内点の値を書き込んだり，rwset_read() 関数によって内点と外点の値を読み込んだりする．

なお，多くの場合には領域 i に関する rwset_decompose() 関数と rwset_build() 関数は同一のスレッド i が呼び出すが，API の仕様上は，別のスレッドが呼び出しても問題ない．また，上記の説明では，$readset_i$ は内点と外点の順序集合であるとしたが，かならずしも内点を含める必要はない．なぜなら，領域 i の内点の値は，rwset_read() 関数によってグローバルアドレス空間から読み出さなくても，スレッド i のローカルアドレス空間に保持されているためである．ただし，とくに内点と外点の順序をオーダリング [116] する場合には，$readset_i$ をそのオーダリングに基づいた内点と外点の順序集合としたうえで，外点の値といっしょに内点の値もグローバルアドレス空間から読み出すようにした方が，プログラマビリティは高い．

5.3 実装

read-write-set の目標は，前節で説明した API によってグローバルインデックスに基づいて並列計算を記述できるようにしつつも，内部的には，図 2.2 に示したメッセージパッシングモデルと同等の，外点の値の通信だけが起きるようにすることである．read-write-set の各関数は，無駄にメモリを消費したり無駄なデータを転送したりすることがないよう，離散アクセスのグルーピングなどを活用して実装されている．本節では，rwset_init() 関数（初期化），rwset_destroy() 関数（破棄），rwset_decompose() 関数（内点の定義），rwset_build() 関数（内点と外点の定義），rwset_write() 関数（内点の値の write），rwset_read() 関数（内点と外点の値の read）の実装について説明する．なお，図 5.2，図 5.3，図 5.4，図 5.5，図 5.7，図 5.6 では，グローバルアドレス空間をオレンジ色で，ローカルアドレス空間を青色で描いている．

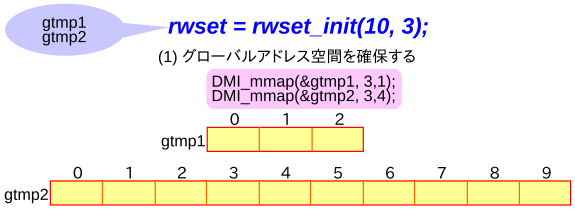

図 5.2 rwset_init() 関数の実装.

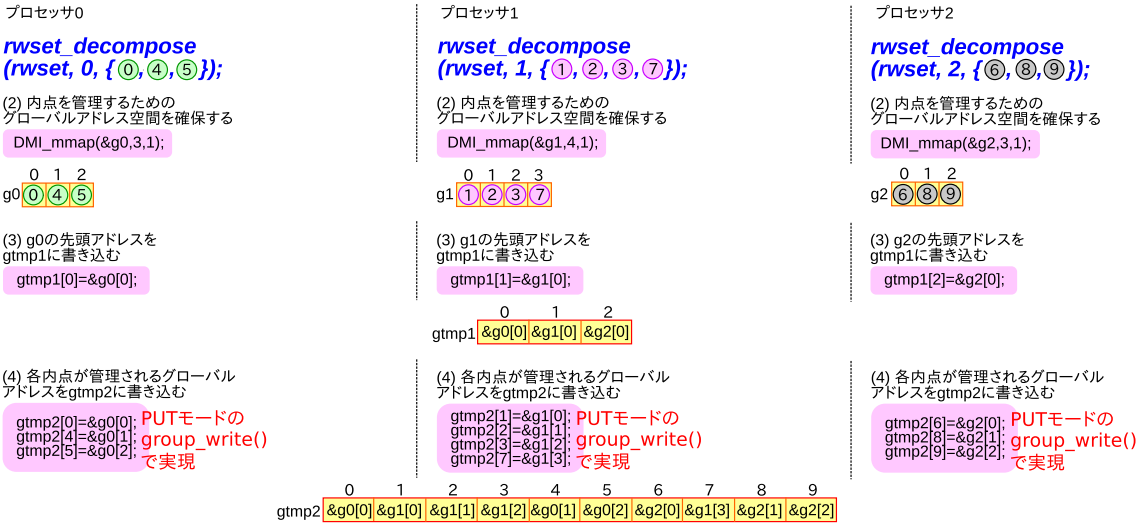

図 5.3 rwset_decompose() 関数の実装.

5.3.1 初期化

rwset_init(*element_num*, *domain_num*) 関数が呼ばれると，*domain_num* 個の要素を持ったグローバルアドレス空間 *gtmp1* と，*element_num* 個の要素を持ったグローバルアドレス空間 *gtmp2* を確保する（図 5.2 (1)）．一般に，*element_num* は大きい値をとりうるため，*gtmp2* 全体がいずれかのプロセスのメモリプールに割り当てられてしまうことがないよう，*gtmp2* のページサイズはある程度小さくとる．現在の実装では，*gtmp2* のページ数が，*domain_num*×20 になるようにページサイズを決めている．rwset_init() 関数の返り値である read-write-set のハンドル *rwset* には，*gtmp1* と *gtmp2* の情報が記録されている．

5.3.2 破棄

グローバルアドレス空間 *gtmp1* と *gtmp2* を解放する．

5.3.3 内点の定義

rwset_decompose(*rwset*, *i*, $writeset_i$) 関数が呼ばれると，第 1 に，$|writeset_i|$ 個の要素を持つグローバルアドレス空間 g_i を確保する（図 5.3 (2)）．g_i は，のちに領域 i の内点の値を管理するためのグ

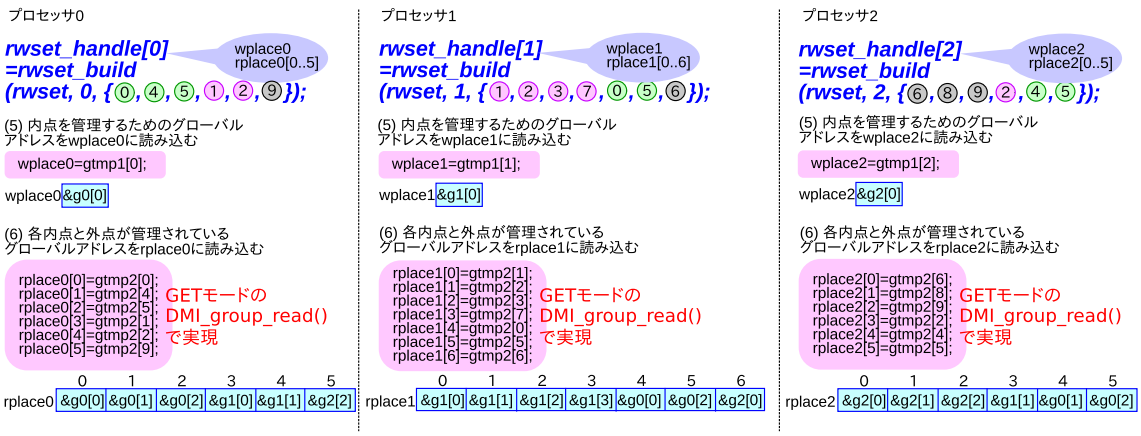

図 5.4 rwset_build() 関数の実装.

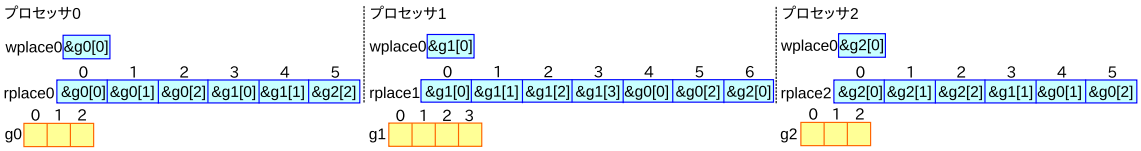

図 5.5 rwset_build() 関数が完了した直後の状態.

ローバルアドレス空間である．通常，領域 i の内点の書き込みが複数のスレッドから行われることはないため，g_i 全体を 1 個のコヒーレンシ粒度にすれば良く，g_i のページサイズは $|writeset_i|$ とする．第 2 に，g_i のグローバルアドレスを，$gtmp1[i]$ に記録する（図 5.3 (3)）．第 3 に，順序集合 $writeset_i$ に含まれる各内点 j について，内点 j の値を管理するグローバルアドレス空間 g_i 上のアドレスを，$gtmp2[j]$ に書き込む（図 5.3 (4)）．たとえば，領域 0 について考えたとき，内点 0 の値は $g_0[0]$ で，内点 4 の値は $g_0[1]$ で，内点 5 の値は $g_0[2]$ で管理されることになるため，$gtmp2[0]$ には $g_0[0]$ のグローバルアドレスを，$gtmp2[4]$ には $g_0[1]$ のグローバルアドレスを，$gtmp2[5]$ には $g_0[2]$ のグローバルアドレスを書き込む．この書き込み操作は離散的なアクセスとなるため，離散アクセスのグルーピングを活用し，PUT モードの group_write() 関数によって実現する．結果的に，$gtmp2[j]$ には，「内点 j の値が管理されるグローバルアドレス」が記録された状態になる．すべての領域 i について，rwset_decompose() 関数が呼び出されることで，$gtmp2$ が完成する．

離散アクセスのグルーピングを利用しているため，rwset_decompose() 関数で発生する通信量は $O(|writeset_i|)$ である．

5.3.4 内点と外点の定義

スレッド i によって rwset_build($rwset$, i, $readset_i$) 関数が呼ばれると，第 1 に，$gtmp1[i]$ の値が，スレッド i のローカルアドレス空間 $wplace_i$ に読み込まれる（図 5.3 (5)）．結果的に，$wplace_i$ には，「領域 i の内点を管理するグローバルアドレス空間 g_i」が記録された状態になる．このように，

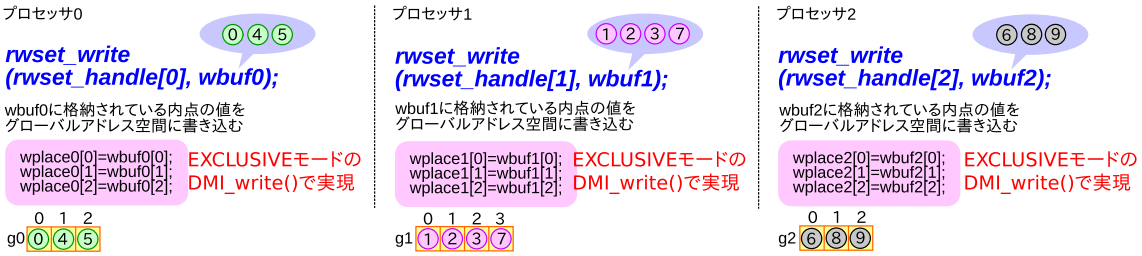

図 5.6 rwset_write() 関数の実装.

rwset_decompose() 関数と rwset_build() 関数の間で，いったん *gtmp1* を経由して g_i の値をやりとりする理由は，一般には，領域 i に対する rwset_decompose() 関数と rwset_build() 関数を呼び出すスレッドが異なる可能性があるためである．第 2 に，順序集合 $readset_i$ に含まれる各節点 j について，グローバルアドレス空間上の $gtmp2[j]$ の値を読み出し，スレッド i のローカルアドレス空間上の配列 $rplace_i$ に格納する（図 5.3（6））．たとえば，領域 0 について考えたとき，順序集合 $readset_i$ には節点 0，節点 4，節点 5，節点 1，節点 2，節点 9 がこの順に含まれるので，$rplace_0[0]$ に $gtmp2[0]$ の値を，$rplace_0[1]$ に $gtmp2[4]$ の値を，$rplace_0[2]$ に $gtmp2[5]$ の値を，... などと格納する．この読み込み操作は離散的なアクセスとなるため，離散アクセスのグルーピングを活用し，GET モードの group_read() 関数によって実現する．ここで，$gtmp2[j]$ には「内点 j の値が管理されるグローバルアドレス」が記録されていることをふまえると，結果的に，$rplace_i[j]$ には「順序集合 $readset_i$ の j 番目の節点が管理されるグローバルアドレス」が記録されることになる．rwset_build() 関数の返り値である領域 i のハンドル $rwset\_handle_i$ には，$wplace_i$ と配列 $rplace_i$ が記録されている．

rwset_build() 関数で発生する通信量は $O(|readset_i|)$ である．

すべての領域 i に対して rwset_build() 関数が完了したあとの状態を図 5.5 に示す．以上の操作においては，グローバルアドレス空間 *gtmp2* のページサイズをある程度小さくとることで *gtmp2* のオーナーがすべてのプロセスに分散されており，かつ，*gtmp2* に対しては GET モードの group_read() 関数と PUT モードの group_write() 関数しか発行しないため，いずれかのプロセスのメモリプールの消費量が膨れあがることはない．

5.3.5 内点の値の write

rwset_write($rwset\_handle_i$, $wbuf_i$) 関数が呼ばれると，ローカルアドレス空間上の配列 $wbuf_i$ のデータが，グローバルアドレス空間 g_i に EXCLUSIVE モードで write される（図 5.6）．

rwset_write() 関数は通信をともなうことなくローカルに完了する．

5.3.6 内点と外点の値の read

rwset_read($rwset\_handle_i$, $rbuf_i$) 関数が呼ばれると，ローカルアドレス空間上の配列 $rplace_i$ に記録されている各グローバルアドレスに格納されている値が，その順序でローカルアドレス空間上の配列 $rbuf_i$ に読み込まれる（図 5.7）．この読み込み操作は，離散アクセスのグルーピングを活用して，GET モードの group_read() 関数によって実現する．

図 5.7 rwset_read() 関数の実装.

rwset_read() 関数で発生する通信量は，$readset_i$ に含まれる外点の個数を k_i とすると $O(k_i)$ である．結局，**rwset_write() 関数では通信は発生せず，rwset_read() 関数では外点の値の通信しか発生しない**.

5.4 要約

有限要素法，マルチグリッド法，ページランク計算，最短路計算など，節点間の複雑な結合関係に基づいて節点の値を更新するグラフ計算としてモデル化できる並列計算は数多い．本章では，そのような非定型な並列計算をグローバルインデックスに基づいて記述できるようにしつつも，内部的には外点の値の通信しか発生させないような API として，read-write-set を設計して実装した.

第 6 章

評価 I：グローバルアドレス空間の性能とプログラマビリティ

本章では，第 **3** 章で設計し，第 **4** 章で実装したグローバルアドレス空間の性能とプログラマビリティを評価する．本章では高性能並列科学技術計算に対する基本的な性能とプログラマビリティを評価することを主な目的とし，再構成に対する詳しい評価は第 **10** 章で行う．

6.1 実験環境

実験環境としては，8 プロセッサ（ハイパースレッディングにより論理的には 16 プロセッサ）のノードを 10Gbit イーサネットで 16 個接続した，合計 128 プロセッサのクラスタ環境を使用した．各ノードの構成を表 6.1 に示す．このクラスタ環境には，RAID によって構成された 20 TB のディスクが接続されており，このディスクは NFS によってすべてのノード間で共有されている．**6.3.5** 節で述べるディスクスワップと HDD の性能評価実験をのぞいては，実験で使用するすべてのデータは，この NFS で共有されたディスクに格納した．

表 6.1 実験環境の各ノードのハードウェア構成．

マシン名	Dell PowerEdge R610
CPU	Intel Xeon E5530×2
各 CPU のプロセッサ数	4 プロセッサ（ハイパースレッディングにより 8 プロセッサ）
各 CPU のキャッシュ	L1：64 KB×4，L2：256 KB×4，L3：8 MB×1
メモリ	2 GB×12，周波数は 1066 MHz
スワップ領域	24 GB
HDD	500 GB×2
NIC	NetXtreme II BCM57711 10Gigabit PCIe，10Gbit イーサネット
OS	GNU Linux，カーネル 2.6.26-2-amd64

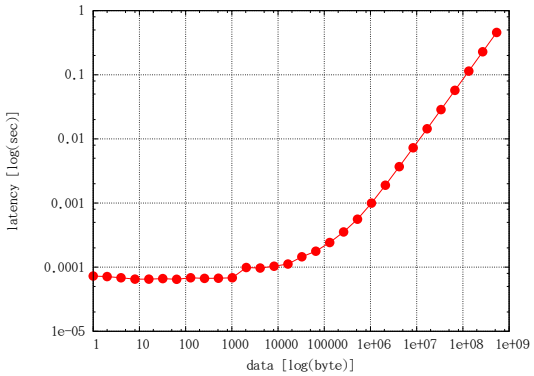

図 6.1　実験環境の TCP レイテンシ.

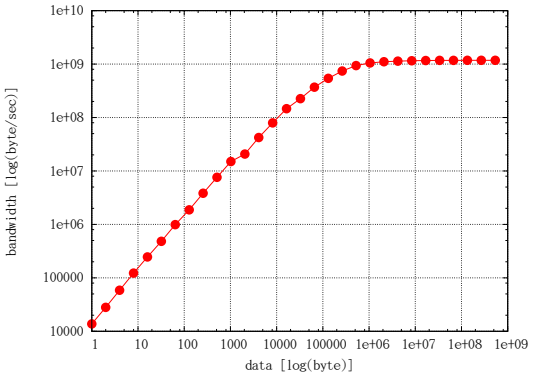

図 6.2　実験環境の TCP バンド幅.

トランスポート層のプロトコルとしては TCP を使用し，輻輳制御アルゴリズムとしては，Linux カーネル 2.6.26-2-amd64 に標準の cubic を使用した．また，TCP_NODELAY は無効にし，Nagle アルゴリズムを禁止した．このクラスタ環境における 2 ノード間で ping-pong 通信を行った場合のレイテンシおよびバンド幅を，それぞれ図 6.1 と図 6.2 に示す．グラフ中の各点は 100 回の実行時間の平均値を表す．データサイズが十分に小さい場合のレイテンシは約 65 マイクロ秒であり，データサイズが十分に大きい場合のバンド幅は約 8.76 GBit/秒である．

処理系のコンパイラとしては gcc 4.3.2 を使用し，コンパイラオプションとしては-O3 を使用した．DMI と比較する対象の処理系としては MPI を使用し，各アプリケーションを同一のアルゴリズムで DMI と MPI の両方で記述して，性能およびプログラマビリティを比較した．比較対象の処理系として MPI を採用した理由は，多くの並列分散プログラミング処理系のなかでも MPI は性能やスケーラビリティにきわめて優れており，HPC 分野において高性能な並列科学技術計算を記述する際のデファクトスタンダードになっているためである．したがって，性能面では，MPI と同等かそれ以上の性能を達成できれば，HPC 分野において有用な処理系であることを主張できると考えられる．MPI の処理系としては，mpich2 1.2.1p1 と OpenMPI 1.4.2 を使用した．MPI を n プロセッサで実行する場合には，8 個の MPI プロセスを $\lfloor n/8 \rfloor$ 個のノードに生成し，残りの $n - 8 \times \lfloor n/8 \rfloor$ 個の MPI プロセスを別の 1 個のノードに生成した．また，DMI を n プロセッサで実行する場合には，各ノードに 1 個の DMI プロセスを生成したうえで，8 個の DMI スレッドを $\lfloor n/8 \rfloor$ 個のノードに生成し，残りの $n - 8 \times \lfloor n/8 \rfloor$ 個の DMI スレッドを別の 1 個のノードに生成した．すなわち，各プロセッサに 1 個の DMI スレッドまたは 1 個の MPI プロセスが割り当てられるようにした．よって，本章でアルゴリズムを記述する際には，簡単のため，「DMI スレッドまたは MPI プロセス」のことを単に「プロセッサ」と呼ぶことにする．**6.3.5** 節の遠隔スワップの実験をのぞいては，DMI における各プロセスのメモリプールの容量は 24 GB に設定した．

実験に使用する乱数としては，原始多項式 $x^{521} + x^{32} + 1$ に基づく 64 ビットの M 系列乱数 [200] を使用した．乱数の精度が十分であることは事前に確認した．

6.2 各実験の意図

本章では多くの実験の結果と考察を示すが，各実験の意図を事前にまとめておく．

第 1 に，マイクロベンチマークとして以下のものを評価する．**6.3.1** 節では，グローバルアドレス空間に対する read/write の基本性能とオーバヘッドを評価する．**6.3.2** 節では，排他制御を題材として，選択的キャッシュ read/write の効果について評価する．**6.3.3** 節では，Allreduce を題材として，選択的キャッシュ read/write およびアドレスの変更監視の効果について評価する．**6.3.4** 節では，Broadcast を題材として，データ転送の動的負荷分散の効果を評価する．**6.3.5** 節では，STREAM ベンチマークを題材として，遠隔スワップの性能，非同期 read/write の効果，ページ置換の効果を評価する．

第 2 に，基本的なアプリケーションとして以下のものを評価する．**6.5.1** 節から **6.5.9** 節までは，それぞれ，NAS Parallel Benchmark の EP，マンデルブロ集合の描画，横ブロック分割による行列行列積，Fox アルゴリズムによる行列行列積，ランダムサンプリングソート，N 体問題，ヤコビ法による PDE ソルバを題材として，DMI と MPI のプログラマビリティとスケーラビリティを比較する．また，**6.5.2** 節の NAS Parallel Benchmark の EP と **6.5.4** 節のマンデルブロ集合の描画では，DMI において並列計算を実行中に動的にノードを参加/脱退させた場合の挙動を評価する．**6.5.5** 節の横ブロック分割による行列行列積と **6.5.6** 節の Fox アルゴリズムによる行列行列積では，データ転送の動的負荷分散が実際のアプリケーションに与える効果を評価する．**6.5.7** 節のランダムサンプリングソートでは，非同期 read/write が実際のアプリケーションに与える効果を評価する．

第 3 に，非定型で応用的なアプリケーションとして以下のものを評価する．**6.6.1** 節から **6.6.3** 節まででは，それぞれ，非定型な領域分割をともなう有限要素法による応力解析，大規模な Web グラフのページランク計算，大規模な Web グラフの最短路計算を題材として，DMI と MPI の性能とプログラマビリティを比較する．**6.6.4** 節では，DMI が採用している PGAS モデルにおける単方向通信の利点を活かして，Web グラフの最短路計算を非同期的なアルゴリズムで実装した場合の性能を評価する．

6.3 マイクロベンチマーク

6.3.1 read/write のオーバヘッド

6.3.1.1 実験設定

DMI における read/write の基本性能を評価した．第 1 に，DMI_read() 関数に関して，さまざまなデータサイズのデータを INVALIDATE モードで read したとき，read フォルトが発生する場合と発生しない場合のそれぞれについて，「全体の実行時間（total）」，「全体の実行時間のうち，グローバルアドレス空間からローカルアドレス空間へのメモリコピーに要する時間（memcpy）」，「全体の実行時間のうち，オーナーからのページ転送に要する時間（communication）」を調べた．第 2 に，DMI_write() 関数に関して，さまざまなデータサイズのデータを PUT モードで write したとき，write フォルトが発生する場合と発生しない場合のそれぞれについて，「全体の実行時間（total）」，「全体の実行時間のうち，ローカルアドレス空間からグローバルアドレス空間へのメモリコピーに要する時間（memcpy）」，

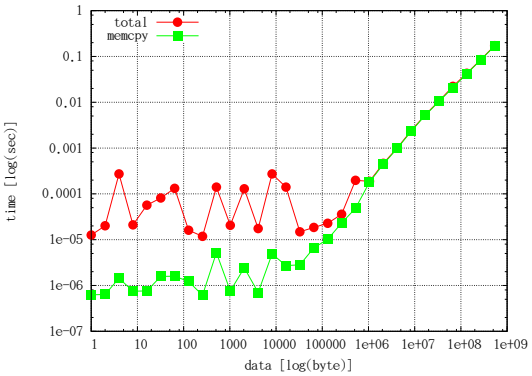

図 6.3　read フォルトが発生しない場合の実行時間の内訳.

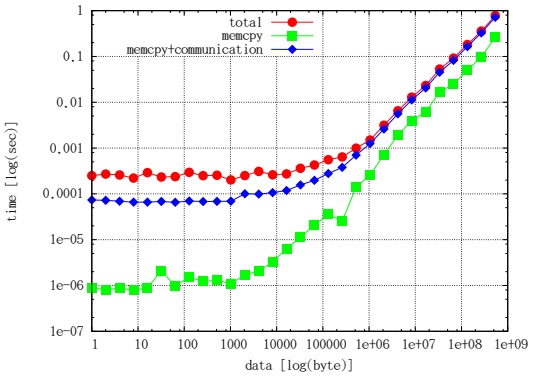

図 6.4　read フォルトが発生する場合の実行時間の内訳.

「全体の実行時間のうち，オーナーへのデータ転送に要する時間（communication）」を調べた．なお，簡単化のため，オーナー以外にページをキャッシュしているプロセス数は 0 とし，write フォルトにともなって invalidate 要求や update 要求が発行されないようにした．各データサイズについて 100 回測定を行い，平均時間を算出した．

6.3.1.2　結果と考察

第 1 に，DMI_read() 関数について，read フォルトが発生しない場合の実行時間の内訳と，read フォルトが発生する場合の実行時間の内訳を，それぞれ図 6.3 と図 6.4 に示す．図 6.3 より，データサイズが 16 KB 以下では，memcpy が total に占める割合は 0.5%~7.6% 程度であり，多くの時間がオーバヘッドに消費されてしまっていることがわかる．このオーバヘッドは，ページテーブルにアクセスするための排他制御，read/write を行っている最中に該当のグローバルアドレス空間が解放されてしまうのを防ぐための排他制御，ページフォルトに備えるための各種データ構造の malloc/free，この read/write が非同期化されることに備えるための各種データ構造の malloc/free などに起因している．このように，多くの排他制御や malloc/free が必要になっているのは，現在の DMI_read() 関数の実装が十分に洗練されていないためである．実装上本質的に必要なオーバヘッドは，ページテーブルにアクセスするための排他制御だけであると考えられ，実装をより洗練させてオーバヘッドを削減することが重要である．一方で，データサイズが 1 MB 以上では，memcpy が total に占める割合が 95% 以上になり，オーバヘッドはほぼ無視できるようになる．また，図 6.4 より，データサイズが 512 B 以下では，memcpy+communication が total に占める割合は 22%~30% 程度であり，残りの時間がオーバヘッドに消費されている．一方で，データサイズが 128 MB 以上では，memcpy+communication が total に占める割合が 90% 以上になる．

第 2 に，DMI_write() 関数について，write フォルトが発生しない場合の実行時間の内訳と，write フォルトが発生する場合の実行時間の内訳を，それぞれ図 6.5 と図 6.6 に示す．グラフ中の各点は 100 回の実行時間の平均値を表す．図 6.5 より，データサイズが 8 KB 以下では，memcpy が total に占め

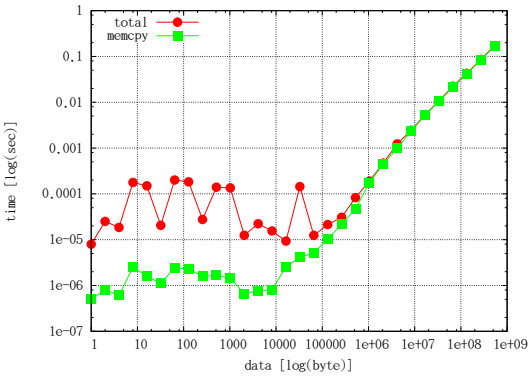

図 6.5　write フォルトが発生しない場合の実行時間の内訳.

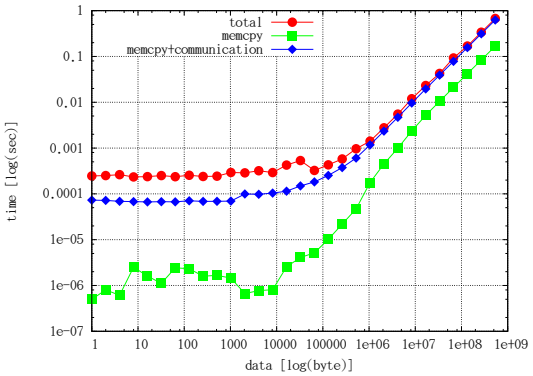

図 6.6　write フォルトが発生する場合の実行時間の内訳.

る割合は 1.0%~6.2% 程度であり，データサイズが 2 MB 以上では，95% 以上になることがわかる．また，図 6.6 より，データサイズが 1 KB 以下では，memcpy+communication が total に占める割合は 23%~30% 程度であり，データサイズが 32 MB 以上では，90% 以上になることがわかる．ページをキャッシュしているプロセスが存在すれば，invalidate 要求や update 要求の発行とその応答の回収が必要になるため，より多くの時間を要する．

オーバヘッドの大小はともかく，DMI_read() 関数/DMI_write() 関数はメモリコピーを必要とするため，共有メモリ環境上の通常の read/write と比較すると多くの時間を要する．したがって，**3.2.6** 節で指摘したように，DMI のプログラム開発においては，ページサイズを必要十分に大きくしたり，1 回の DMI_read() 関数/DMI_write() 関数でできるかぎり大きなグローバルアドレス領域を read/write したりすることで，DMI の API の呼び出し回数を少なくすることが性能上重要である．

6.3.2　排他制御における選択的キャッシュ read/write の効果

6.3.2.1　実験設定

図 4.19 の排他制御のアルゴリズムを題材にして，選択的キャッシュ read/write の効果を評価した．図 4.19 の排他制御アルゴリズムでは，構造体 mutex_t のメンバ変数に対して，23 行目，27 行目，35 行目，38 行目で write を，21 行目で fetch-and-store を，37 行目で compare-and-swap を行い，20 行目の直前，33 行目の直前，38 行目の直前で read を行っているが，これらの read/write/fetch-and-store/compare-and-swap を発行するモードを変化させることで，排他制御の性能がどのように変化するかを調べた．調べる組み合わせとしては，上記 4 ヶ所の write と 1 ヶ所の fetch-and-store と 1 ヶ所の compare-and-swap をすべて X モードで行い，上記 3 ヶ所の read をすべて Y モードで行うとしたとき，(X, Y) = (PUT, GET)，(X, Y) = (PUT, INVALIDATE)，(X, Y) = (PUT, UPDATE)，(X, Y) = (EXCLUSIVE, GET)，(X, Y) = (EXCLUSIVE, INVALIDATE)，(X, Y) = (EXCLUSIVE, UPDATE) の 6 種類の場合を調べた．128 プロセッサを使用し，各プロセッサが 300 回の排他制御（lock() 関数と unlock() 関数の呼び出し）を行うのに要した実行時間を測定し，

表 6.2　mutex における選択的キャッシュ read/write と実行時間の関係（128 プロセッサ実行時）.

種類（X，Y）	実行時間 [sec]
PUT，GET	0.1142
PUT，INVALIDATE	0.1477
PUT，UPDATE	0.1361
EXCLUSIVE，GET	0.1838
EXCLUSIVE，INVALIDATE	0.1810
EXCLUSIVE，UPDATE	0.1714

排他制御 1 回あたりの平均時間を算出した．本実験では，128 プロセッサが排他制御を争うことになり，同一のグローバルアドレスに対してほぼ同時に write/fetch-and-store/compare-and-swap を行うため，write ローカリティは非常に低い.

6.3.2.2　結果と考察

各組み合わせに対して，排他制御 1 回あたりに要した平均時間を表 6.2 に示す．表 6.2 より，（X，Y）=（PUT，GET）の場合が最速であり，もっとも遅い（X，Y）=（EXCLUSIVE，GET）の場合より 60% も高速である．read のモードとしていずれのモードを使用する場合でも，PUT モードよりも EXCLUSIVE モードの方が遅い理由は，本実験のように write ローカリティの低いプログラムでは，EXCLUSIVE モードを使用するとオーナーが頻繁に移動してしまい，最新ページの転送やオーナー追跡のためのオーバヘッドが増大するためである．また，write のモードとして PUT モードを使用したとき，GET モードよりも INVALIDATE モードや UPDATE モードの方が遅い理由は，本実験のように多数のプロセッサが頻繁に write する状況では，データを read してから次に read するまでの間にそのデータが更新されている可能性が高く，データをキャッシュする意味がないうえに，多数の invalidate 要求や update 要求が発生してしまうためである．この結果から，write ローカリティや read の頻度に応じて，選択的キャッシュ read/write を使い分けることが性能上重要であることがわかる.

6.3.3　Allreduce における選択的キャッシュ read/write などの効果

6.3.3.1　実験設定

図 3.8 の Allreduce を題材にして，選択的キャッシュ read/write およびアドレスの変更監視の効果を評価した．Allreduce あるいはその部分的機能であるバリアは，反復計算型のアプリケーションにおいて頻繁に利用されるため，その性能はきわめて重要である．たとえば，**6.6.1** 節で述べる有限要素法では，図 6.37 に示す BiCGSafe 法によって反復計算を行うが，BiCGSafe 法では 1 イテレーションのなかに 22 回の Allreduce またはバリアが含まれる.

本実験では，以下の 6 種類の場合の性能を比較した：

- 図 3.8 において (##) で示した行の DMI_read() 関数のモードを GET モードにする場合（DMI (GET)）.
- 図 3.8 において (##) で示した行の DMI_read() 関数のモードを INVALIDATE モードにする場

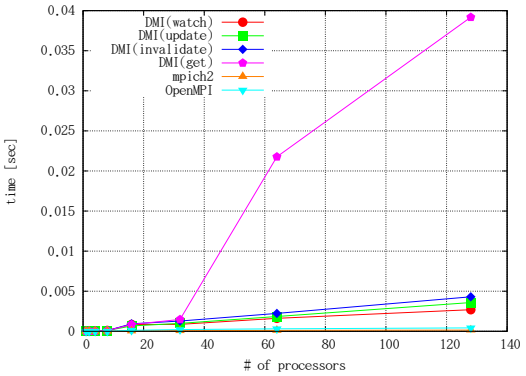

図 6.7　Allreduce の実行時間.

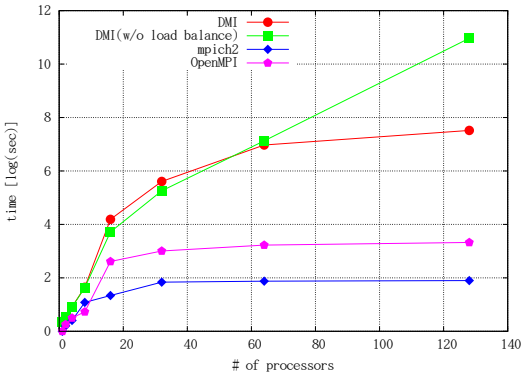

図 6.8　Broadcast の実行時間.

合（DMI（INVALIDATE））.

- 図 3.8 において (##) で示した行の DMI_read() 関数のモードを UPDATE モードにする場合（DMI（UPDATE））.
- **3.5.3** 節で述べたように，DMI_read() 関数をビジーウェイトで呼び出すのではなく，かわりに DMI_watch() 関数によってアドレスの変更監視を行う場合（DMI（watch））.
- mpich2 の MPI_Allreduce() 関数を使った場合（mpich2）.
- OpenMPI の MPI_Allreduce() 関数を使った場合（OpenMPI）.

上記の各場合に対して，n 個のプロセッサを使用して 300 回の Allreduce を行うのに要した実行時間を測定した．ただし，すべてのプロセッサが 300 回連続で Allreduce を休むことなく呼び出すわけではなく，各プロセッサ i は，各 Allreduce を呼び出す直前に $0.05(i+1)/n$ 秒の休止を行うようにした．これは，一般の並列プログラムにおけるプロセッサ間の負荷バランスの崩れをシミュレートするためである．一般に，多数のプロセッサで Allreduce を呼び出す場合には，すべてのプロセッサが Allreduce を同一のタイミングで呼び出すことはまれであり，重要なのは，各プロセッサが Allreduce を呼び出すタイミングが多少ずれた場合の Allreduce の性能だからである．以上の条件のもとで n 個のプロセッサを使って 300 回の Allreduce を行い，プロセッサ $n-1$ が Allreduce に要していた実行時間を測定して，Allreduce1 回あたりの平均時間を算出した．ここで，プロセッサ $n-1$ はもっとも長い休止を行うため，すべてのプロセッサのなかで一番最後に Allreduce に突入するプロセッサである．

6.3.3.2　結果と考察

プロセッサ数を変化させた場合の Allreduce1 回あたりの平均時間を図 6.7 に示す．128 プロセッサを使用した場合の，DMI（GET），DMI（INVALIDATE），DMI（UPDATE），DMI（watch），mpich2，OpenMPI の実行時間は，それぞれ 0.0391 秒，0.00430 秒，0.00358 秒，0.00269 秒，0.000152 秒，0.000423 秒である．第 1 に，DMI（GET）の実行時間がとくに遅い理由は，図 3.8 において，ビジーウェイトしている最中の (##) の DMI_read() 関数が毎回ページフォルトを引き起こし，オーナーとの通信を引き起こすためである．第 2 に，実行時間が DMI（GET）>DMI（INVALIDATE）

>DMI（UPDATE）となっている理由は，引き起こされるページフォルトの回数に起因している．DMI（UPDATE）の場合には，*wait_addr* のデータが update 型でキャッシュされているため，(##) の DMI_read() 関数がページフォルトを引き起こすことは（1 回目の Allreduce における 1 回目の DMI_read() 関数をのぞいては）ありえない．DMI（INVALIDATE）の場合には，*wait_addr* のデータが invalidate 型でキャッシュされているため，ページフォルトの回数は 0 回ではないが，DMI（GET）よりははるかに少なくなる．第 3 に，DMI（watch) の方が DMI（UPDATE）より性能が 33% 高い理由は，ビジーウェイトを行うか行わないかの性能差に起因している．両者ともページフォルトの回数は 0 回であり内部的に発生する通信もまったく同じであるが，DMI（UPDATE）の場合には，計算スレッドがビジーウェイトを行っているために，それが receiver スレッドや handler スレッドなどの他の管理用スレッドの実行を阻害してしまう．この結果より，DMI におけるアドレスの変更監視は，高性能な同期を実現するうえで重要な API であるといえる．第 4 に，DMI の性能が mpich2 や OpenMPI より劣る理由は特定できていないが，DMI（UPDATE）や DMI（watch）ではオーナーを根とする flat tree のトポロジに沿って Allreduce が実現されるのに対して，mpich2 や OpenMPI では Recursive Halving[176] などのより効率的なトポロジ上で Allreduce が実現されている可能性が考えられる．

6.3.4 Broadcast におけるデータ転送の動的負荷分散の効果

6.3.4.1 実験設定

Broadcast を題材にして，**3.4** 節で述べたデータ転送の動的負荷分散の効果を評価した．1 個のプロセッサが n 個のプロセッサに対して 512 MB のデータを Broadcast するのに要する実行時間を，「動的負荷分散なしの DMI（DMI（w/o load balance））」，「動的負荷分散ありの DMI（DMI）」，「mpich2 の MPI_Broadcast() 関数（mpich2）」，「OpenMPI の MPI_Broadcast() 関数（OpenMPI）」の 4 種類の場合について比較した．5 回の Broadcast の実行時間を測定し，Broadcast1 回あたりの平均時間を算出した．

6.3.4.2 結果と考察

プロセッサ数を変化させた場合の Broadcast1 回あたりの平均時間を図 6.8 に示す．DMI における Broadcast では，図 3.6（E）に示すような完全な二項木状のトポロジに沿ってページが転送される．128 プロセッサを使用した場合の DMI は 7.51 秒，DMI（w/o load balance）は 10.9 秒であり，データ転送の負荷分散によって 45% もの性能向上を達成できている．一方で，DMI は，mpich2 や OpenMPI の MPI_Broadcast() 関数と比較するとかなり遅い．この原因は特定できていないが，Allreduce と同様に，データ転送のトポロジの違いが関係しているのではないかと考えられる．

6.3.5 STREAM ベンチマークにおける遠隔スワップの性能

6.3.5.1 実験設定

第 1 に，STREAM ベンチマーク [130] における copy, scale, add, triadd の 4 種類の演算を題材として，DMI における遠隔スワップの性能および非同期 read/write の効果を評価した．この STREAM ベンチマークでは，各要素が double 型の 12 GB の配列 A, B, C と定数 s を用意し，copy：$\forall i\ C[i] = A[i]$，scale：$\forall i\ B[i] = sC[i]$，add：$\forall i\ C[i] = A[i] + B[i]$，triadd：$\forall i\ A[i] = B[i] + sC[i]$ の 4 種類の演算

を，以下の 4 種類の方法で行って実行時間を測定した：

- 通常のメモリアクセスによって演算を行った場合（swap）．**6.1** 節で述べたように，1 ノード内のメモリは 24 GB，スワップ領域は 24 GB であるため，合計 24 GB のメモリを要求する copy と scale，合計 36 GB のメモリを要求する add と triadd では，OS によるディスクスワップが発動した．
- HDD から必要なデータを少しずつ read/write して演算を行った場合（HDD）．配列 A，B，C を HDD に格納し，メモリ上に 64 MB の配列 a，b，c を用意した．各演算を行う際には，(1) HDD 上の各配列 A，B，C（のうち必要なもの）からデータを 64 MB ずつメモリ上の配列 a，b，c に読み込み，(2) メモリ上の配列 a，b，c を使って演算し，(3) 64 MB の演算結果を HDD 上の配列に書き込む，という操作を 12 GB/64 MB=192 回繰り返した．OS のページキャッシュの影響を除外するため，1 回の演算を行うたびにページキャッシュをフラッシュし，確実に HDD から配列が読み込まれるようにした．
- DMI の遠隔スワップを利用して read/write して演算を行った場合（DMI）．16 ノード上の 16 プロセスを利用し，3 個の 12 GB の配列 A，B，C を，それぞれページサイズ 64 MB でグローバルアドレス空間上に確保した．次に，各配列 A，B，C に関して，先頭から $i \times 768$ MB 以上 $(i+1) \times 768$ MB 未満 $(0 \leq i < 16)$ のアドレス領域が，プロセス i のメモリプールにはオーナー権をともなった DOWN_VALID 状態で存在し，他のプロセスのメモリプールには INVALID 状態で存在するように各配列を分散配置した．さらに，プロセス 0 は，ローカルアドレス空間上に 64 MB の配列 a，b，c を確保した．各演算を行う際には，(1) プロセス 0 はグローバルアドレス空間上の各配列 A，B，C（のうち必要なもの）からデータを 64 MB ずつメモリ上の配列 a，b，c に GET モードで read し，(2) メモリ上の配列 a，b，c を使って演算し，(3) 64 MB の演算結果をグローバルアドレス空間上の配列に PUT モードで write する，という操作を 12 GB/64 MB=192 回繰り返した．なお，1 回の演算を行うたびに配列の分散配置をやりなおした．
- DMI の遠隔スワップを利用して非同期 read/write を使って演算を行った場合（DMI（async））．上記の DMI においてグローバルアドレス空間から 64 MB ずつ read するときに，非同期 read を用いて 20×64 MB=1.28 GB 先のデータまでをプリフェッチした．さらに，グローバルアドレス空間に対して 64 MB ずつ write するときに，非同期 write を用いて 20×64 MB=1.28 GB 前のデータまでをポストストアした．

上記の 4 種類の方法に関して各演算に対して 5 回の測定を行い，各演算 1 回あたりに要する平均時間を算出した．

第 2 に，ページ置換の効果を観察するため，各プロセスにおけるメモリプールの使用量を 4 GB に設定したうえで，上記の DMI の場合に関して，PUT モードで write するかわりに，EXCLUSIVE モードで write するようにプログラムを書きなおした．この状況で copy 演算を行うと，配列 A と C のすべてのページのオーナーがプロセス 0 に移動することになるため，プロセス 0 のメモリプールには合計 24 GB のデータが転送されることになる．このとき，0.4 秒ごとにプロセス 0 のメモリプールの使用量

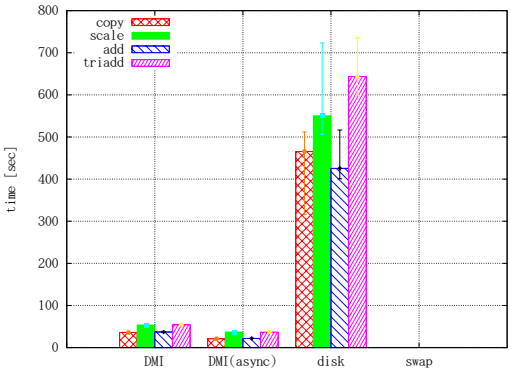

図 6.9　STREAM ベンチマークの実行時間比較（swap は値が大きすぎるためグラフ中にプロットしていないが，swap における copy，scale，add，triadd の実行時間は，それぞれ，13321 秒，43657 秒，24866 秒，49376 秒である）．

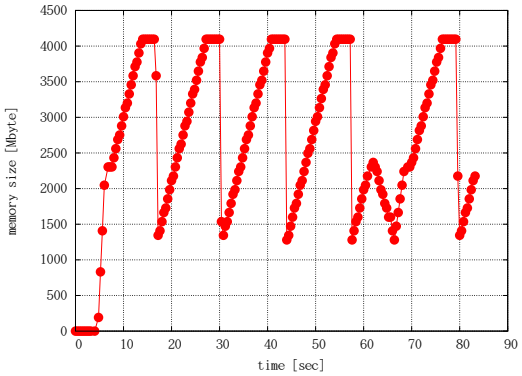

図 6.10　STREAM ベンチマークにおけるプロセス 0 のメモリプールの消費量の時間的変化．

を観測し，ページ置換が正常に動作してメモリプールの使用量が 4 GB 程度に抑えられているかどうかを検証した．

6.3.5.2　結果と考察

第 1 に，4 種類の方法に対する STREAM ベンチマークの結果を図 6.9 に示す．バーは 5 回の測定値の平均値を表し，エラーバーの上端と下端はそれぞれ 5 回の測定値の最大値と最小値を表す．図 6.9 より，copy，scale，add，triadd のそれぞれに関して，DMI は HDD の 22.1 倍，14.8 倍，19.5 倍，17.5 倍の性能を達成しており，swap の 633 倍，1179 倍，1141 倍，1345 倍の性能を達成している．この結果より，DMI における遠隔スワップは，メモリインテンシブな（並列）計算に対して，ローカルな HDD やディスクスワップよりもはるかに高性能な記憶階層を提供できているといえる．さらに，copy，scale，add，triadd のそれぞれに関して，DMI（async）は DMI より，1.70 倍，1.44 倍，1.69 倍，1.47 倍の性能を達成しており，非同期 read/write によるプリフェッチとポストストアが通信時間を隠蔽するための効果的な手段として機能していることがわかる．

第 2 に，DMI の遠隔スワップにおいて，copy 演算全体を通じてプロセス 0 に合計 24 GB のデータを読み込んだ場合，プロセス 0 のメモリプールの使用量がどのように変化したかを図 6.10 に示す．図 6.10 より，メモリプールの使用量が 4 GB に達した付近でページ置換が発動していることが読みとれる．**4.3** 節で述べたように，現在の実装では，ページ置換の発動時にはメモリプールの最大使用可能量の 0.7 倍の容量のページを追い出そうと試みるため，ページ置換が終了した時点でのメモリプールの使用量が，おおよそ 4 GB×(1 − 0.7) =1.2 GB になっている．

表 6.3　プログラム行数の比較.

種類	MPI の行数 [行]	DMI の行数 [行]
NAS Parallel Benchmark の EP	201	230
マンデルブロ集合の描画	276	235
横ブロック分割による行列行列積	126	178
Fox アルゴリズムによる行列行列積	202	229
ランダムサンプリングソート	400	427
N 体問題	178	215
ヤコビ法	147	168
有限要素法	2572	2368
ページランク計算	738	693
同期的な最短路計算	747	645
非同期的な最短路計算	自然には記述できない	651

6.4　プログラマビリティの比較

以降で評価する 11 種類のアプリケーションに対する MPI のプログラム行数と DMI のプログラム行数を比較したものを，表 6.3 に示す．この行数は，コメント行および空行をのぞいた行数である．NAS Parallel Benchmark の EP からヤコビ法までが，定型的で基本的なアプリケーションであり，有限要素法から非同期的な最短路計算までが，グローバルアドレス空間への非定型なアクセスをともなう応用的なアプリケーションである．

表 6.3 より，基本的なアプリケーションについては，DMI のプログラム行数は MPI とほぼ同じであり，むしろやや長いことがわかる．この理由は，第 **3** 章で述べたように，DMI の API は，強力な性能最適化を見通しよく施せるようにすることを優先して設計されていて，グローバルアドレス空間に対して透過的に read/write できるわけではないためである．DMI と MPI で基本的なアプリケーションを同一のアルゴリズムによって記述した場合，DMI のプログラムは，MPI のプログラムにおける MPI_Send() 関数/MPI_Recv() 関数を DMI_read() 関数/DMI_write() 関数に置き換えただけのようなプログラムになることが多い．

一方で，応用的なアプリケーションについては，DMI のプログラム行数は MPI よりも有意に短い．この理由は，DMI では read-write-set を用いることで，非定型な並列計算をグローバルビュー型のグローバルアドレス空間モデルに基づいて記述できるためである．応用的なアプリケーションにおける DMI のプログラマビリティについては，**6.6.1** 節から **6.6.4** 節までで個々に議論する．

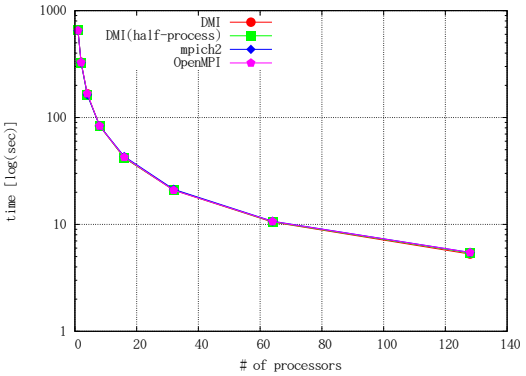

図 6.11　NAS Parallel Benchmark の EP の実行時間.

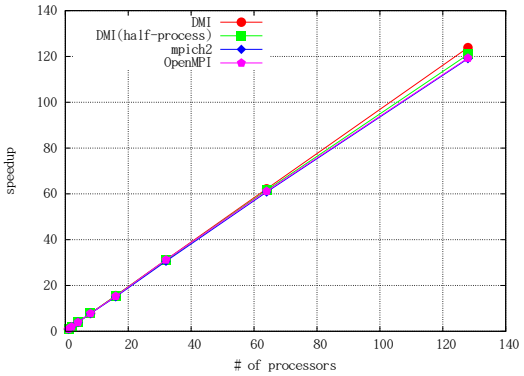

図 6.12　NAS Parallel Benchmark の EP のウィークスケーラビリティ.

6.5　基本的なアプリケーション

6.5.1　NAS Parallel Benchmark の EP

6.5.1.1　実験設定

NAS Parallel Benchmark の EP を題材にして，DMI と mpich2 と OpenMPI の性能を比較した[*1]．NAS Parallel Benchmark の EP では，$a = 2^{32}$ 個の乱数の組 (x_j, y_j) を生成し，$t_j = x_j^2 + y_j^2 \leq 1$ を満たす (x_j, y_j) に対して $\max(|x_j\sqrt{(-2\log t_j)/t_j}\,|, |y_j\sqrt{(-2\log t_j)/t_j}\,|)$ の度数分布を計算する．n 個のプロセッサで実行する場合の並列アルゴリズムは以下のとおりである：

(1) $a = 2^{32}$ 個を，均等な n 個のタスクに分割する.
(2) 各プロセッサ i は，i 番目のタスクを担当し，そのタスクに対応する度数分布を計算する.
(3) Reduce によって，すべてのプロセッサが計算した度数分布を足し合わせて，全体の度数分布を計算する.

6.5.1.2　結果と考察

プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.11 に，そのウィークスケーラビリティを図 6.12 に示す．ウィークスケーラビリティのグラフの縦軸は，（処理系 X を使って 1 個のプロセッサで実行した場合の実行時間）/（処理系 X を使って n 個のプロセッサで実行した場合の実行時間）を示す．グラフ中の DMI（half-process）は，第 **9** 章で導入する改造カーネル上での実験結果であり，これについては **10.4** 節で説明する．また，128 プロセッサで実行した場合における，全体の実行時間と「計算実行時間」を図 6.13 に示す．ここで，「計算実行時間」とは，DMI_read()

[*1] MPI のプログラムは NAS Parallel Benchmark の公式サイトから配布されているが，それらは Fortran で記述されている．本実験では，コンパイラの違いに左右されることなく DMI と MPI の性能を比較するために，配布されている MPI のプログラムを C 言語で書きなおしたものを使用した.

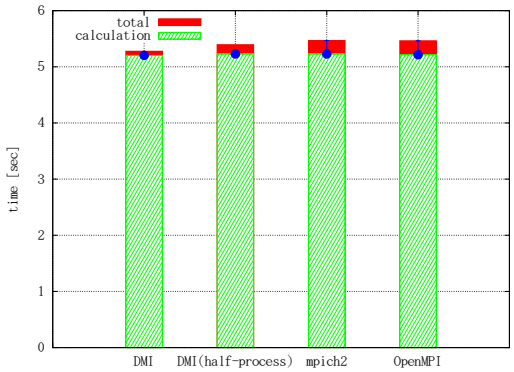

図 6.13　NAS Parallel Benchmark の EP における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）．

関数/DMI_write() 関数，MPI_Recv() 関数/MPI_Send() 関数などの通信用の関数に消費された時間をすべてのぞいた，純粋にアプリケーションの計算に消費されていた時間を意味する．128 プロセッサで実行する場合，「計算実行時間」としては 128 個の値が得られることになるが，図 6.13 では，128 個の「計算実行時間」の平均値を緑色のバーおよび青色の点で表し，その最大値と最小値を青色のエラーバーとして表している．

図 6.13 において大部分の時間が計算実行時間に消費されていることからわかるように，NAS Parallel Benchmark の EP は embarrassingly parallel なアプリケーションであり，必要となる通信は複数回の Reduce 操作のみである．よって，処理系による性能差はほとんど見られず，図 6.11 や図 6.12 では理想的なリニアスケーラビリティが得られている．とくに，DMI は mpich2 や OpenMPI と同等の性能を達成できている．

6.5.2　NAS Parallel Benchmark の EP における再構成

6.5.2.1　実験設定

NAS Parallel Benchmark の EP について，DMI を使って非同期的にプロセスを参加/脱退させる実験を行った．アルゴリズムは次のとおりである：

(1) $a = 2^{34}$ 個を，均等な 1024 個のタスクに分割する．
(2) 図 3.9 に示す要領で，新たに参加してきたプロセスに対してはプロセッサ数個のスレッドを生成し，脱退しようとしているプロセスからは，そのプロセス上の各スレッドに対して終了通知を送ることでスレッドを回収する．
(3) 各スレッド i は，自分のスレッドに対して終了通知が届いていないかどうかを検査し，届いていればすぐに終了する．届いていなければ，1024 個のタスクのうち未処理のタスクを 1 個とってきて，タスクを実行し，計算結果としての度数分布をグローバルアドレス空間に書き込んだあと，再び（3）に戻る．

(4) 1024 個のすべてのタスクが終了した時点で，いずれか 1 個のスレッドが 1024 個の度数分布を足し合わせて全体の度数分布を計算する．

1 ノードあたり 1 プロセスを生成することとし，初期的にはノード 0 からノード 3 の 4 ノードで実行し（合計 4 ノード，32 プロセッサ），しばらくしてからノード 4 からノード 15 の 12 ノードを参加させ（合計 16 ノード，128 プロセッサ），さらにしばらくしてからノード 0 からノード 7 の 8 ノードを脱退させた（合計 8 ノード，64 プロセッサ）．このとき，スレッドがどのように生成/破棄され，各スレッドが 1024 個のタスクをどのように処理するのかの様子を調べた．

6.5.2.2 結果と考察

スレッドの生成/破棄の様子と各スレッドがタスクを処理する様子を図 6.14 に可視化する．図 6.14 において，横軸は時間，縦軸は 128 個のスレッド，青い長方形（task）は 1 個のタスク，緑色の長方形（wait）は何らかの処理を待機していることを表す．NAS Parallel Benchmark の EP の場合，すべてのタスクの負荷はほぼ均等なので，青い長方形の横幅はほぼ等しくなっている．たとえばスレッド 40 は，途中で生成され，4 個のタスクを実行したあと，破棄されたことがわかる．図 6.14 より，以下の挙動が起きたことが読みとれる：

(1) 時刻 0〜2 秒の間にノード 0 からノード 3 が参加して，それらのノード上にスレッド 0 からスレッド 31 が生成された．
(2) 時刻 10〜14 秒の間にノード 4 からノード 15 が参加して，それらのノード上にスレッド 32 からスレッド 127 が生成された．
(3) 時刻 16〜28 秒の間にノード 0 からノード 7 が脱退して，それらのノード上に存在していたスレッド 16 からスレッド 31，スレッド 40 からスレッド 55，スレッド 64 からスレッド 71，スレッド 80 からスレッド 87，スレッド 96 からスレッド 103，スレッド 120 からスレッド 127 までが破棄された．
(4) 最後にスレッド 104 が，すべてのタスクの終了を待機して，全体の度数分布の計算を行った．

図 6.14 より，DMI は，**非同期的なプロセスの参加/脱退を越えて，動的に並列度を増減させながら並列計算を継続実行できた**ことがわかる．なお，プロセスの脱退に時間的な幅が生じている理由は，スレッドの終了通知を出してから実際に各スレッドが終了するまでに若干の時間を要することと，すべてのプロセスを 1 個ずつ脱退させていることに起因している．具体的には，NAS Parallel Benchmark の EP の場合には，スレッドの終了が可能なタイミングがタスクとタスクの間に限定されるため，ある時点でスレッド i の終了通知を出したとしても，その時点でスレッド i が実行しているタスクが終了するまでは，スレッド i は終了してくれない．この遅延が各プロセスの脱退に要する時間を伸ばし，かつ，図 3.9 のようなプログラムの書き方では，プロセスの脱退は 1 個ずつ順に行われることになるため，すべてのプロセスが脱退するまでに時間を要してしまう．改善策としては，プロセスの脱退を処理するときに，図 3.9 のように，「プロセス i 上のスレッドに終了通知を出す → プロセス i 上のスレッドを回収する → プロセス i を脱退させる」という処理を脱退しようとしている各プロセス i に対して繰り返すのではなく，「脱退しようとしているすべてのプロセス上のスレッドに終了通知を出す → 脱退しようと

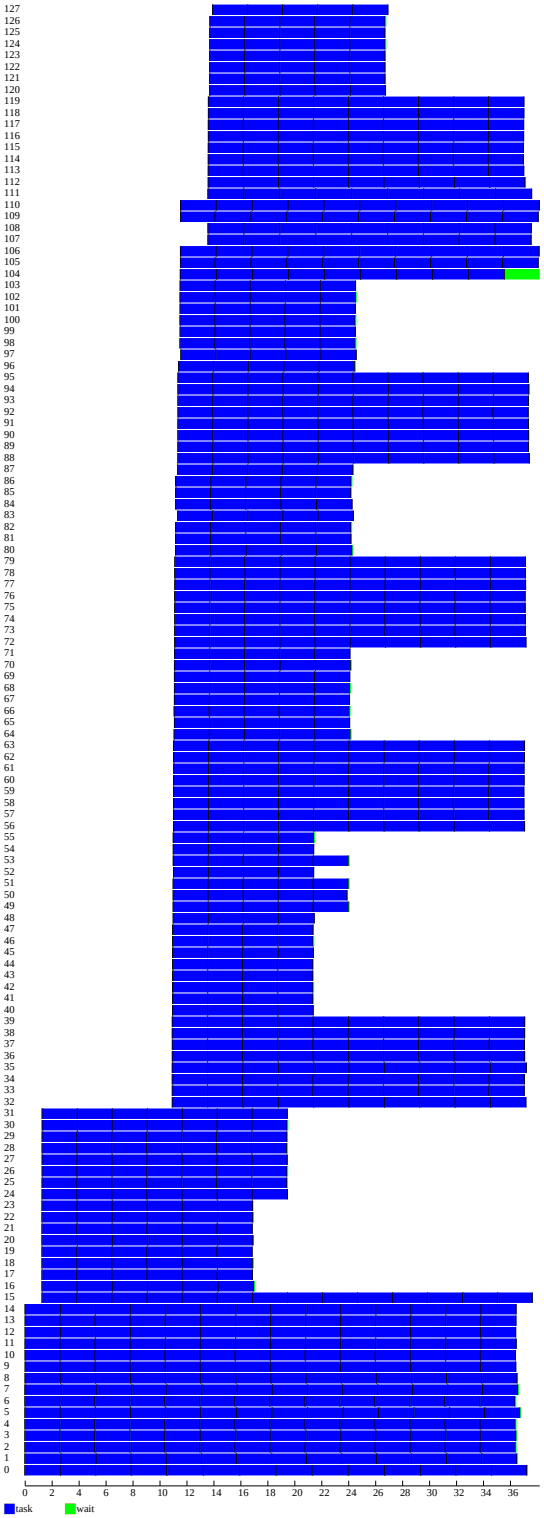

図 6.14　NAS Parallel Benchmark の EP を動的に再構成した場合における，各プロセッサに対するタスク割り当ての様子.

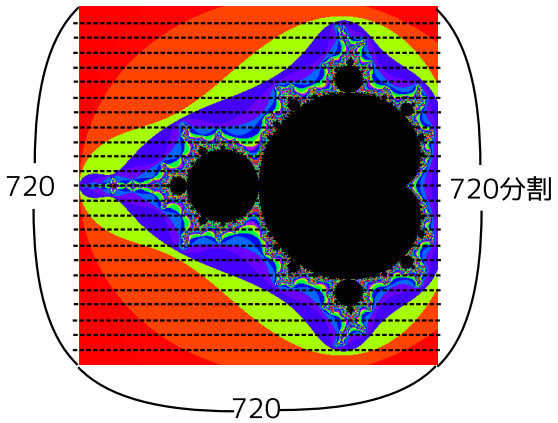

図 6.15　マンデルブロ集合.

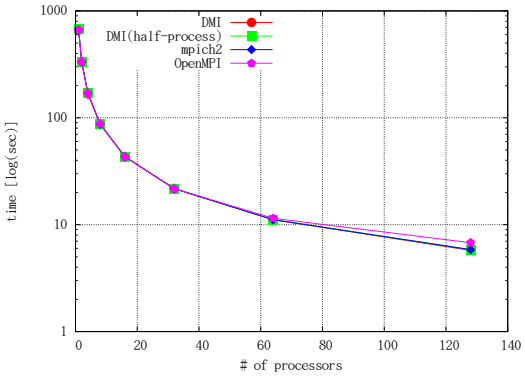

図 6.16　マンデルブロ集合の描画の実行時間.

しているすべてのプロセス上のスレッドを回収する → 脱退しようとしているすべてのプロセスを脱退させる」というように，まとめてスレッドの終了通知と回収を行う方法が考えられる．これにより，各スレッドに終了通知を出してから各スレッドが実際に終了するまでの遅延を隠蔽できる．

6.5.3　マンデルブロ集合の描画

6.5.3.1　実験設定

マンデルブロ集合の描画を題材にして，DMI と mpich2 と OpenMPI の性能を比較した．マンデルブロ集合とは，$z_0 = 0$，$z_{n+1} = z_n^2 + c$ で定義される複素数列 $\{z_n\}$ が $n \to \infty$ で発散しないような複素数 c の範囲を描画する問題である．マンデルブロ集合の描画は embarrassingly parallel なアプリケーションであるが，発散判定までの演算回数が描画範囲によって大きく異なることをふまえて，以下のようなアルゴリズムを用いた：

(1) 横 720× 縦 720 の描画領域全体を横方向に 720 分割し，720 個のタスクを用意する（図 6.15）.
(2) 各プロセッサ i は，タスクが存在するかぎり，未処理のタスクを 1 個とってきて，そのタスクが指定する描画領域内の各点 c について発散判定を行う．描画結果をプロセッサ 0 に送信したあと，再び（2）に戻る.

なお，発散を判定するまでのイテレーション数の上限値は 1000000 とした．図 6.15 における黒い領域が発散した領域であり，もっとも多くの演算回数を必要とする.

6.5.3.2　結果と考察

プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.16 に，そのウィークスケーラビリティを図 6.17 に示す．また，128 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.18 に示す.

図 6.17 より，DMI は mpich2 と同等のウィークスケーラビリティを達成できていることがわかる．DMI および mpich2 のスケーラビリティがリニアスケーラビリティより若干劣っている理由は，図

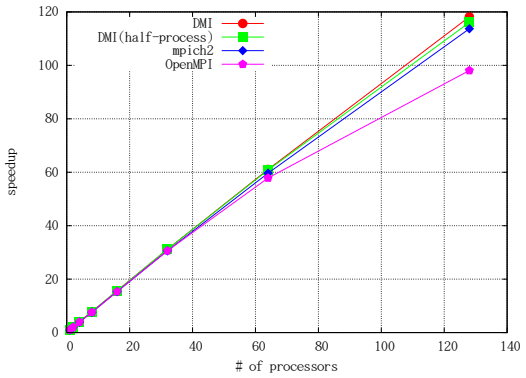

図 6.17　マンデルブロ集合の描画のウィークスケーラビリティ．

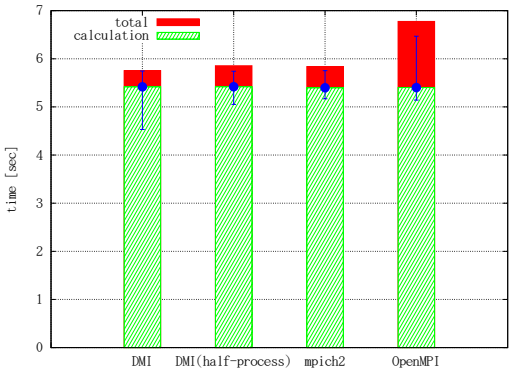

図 6.18　マンデルブロ集合の描画における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）．

6.18 において DMI と mpich2 のエラーバーの上限値と全体の実行時間との差が小さいことから判断すると，通信のためのオーバヘッドが原因ではなく，各タスクの負荷が異なるためにプロセッサ間の計算実行時間にばらつきが生じていることが主因だとわかる．実際，このマンデルブロ集合の描画において発生する通信量は，合計 $720 \times 720 \times \text{sizeof(double)}$=3.95 MB 程度にすぎない．また，128 プロセッサ実行時の OpenMPI の速度向上度が DMI や mpich2 よりも低い理由は，偶然，OpenMPI の実験においてタスクの負荷分散がうまく均等化しなかっただけだと考えられる．なぜなら，図 6.18 を見ると，DMI，mpich2，OpenMPI における計算実行時間の平均値はほぼ一致しており，OpenMPI の全体の実行時間が DMI や mpich2 よりも遅いことの直接の原因は，OpenMPI におけるエラーバーの上限値が DMI や mpich2 におけるエラーバーの上限値よりも高くなっていることに起因しているためである．いずれにせよ，DMI は mpich2 や OpenMPI と同等の性能を達成できている．

6.5.4　マンデルブロ集合の描画における再構成

6.5.4.1　実験設定

マンデルブロ集合の描画について，DMI を使って非同期的にプロセスを参加/脱退させる実験を行った．アルゴリズムは次のとおりである：

(1) 横 1024× 縦 1024 の描画領域全体を横方向に 1024 分割し，1024 個のタスクを用意する．
(2) **6.5.2** 節で述べた NAS Parallel Benchmark の EP と同様にして，参加/脱退するプロセスに対して動的にスレッドを生成/破棄する．
(3) 各スレッド i は，自分のスレッドに対して終了通知が届いていないかどうかを検査し，届いていればすぐに終了する．届いていなければ，未処理のタスクを 1 個とってきて，そのタスクが指定する描画領域内の各点 c について発散判定を行う．描画結果をプロセッサ 0 に送信したあと，再び（3）に戻る．

6.5.2 節で述べた NAS Parallel Benchmark の EP と同様にして，初期的にはノード 0 からノード 3 の 4 ノードで実行し（合計 4 ノード，32 プロセッサ），しばらくしてからノード 4 からノード 15 の 12 ノードを参加させ（合計 16 ノード，128 プロセッサ），さらにしばらくしてからノード 0 からノード 7 の 8 ノードを脱退させた（合計 8 ノード，64 プロセッサ）．

6.5.4.2 結果と考察

スレッドの生成/破棄の様子と各スレッドがタスクを処理する様子を図 6.19 に可視化する．マンデルブロ集合の描画の場合，各タスクの負荷が異なるため，青い長方形の幅が異なっている．図 6.19 より，以下の挙動が起きたことが読みとれる：

(1) 時刻 0〜2 秒の間にノード 0 からノード 3 が参加して，それらのノード上にスレッド 0 からスレッド 31 が生成された．
(2) 時刻 4〜9 秒の間にノード 4 からノード 15 が参加して，それらのノード上にスレッド 32 からスレッド 127 が生成された．
(3) 時刻 18〜135 秒の間にノード 0 からノード 7 が脱退して，それらのノード上に存在していたスレッド 8 からスレッド 47，スレッド 80 からスレッド 103 までが破棄された．

図 6.19 より，DMI は，非同期的なプロセスの参加/脱退を越えて，動的に並列度を増減させながら並列計算を継続実行できたことがわかる．ところが，マンデルブロ集合の描画の場合には 1 個のタスクの実行時間が長くなる場合があるため，スレッドに終了通知を送ってから回収できるまでの遅延が大きく，1 個ずつプロセスを脱退させる方法では，すべてのプロセスの脱退が完了するまでに長い時間を要してしまっている．よって，**6.5.2** 節で述べたように，脱退しようとしているすべてのプロセス上のスレッドへの終了通知と回収をまとめて処理するような改善が必要である．

6.5.5 横ブロック分割による行列行列積

6.5.5.1 実験設定

8192×8192 のサイズの行列 A，B，C に対して，横ブロック分割によって行列行列積 $AB = C$ を計算するアプリケーションを題材にして，DMI と mpich2 と OpenMPI の性能を比較した．n 個のプロセッサで実行する場合のアルゴリズムは以下のとおりである：

(1) プロセッサ 0 は，行列 A を横方向に n 個の均等なブロックに分割し，それをすべてのプロセッサに Scatter する．
(2) プロセッサ 0 は，行列 B を Broadcast する．
(3) 各プロセッサ i は，プロセッサ 0 から受信した横ブロック部分行列 A_i と行列 B を用いて，部分行列行列積 $A_iB = C_i$ を計算する．なお，部分行列行列積はキャッシュヒット率を考慮して ikj ループで記述する．
(4) 各プロセッサ i は部分行列 C_i をプロセッサ 0 に Gather する．

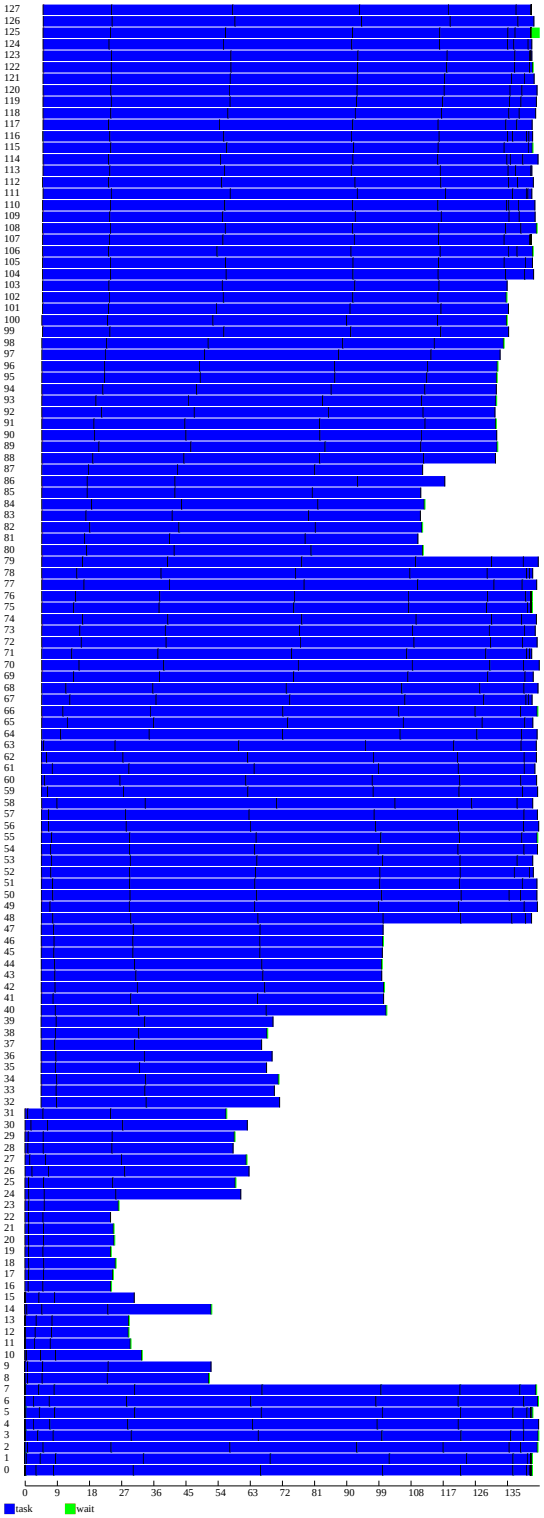

図 6.19　マンデルブロ集合の描画を動的に再構成した場合における，各プロセッサに対するタスク割り当ての様子.

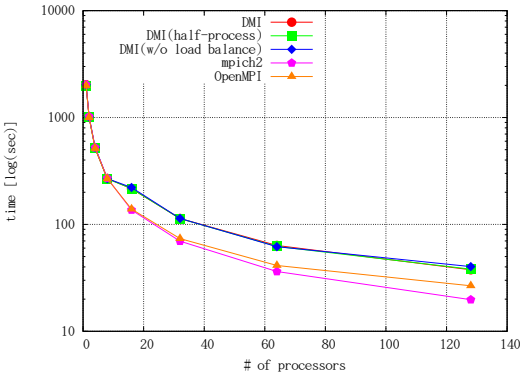

図 6.20　横ブロック分割による行列行列積の実行時間.

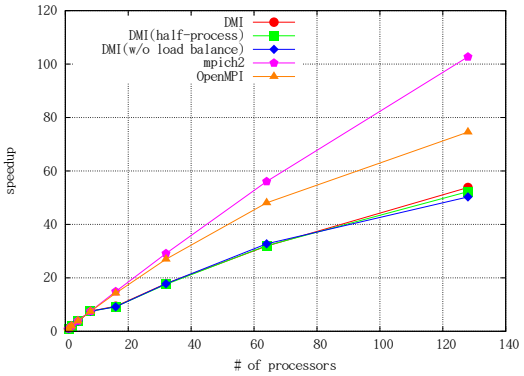

図 6.21　横ブロック分割による行列行列積のウィークスケーラビリティ.

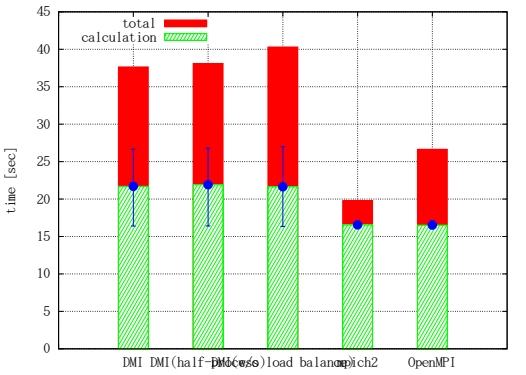

図 6.22　横ブロック分割による行列行列積における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）.

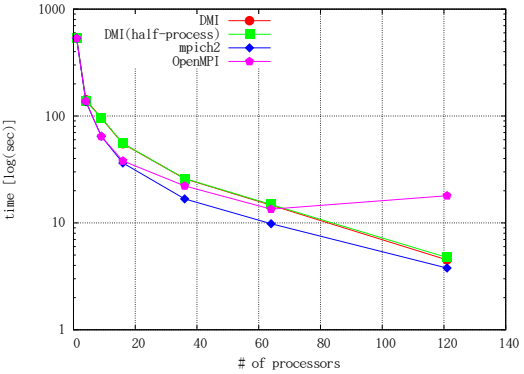

図 6.23　Fox アルゴリズムによる行列行列積の実行時間.

6.5.5.2　結果と考察

プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.20 に，そのウィークスケーラビリティを図 6.21 に示す．グラフ中の DMI（w/o load balance）は，DMI においてデータ転送の動的負荷分散を無効化させた場合の結果である．また，128 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.22 に示す.

第 1 に，DMI と mpich2 と OpenMPI の性能差について考える．図 6.20 および図 6.21 より，DMI の性能は mpich2 や OpenMPI よりも大きく劣っているが，図 6.22 より，性能劣化の原因は DMI における通信の遅さに起因することがわかる．具体的には，この行列行列積では，$8192 \times 8192 \times \mathrm{sizeof(double)} = 512$ MB の行列 B を Broadcast する操作が通信上の性能差を生む主因になっていることが，別のプロファイリングからわかっている．mpich2 や OpenMPI がどのような通信トポロジで Broadcast を実装しているかは未調査であるが，DMI における二項木状の通信トポロジとの違いが性能差に現れている

と考えられる．なお，図 6.22 において，mpich2 や OpenMPI の計算実行時間はすべてのプロセッサ間でほぼ均等化しているのに対して，DMI の計算実行時間が大きくばらついているのは次のような理由による．mpich2 や OpenMPI では，MPI_Broadcast() 関数などの集合関数が返るタイミングはすべてのプロセッサでほぼ揃うため，すべてのプロセッサが同時に計算を開始できる．そして，本実験の行列行列積では各プロセッサに対する計算負荷は完全に均等なので，すべてのプロセッサの計算実行時間は均等化する．これに対して，DMI では，Broadcast はすべてのプロセッサが DMI_read() 関数を呼び出すことによって実現されるため，DMI_read() 関数が早く返ったプロセッサから順に計算に移行することになる．よって，早く計算に移行したプロセッサは，DMI の receiver スレッドや handler スレッドなどの管理用スレッドが，他のプロセッサが発行した DMI_read() 関数に起因する read フォルトを処理するためにバックグラウンドで動作している最中に計算を実行することになるため，純粋に計算だけに集中できる場合と比較すると性能が落ちてしまう．とくに，DMI の計算実行時間のエラーバーの下限値は，各プロセスのなかで一番遅く DMI_read() 関数が返ったプロセッサの計算実行時間を表しているが，この下限値が mpich2 や OpenMPI の計算実行時間にほぼ一致している理由は，この下限値が，DMI の管理用スレッドに邪魔されることなく計算だけに集中できた場合の実行時間を表しているからである．いずれにせよ，DMI の性能を向上させるためには，Broadcast のような集合通信の最適化が重要であるといえる．

第 2 に，DMI と DMI（w/o load balance）の性能差について考える．図 6.22 より，128 プロセッサで実行した場合，DMI の全体の実行時間は DMI（w/o load balance）の全体の実行時間より 2.65 秒高速である．これは，**6.3.4** 節において 512 MB の Broadcast を単体で実験した場合に，DMI が DMI（w/o load balance）よりも 3.45 秒高速であったことをふまえると，妥当な結果であると考えられる．

6.5.6 Fox アルゴリズムによる行列行列積

6.5.6.1 実験設定

5280×5280 のサイズの行列 A，B，C に対して，Fox アルゴリズム [200] によって行列行列積 $AB = C$ を計算するアプリケーションを題材にして，DMI と mpich2 と OpenMPI の性能を比較した．n 個のプロセッサで実行する場合のアルゴリズムは以下のとおりである[*2] ：

(1) プロセッサ 0 は，行列 A，B を，それぞれ $\sqrt{n} \times \sqrt{n}$ 個の小行列に分割する．行列 A と B に対して，上から i（$0 \leq i < \sqrt{n}$）番目，左から j（$0 \leq j < \sqrt{n}$）番目の小行列を，それぞれ $A_{i,j}$，$B_{i,j}$ と表すことにする．プロセッサ 0 は，各 $A_{i,j}$ と $B_{i,j}$ をプロセッサ $i\sqrt{n} + j$ に対して送信する．以下では，プロセッサ $i\sqrt{n} + j$ を，2 次元的には第 i 行第 j 列に位置しているプロセッサであるという意味で $P_{i,j}$ と表す．

(2) 各プロセッサ i は，以下のステップを $\sqrt{n}$ 回繰り返す：

  (a) 第 k ステップにおいて，$z(i,k) = (i+k) \mod \sqrt{n}$ とすると，プロセッサ $P_{i,z}$ が，自分が持っている $A_{i,z(i,k)}$ を，自分と同一行のプロセッサたち $P_{i,*}$ に対して Broadcast する．

  (b) 各プロセッサ $P_{i,j}$ は，$C_{i,j} = C_{i,j} + A_{i,z(i,k)}B_{z(i,k),j}$ を計算する．

[*2] 簡単のため，n は平方数とする．

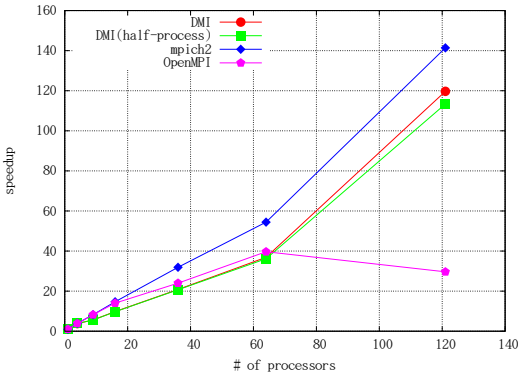

図 6.24　Fox アルゴリズムによる行列行列積のウィークスケーラビリティ．

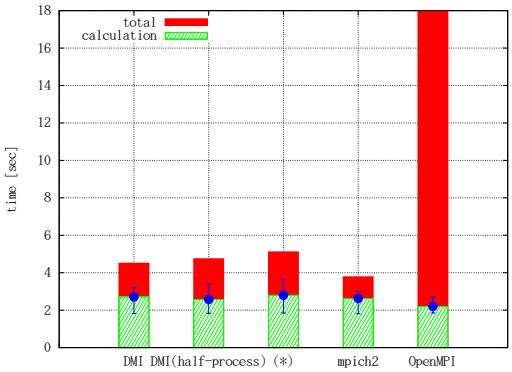

図 6.25　Fox アルゴリズムによる行列行列積における，全体の実行時間に占める計算実行時間の割合（121 プロセッサ実行時，(*) は DMI(w/o load balance) を表す）．

(c) 各プロセッサ $P_{i,j}$ は，列方向において自分の上のプロセッサ $P_{(i-1) \mod \sqrt{n},j}$ に対して $B_{z(i,k),j}$ を送信すると同時に，列方向において自分の下のプロセッサ $P_{(i+1) \mod \sqrt{n},j}$ から $B_{z(i+1,k),j}$ を受信する．なお，$z(i+1,k) = z(i,k+1)$ であるから，ここで受信した $B_{z(i+1,k),j}$ が，次の第 $k+1$ ステップにおいて $C_{i,j}$ への加算に利用する $B_{z(i,k+1),j}$ になることに注意する．

(3) 各プロセッサ $P_{i,j}$ 上の小行列 $C_{i,j}$ をプロセッサ 0 に Gather する．

6.5.6.2　結果と考察

プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.23 に，そのウィークスケーラビリティを図 6.24 に示す．グラフ中の DMI（w/o load balance）は，DMI においてデータ転送の動的負荷分散を無効化させた場合の結果である．また，121 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.25 に示す．

121 プロセッサ実行時の DMI，mpich2，OpenMPI の全体の実行時間は，それぞれ 4.50 秒，3.78 秒，17.9 秒である．ここで，横ブロック分割による行列行列積では DMI の性能が mpich2 や OpenMPI よりも大きく劣っていたにもかかわらず，Fox アルゴリズムの場合には，DMI が mpich2 と同等でかつ OpenMPI よりも高い性能を達成できているのは，次のような理由による．Fox アルゴリズムでは，Broadcast は同一行のプロセッサに対してしか行われない．そして，121 プロセッサで実行する場合であっても各行には 11 プロセッサしか存在しないため，1 ノードに 8 個の DMI スレッドまたは MPI プロセスを起動する本実験では，たかだか 3 ノード間での通信しか行われない．したがって，横ブロック分割の場合とは異なり，Fox アルゴリズムでは Broadcast の対象となるノード数が非常に少なく，実質的には point-to-point な通信しか発生しないため，横ブロック分割による行列行列積で見られたような，Broadcast の通信トポロジの違いによる性能差が現れなかったものと考えられる．なお，121 プロ

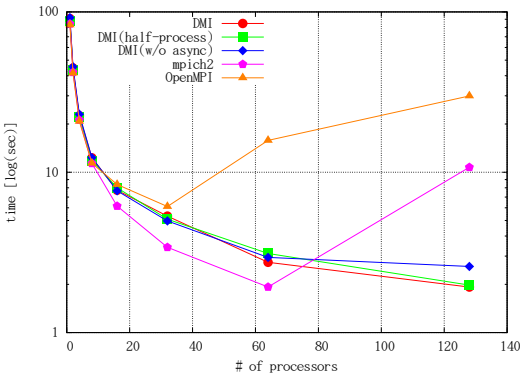

図 6.26　ランダムサンプリングソートの実行時間.

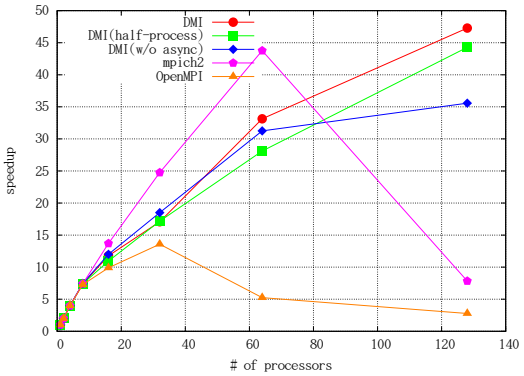

図 6.27　ランダムサンプリングソートのウィークスケーラビリティ.

セッサ実行時の OpenMPI の性能が著しく遅い理由については，図 6.25 より OpenMPI における通信の遅さが原因だとはわかるが，それ以上の理由は特定できていない.

6.5.7　ランダムサンプリングソート

6.5.7.1　実験設定

$a = 512 \times 10^6$ 個の整数のランダムサンプリングソート [200] を題材にして，DMI と mpich2 と OpenMPI の性能を比較した．n 個のプロセッサで実行する場合のアルゴリズムは以下のとおりである：

(1) 各プロセッサ i は，初期的に a/n 個のデータを持っている．各プロセッサ i は，自分が持っている a/n 個のなかからランダムに $z = 1280$ 個のデータを選び，これをプロセッサ 0 に送信する.
(2) プロセッサ 0 は，すべてのプロセッサから集めた nz 個のデータをクイックソートし，この nz 個のデータの n 分位点を求める.
(3) プロセッサ 0 は，求めた $n-1$ 個の n 分位点をすべてのプロセッサに対して Broadcast する.
(4) 各プロセッサ i は，プロセッサ 0 から受信した n 分位点に基づいて，自分の持っている a/n 個のデータを n 個の集合に振り分ける．このとき，j（$0 \le j < n$）番目の集合には，第 $j-1$ 分位点以上第 j 分位点未満のデータを入れる．各プロセッサ i は，各 j に対して，j 番目の集合をプロセッサ j に対して送信する．ここで All-to-all 型の通信が発生する.
(5) 各プロセッサ i は，すべてのプロセッサから受信した合計 n 個の集合に含まれるすべてのデータを対象にして，クイックソートを行う.

6.5.7.2　結果と考察

プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.26 に，そのウィークスケーラビリティを図 6.27 に示す．また，128 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.28 に示す．DMI では，n 分位点に基づいて n 個の集合を作ったあとで

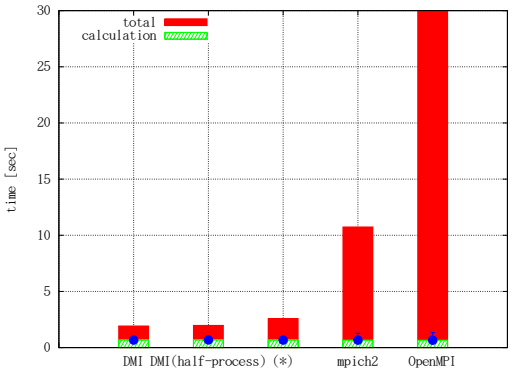

図 6.28　ランダムサンプリングソートにおける，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時，(*) は DMI(w/o async) を表す）．

All-to-all を行うときに非同期 read を利用している．具体的には，DMI において All-to-all を実現する場合，(1) 各プロセッサ i は a/n 個のデータを n 個の集合に振り分けたあとでそれらのデータをグローバルアドレス空間に対して write する，(2) バリアによってすべてのプロセッサを同期する，(3) 各プロセッサ i は各プロセッサ j から受信すべきデータをグローバルアドレス空間から read する，という 3 種類の操作が行われる．とくに (3) においては，各プロセッサ i は，グローバルアドレス空間上の n ヶ所から，GET モードでデータを read することになる．ここで，n 個の read を通常の read として発行してしまうとこれら n 個の read が逐次的に行われてしまうが，非同期 read として発行することにより n 個の read を並列に発行することができる[*3]．比較のため，ここで非同期 read を使わずに通常の read を使った場合の結果を DMI（w/o async）としてグラフ中に示す．

第 1 に，DMI と mpich2 と OpenMPI の性能差について考える．図 6.26 と図 6.27 より，DMI は mpich2 や OpenMPI より優れたスケーラビリティを達成していることがわかる．図 6.28 より，この性能差は通信実行時間の差に起因していることが読みとれる．また，この通信実行時間の大部分は All-to-all が支配していることが，別のプロファイリングからわかっている．乱数の一様性を考慮すれば，この All-to-all では，すべてのプロセッサ間で $512 \times 10^6 \times \mathrm{sizeof(int)}/128/128 = 119$ MB 程度のデータが送受信される．詳しくは **6.6.2** 節で述べるが，mpich2 や OpenMPI では密な All-to-all の性能が悪いことがわかっており，本実験ではその性能差が現れたものと考えられる．

第 2 に，DMI と DMI（w/o async）の性能差について考える．128 プロセッサ実行時には DMI は DMI（w/o async）よりも 30% 高速であり，非同期 read の有効性を確認できる．

[*3] なお，離散アクセスのグルーピングを使っても，非同期 read を使った場合と同様の通信を起こすことができる．

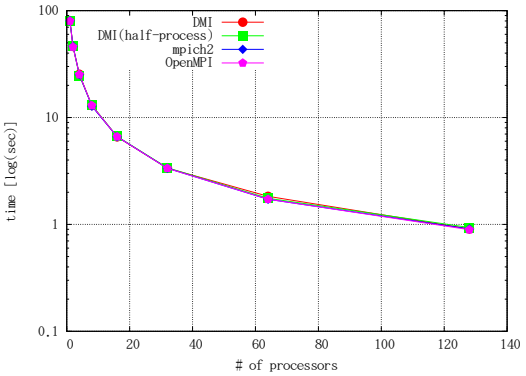

図 6.29　N 体問題の実行時間.

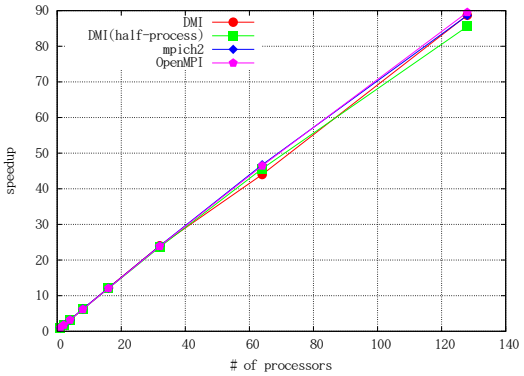

図 6.30　N 体問題のウィークスケーラビリティ.

6.5.8　N 体問題

6.5.8.1　実験設定

N 体問題を題材にして，DMI と mpich2 と OpenMPI の性能を比較した．n 個のプロセッサで実行する場合のアルゴリズム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) $l_1 \times l_2 \times l_3$ 個の格子状に並んだ粒子を n 等分し，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) 各プロセッサ i は以下の処理を反復する：
    (a) 各プロセッサ i は，プロセッサ i 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．つまり，粒子の位置を Allgather することで，すべてのプロセッサがすべての粒子の位置を把握できるようにする．
    (b) 各プロセッサ i は，すべての粒子の位置に基づいて，そのプロセッサ i の担当範囲の粒子とすべての粒子との相互作用を計算し，プロセッサ i の担当範囲の粒子の位置と速度を更新する．

本実験では $l_1 = l_2 = l_3 = 24$，反復回数は 10 回とし，イテレーション 1 回あたりの平均時間を算出した．

6.5.8.2　結果と考察

プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.29 に，そのウィークスケーラビリティを図 6.30 に示す．また，128 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.31 に示す．グラフ中の値は，すべて 1 イテレーションあたりの実行時間である．図 6.29 と図 6.30 より，DMI は mpich2 と OpenMPI と同等の性能を達成できていることがわかる．

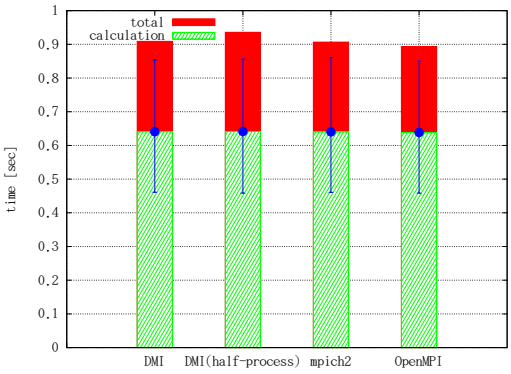

図 6.31　N 体問題における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）.

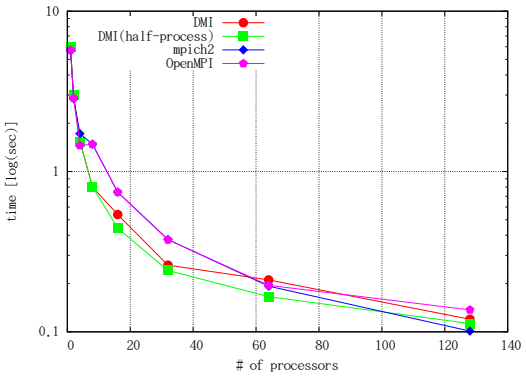

図 6.32　ヤコビ法の実行時間.

6.5.9　ヤコビ法による PDE ソルバ

6.5.9.1　実験設定

ヤコビ法による PDE（Partial Differential Equation）ソルバを題材にして，DMI と mpich2 と OpenMPI の性能を比較した．このヤコビ法では，3 次元立方体物体を 512^3 個の要素に分割して，27 点ステンシルのヤコビ反復法を用いて熱伝導偏微分方程式を解く．n 個のプロセッサで実行する場合のアルゴリズムは以下のとおりである：

(1) 3 次元立方体物体の領域全体を，n 個の均等な領域に 1 次元分割する．よって，各領域は左右 2 つの隣接領域を持ち，それぞれ 514^2 個の ghost 要素を持つ.
(2) 各プロセッサ i は以下の手順を反復する：
  (a) 各プロセッサ i は，プロセッサ $i-1$ とプロセッサ $i+1$ に対して，それぞれ，プロセッサ $i-1$ にとっての ghost 要素の値と，プロセッサ $i+1$ にとっての ghost 要素の値を送信する.
  (b) 各プロセッサ i は，プロセッサ $i-1$ とプロセッサ $i+1$ から，プロセッサ i の ghost 要素の値を受信する.
  (c) 各プロセッサ i は，プロセッサ i の担当領域に関して，27 点ステンシルの計算を行う.

反復回数は 10 回とし，イテレーション 1 回あたりの平均時間を算出した.

6.5.9.2　結果と考察

プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.32 に，そのウィークスケーラビリティを図 6.33 に示す．また，128 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.34 に示す．グラフ中の値は，すべて 1 イテレーションあたりの実行時間である．図 6.32 と図 6.33 より，DMI は mpich2 と同程度で，OpenMPI よりも高い性能を達成できて

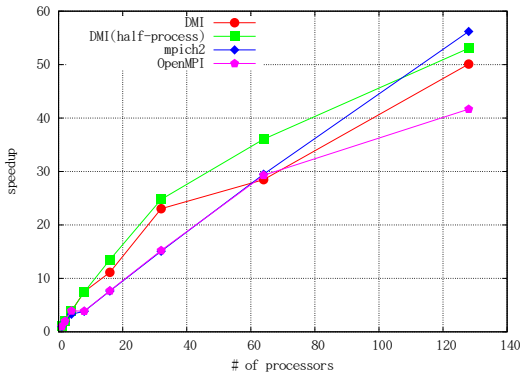

図 6.33　ヤコビ法のウィークスケーラビリティ．

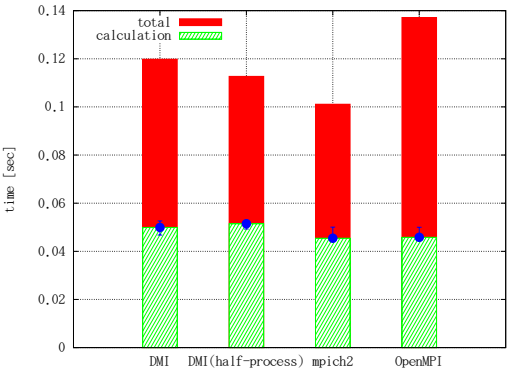

図 6.34　ヤコビ法における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）．

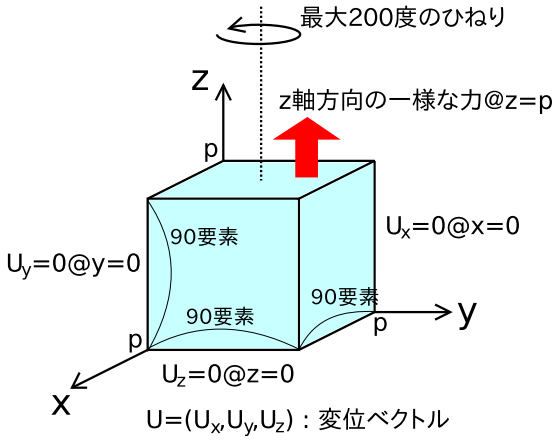

図 6.35　有限要素法による応力解析．

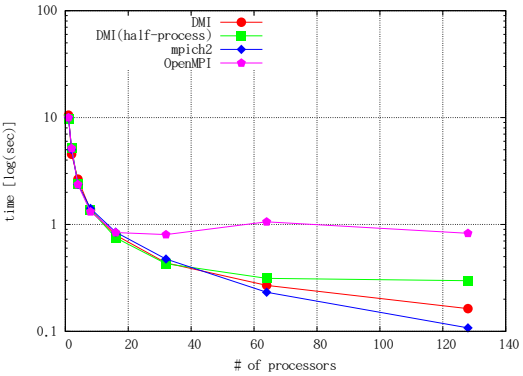

図 6.36　有限要素法の実行時間．

いることがわかる．

6.6　応用的なアプリケーション

本節では，有限要素法による応力解析，大規模な Web グラフのページランク計算と最短路計算を題材にして，DMI の性能とプログラマビリティを評価した．これらはグローバルアドレス空間への非定型なアクセスをともなう応用的なアプリケーションであり，**read-write-set を用いて記述した**．

6.6.1　有限要素法による応力解析

6.6.1.1　実験設定

3 次元立方体物体に対して，図 6.35 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

solve $A\boldsymbol{x} = \boldsymbol{b}$:
K = preconditioned matrix of A
$\boldsymbol{r_0} = \boldsymbol{b} - A\boldsymbol{x}$
initialize vectors $\boldsymbol{x_0}, \boldsymbol{r_0}, \boldsymbol{r_0^*}, \boldsymbol{p_0}, \boldsymbol{u_0}, \boldsymbol{y_0}, \boldsymbol{v_0}$ properly
initialize $\beta_{-1}, \xi_0, \eta_0$ properly
for $n = 0, 1, 2, \ldots$ until convergence **do**
$\boldsymbol{p_n} = K^{-1}\boldsymbol{r_n} + \beta_{n-1}(\boldsymbol{p_{n-1}} - \boldsymbol{u_{n-1}})$
$A\boldsymbol{p_n} = AK^{-1}\boldsymbol{r_n} + \beta_{n-1}(A\boldsymbol{p_{n-1}} - A\boldsymbol{u_{n-1}})$
$\alpha_n = (\boldsymbol{r_0^*}, \boldsymbol{r_n})/(\boldsymbol{r_0^*}, A\boldsymbol{p_n})$
$\xi_n = ((\boldsymbol{y_n}, \boldsymbol{y_n})(\boldsymbol{v_n}, \boldsymbol{r_n}) - (\boldsymbol{y_n}, \boldsymbol{r_n})(\boldsymbol{v_n}, \boldsymbol{y_n}))/((\boldsymbol{v_n}, \boldsymbol{v_n})(\boldsymbol{y_n}, \boldsymbol{y_n}) - (\boldsymbol{y_n}, \boldsymbol{v_n})(\boldsymbol{v_n}, \boldsymbol{y_n}))$
$\eta_n = ((\boldsymbol{v_n}, \boldsymbol{v_n})(\boldsymbol{y_n}, \boldsymbol{r_n}) - (\boldsymbol{y_n}, \boldsymbol{v_n})(\boldsymbol{v_n}, \boldsymbol{r_n}))/((\boldsymbol{v_n}, \boldsymbol{v_n})(\boldsymbol{y_n}, \boldsymbol{y_n}) - (\boldsymbol{y_n}, \boldsymbol{v_n})(\boldsymbol{v_n}, \boldsymbol{y_n}))$
$\boldsymbol{u_n} = K^{-1}(\xi_n A\boldsymbol{p_n} + \eta_n \boldsymbol{y_n}) + \eta_n \beta_{n-1}\boldsymbol{u_{n-1}}$
$\boldsymbol{z_n} = \xi_n K^{-1}\boldsymbol{r_n} + \eta_n \boldsymbol{z_{n-1}} - \alpha_n \boldsymbol{u_n}$
$\boldsymbol{y_{n+1}} = \xi_n AK^{-1}\boldsymbol{r_n} + \eta_n \boldsymbol{y_n} - \alpha_n A\boldsymbol{u_n}$
$\boldsymbol{x_{n+1}} = \boldsymbol{x_n} + \alpha_n \boldsymbol{p_n} + \boldsymbol{z_n}$
$\boldsymbol{r_{n+1}} = \boldsymbol{r_n} - \alpha_n A\boldsymbol{p_n} - \boldsymbol{y_{n+1}}$
$\beta_n = (\alpha_n/\xi_n)(\boldsymbol{r_0^*}, \boldsymbol{r_{n+1}}/(\boldsymbol{r_0^*}, \boldsymbol{r_n}))$
endfor

図 6.37 BiCGSafe 法のアルゴリズム.

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 [13] 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．n 個のプロセッサで実行する場合のアルゴリズムの概要は以下のとおりである．詳細は著者の資料 [197] を参考にされたい：

(1) 立方体物体を n 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 [36] とフィルインを考慮したとき，領域間の計算負荷が均等化するように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 [116] によって各領域内の節点を並び替える．
(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，$\boldsymbol{x}$ は求めるべき各節点の変位ベクトルである．
(3) 連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を，図 6.37 に示すような BiCGSafe 法 [64] と呼ばれる反復法を用いて，解ベクトル $\boldsymbol{x}$ が収束するまで反復計算を行う．図 6.37 からわかるように，BiCGSafe 法では 1

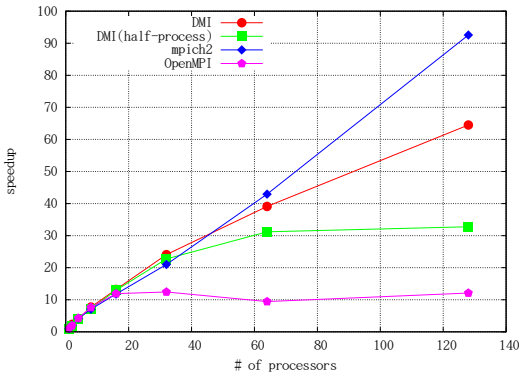

図 6.38　有限要素法のウィークスケーラビリティ．

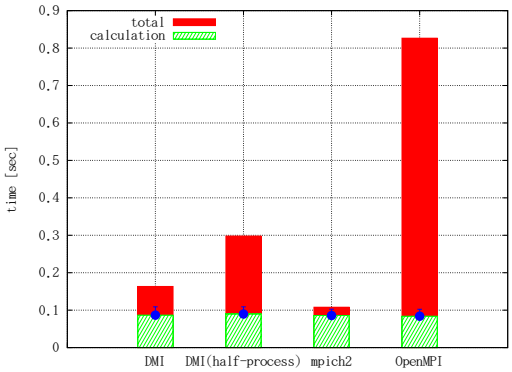

図 6.39　有限要素法における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）．

イテレーションあたり 22 回もの Allreduce やバリアが必要になる．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 [36] に基づく前処理を適用する．

6.6.1.2　結果と考察

第 1 に，アプリケーションの結果について述べる．128 プロセッサで実行した場合に解が収束するまでに要した反復回数は，mpich2 と OpenMPI が 175 回，DMI が 171 回だった．なお，収束するまでの反復回数が異なるのは次の理由による．図 6.37 における BiCGSafe 法では多数回の内積計算が必要であり，この内積計算は，すべてのプロセッサが計算した部分和を Allreduce によって足し込むことで行われる．このとき，（実装にも依存するが）mpich2 や OpenMPI では，MPI_Allreduce() 関数によって部分和が足し込まれる順序がつねに一定であるため，結果として得られる内積は何度実行しても同一の値になり，収束までに要する反復回数はつねに 175 回になる．これに対して，図 3.8 に示した DMI の Allreduce では，128 個のプロセッサの部分和を足し込む順序は非決定的であり，内積に若干の誤差が生じるため，収束までに要する反復回数が非決定的になる．

第 2 に，性能について述べる．プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.36 に，そのウィークスケーラビリティを図 6.38 に示す．また，128 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.39 に示す．グラフ中の値は，すべて 1 イテレーションあたりの実行時間である．図 6.36 および図 6.38 より，**DMI の性能は mpich2 よりはやや劣るが，OpenMPI よりはとても優れている**．

まず，DMI と mpich2 との性能差について考える．図 6.39 より，128 プロセッサで実行した場合における DMI と mpich2 との全体の実行時間差は 0.0554 秒であり，DMI と mpich2 との性能差は通信実行時間の差に起因していることがわかる．BiCGSafe 法の 1 イテレーションにおける通信は，領域間の ghost 節点のやりとり 2 回と Allreduce22 回から構成される．ここで，図 6.7 における

データセット名	medium0.01	medium0.1	large0.01	large0.1
節点数 $\|V\|$	1.28 億	1.28 億	12.8 億	12.8 億
エッジ数 $\|E\|$	4.48 億	4.48 億	44.8 億	44.8 億
サブグラフ間のエッジカット	0.0448 億	0.448 億	0.448 億	4.48 億
入力ファイルのサイズ	7.81 GB	7.81 GB	78.1 GB	78.1 GB

表 6.4　Web グラフのデータセット.

Allreduce 単体の性能評価において，mpich2 の MPI_Allreduce() 関数は DMI の Allreduce よりも 0.00242 秒だけ速いことをふまえると，BiCGSafe 法の 1 イテレーションあたりでは，Allreduce に関して $0.00242 \times 22 = 0.0532$ 秒の通信実行時間差が現れると考えられる．すなわち，Allreduce の性能差によって，DMI と mpich2 との全体の実行時間差をほぼ説明づけることができる．よって，DMI の性能を向上させるためには Allreduce の高速化が重要であるといえる．

次に，OpenMPI の性能の低さについて考える．BiCGSafe 法における領域間の ghost 節点のやりとりでは，各プロセッサは周囲の 7~26 個のプロセッサと point-to-point な通信を行うが，OpenMPI ではこの point-to-point な通信が遅いことがわかっている．たとえば，2 ノード間で 65536 バイトのデータを 10000 回 ping-pong する実験を行った場合，mpich2 では 2.39 秒を要するが，OpenMPI では 9.03 秒も要する．

第 3 に，プログラマビリティについて述べる．表 6.3 より，MPI のプログラムは 2572 行，DMI のプログラムは 2368 行である．ここで，DMI と MPI の 204 行の差は，ghost 節点のやりとりをグローバルビュー型で記述できるか，あるいはローカルビュー型で記述する必要があるかの違いに起因している．この有限要素法では非定型な領域分割が行われるため，ローカルビュー型の MPI では，各節点がどのランクのプロセスのローカルアドレス空間のどの位置に格納されているかを対応づけたうえで，どのランクのプロセスがどのランクのプロセスに対してどの位置のデータを送受信するべきかに関する非常に煩雑な計算が必要になる．204 行という数字自体はそれほど大きいものではないが，この 204 行で記述する計算は非常に煩雑でありバグの原因となりやすいことを強調しておく．これに対して，DMI では，**read-write-set を使うことにより，各節点の read/write をグローバルビュー型で記述できるため生産性が高い**.

6.6.2　Web グラフのページランク計算

6.6.2.1　大規模な Web グラフの生成

一般に，グラフにおける各節点への入次数を対数正規分布にしたがって決定することにより，実世界のソーシャルグラフや Web サイトの相互リンク関係を表現するグラフによく似たグラフを生成できることが知られている [129]．そこで本実験では，ページランク計算や最短路計算の対象となる Web グラフ G を，以下の数理モデルにしたがって生成した：

- グラフ $G = (V, E)$ は，128 個のサブグラフ G_0，G_1，...，G_{127} から構成される．エッジは有向エッジとする．

- 各サブグラフ G_i の節点数はどれも等しく b である．よって，グラフ G の節点数は $128b$ である．
- 各節点 v_i（$0 \leq i < 128b$）への入次数を，対数正規分布にしたがう乱数によって決定する．いい換えると，入次数が d であるような節点の個数が対数正規分布：

$$p(d) = \frac{1}{d\sigma\sqrt{2\pi}} e^{-((\ln d - \mu)/\sigma)^2/2}$$

  にしたがうように各節点の入次数を決定する．ここで，μ と σ は，それぞれ，この対数正規分布に対応する正規分布の平均と標準偏差である．本実験では，対数正規分布の平均が 4，標準偏差が 1.3 となるように μ と σ を設定する．すなわち，各節点に入るエッジ数が平均 4 本で標準偏差が 1.3 となるように設定する．
- 各節点 v_i に入るエッジの始点は，確率 $1-p$ で v_i が属しているサブグラフ内の節点からランダムに選択し，確率 p で v_i が属していないサブグラフ内の節点からランダムに選択する．つまり，平均して $p|E|$ 本のエッジは異なるサブグラフ間にまたがり，平均して $(1-p)|E|$ 本のエッジは同一サブグラフ内に収まっているように，エッジたち E を生成する．p が小さいほどサブグラフ間のエッジカットが小さくなる[⋆4]．
- 各エッジには 0 以上 1 未満の一様乱数によって重みを与える．

上記の数理モデルに基づき，本実験では表 6.4 に示す 4 種類のデータセットを生成した．n 個のプロセッサで実行する場合には，プロセッサ i にはサブグラフ $G_{in/128}$，$G_{in/128+1}$，$\ldots$，$G_{(i+1)n/128}$ を担当させた[⋆5]．各サブグラフ G_i を処理する場合の通信量は，サブグラフ G_i の外点数に比例し，計算量は，サブグラフ G_i 内の節点に入るエッジ数に比例する．データセット medium0.01 と medium0.1 について，128 個の各サブグラフの外点数とエッジ数の分布を調べた結果を図 6.40 と図 6.41 に示す．図 6.40 と図 6.41 より，外点数の分布は $1.035 \times 10^6 \pm 0.04\%$，エッジ数の分布は $3.5 \times 10^6 \pm 0.11\%$ に収まっており，128 個のサブグラフを通じて通信量と計算量の負荷バランスは良好であることがわかる．

6.6.2.2 実験設定

Web グラフ $G = (V, E)$ と各節点 $v_i \in V$ に対して，集合 $adj^+(v_i)$ と集合 $adj^-(v_i)$ を以下のように定義する：

$$adj^+(v_i) = \{v_j \mid (v_i, v_j) \in E\}, \quad adj^-(v_i) = \{v_j \mid (v_j, v_i) \in E\}.$$

このとき，各節点 v_i のページランクは，以下の漸化式が「収束」したときの $rank(v_i, t)$ の値として定義される [129]：

$$rank(v_i, t) = \begin{cases} 1/n & \text{if } t = 0, \\ (1-c)/n + c\sum_{v_j \in adj^-(v_i)} rank(v_j, t-1)/|adj^+(v_j)| & \text{if } t \geq 1. \end{cases} \tag{6.1}$$

[⋆4] 本来ならば，最初にグラフ G を生成して，METIS[100] や ParMETIS[99] などのグラフ分割ライブラリを使用してグラフ G をサブグラフに分割するべきである．しかし，著者の環境では大規模なグラフに対して METIS も ParMETIS も正常に動作しなかったため，ここで説明したように，サブグラフを最初に作ってからサブグラフ間に適当な個数のエッジを張っていく方法をとった．

[⋆5] 簡単のため n は 128 の約数とする．

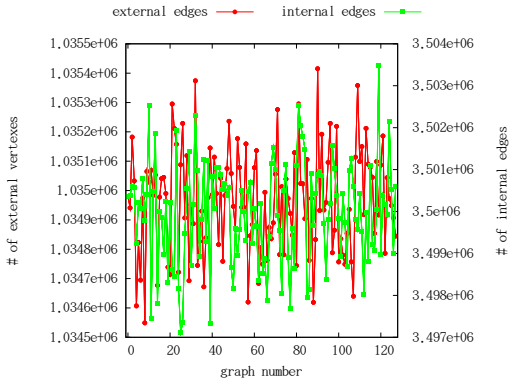

図 6.40　サブグラフ間の外点数とエッジ数のバランス（medium0.01）.

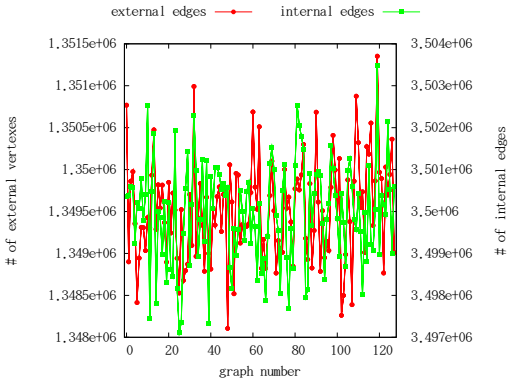

図 6.41　サブグラフ間の外点数とエッジ数のバランス（medium0.1）.

ここで漸化式 (6.1) が「収束」するとは,

$$\sum_{v_i \in V} |rank(v_i, t) - rank(v_i, t-1)| < \epsilon$$

が成り立つことである．c は減衰定数であり，本実験では $c = 0.85$ とした.

アルゴリズムとしては，漸化式 (6.1) をそのまま反復計算として表現した．すなわち，各プロセッサ i は以下の処理を反復する：

(1) 各プロセッサ i は，プロセッサ i が担当するサブグラフの外点の値を取得する.
(2) 各プロセッサ i は，プロセッサ i が担当するサブグラフの内点の値を，式 (6.1) にしたがって更新する.
(3) すべてのプロセッサが同期する.

DMI の場合には，read-write-set を用いて，(1) 各プロセッサ i が rwset_write() 関数によってグローバルアドレス空間に対して内点の値を書き込んだあと，(2) すべてのプロセッサが同期し，(3) 各プロセッサ i が rwset_read() 関数によってグローバルアドレス空間から外点の値を読み出す，というように記述した．反復回数は 10 回とし，1 イテレーションあたりの平均時間を算出した.

6.6.2.3　結果と考察

第 1 に，性能について述べる．データセット medium0.01 と medium0.1 のそれぞれに関して，プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.42 と図 6.43 に，そのウィークスケーラビリティを図 6.44 と図 6.45 に示す．また，128 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.46 と図 6.47 に示す．グラフ中の値は，すべて 1 イテレーションあたりの実行時間である．図 6.42，図 6.43，図 6.44，図 6.45 より，**DMI は mpich2 や OpenMPI よりとても高い性能を達成できている**ことがわかる．とくに，エッジカットが少なく通信量の少ないデータセット medium0.01 において，mpich2 や OpenMPI はスケーラビリティが大きく鈍

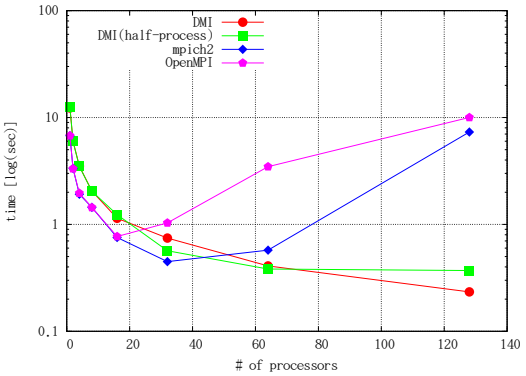

図 6.42 ページランク計算（データセット medium0.01）の実行時間.

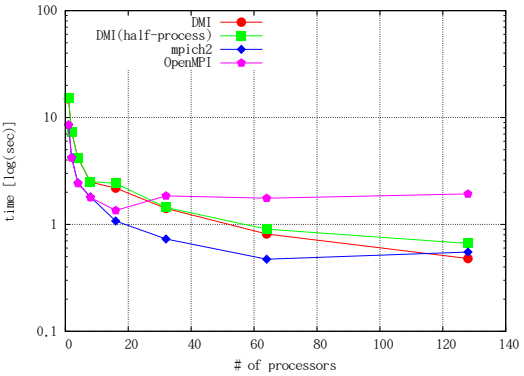

図 6.43 ページランク計算（データセット medium0.1）の実行時間.

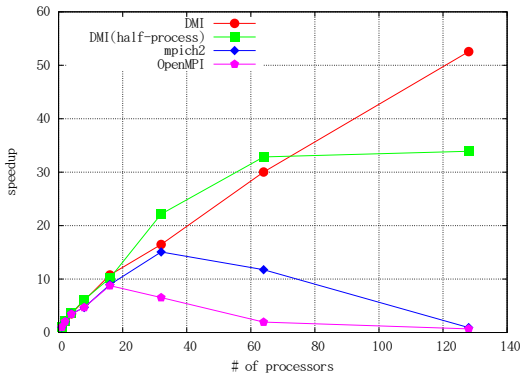

図 6.44 ページランク計算（データセット medium0.01）のウィークスケーラビリティ.

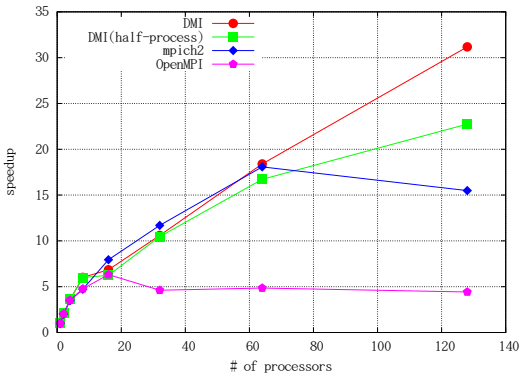

図 6.45 ページランク計算（データセット medium0.1）のウィークスケーラビリティ.

るのに対して，DMI は良好なスケーラビリティを達成している．

DMI と mpich2 と OpenMPI の性能差の原因について分析する．まず，図 6.46 より，性能差の大部分は通信実行時間の差に起因していることがわかる．この通信は外点の値の取得にかかわるもので，グラフ G のエッジの始点を一様乱数によって選択していることをふまえると，各サブグラフは他のすべてのサブグラフ内にほぼ等しい個数の外点を持ち，ほぼ一様な All-to-all 型の通信が発生する．具体的には，128 プロセッサで実行する場合，データセット medium0.01 では各プロセッサは自分以外の 127 個のプロセッサと各 2.15 KB の通信を行い，データセット medium0.1 では各プロセッサは自分以外の 127 個のプロセッサと各 21.5 KB の通信を行う．以上をふまえて，ページランク計算における DMI と mpich2 と OpenMPI の性能差を説明づけるために，一様な All-to-all 型の通信に対する DMI と mpich2 と OpenMPI の潜在的な性能を調べた．図 6.48 は，横軸のデータサイズ x に対して，128 個の各プロセッサが自分以外の 127 個のプロセッサと各 x バイトを通信する場合の実行時間を示した

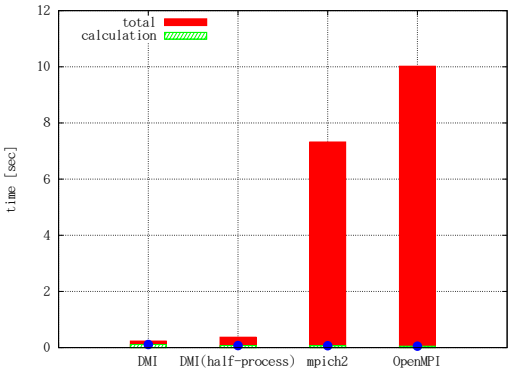

図 6.46 ページランク計算（データセット medium0.01）における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）.

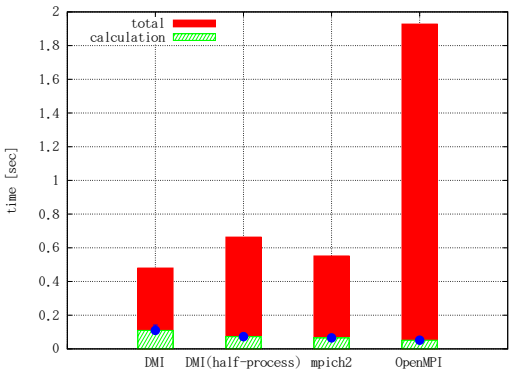

図 6.47 ページランク計算（データセット medium0.1）における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）.

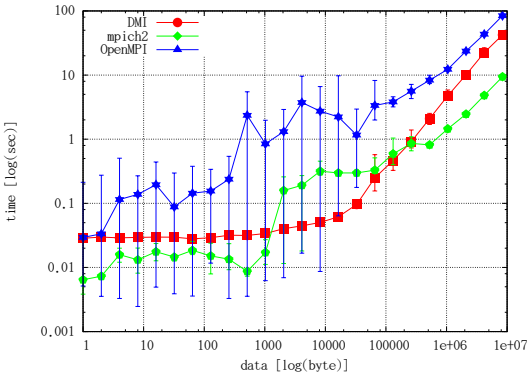

図 6.48 128 個の各プロセッサが 127 個のプロセッサと通信する場合の，データサイズと実行時間の関係.

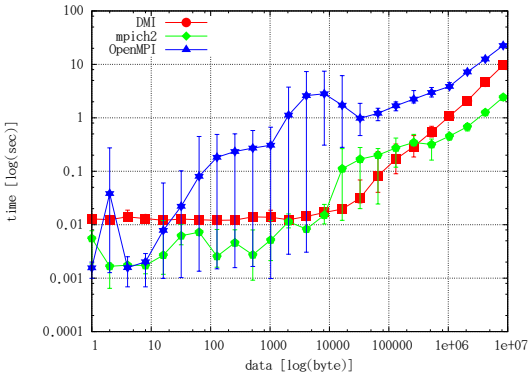

図 6.49 128 個の各プロセッサが 31 個のプロセッサと通信する場合の，データサイズと実行時間の関係.

グラフである．グラフ中の各点は 10 回の測定値の平均値であり，エラーバーの上限値と下限値は，それぞれ 10 回の測定値の最大値と最小値を示す．同様に，図 6.49 は，128 個の各プロセッサ i が自分以外の 31 個のプロセッサ $\mathrm{mod}(i+4,128)$，$\mathrm{mod}(i+8,128)$，...，$\mathrm{mod}(i+124,128)$ と各 x バイトを通信する場合の実行時間を示したグラフである．図 6.50 は，128 個の各プロセッサ i が自分以外の 7 個のプロセッサ $\mathrm{mod}(i+16,128)$，$\mathrm{mod}(i+32,128)$，...，$\mathrm{mod}(i+112,128)$ と各 x バイトを通信する場合の実行時間を示したグラフである．図 6.51 は，128 個の各プロセッサ i が自分以外の 1 個のプロセッサ $\mathrm{mod}(i+64,128)$ と各 x バイトを通信する場合の実行時間を示したグラフである．これらのグラフから以下の事実が読みとれる：

- OpenMPI は通信時間のばらつきが非常に大きい．通信時間の安定性という点では DMI がもっと

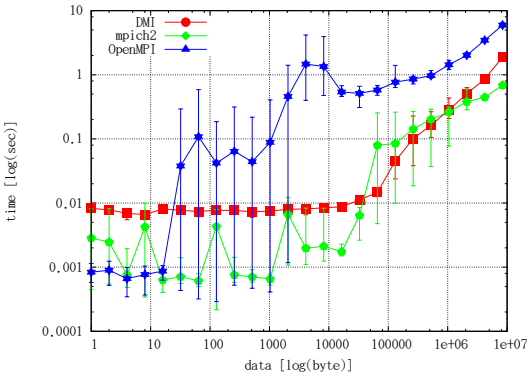

図 6.50　128 個の各プロセッサが 7 個のプロセッサと通信する場合の，データサイズと実行時間の関係.

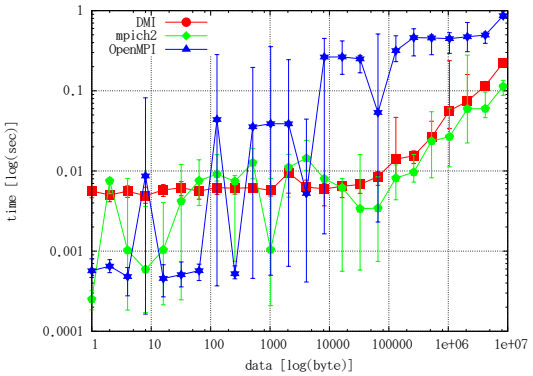

図 6.51　128 個の各プロセッサが 1 個のプロセッサと通信する場合の，データサイズと実行時間の関係.

も安定している.

- 1 KB 以上のデータサイズでは，OpenMPI の通信性能は DMI と mpich2 よりも非常に悪い.
- 図 6.48 において，データセット medium0.01 で発生する各 2.15 KB の通信に要する時間を読みとると，DMI，mpich2，OpenMPI では，それぞれおよそ 0.0401 秒，0.158 秒，1.31 秒である．また，データセット medium0.1 で発生する各 21.5 KB の通信に要する時間を読みとると，DMI，mpich2，OpenMPI では，それぞれおよそ 0.0787 秒，0.297 秒，1.69 秒である．この性能差が，ページランク計算の各イテレーションの性能差に反映されているものと考えられる.

6.6.3　同期的なアルゴリズムによる Web グラフの最短路計算

6.6.3.1　実験設定

6.6.2.1 節で生成したグラフ $G = (V, E)$ に関して，節点 v_0 から他のすべての節点までの最短路を計算した．節点 v_0 から各節点 v_i までの最短路は，以下の漸化式が「収束」したときの $spath(v_i, t)$ の値として定義される：

$$spath(v_i, t) = \begin{cases} 0 \quad \textbf{if } t = 0 \quad \textbf{and} \quad i = 0, \\ \infty \quad \textbf{if } t = 0 \quad \textbf{and} \quad i \neq 0, \\ \min_{v_j \in adj^-(v_i)}(spath(v_i, t-1), spath(v_j, t-1) + weight_{v_j \to v_i}) \quad \textbf{if } t \geq 1 \end{cases} \tag{6.2}$$

ここで漸化式 (6.2) が「収束」するとは，すべての節点 v_i に対して $spath(v_i, t+1) = spath(v_i, t)$ が成り立つことをいう.

アルゴリズムとしては，**6.6.2** 節におけるページランク計算と同様に，漸化式 (6.2) をそのまま各プロセッサの反復計算として表現した．ページランク計算とは，漸化式における更新式が異なるだけで，プログラムの構造やプロセッサ間で発生する通信は同じである.

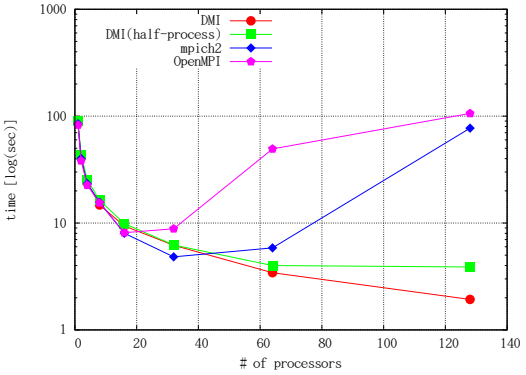

図 6.52 同期的な最短路計算（データセット medium0.01）の実行時間.

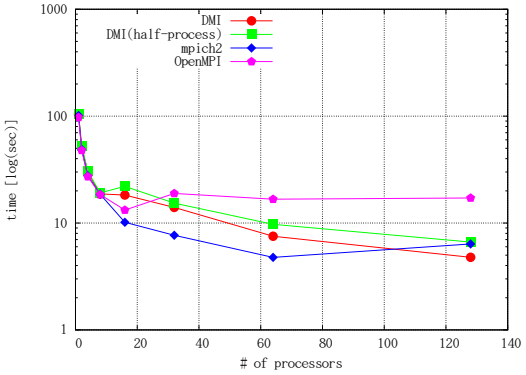

図 6.53 同期的な最短路計算（データセット medium0.1）の実行時間.

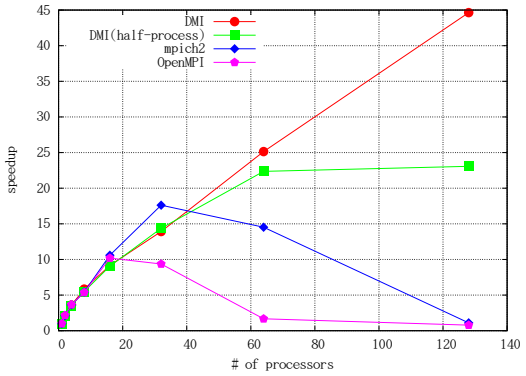

図 6.54 同期的な最短路計算（データセット medium0.01）のウィークスケーラビリティ.

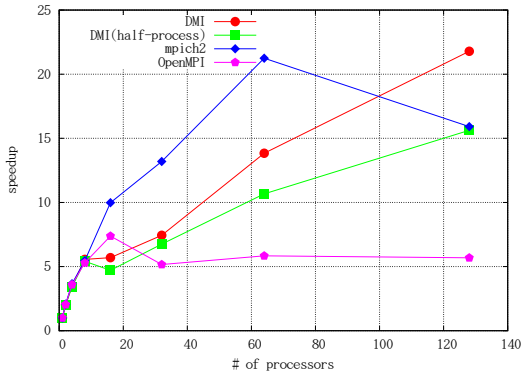

図 6.55 同期的な最短路計算（データセット medium0.1）のウィークスケーラビリティ.

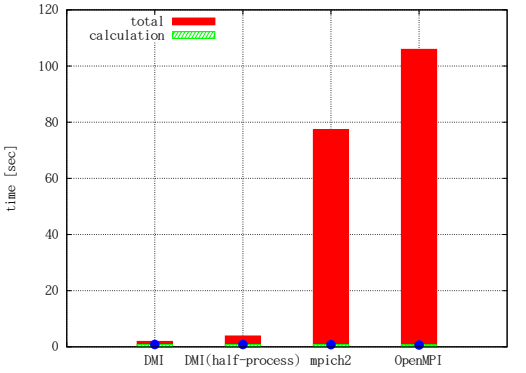

図 6.56 同期的な最短路計算（データセット medium0.01）における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）.

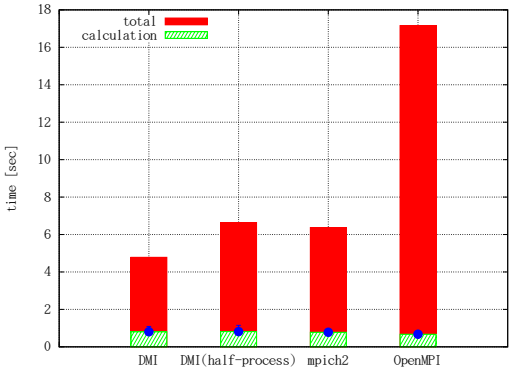

図 6.57 同期的な最短路計算（データセット medium0.1）における，全体の実行時間に占める計算実行時間の割合（128 プロセッサ実行時）.

```
01: each_thread(subgraph G_i):
02:     while 1 do
03:         changed = do_iteration_asynchronously(G_i)
04:         if changed == 0 then
05:             changed = do_iteration_synchronously(G_i)
06:             changed_sum = allreduce(changed, SUM)
07:             if changed_sum == 0 then
08:                 break;
09:             endif
10:         endif
11:     endwhile
```

図 6.58　最短路計算の非同期的なアルゴリズム.

6.6.3.2　結果と考察

第 1 に，アプリケーションの結果について述べる．データセット medium0.1 では，DMI，mpich2，OpenMPI ともに，最短路が求まるまでに 53 回の反復計算を要した．節点 v_0 から到達不可能な節点が 1638 個存在した．残りの節点のなかで，最短路が最大であったものは 9.54 だった．

第 2 に，性能について述べる．データセット medium0.01 と medium0.1 のそれぞれに関して，プロセッサ数を変化させた場合における，DMI，mpich2，OpenMPI の実行時間を図 6.52 と図 6.53 に，そのウィークスケーラビリティを図 6.54 と図 6.55 に示す．また，128 プロセッサで実行した場合における，全体の実行時間と計算実行時間を図 6.56 と図 6.57 に示す．プロセッサ間の通信パターンはページランク計算と同様のため，グラフの形状はページランク計算の場合と似ている．DMI は mpich2 や OpenMPI よりとても高い性能を達成できている．

第 3 に，プログラマビリティに関しても，ページランク計算と同様のことがいえる．

6.6.4　非同期的なアルゴリズムによる Web グラフの最短路計算

6.6.4.1　実験設定

ページランク計算の場合には，すべての節点 v_i のページランクの総和がつねに 1 になる必要があるため，すべてのプロセッサ間で反復計算を同期的に行わなければならない．いい換えると，すべての節点において「値の更新速度」が一致している必要がある．これに対して，最短路計算の場合には，演算子 min は可換性と結合性を持つので，反復計算を同期的に行う必要はなく，節点によって「値の更新速度」が一致していなくても問題ない．そこで，図 6.58 に示すような非同期的な最短路計算のアルゴリズムを考える．図 6.58 において，do_iteration_synchronously(G_i) 関数は，**6.6.3** 節で述べたアルゴリズムと同様に，各サブグラフ G_i に対する値の更新を同期的に行う関数であり，DMI の場合には以下の処理を行う：

(1) すべてのプロセッサが同期する．
(2) 各プロセッサ i が rwset_write() 関数によってグローバルアドレス空間に対して内点の値を書き込む．
(3) すべてのプロセッサが同期する．

(4) 各プロセッサ i が rwset_read() 関数によってグローバルアドレス空間から外点の値を読み出す.

これに対して，do_iteration_asynchronously(G_i) 関数は，各サブグラフ G_i に対する値の更新を非同期的に行う関数であり，DMI の場合には以下の処理を行う：

(1) **同期することなく**，各プロセッサ i が rwset_write() 関数によってグローバルアドレス空間に対して内点の値を書き込む.
(2) **同期することなく**，各プロセッサ i が rwset_read() 関数によってグローバルアドレス空間から外点の値を読み出す.

したがって，rwset_read() 関数によって読み込まれる外点の値が，いつの時点で隣接プロセッサによって書き込まれた値であるかに関する保証はない．do_iteration_synchronously(G_i) 関数および do_iteration_asynchronously(G_i) 関数は，返り値として，サブグラフ G_i の内点のうち値に変更があった節点の個数を返す．以上をふまえると，図 6.58 のアルゴリズムは以下のように動作する：

(1) 各プロセッサ i は，サブグラフ G_i の内点の値に変更がなくなるまで，他のプロセッサとは独立に非同期的に内点の値の更新を行う（3 行目）.
(2) 各プロセッサ i は，サブグラフ G_i の内点の値に変更がなくなった場合，同期的に内点の値の更新を行おうとして，do_iteration_synchronously(G_i) 関数を呼ぶ（5 行目）．この do_iteration_synchronously(G_i) 関数は先頭で同期を必要とするので，do_iteration_synchronously(G_i) 関数による同期的な内点の値の更新が始まるのは，すべてのプロセッサが do_iteration_synchronously(G_i) 関数を呼んだ時点である．いい換えると，すべてのプロセッサが，「自分の担当するサブグラフには内点の値の更新がなくなった」と思った時点で，do_iteration_synchronously(G_i) 関数が実行される.
(3) do_iteration_synchronously(G_i) 関数の結果として 0 が返れば（7 行目），グラフ G のすべての内点の値に更新がなくなったことを意味するので，反復計算を終了する．そうでなければ，(1) に戻る.

この非同期的なアルゴリズムは，**DMI では read/write による単方向通信を記述できるからこそ自然に記述できている**という点を強調したい．メッセージパッシングモデルのような send/receive による双方向通信のプログラミングモデルでは，データを通信するために相手側の協力を必要になるため，このような非同期的なアルゴリズムを記述するのは難しい．なお，MPI-2 では，send/receive による双方向通信に加えて get/put による単方向通信も利用できるため，このような非同期的なアルゴリズムを記述することは可能であるが，事前にウィンドウを登録する必要があるなど，DMI と比較すると API が煩雑で使いにくい．また，**DMI が read/write に関して Sequential Consistency を保証している**という点も重要である．Lazy Release Consistency などの緩和型のコンシステンシモデルを採用する分散共有メモリ処理系 [15, 204] では，write した結果を後続の read に反映させるために排他制御が必要になり，すべてのプロセッサを独立に実行させることはできない.

表 6.5 128 プロセッサ実行時の最短路計算の実行時間比較 [sec].

データセット名	medium0.01	medium0.1	large0.01	large0.1
mpich2	142.45	31.20	108.63	125.70
OpenMPI	360.39	105.44	200.40	281.50
DMI（同期的）	9.06	21.82	103.03	256.48
DMI（非同期的）	11.64	17.34	107.39	208.35

本実験では，同期的なアルゴリズムで記述した DMI，mpich2，OpenMPI と，非同期的なアルゴリズムで記述した DMI について，性能を比較した．

6.6.4.2 結果と考察

第 1 に，DMI における非同期的な最短路計算（データセット large0.1）において，時系列的に各スレッドがどのような挙動を行ったかを図 6.59 に可視化する．図 6.59 では，横軸が時間，縦軸が 128 個のスレッド，黄色の長方形（wait）が待機している時間，その他の色の長方形（iterXXXXX）が 1 イテレーションを表す．また，（黄色以外の）各色は実行されたプロセスを表しており，たとえば凡例の iter17526 は，そのイテレーションがプロセス 17526 で実行されたことを意味している．図 6.59 から以下のことが読みとれる：

- 初期的には，すべてのスレッドが do_iteration_asynchronously() 関数を実行するが，節点 v_0 を始点とするエッジを持っていないサブグラフは，すべての内点の値は ∞ のままで更新されない．よって，スレッド 0 以外は，担当するサブグラフ内の内点の値に更新を観測できず，do_iteration_synchronously() 関数を呼び出す．そして，すべてのスレッドが do_iteration_synchronously() 関数を呼び出すまで待機する（4 秒付近）．スレッド 0 のみが do_iteration_asynchronously() 関数を繰り返し呼び出し，やがてサブグラフ G_0 の内点の値に更新がなくなった時点で，do_iteration_synchronously() 関数を呼び出す（40 秒付近）．
- すべてのスレッドが再び do_iteration_asynchronously() 関数を実行し始める（40 秒付近）．この時点では，サブグラフ G_0 のなかで節点 v_0 から到達可能な節点の値が ∞ 以外になっているため，そのうちの少なくとも 1 個の節点を始点とするエッジを持つサブグラフでは，その内点の値に更新が行われることになる．すると，その更新が契機となって，さらに更新が広く伝播し，すべてのスレッドが独立に do_iteration_asynchronously() 関数を呼び出し続ける状態となる．
- やがて 190 秒付近で do_iteration_synchronously() 関数による 2 回目の同期が行われ，200 秒付近で 3 回目の同期が行われ，210 秒付近で 4 回目の同期が行われ，最後にすべてのスレッドで内点の値の更新が観測されなくなって，アルゴリズムが停止する．

第 2 に，4 種類のデータセットに関して，同期的なアルゴリズムで記述した DMI，mpich2，OpenMPI と，非同期的なアルゴリズムで記述した DMI について，128 プロセッサで実行した場合の実行時間を表 6.5 に比較する．

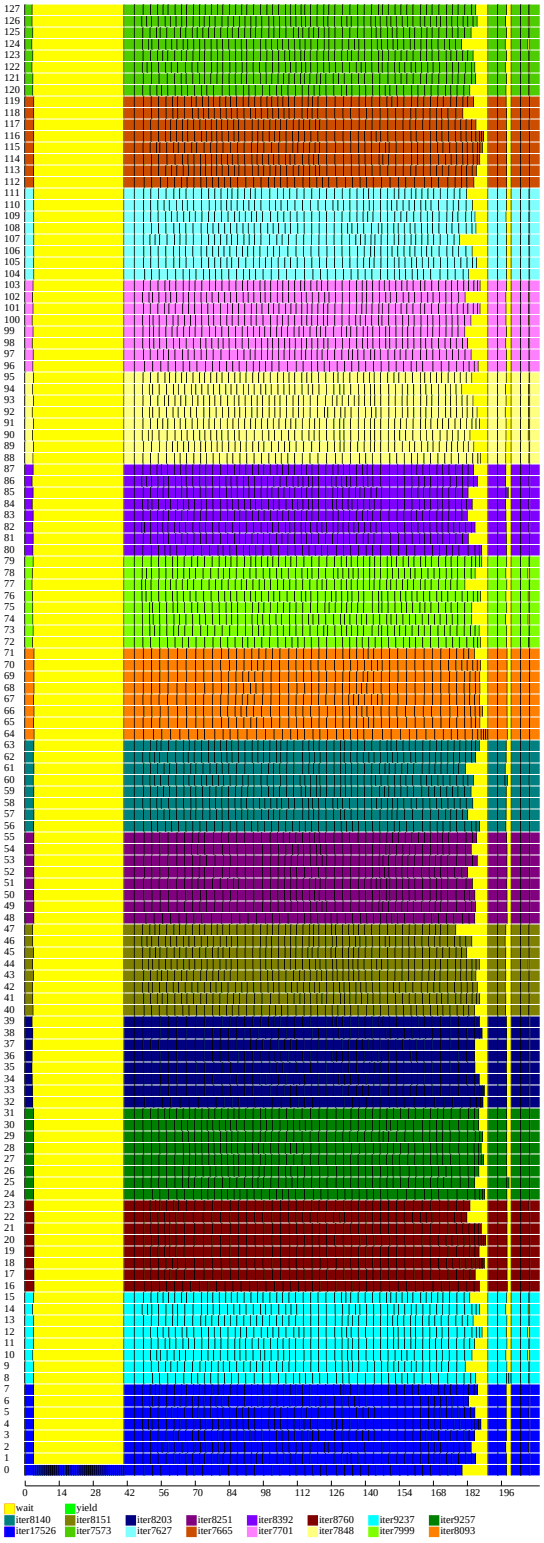

図 6.59　非同期的な最短路計算（large0.1）における各スレッドの挙動.

表 6.5 より，データセット large0.1 をのぞいては，DMI（同期的）と DMI（非同期的）の両方ともが mpich2 および OpenMPI よりも高い性能を達成している．また，全体のエッジ数に対して相対的にエッジカット数の少ないデータセットである medium0.01 や large0.01 では，DMI（非同期的）と DMI（同期的）の性能はほぼ等しいが，相対的にエッジカット数の多いデータセットである medium0.1 や large0.1 では，DMI（非同期的）の方が DMI（同期的）より性能がよい．エッジカット数の割合によってこのような傾向が出る理由は特定できていないが，いずれにせよ，この結果は，**read/write による単方向通信を活かした非同期的なアルゴリズムによって，同期的なアルゴリズム以上の性能を達成できる**場合があることを示しており，非同期化が可能な他のアルゴリズムに対しても，DMI を用いて非同期化することによってさらなる性能向上が得られる可能性を示唆している．

6.7 要約

本章では，各種マイクロベンチマーク，基本的なアプリケーション，応用的なアプリケーションを用いて性能とプログラマビリティの評価を行った．全体を要約すると以下のとおりである：

- 定型的で基本的なアプリケーションに関しては，DMI のプログラム行数は MPI のプログラム行数とほぼ等しく，プログラマビリティ上の利点はない．これは，DMI の API 設計が性能最適化を重視して設計されているためであり，DMI のプログラムは，MPI のプログラムにおける MPI_Send() 関数/MPI_Recv() 関数を DMI_read() 関数/DMI_write() 関数に置き換えただけのようなプログラムとなる．
- **非定型で応用的なアプリケーションに関しては，DMI のプログラム行数は MPI のプログラム行数より有意に短く，DMI の方がプログラマビリティが高い**．この理由は，MPI では節点番号とローカルインデックスを対応づけるための煩雑な計算が必要となるのに対して，DMI では read-write-set を用いることで，非定型な並列計算をグローバルビュー型のグローバルアドレス空間モデルに基づいて記述できるためである．
- 性能とスケーラビリティに関してはアプリケーション依存であるが，総じていえば，**DMI は，mpich2 と同等で，OpenMPI よりも高い性能を達成している**といえる．NAS Parallel Benchmark の EP，マンデルブロ集合の描画，N 体問題など，比較的通信量の少ないアプリケーションの性能は，DMI≈mpich2≈OpenMPI となる．横ブロック分割による行列行列積など，大容量のデータの集合通信が必要となるアプリケーションの性能は，mpich2>OpenMPI>DMI となる．ランダムサンプリングソート，ページランク計算，最短路問題など，プロセッサ間の密な全対全通信が要求されるアプリケーションの性能は，DMI>mpich2>OpenMPI となる．

第 7 章

スレッド増減に基づく並列計算の再構成

本章からはじまる 3 つの章で，それぞれ，rescale，thread-move，half-process-move の 3 種類のプログラミングモデルに基づく並列計算の再構成について述べる．本章では，rescale のプログラミングモデルに基づく並列計算の再構成について述べる．

7.1 全体像

DMI では，プロセスの非同期的な参加/脱退を越えてコヒーレンシが維持される高性能なグローバルアドレス空間を提供している．また，**3.7** 節のプログラム例に示すように，プロセスの参加/脱退に対応してスレッドを動的に生成/破棄することによって，計算規模を自由に拡張/縮小させることができ，並列計算を再構成することができる．実際に，**6.5.2** 節および **6.5.4** 節では，それぞれ NAS Parallel Benchmark の EP とマンデルブロ集合の描画に関して，スレッド数を動的に増減させることで並列計算を再構成できることを実証した．

しかし，**3.7** 節で述べたプログラムの記述方法は，決してプログラマビリティや記述力が高いとはいえない．第 1 に，この記述方法では，プログラマがプロセスの参加/脱退を明示的に処理したり，その参加/脱退に対応してスレッドを動的に生成/破棄しなければならない．第 2 に，この記述方法では，大量のタスクがグローバルアドレス空間上に配置されていて，未処理のタスクを各スレッドが 1 個ずつとってきては処理するような，いわゆるマスタワーカ方式の並列計算しか自然には記述できない．実際に，**6.5.2** 節で述べた NAS Parallel Benchmark の EP も **6.5.4** 節で述べたマンデルブロ集合の描画も，グローバルアドレス空間上に配置された 1024 個のタスクをスレッドたちが 1 個ずつ奪って処理するように記述しているにすぎない．たとえば，有限要素法などの SPMD 型のプログラムを記述しようとした場合には，どのように記述すれば，SPMD 型の構造を自然に表現しつつ，並列計算の再構成を実現できるかはまったく自明ではない．したがって，より記述力もプログラマビリティも高い，並列計算の再構成のためのプログラミングモデルが要請されているといえる．

そこで，本研究では，並列計算の再構成のためのプログラミングモデルとして，rescale，thread-move，half-process-move の 3 種類を提案し，それぞれの性能とプログラマビリティを比較検討する．これらのプログラミングモデルの概要については，**1.3.3** 節で述べたが，改めて特徴を要約すると以下のよう

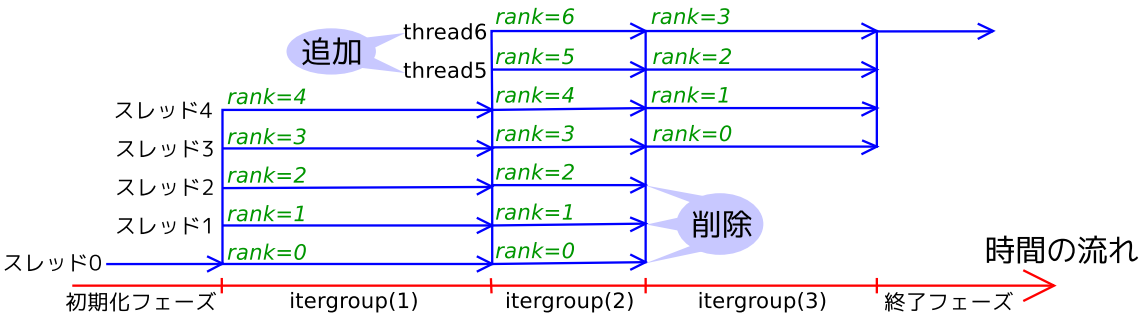

図 7.1　再構成をともなう並列反復計算における実行フロー.

になる：

rescale　再構成にともなってスレッドを生成/破棄することで，つねに 1 プロセッサあたり 1 スレッドが割り当てられるようにする．そのため性能はよいが，データのチェックポイント/リストアなどのコードが必要になるためプログラマビリティは低い．

thread-move　プログラマは大量のスレッドを生成しておくだけでよく，処理系が，透過的なスレッド移動によって，それら大量のスレッドを利用可能なノードに動的にマッピングしてくれる．よって，プログラマが再構成を意識しなくてもよいという点ではプログラマビリティは高いが，1 プロセッサに複数スレッドが割り当てられることによる性能低下が起きる．また，スレッド移動にともなうプログラミング制約が存在する．

half-process-move　スレッドとプロセスの「中間」の機能を持つ新たなカーネルプリミティブを導入することで，thread-move におけるプログラミング制約を撤廃し，真に透過的なスレッド移動を実現する．そのためプログラマビリティはきわめて高いが，1 プロセッサに複数スレッドが割り当てられることによる性能低下は依然として起きる．

このうち，本節では rescale のプログラミングモデルに基づく並列計算の再構成について述べる．

7.2　プログラミングモデル

7.2.1　基本アイディア

一般的に，再構成をともなう SPMD 型の並列反復計算の実行フローは図 7.1 のようになる．すなわち，(1) 1 スレッドで初期化フェーズを行う，(2) しばらくの間，あるスレッド集合で反復計算を実行する，(3) ノード集合の変化にともなってスレッド集合が変化し，新たなスレッド集合で再びしばらくの間反復計算を実行する，(4) やがて反復計算が終了し，最後に 1 スレッドが終了フェーズを行う．ここで，スレッド集合が変化することなく実行されている期間のことを itergroup と呼ぶことにする．たとえば，図 7.1 の実行フローは，3 個の itergroup から構成されている．

さて，この実行フローを自然にかつ柔軟に表現するためには，以下の 3 つの要請を満たすプログラミングモデルが必要である：

```
01: void DMI_main(int argc, char **argv) {
02:     struct data_t data; /* the data to be checkpointed and restored */
03:     int node_num = atoi(argv[1]); /* the initial number of nodes */
04:     ...; /* an initialization phase of this application */
05:     int64_t addr = DMI_mmap(sizeof(data), 1); /* allocate a global address space */
06:     data.iter = 0; /* a current iteration number */
07:     DMI_write(addr, sizeof(data), &data, PUT); /* write to the global address space */
08:     DMI_gather_nodes(node_num); /* wait until node_num nodes gather */
09:     DMI_rescale(addr);
10:     DMI_munmap(addr); /* deallocate the global address space */
11:     ...; /* a finalization phase of this application */
12: }
13:
14: int DMI_itergroup(int rank, int pnum, int64_t addr) {
15:     struct data_t data;
16:     int iter;
17:     DMI_read(addr, sizeof(data), &data, GET); /* restore */
18:     ...; /* calculate the role of this thread based on rank and pnum */
19:     for (iter = 0; iter < 100 && data.iter < ITER_MAX; iter++, data.iter++) {
20:         ...; /* the body of each iteration */
21:     }
22:     DMI_write(addr, sizeof(data), &data, PUT); /* checkpoint */
23:     return data.iter != ITER_MAX;
24: }
```

図 7.2　rescale のプログラミングモデル（単純化されたもの）.

- 各 itergroup は SPMD 型のプログラムとして記述できる必要がある.
- itergroup と itergroup の間ではスレッド集合が変化する．よって，各 itergroup の最後では，それ以降の実行を継続するために必要なすべてのデータをグローバルアドレス空間に書き出す必要がある．また，各 itergroup の最初では，直前の itergroup によって保存されたデータをグローバルアドレス空間から読み込む必要がある.
- 各 itergroup がいつ終了するべきかは柔軟にプログラマブルである必要がある．つまり，参加ノード数，脱退ノード数，それらのノードの資源量などの情報に基づいて，どのような条件が成立したときに itergroup を終了させてスレッド集合を変化させるべきかは，柔軟に制御できることが望ましい.

7.2.2　単純化されたプログラミングモデル

前節で述べた観察に基づき，rescale では，図 7.2 に示すプログラミングモデルを提案する．このプログラミングモデルでは，図 7.1 における初期化フェーズと終了フェーズが DMI_main() 関数として表現され，各 itergroup が DMI_itergroup() 関数として表現されている.

まず，DMI の実行が開始されると，DMI は 1 個のスレッドを立ち上げて，DMI_main() 関数を実行し始める（1 行目）．よって，プログラマは，DMI_main() 関数の先頭に，プログラムの実行にとって必要なグローバルアドレス空間の確保や初期化などの初期化フェーズを記述することができる（4

行目)．DMI_gather_nodes() 関数によって *node_num* 個のノードが集まるまで待機したあと（8 行目)，DMI_rescale(*addr*, *node_num*) 関数を呼び出すと，一番最初の itergroup の実行が始まる（9 行目)．すると，DMI は，その時点で参加している *node_num* 個の各ノード上にそのノードのプロセッサ数と等しい個数のスレッドを生成する．これらの各スレッドは，DMI_itergroup(int *rank*, int *pnum*, int64_t *addr*) 関数を実行し始める（14 行目)．このとき，*pnum* には生成されたスレッド数が渡され，*rank* には各スレッドのランクが渡される（$0 \leq rank < pnum$)．また，*addr* には，プログラマが DMI_rescale() 関数の第 1 引数に渡した *addr* がそのまま渡される．よって，プログラマは，*addr* が指すグローバルアドレス空間にさまざまなデータを格納することにより，DMI_main() 関数から DMI_itergroup() へ任意のデータを渡すことができる．

一般的には，DMI_itergroup() 関数の中身は以下の流れで記述する：

(1) その itergroup を実行するために必要なデータをグローバルアドレス空間から読み込む（データのリストア)．
(2) *rank* と *pnum* に基づいて SPMD 型の並列計算を行う．
(3) 以降の itergroup を実行するために必要なデータをグローバルアドレス空間に書き込む（データのチェックポイント)．

図 7.2 の例では，チェックポイント/リストアするべきデータとして，現在のイテレーション数をグローバルアドレス空間に書き込んでいる．

ノードの参加/脱退とそれにともなうスレッド集合の変化は，itergroup と itergroup の間でしか実現できない．よって，外部から指示されるノードの参加/脱退の要求に対して応答性よく反応するためには，各 itergroup の実行時間がある程度短い必要がある．これは，もっとも単純には，適切な反復回数ごとに DMI_itergroup() 関数を終了させることで実現できる．たとえば，図 7.2 の例では 100 イテレーションごとに DMI_itergroup() 関数を終了させている（19 行目)．

プログラム全体の終了は，ランク 0 のスレッドの DMI_itergroup() 関数の返り値で指示する．ランク 0 のスレッドが DMI_itergroup() 関数の返り値として 0 を返すと，その itergroup が最後の itergroup であると見なされ，最初に呼び出した DMI_rescale() 関数が返る（9 行目)．よって，プログラマは，DMI_rescale() 関数のあとに，グローバルアドレス空間の解放などの終了フェーズを記述することができる（11 行目)．一方で，ランク 0 のスレッドが DMI_itergroup() 関数の返り値として 0 以外を返すと，DMI によって透過的にノード集合とスレッド集合が再構成されたあと，その時点で参加している各ノード上に，そのノードのプロセッサ数と等しい個数のスレッドが生成される．そして，それらの各スレッドは，新しい *rank* と *pnum* に基づいて，DMI_itergroup(int *rank*, int *pnum*, int64_t *addr*) 関数を実行し始める．つまり，新しい itergroup の実行が始まる．

7.2.3 より高度なプログラミングモデル

図 7.2 で示したプログラムは意図とおりに動作する．しかし，一般のアプリケーションではデータをチェックポイント/リストアするのに無視できない時間がかかることをふまえると，ノードの参加/脱退が要求されていないにもかかわらず，つねに 100 イテレーションごとに itergroup を仕

```
01: void DMI_main(int argc, char **argv) {
02:     struct data_t data; /* the data to be checkpointed and restored */
03:     int node_num = atoi(argv[1]); /* the initial number of nodes */
04:     ...; /* an initialization phase of this application */
05:     int64_t addr = DMI_mmap(sizeof(data), 1); /* allocate a global address space */
06:     data.iter = 0; /* a current iteration number */
07:     DMI_write(addr, sizeof(data), &data, PUT); /* write to the global address space */
08:     DMI_gather_nodes(node_num); /* wait until node_num nodes gather */
09:     DMI_rescale(addr);
10:     DMI_munmap(addr); /* deallocate the global address space */
11:     ...; /* a finalization phase of this application */
12: }
13:
14: int DMI_itergroup(int rank, int pnum, int64_t addr) {
15:     struct data_t data;
16:     DMI_read(addr, sizeof(data), &data, GET); /* restore */
17:     ...; /* calculate the role of this thread based on rank and pnum */
18:     while (data.iter < ITER_MAX) {
19:         if (DMI_check_reconf()) {
20:             break;
21:         }
22:         ...; /* the body of each iteration */
23:         data.iter++;
24:     }
25:     DMI_write(addr, sizeof(data), &data, PUT); /* checkpoint */
26:     return data.iter != ITER_MAX;
27: }
28:
29: int DMI_judge_reconf(DMI_node_t *in_nodes, DMI_node_t *out_nodes,
          DMI_node_t *cur_nodes, int in_node_num, int out_node_num, int cur_node_num) {
30:     return in_node_num + out_node_num >= 4;
31: }
```

図 7.3　rescale のプログラミングモデル（より高度なもの）.

切りなおすのは無駄である．このような場合には，図 7.3 に示すように，DMI_check_reconf() 関数を使うことで，本当にノード集合の増減が必要になったときにのみ itergroup を終了させることが可能となる．具体的には，プログラマが DMI_check_reconf() 関数を呼び出すと（19 行目），DMI は，すべてのスレッドで同期をとったあと，DMI_judge_reconf(DMI_node_t *in_nodes, DMI_node_t *out_nodes, DMI_node_t *cur_nodes, int in_node_num, int out_node_num, int cur_node_num) 関数を呼び出して実行し（29 行目），その返り値を DMI_check_reconf() 関数の返り値とする．ここで，DMI_judge_reconf() 関数の引数としては，その時点で参加しようとしているノードの集合 *in_nodes* とその数 *in_node_num*，その時点で脱退しようとしているノードの集合 *out_nodes* とその数 *out_node_num*，その時点で参加中のノードの集合 *cur_nodes* とその数 *cur_node_num* が渡される．要するに，プログラマは，これらの情報に基づいてノード集合の増減を引き起こすための条件を DMI_judge_reconf() 関数に記述することにより，DMI_check_reconf() 関

```
01: function itergroup_wrapper(thread_id, args_addr):
02:     while 1 do
03:         pnum := $thread_num
04:         barrier(pnum + 1) /* barrier A */
05:         my_rank := $ranks[thread_id]
06:         ret := DMI_itergroup(my_rank,
                pnum, args_addr) /* itergroup */
07:         if my_rank == 0 and ret == 0 then
08:             $exit_flag := 1
09:         endif
10:         barrier(pnum + 1) /* barrier B */
11:         barrier(pnum + 1) /* barrier C */
12:         if $flags[thread_id] == 1 then
13:             break
14:         endif
15:     endwhile
16:
17: function DMI_rescale(args_addr):
18:     $exit_flag := 0
19:     foreach thread_id in maximal # of threads do
20:         $flags[thread_id] := 0
21:     endforeach
22:     pnum := 0
23:     RunningThread := ∅
24:     while 1 do
25:         NewNode := ∅
26:         DeleteNode := ∅
27:         NewThread := ∅
28:         DeleteThread := ∅
29:         old_pnum := pnum
30:         foreach node in the joining nodes do
31:             NewNode := NewNode ∪ {node}
32:             foreach i in # of processors of the node
    node do
33:                 thread_id := a unique thread ID
34:                 NewThread :=
                        NewThread ∪ {thread_id}
35:                 RunningThread :=
                        RunningThread ∪ {thread_id}
36:                 nodes[thread_id] := node
37:                 pnum := pnum + 1
38:             endforeach
39:         endforeach
40:         foreach node in the leaving nodes do
41:             DeleteNode := DeleteNode ∪ {node}
42:             foreach thread_id in thread IDs of all
                    the threads on the node node do
43:                 DeleteThread :=
                        DeleteThread ∪ {thread_id}
44:                 RunningThread :=
                        RunningThread \ {thread_id}
45:                 pnum := pnum − 1
46:                 $flags[thread_id] := 1
47:             endforeach
48:         endforeach
49:         $thread_num := pnum
50:         barrier(old_pnum + 1) /* barrier C */
51:         rank := 0
52:         foreach thread_id in RunningThread do
53:             $ranks[thread_id] := rank
54:             rank := rank + 1
55:         endforeach
56:         foreach node in NewNode do
            handle the joining of the node node
57:         endforeach
58:         foreach thread_id in NewThread do
59:             handle[thread_id] := thread_create(
            nodes[thread_id], itergroup_wrapper,
            thread_id, args_addr)
            /* create a thread on the node
            nodes[thread_id]. This thread invokes
            itergroup_wrapper(thread_id, args_addr) */
60:         endforeach
61:         barrier(pnum + 1) /* barrier A*/
62:         foreach thread_id in DeleteThread do
63:             thread_join(handle[thread_id])
                /* retrieve a thread */
64:             $flags[thread_id] := 0
65:         endforeach
66:         foreach node in DeleteNode do
67:             handle the leaving of the node node
68:         endforeach
69:         barrier(pnum + 1) /* barrier B */
70:         if $exit_flag == 1 then
71:             break
72:         endif
73:     endwhile
74:     ...
```

図 7.4　DMI_rescale() 関数のアルゴリズムと，DMI_itergroup() 関数のラッパー関数のアルゴリズム.

数の返り値を操作し，それによって itergroup の終了を明示的に制御することができる.

7.3 実装

ある itergroup が終了したとき，ノード集合とスレッド集合を再構成するためには，もっとも単純には，(1) その itergroup を実行していたすべてのスレッドを回収して，(2) ノードの参加/脱退を処理したあと，次の itergroup を実行するノード集合を決定し，(3) それらの各ノード上にスレッドを生成して，それらのスレッド集合で次の itergroup を実行すればよい．しかし，再構成のたびにすべてのスレッドを回収するのは無駄である．なぜなら，一般的な再構成においては，ノード集合の全部ではなくごく一部だけが変化する場合が多いと考えられ，そのような場合に，再構成を越えて参加し続けるノード上のスレッドもいったん回収して再び生成しなおすのは無駄だからである．そこで DMI では，**再構成を越えて参加し続けるノード上のスレッドを回収することなく，ノード集合とスレッド集合を再構成する**アルゴリズムを提案する．

前節で述べた rescale のプログラミングモデルにおいて，再構成に関わっているのは DMI_rescale() 関数の内部と，DMI_itergroup() 関数の実行前後である．よって，図 7.4 には，各 DMI_itergroup() 関数がどのように呼び出されるか（itergroup_wrapper()）と，DMI_rescale() 関数がどのように実装されているか（DMI_rescale()）のアルゴリズムを示す．図 7.4 において，$\$v$ はグローバルアドレス空間上の変数 v を表し，その他の変数はそのスレッドのローカルアドレス空間上の変数を表す．具体的には，$\$thread\_num$ は「その itergroup を実行するスレッド数」を，$\$exit\_flag$ は「この itergroup が最後の itergroup かどうか」を，$\$ranks[thread\_id]$ は「スレッド ID が $thread\_id$ のスレッドのランク」を，$\$flags[thread\_id]$ は「スレッド ID が $thread\_id$ のスレッドが終了すべきかどうか」を表す変数である．ここで，スレッド ID とは各スレッドに固有の識別番号を意味する．各スレッドのスレッド ID はそのスレッドが生成されてから破棄されるまで変化しないが，各スレッドのランクは itergroup が変化すれば変化しうる．

また，barrier($pnum$) 関数は $pnum$ 個のスレッドでバリアを行う関数である．とくに，barrier($pnum + 1$) 関数の意味は，itergroup を実行している $pnum$ 個のスレッドに加えて，DMI_rescale() 関数を実行しているスレッドもバリアに加わっていることを意味する．図 7.4 では 3 種類のバリアを巧みに組み合わせているが，それぞれのバリアは主に次の意味を持つ．バリア A（4 行目と 61 行目）は，DMI_rescale() 関数を実行しているスレッドが$\$ranks[*]$ の値を確定させたことを保証する．バリア B（10 行目と 69 行目）は，ランク 0 のスレッドが$\$exit_flag$ の値を確定させたことを保証する．バリア C（11 行目と 50 行目）は，DMI_rescale() 関数を実行しているスレッドが $\$thread\_num$ と$\$flags[*]$ を確定させたことを保証する．

なお，スレッドの回収処理（63 行目）とノードの脱退処理（67 行目）をバリア A の前ではなく後で行っている理由は，バリア A をできるかぎり速く実行するためである．バリア A が，各 itergroup を開始するタイミングを決定しているため，バリア A を速く実行できるほど，itergroup を速く開始できることになる．よって，スレッドの回収処理やノードの脱退処理をバリア A の後に回すことで，これらの重い処理の完了を待つことなく新しい itergroup に移行できるようにしている．

7.4 要約：利点と欠点

本章では，スレッド増減に基づく並列計算の再構成を実現するプログラミングモデルとして rescale を設計して実装した．rescale では，適当なイテレーション数ごとに，計算を実行するために必要なデータをグローバルアドレス空間に対してチェックポイント/リストアするようにプログラムを記述しておくと，処理系が，そのチェックポイントとリストアの「間」でプロセスの参加/脱退を処理して，つねに 1 プロセッサに 1 スレッドが割り当てられるように並列計算を再構成してくれる．また，不必要な再構成が起きることがないよう，再構成を引き起こすための条件を明示的に指示することも可能である．さらに，再構成を越えて参加し続けるノード上のスレッドを回収することなく，ノード集合とスレッド集合を再構成するアルゴリズムを新たに提案している．SPMD 型の反復計算を対象にして，プログラマにデータのチェックポイント/リストアを記述させることで並列計算の再構成を実現する既存研究としては，**2.2.2.6** で述べた SRS[173]，DyRecT[69]，DRMS[103]，PCM[128, 126] などがあるが，これらはいずれもメッセージパッシングモデルに基づくものであって，グローバルアドレス空間モデルに基づく DMI とは異なる．

rescale の利点は，つねに 1 プロセッサあたり 1 スレッドが割り当てられるため，無駄なオーバヘッドが引き起こされず性能がよい点である．一方で，第 1 の欠点は，適当なイテレーション数ごとに itergroup を終了させたり，データのチェックポイント/リスタートのためのコードを記述したりする必要があるため，プログラマビリティが低い点である．また，**10.6** 節で評価するように，有限要素法などの複雑なアプリケーションでは，どのデータをチェックポイント/リストアすればよいかが自明でないことも多い．第 2 の欠点は，rescale は SPMD 型の同期的な反復計算しかサポートできない点である．そもそも，DMI の本来の設計コンセプトから考えると，rescale は記述力を限定しすぎたプログラミングモデルであるといえる．第 **3** 章で述べたように，DMI では，SPMD 型のプログラムにかぎらず，より動的な並列性を表現できるようにするために，スレッドの生成/破棄によって並列性を表現できるようになっている．また，DMI のグローバルアドレス空間は，プロセスの非同期的な参加/脱退に対応できるように設計されている．これに対して，rescale は SPMD 型の同期的な反復計算に特化しており，DMI の基礎的な設計を十分に活かしたプログラミングモデルであるとはいえない．

次章で述べる thread-move では，透過的なスレッド移動を実現することで，並列計算の再構成に対するプログラマビリティを大幅に改善する．

第 8 章

透過的なスレッド移動に基づく並列計算の再構成

本章では，thread-move のプログラミングモデルに基づく並列計算の再構成について述べる．

8.1 全体像

thread-move では，プログラマは単に十分な数のスレッドを生成するだけでよい．すると，あとは処理系が，透過的なスレッド移動を通じて，それら大量のスレッドを各時点で利用可能なノードにマッピングしてくれる．よって，thread-move は，プログラマが並列計算の再構成を意識する必要がないという点ではプログラマビリティは高いが，**2.2.4** 節で指摘したように，透過的なスレッド移動を実現するには 2 つの問題がある．

第 1 の問題は，各スレッドでグローバル変数が使えないなど，一定のプログラミング制約が課せられる点である．スレッド移動の既存研究 [18, 19, 105, 49, 92, 90, 91, 89, 56, 183, 42, 85, 132, 193, 95, 192, 194, 114] ではこのプログラミング制約が正確に語られることはなかったが，**8.3** 節ではプログラミング制約を厳密に記述する．そのうえで，たしかに理解しにくいプログラミング制約が課せられるものの，通常の並列科学技術計算を記述するための記述力は保たれていることを確認する．

第 2 の問題は，スレッドが使用するメモリ領域を移動元プロセスと移動先プロセスとで同一のアドレスに配置しないと，スレッド移動時にポインタが無効化してしまう問題である．多くのスレッド移動の既存研究では，この問題を iso-address[18, 19, 132] の手法によって解決しているが，iso-address では計算規模が CPU のアドレス空間全体のサイズに制限されてしまうため，今後ますます計算規模が増大するにつれて手法が限界に達する可能性がある．そこで，thread-move では，計算規模がアドレス空間のサイズに制限されない新しいスレッド移動手法として **random-address** を提案する．**8.4** 節では random-address について述べ，**8.5** 節で random-address に基づくスレッド移動の実装について述べる．**8.6** 節では random-address の有効性をシミュレーションによって検証する．

以降では，議論を具体化させるために DMI の用語に即して説明する場合があるが，本章で述べるプログラミング制約，random-address の手法，スレッド移動の実装は，DMI にかぎらず，スレッド移動

を行う一般の処理系に対しても適用できるものである．

8.2 プログラミングモデル

thread-move におけるプログラミングモデルを図 8.1 に示す．

一番最初に生成されたプロセス 0 は DMI_main() 関数を呼び出す（5 行目）．ここで，他のプロセスが参加するのを待機することなくすぐにスレッドたちを生成してしまうと，それらのスレッドたちがすべてプロセス 0 上に生成されてしまう．当然，あとからプロセスを動的に参加させることで，プロセス 0 上のスレッドたちは他のプロセスに移動していくが，スレッド移動のコストは無視できないため，いったんプロセス 0 に生成した大量のスレッドたちを他のプロセスに移動させようとするのは非効率的である．そこで，DMI_gather_nodes() 関数を使ってある程度の個数のプロセスが参加するのを待機したあとで（18 行目），スレッドたちを生成する．この場合，生成されるスレッドたちは，その時点で参加しているプロセスたちに対して均等に分散される．スレッドを生成するには，まず DMI_scheduler_init() 関数を呼び出してスレッドスケジューラを初期化したあと（19 行目），DMI_scheduler_create() 関数を呼び出せばよい（21 行目）．生成された各スレッドは DMI_thread() 関数から実行を開始する．このとき，DMI_scheduler_create() 関数の第 3 引数に渡したグローバルアドレスはそのまま DMI_thread() 関数の引数に渡されるため，このグローバルアドレスに任意のデータを格納しておくことにより，DMI_main() 関数から各 DMI_thread() 関数へ任意のデータを渡すことができる．また，DMI_scheduler_create() 関数の第 2 引数にはこのスレッドのハンドルが返る．このハンドルを利用して，終了したスレッドを DMI_scheduler_join() 関数で回収したり（24 行目），DMI_scheduler_detach() 関数でデタッチしたりできる．

thread-move における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，スレッド移動が必要であると DMI が判断した任意の時点でスレッド移動が起きるわけではなく，その時点以降で，移動対象のスレッドがはじめて 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，スレッド移動の必要が生じていなければ何も行わずにすぐに返り，スレッド移動の必要が生じていれば，内部でスレッド移動を行い，移動先のプロセスで返る．すなわち，あるスレッドを移動する必要が生じていても，そのスレッドが DMI_yield() 関数を呼び出さないかぎりスレッド移動は起きない．プリエンプティブなスレッド移動を行わない理由は，ユーザプログラムにとって不都合なタイミングでスレッド移動が起きないようにするためである．したがって，プロセスの動的な参加/脱退に対応してスレッド移動を応答性よく実現するためには，各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，ユーザプログラムを記述しておく必要がある．たとえば，反復計算を行うユーザプログラムであれば，たとえば図 8.1 に示すように，各イテレーションの先頭に DMI_yield() 関数を記述すればよい．また，**7.2** 節で述べたように，デフォルトの DMI では少なくとも 1 個以上のプロセスの参加/脱退があった場合に再構成の必要性を検知してスレッドスケジューリングを発動するが，DMI_judge_reconf() 関数をユーザプログラムに定義することで，DMI が再構成の必要性を検知するために必要な条件を指示す

```
01: typedef struct arg_t {
02:     ...; /* arguments for a thread */
03: }arg_t;
04:
05: void DMI_main(int argc, char **argv) {
06:     arg_t arg;
07:     int rank, pnum, node_num;
08:     int64_t sched_addr, arg_addr, handle[THREAD_MAX];
09:
10:     pnum = atoi(argv[1]); /* the number of threads */
11:     node_num = atoi(argv[2]); /* the initial number of nodes */
12:     DMI_mmap(&sched_addr, sizeof(DMI_scheduler_t), 1);
          /* a global address space for a thread scheduler */
13:     DMI_mmap(&arg_addr, sizeof(arg_t) * pnum, 1);
          /* a global address space for storing arguments for threads */
14:     for (rank = 0; rank < pnum; rank++) {
15:         arg = ...; /* set the arguments for each thread */
16:         DMI_write(arg_addr + rank * sizeof(arg_t), sizeof(arg_t), &arg, EXCLUSIVE);
              /* write the arguments to the global address space */
17:     }
18:     DMI_gather_nodes(node_num); /* wait until node_num nodes gather */
19:     DMI_scheduler_init(sched_addr); /* initialize the thread scheduler */
20:     for (rank = 0; rank < pnum; rank++) { /* create threads */
21:         DMI_scheduler_create(sched_addr, &handle[rank], arg_addr + rank * sizeof(arg_t));
22:     }
23:     for (rank = 0; rank < pnum; rank++) { /* join the threads */
24:         DMI_scheduler_join(sched_addr, handle[rank]);
25:     }
26:     DMI_scheduler_destroy(sched_addr); /* destroy the thread scheduler */
27:     DMI_munmap(sched_addr);
28:     DMI_munmap(arg_addr);
29:     return;
30: }
31:
32: int64_t DMI_thread(int64_t arg_addr) { /* each thread */
33:     arg_t arg;
34:     int iter;
35:     DMI_read(arg_addr, sizeof(arg_t), &arg, GET); /* read the argument for this thread */
36:     for (iter = 0; iter < ITER_MAX; iter++) {
37:         DMI_yield(); /* give the thread scheduler a chance for migrating this thread */
38:         ...; /* the body of each iteration */
39:     }
40:     return DMI_NULL;
41: }
42:
43: int DMI_judge_reconf(DMI_node_t *in_nodes, DMI_node_t *out_nodes,
       DMI_node_t *cur_nodes, int in_node_num, int out_node_num, int cur_node_num) {
44:     return in_node_num + out_node_num >= 4;
45: }
```

図 8.1 thread-move のプログラミングモデル.

ることができる．

このように thread-move では，マルチスレッドプログラミングと同様のプログラミングによって，再構成可能な並列計算を簡単に記述できる．ただし，**8.3** 節で述べるように，thread-move にはいくつかのプログラミング制約がともなう．

8.3 プログラミング制約

本節では，まず **8.3.1** 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 **8.3.2** 節でプログラミング制約を記述する．さらに，**8.3.4** 節でそのプログラミング制約を緩和する．

8.3.1 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．thread-move では，アドレス領域全体を以下の 6 種類に分類してモデル化する（図 8.2）．なお，このモデル化は DMI の実装に特化したモデル化ではなく，スレッド移動を実現しようとする一般の処理系を対象としたモデル化であることを強調しておく：

$register_i^p$　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．

$static^p$　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．

global　グローバルアドレス空間領域．DMI の用語に即していえば，*global* 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．*global* は（DMI などの）処理系によってすべてのプロセスで共有されるように管理される．

$threadheap_i^p$　プロセス p 内のスレッド i のヒープ領域．マルチスレッド型の処理系において各スレッドを粒度としたスレッド移動を実現するためには，プロセスが使用するアドレス領域のなかで，各スレッドが使用するアドレス領域が明確になっている必要がある．よって，あるプロセスのヒープ領域全体のなかで，どの部分がすべてのスレッドによって共有利用されているヒープ領域で，どの部分が個々のスレッドだけによって利用されているヒープ領域なのかを，処理系が判別できなければならない．このモデル化では，前者のヒープ領域が $processheap^p$ であり，後者のヒープ領域が $threadheap_i^p$ である．一般に，処理系が $processheap^p$ と $threadheap_i^p$ を判別できるようにするためには，$threadheap_i^p$ を確保/解放するための手段として，通常の mmap() 関数/mremap() 関数/munmap() 関数とは区別された API が必要である．DMI の用語に即していえば，$threadheap_i^p$ 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

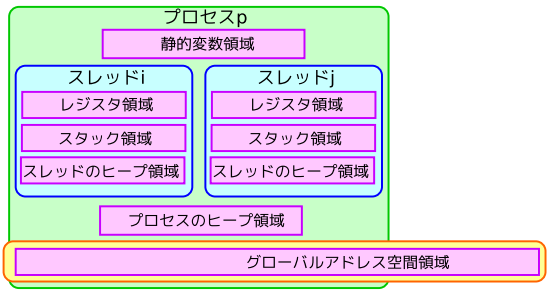

図 8.2 thread-move におけるアドレス領域のモデル化.

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

    pthread_mutex_lock(&mutex);
    if (buf == NULL) {
        buf = malloc(4096);
    }
    ...; /* make a string on buf using format */
    write(1, buf, strlen(buf));
    pthread_mutex_unlock(&mutex);
}
```

図 8.3 printf() 関数の実装例.

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[*1]．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．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．$threadheap_i^p$ は（DMI などの）処理系によってスレッドローカルに管理される．

8.3.2 プログラミング制約

前節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正確に述べる．**8.2** 節で述べたように，DMI では，ユーザプログラムから DMI_yield() 関数を呼び出してもらい，その DMI_yield() 関数のなか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，$register_i^p$，$stack_i^p$，$threadheap_i^p$，$global$ のいずれかのアドレス領域に含まれている[*2]．

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，

[*1] 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_realloc() 関数/ DMI_thread_free() 関数を提供している．

[*2] なお，このプログラミング制約は **8.3.4** 節において少し緩和していいなおされる．

$static^p$ と $processheap^p$ 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 f() を考える．このとき，f() 関数のなかでは $processheap^p$ を使用することになるが，f() 関数が返った時点ではスレッドの実行を継続するために必要なデータは $processheap^p$ には残っていないため，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．同様の例として，DMI_read() 関数や DMI_write() 関数などの DMI の API は内部的には malloc() 関数を使用しているが，すべての DMI の API は，その API の実行が終了した時点で $static^p$ と $processheap^p$ 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 の API を呼び出しても，(当然ながら) プログラミング制約には違反しない[⋆3] ．

要約すると，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，一定の記述力は保たれている：

- 各スレッドのヒープ領域に対する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 の用語に即していえば，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．
- グローバルアドレス空間の確保/解放と，グローバルアドレス空間に対する read/write．DMI の用語に即していえば，DMI_mmap() 関数/DMI_munmap() 関数/ DMI_mremap() 関数によるグローバル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() 関数や同期操作などの DMI の API の呼び出し．

したがって，スレッド間で共有する必要のあるデータはグローバルアドレス空間を使用し，各スレッドに固有のデータは各スレッドのヒープ領域を使用するようにユーザプログラムを記述すれば，第 **10** 章で評価するような多くの並列科学技術計算を十分に記述することができる．

8.3.3 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 の移動が必要とされていれば，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] ただし，非同期な DMI の API 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 $static^p$ と $processheap^p$ に残っている．よって，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．

(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 を復帰させ，DMI_yield() 関数を返す．

global 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らかの処理を行う必要はない．なぜなら，そもそもグローバルアドレス空間とは，系内のどのスレッドがどのプロセスからアクセスしてもデータが read/write できるよう，コヒーレンシが維持されているものだからである．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 **8.4** 節で述べる．

8.3.4 プログラミング制約の緩和

8.3 節において，スレッド移動にともなう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，このプログラミング制約はグローバル変数の使用を完全に禁止しており，これはプログラマに対して相当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する可能性のあ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，printf() 関数，sin() 関数などの，内部でグローバル変数を使用しうる 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，**8.3** 節で述べたプログラミング制約は若干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，内部的に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，printf() 関数を例にとり，プログラミング制約をどのように緩和できるかについて議論する．

たとえば，図 8.3 に示すような実装の printf() 関数を考える．実際の 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 *buf* を再利用する実装になっている．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，printf() 関数が返る実装になっている．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一度でも printf() 関数を呼び出してしまっているならば，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，スレッド移動の際には $static^p$ のデータは移動させないため，スレッド i が，移動後に移動先プロセス 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．スレッド i が移動する時点におけるプロセス p の *buf* の値を buf_p，プロセス q の *buf* の値を buf_q と表すことにする．いまの場合，スレッド移動の前にスレッド i が少なくとも 1 回はプロセス p において printf() 関数を呼び出している状況を考えているため，

buf_p と buf_q の組み合わせとしては，「$buf_p \neq$ NULL かつ $buf_q =$ NULL」または「$buf_p \neq$ NULL かつ $buf_q \neq$ NULL」の 2 とおりが考えられる．第 1 に，$buf_p \neq$ NULL かつ $buf_q =$ NULL の場合には，スレッド i が移動後にプロセス q において printf() 関数を呼び出すと，プロセス q において printf() 関数の 1 回目の呼び出しが起きることになり，buf_q に新たにメモリが割り当てられるが，ここでは何の問題も生じない．第 2 に，$buf_p \neq$ NULL かつ $buf_q \neq$ NULL の場合には，スレッド i が移動後にプロセス q において printf() 関数を呼び出すと，プロセス q において printf() 関数の 2 回目以降の呼び出しが起きることになり，すでに buf_q に割り当てられているメモリを使用して printf() 関数が実行されるが，ここでも何の問題も生じない．このように，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_p 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_q の値に基づいて正しく実行される．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 printf() 関数が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．なお，以上の議論はグローバル変数の使用の可否にかぎったものではなく，一般には，$static^p$ や $processheap^p$ のアドレス領域の使用の可否に関するものである．したがって，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，$register_i^p$，$stack_i^p$，$threadheap_i^p$，$global$ 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 $static^p$ や $processheap^p$ 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可能な場合がある，ということである．実際に，スレッドセーフな libc 共有ライブラリの関数のなかには安全に呼び出せるものも多い．

8.3.5 プログラミング制約に関する関連研究

ここまでスレッド移動にともなうプログラミング制約について議論してきたが，本節では，既存のスレッド移動の研究におけるプログラミング制約についてまとめる．

第 1 に，PM2[18, 19]，Adaptive MPI[80, 81]，MigThread[92, 90, 91]，Arachne[56] など，マルチスレッド型の処理系でスレッド移動を実現する多くの既存研究では，そもそもグローバル変数の使用が禁止されている．しかし，プログラミング制約が正確に議論されているわけではなく，ライブラリ関数の使用の可否についても議論されないまま printf() 関数などが使用されている．これは，前節で述べた

ように，現実問題としては，スレッドセーフな libc 共有ライブラリの関数は，内部的にグローバル変数を使用していたとしても安全に呼び出せてしまう場合が多いため，これらの既存研究では問題にされていなかったものだと思われる．

第 2 に，Windows 環境においてスレッド移動を実現する Tern[102] 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 [161] の移動がサポートされており，プログラマはグローバル変数のかわりにスレッドローカルストレージを利用できる．このアプローチは，そもそもプログラマの視点で機能的に要請されているものは，グローバル変数ではなくスレッドローカルストレージであるという意味において，合理的である．すなわち，透過的にスレッド移動が行われる処理系において，プログラマが「グローバル変数を使いたい」と思う場合に必要とされているものは，「各スレッドのどの関数からでも触れる変数（＝スレッドローカルストレージ）」であって，「そのスレッドが属しているプロセス内のすべてのスレッドから触れる変数（＝真の意味でのグローバル変数）」ではない．なぜなら，透過的にスレッド移動が行われる処理系では，「そのスレッドが属しているプロセス内のすべてのスレッド」がどれなのかは通常はプログラマからは見えないし，そもそもプログラマに見せるべきではないため，プログラマにとっては真の意味でのグローバル変数が使えたとしても意味がないからである．このように，スレッドローカルストレージを活用するアプローチは合理的であるが，（すでにコンパイル済みの）libc 共有ライブラリで使用されているグローバル変数には対応できない．また，Tern が対象としている 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が実装上難しいという問題もある．

第 3 に，Java においてスレッド移動を実現している JESSICA2[194, 95, 193, 110] 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移動することができる．そして，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の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集約して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 言語におけるスレッド移動に応用させることは難しい．また，各スレッドごとにマスタノードに親スレッドを用意したり，グローバル変数へのアクセスのたびにマスタノードに実行を引き戻したりすることは，性能上のボトルネックになる可能性もある．

以上をまとめると，thread-move におけるプログラミング制約は，スレッド移動の既存研究が課しているプログラミング制約を何らか改善しているわけではなく，とくに，PM2，Adaptive MPI，MigThread，Arachne におけるプログラミング制約と同一である．しかし，アドレス領域のモデル化に基づいて，**プログラミング制約を正確に記述した**点に意義があるといえる．

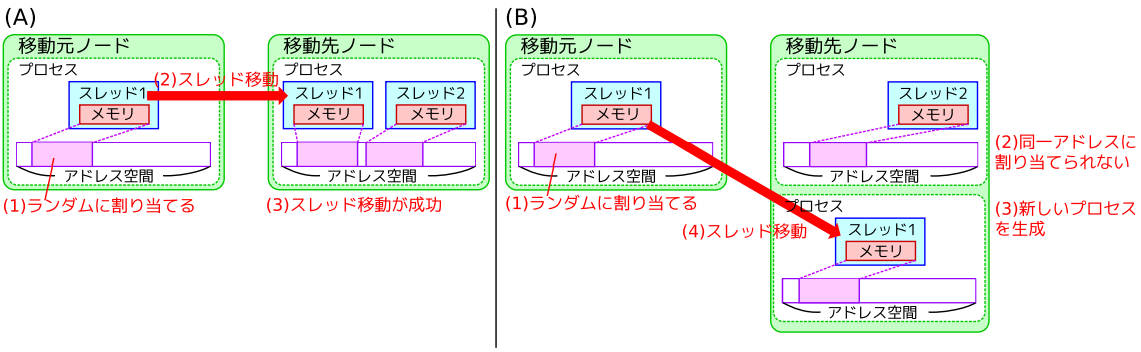

図 8.4　random-address のアルゴリズム．（A）アドレスが衝突しない場合，（B）アドレスが衝突する場合．

8.4　アドレス空間のサイズに制限されないスレッド移動

本節では，計算規模がアドレス空間のサイズに制限されないスレッド移動の手法と，そのためのアドレス空間管理の手法について述べる．

8.4.1　基本アイディア

thread-move では，計算規模がアドレス空間のサイズ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．つまり，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を知らない．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 8.4（A））．したがって，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 へと移動させることを考える．このとき，「運がよければ」，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 8.4（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' のなかにスレッド i を移動させる（図 8.4（B））.

この random-address では，スレッドの移動先プロセスでアドレスが衝突した場合に，動的に新しいプロセスを生成し，そのプロセスをグローバルアドレス空間に参加させる必要がある．すなわち，random-address は，DMI のグローバルアドレス空間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手法であり，その意味で新規的なスレッド移動の手法である．

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 上の別のプロセス 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 のなか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 のなか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，

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

となるように各 S_i^t を分類したとする．このとき，

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

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，つまりプロセス数が最小にな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，1 ノード内の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．

8.4.2 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なため，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グローバルアドレス空間のコヒーレ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，他の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 スレッドなどの複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に対する擾乱が大きくなる．第 2 に，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しているため，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，あるページを INVALIDATE モードで read 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，できるかぎり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第 B 章で与える）：

問題の概略 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り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，以下のきわめて単純な戦略であることが証明できる（証明は第 B 章で与える）：

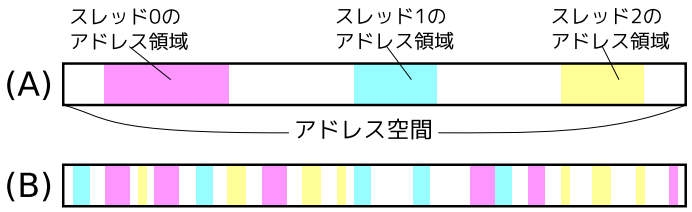

図 8.5 アドレス領域の連続的な使用と離散的な使用．（A）連続的な使用，（B）離散的な使用．

```
01: Z_i : Set of region
02:
03: when thread i is created:
04:     Z_i := ∅
05:     return
06:
07: when thread i mmaps size bytes:
08:     foreach (ptr, length) ∈ Z_i do
09:         p := fixed_mmap(ptr + length, size)
10:         if p != MAP_FAILED then
11:             Z_i := Z_i\{(ptr, length)}
                       ∪{(ptr, length + size)}
12:             reduction(Z_i)
13:             return
14:         endif
15:     endfor
16:     while 1 do
17:         addr := rand(inf, sup)
18:         p := fixed_mmap(addr, size)
19:         if p != MAP_FAILED then
20:             Z_i := Z_i ∪ {(p, size)}
21:             reduction(Z_i)
22:             return
23:         endif
24:     endwhile
25:     return
26:
27: when thread i munmaps region (p, size):
28:     munmap(p, size)
29:     foreach (ptr, length) ∈ Z_i do
30:         if ptr == p and p + size == ptr + length then
31:             Z_i := Z_i\{(ptr, length)}
32:         else if ptr == p and p + size < ptr + length then
33:             Z_i := Z_i\{(ptr, length)}
                       ∪{(p + size, length − size)}
34:         else if ptr < p and p + size == ptr + length then
35:             Z_i := Z_i\{(ptr, length)} ∪ {(ptr, length − size)}
36:         else if ptr < p and p + size < ptr + length then
37:             Z_i := Z_i\{(ptr, length)} ∪ {(ptr, p − ptr)}
                       ∪{(p + size, ptr + length − p − size)}
38:         endif
39:     return
40:
41: when thread i is destroyed:
42:     foreach (ptr, length) ∈ Z_i do
43:         munmap(ptr, length)
44:     endfor
45:     return
```

図 8.6 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．

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

上記の戦略の意味は，**「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 8.5（A）のようにアドレスを使用する方が，図 8.5（B）のようにアドレスを使用するよりも，スレッド移動時のアドレス衝突確率が小さい．したがって，random-address では，各スレッドが使用するアドレス領域（$stack_i^p$ と $threadheap_i^p$）ができるかぎり連続的になるようにアドレス領域を管理する．

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を図 8.6 に示す．図 8.6 のアルゴリズムにおいて，$(ptr, length)$ 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（**8.5.2.4** 節を参照）．Z_i は，スレッド i が現在使用しているアドレス領域の集合を表す．プログラムの進行にともなってアドレス領域の確保/解放が繰り返されると，スレッド i が使用するアドレス領域全体は，いくつかの連続的なアドレス領域に分断されてしまうが，その集合を Z_i として管理する．$|Z_i| = 1$ のとき，スレッド i は真に連続的なアドレス領域を使用していることを意味する．reduction(Z_i) は，アドレス領域の集合 Z_i 内の任意の 2 個以上のアドレス領域に関して，連続したアドレス領域としてまとめられるものを 1 個のアドレス領域にまとめあげる関数とする．

図 8.6 のアルゴリズムの目標は，$|Z_i|$ をできるかぎり小さくすることである．よって，*size* バイトのアドレス領域の確保を行う場合には（DMI の用語では，DMI_thread_mmap() 関数が呼び出された場合には），アドレス領域の集合 Z_i のアドレス領域のうち，末尾を *size* バイトだけ伸ばせるようなアドレス領域を探し，もし見つかればそのアドレス領域を *size* バイトだけ伸ばすことでアドレス領域を確保する．この場合 $|Z_i|$ は増えない．もし見つからなければ，乱数によってアドレスを生成し，*size* バイトのアドレス領域を確保する．この場合 $|Z_i|$ が 1 増える．アドレス領域の解放を行う場合には（DMI の用語では，DMI_thread_munmap() 関数が呼び出された場合には），単にそのアドレス領域を解放する．

8.4.3 特定の知識に基づいたアドレス衝突確率のさらなる最小化

各スレッドが使用するメモリ量やメモリの使用傾向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になる．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* の整数倍にアラインされていなければならない」**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にアドレス衝突確率がどのように変化するか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わかる．極端な例としては，*align* の値がアドレス空間全体のサイズと等しければ，各アドレス空間には 1 個のアドレス領域しか配置できなくなるので，スレッド移動時にはつねにアドレス衝突が発生する．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，もっと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．

8.4.4 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なか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なか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たとしても，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．

8.3.1 節で述べたように，thread-move では，アドレス領域全体を $register_i^p$，$stack_i^p$，$threadheap_i^p$，$static^p$，$processheap^p$，$global$ の 6 種類のアドレス領域に分類して管理する．そして，**8.3.1** 節と **8.3.3** 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 個であり，初期領域に含まれる可能性があるのは，$static^p$，$processheap^p$，$global$ の 3 個である．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 $static^p$，$processheap^p$，$global$ 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，thread-move では，各プロセスが利用可能なアドレス空間全体（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，$processheap^p$，$global$ 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．

とくに，前半部分に関しては，図 8.6 に示すアルゴリズムにしたがって，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，割り当てるアドレス領域を DMI の処理系が制御できる自由度はそもそもないが，$static^p$ 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ありえ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，いずれのプロセスにおいても移動領域が $static^p$ 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すべてのプロセスで同一になるため，移動領域と重なることはありえない．前半部分と後半部分のサイズをどのように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．

8.5 スレッド移動の実装

本節では，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について述べる．

8.5.1 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 を作成することができる．このとき，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．$func$ には，コンテキスト ucp へのはじめてのコンテキストスイッチが起きたときに実行される関数を指定する．さらに，swapcontext(ucontext_t *$oucp$, ucontext_t *ucp) 関数を呼び出すと，この swapcontext() 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 へとコンテキストスイッチすることができる．とくに，2 個のコンテキストの間で swapcontext() 関数を交互に呼び出すことによって，それらのコンテキストを自由にコンテキストスイッチできる．

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，あるプロセス p でスレッド i が生成されたとする．この時点におけるスレッド i のコンテキストを c_0 と名づける．

(2) コンテキスト c_0 は，makecontext(A,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を作り，そのコンテキストを変数 A に格納する．この新しいコンテキストを c_1 と名づける．

(3) コンテキスト c_0 が，swapcontext(B,A) 関数を呼び出す．その結果，現在のコンテキスト c_0 が変数 B 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 で実行され始める．

(4) やがて，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．また，このときスレッド i を移動させる必要が生じていたとする．すると，DMI_yield() 関数は，(3) で保存した変数 B を指定して swapcontext(A,B) 関数を呼び出す．その結果，コンテキスト c_1 が変数 A 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，過去に (3) においてコンテキスト c_0 が呼び出していた 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 $register_i^p$ は変数 A に保存されている．よって，コンテキスト c_0 は，変数 A そのもの（$register_i^p$）と，コンテキスト c_1 のスタック領域（$stack_i^p$）と，（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，これら 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り当てることが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 から受信した変数 A をそのまま指定して swapcontext(B,A) 関数を呼び出す．これにより，過去に（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，スレッドの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 が返してくる成功/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か，スレッド移動先として別のプロセスを選択してスレッド移動を試みる．

8.5.2 システムコールのハイジャック

8.5.2.1 ハイジャックの必要性

8.4.4 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空間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，$processheap^p$，$global$ のアドレス領域は後半部分に割り当てる必要がある．このように，各アドレス領域が使用するアドレスを DMI のような処理系が明示的に制御するためには，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する必要がある．

ここで，各アドレス領域に関して，割り当てるアドレスを明示的に制御することが可能かどうかを確認する．第 1 に，**8.5.1** 節で述べた実装では，$register_i^p$ は ucontext_t 型の変数として扱い，$stack_i^p$ は makecontext() 関数を呼び出すときに明示的に指定できるので，これらのアドレス領域を明示的に制御することは可能である．第 2 に，$threadheap_i^p$ は，定義より，処理系が用意した専用の API によって確保/解放されるアドレス領域であるから，そのアドレス領域を明示的に制御することは可能である．DMI に即していえば，DMI_thread_mmap() 関数/DMI_thread_munmap() 関数/ DMI_thread_mremap() 関数が確保/解放するアドレス領域を明示的に制御すればよい．第 3 に，$processheap^p$ と $global$ は，（malloc() 関数/free() 関数/realloc() 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，これらのシステムコールが確保/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しなければならない．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/brk() 関数をハイジャックする．

8.5.2.2 ハイジャックすべきレイヤの検討

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．ただし，本節で議論および提案するシステムコールのハイジャックの手法は，thread-move の要

素技術としての応用にかぎらず，一般に，**任意のシステムコールをユーザレベルでハイジャックするための汎用的な手法**であることを強調しておく．以下では，カーネル内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 共有ライブラリ内に定義されている mmap() 関数を libc_mmap() 関数，mmap() 関数を何らかのレイヤにおいてハイジャックしたあとに実行したい（処理系のなかに定義されている）mmap() 関数を hijack_mmap() 関数と表記して区別する．

Linux カーネル 2.6 においては，実行バイナリから libc_mmap() 関数が呼び出された場合，以下の動作が起きる [201, 152]：

(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() 関数が呼び出されるのが 1 回目ならば，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[4] 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() 関数の本体が実行されるまでの間に，処理をハイジャックして，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 ヶ所ある．以下ではこの 4 ヶ所におけるハイジャックの概要と問題点について議論する：

（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_mmap() 関数のアドレスが検索されるときに検索対象となる共有ライブラリの検索順序を変更することができ，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る．これにより，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という問題がある．とくに，libc 共有ライブラリ自体は静的リンクされてコンパイルされている場合が多いため，libc 共有ライブラリの malloc() 関数がその内部で呼び出す libc_mmap() 関数をハイジャックすることはできない．したがって，いまの目標からして（1）におけるハイジャックは不十分であるといえる．

（3）におけるハイジャック　libc_mmap() 関数のコードを書き換えて，sys_mmap() 関数の代わりに hijack_mmap() 関数を呼び出させる手法である．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() 関数のプログラム自体を書き換えてコンパイルし，改造 libc 共有ライブラリを作成する．そして，DMI のプログラム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．この手法の問題点は，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プログラムの変更箇所が広範囲に及ぶ点や，各実行環境ごとに改造 libc ライブラリをコンパイルしなければならないため移植性が低い点である．そこで，DMI では，実装を容易化すると同時に移植性を高めるため，動的に libc 共有ライブラリのコードを書き換える手法を提案する．この手法の詳細と得失については次節で議論する．

（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．ところが，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生じ，ユーザレベルで実装するという目標に違反してしまう．また，カーネルモジュールの実行には root 権限が必要になるという点も問題である．

（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により，デバッギプロセスに hijack_mmap() 関数を実行させることが可能になる．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，それがどのプロセスによって発行されたシステムコールなのかは知ることができるが，どの pthread によって発行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．

8.5.2.3　共有ライブラリのコードを動的に書き換える手順

以上の観察をふまえて，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書き換える**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に，プログラム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 共有ライブラリ内の libc_mmap() 関数のコードの先頭部分を上書きして，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ある．なぜなら，hijack_mmap() 関数は，thread-move のアドレス空間管理に基づいて割り当てるべきアドレス領域を決定したあと，いずれは OS から実際にアドレス領域を割り当てるために libc_mmap() 関数を呼び出す必要があるが，libc_mmap() 関数を呼び出した瞬間に再度 hijack_mmap() 関数が呼び出されてしまうため，無限に再帰してしまうからである．つまり，OS から実際に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，libc_mmap() 関数のコードの先頭部分を上書きしてジャンプ命令を書き込む前に，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．要約すると，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) 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 8.7（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上書き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られる．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上書き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 8.7（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 8.7（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 $[libc\_mmap, libc\_mmap + injection)$ に上書きする：

```
mov   hijack_mmap, %rax
```

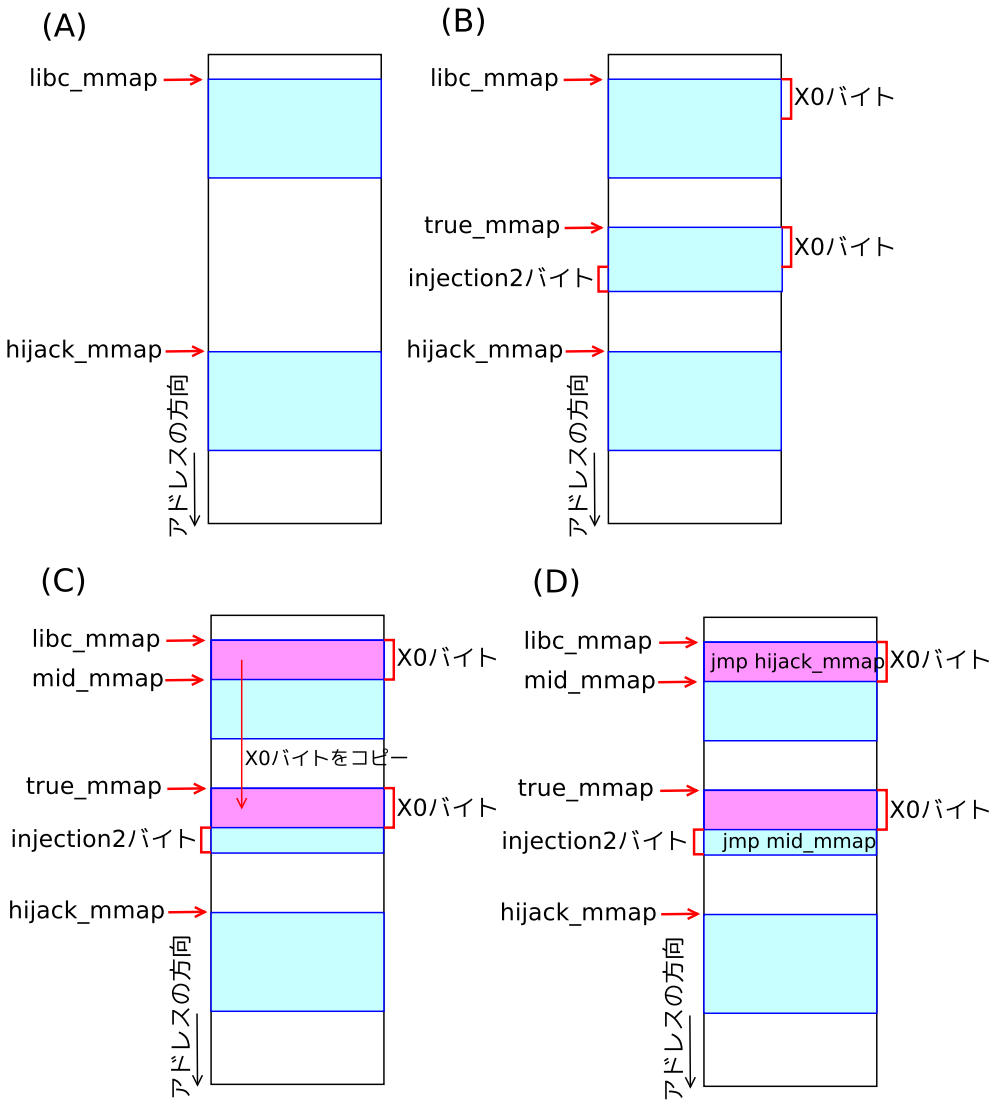

図 8.7 共有ライブラリのコードを動的に書き換える手順.

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 $[true\_mmap + x_0, true\_mmap + x_0 + injection2)$ に上書きする（図 8.7（D））：

mov mid_mmap, %r11

jmpq *%r11

これにより，true_mmap() 関数が呼び出されると，本来アドレス領域 $[libc\_mmap, libc\_mmap + x_0)$ 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，システムコール呼び出しにおける関数呼び出し規約に基づき，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，プログラムの 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() 関数のなかで $processheap^p$ と $global$ のアドレス領域を明示的に制御できるようになる．

なお，この手法は，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．ただし，通常のプログラムではこのような処理はほとんど行われないと考えられる．

8.5.2.4 指定したアドレス領域をメモリマップするアルゴリズム

ここまでの議論では，$register_i^p$，$stack_i^p$，$threadheap_i^p$，$static^p$，$processheap^p$，$global$ 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 **8.4.4** 節で述べた 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」ための関数が必要である．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 8.6 における fixed_mmap() 関数）が必要である．これは通常の mmap() 関数を適当なオプション付きで呼び出すことで実現できそうに思えるが，実は自明には実現できない．本節では，カーネルを変更することなくユーザレベルで上記の仕様の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 $start$ に指定する方法が考えられるが，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とも，アドレス領域 $[start, start + length)$ にアドレス領域が確保される保証はない．実際，**6.1** 節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 4 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．

```
01: void* fixed_mmap(void *start, size_t length, int prot, int flags) {
02:   void *ptr;
03:   int ret;
04:   static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
05:
06:   pthread_mutex_lock(&mutex); /* lock */
07:   errno = 0;
08:   ptr = true_mremap(start, PAGESIZE, PAGESIZE * 2, 0);
        /* try to expand 1 page beginning from start to 2 pages */
09:   if (ptr == MAP_FAILED && errno == EFAULT) {
10:     ptr = true_mmap(start, PAGESIZE, prot, flags | MAP_FIXED, -1, 0);
          /* this certainly succeeds */
11:     assert(ptr != MAP_FAILED);
12:     ptr = true_mremap(start, PAGESIZE, length, 0);
        /* try to expand 1 page beginning from start to length bytes */
13:     if (ptr == MAP_FAILED) { /* if it fails */
14:       ret = true_munmap(start, PAGESIZE); /* cleanup */
15:       assert(ret == 0);
16:     }
17:   } else if (ptr != MAP_FAILED) { /* if the first true_mremap() succeeds */
18:     ret = true_munmap(start + PAGESIZE, PAGESIZE); /* cleanup */
19:     assert(ret == 0);
20:     ptr = MAP_FAILED; /* this fixed_mmap() failed */
21:   }
22:   pthread_mutex_unlock(&mutex);
23:   return ptr;
24: }
```

図 8.8　指定したアドレス領域をメモリマップするアルゴリズム.

以上の観察をふまえて，DMI では，指定したアドレス領域をメモリマップするアルゴリズムとして図 8.8 に示すアルゴリズムを提案する．図 8.8 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* から始まる連続する 2 ページに関して，（1 ページ目の状態，2 ページ目の状態）の組み合わせとしては以下の 5 とお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.
（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() 関数を MAP_FIXED オプション付きで発行する．これにより，1 ページ目のアドレス領域が確実に割り当てられる．いまの場合，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() 関数によってすでに割り当てられているアドレス領域が上書きされてしまうことはない．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．

8.6 シミュレーションによるアドレス衝突確率の評価

8.6.1 実験設定

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 [200] 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全体のサイズを $2^{address}$ 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，サイズが等しい *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サイズはすべて *memory*/*chunk* バイトであるとする．$chunk = 1$ の場合が，真に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* 個の各小アドレス領域の先頭アドレスは，*align* の整数倍の値のなか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 のなかへとスレッド移動させるとき，ノード P のなか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 のなか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序でノード 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*）の各組み合わせに対して 10

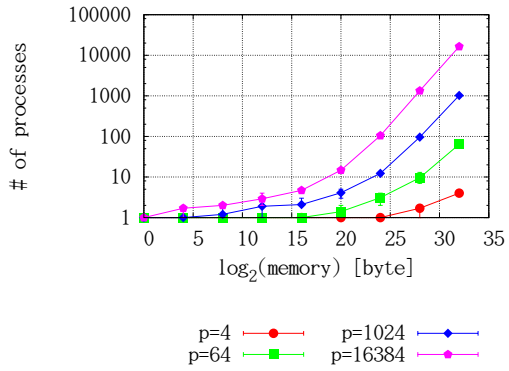

図 8.9 $N(32, process, memory, 1, 1)$.

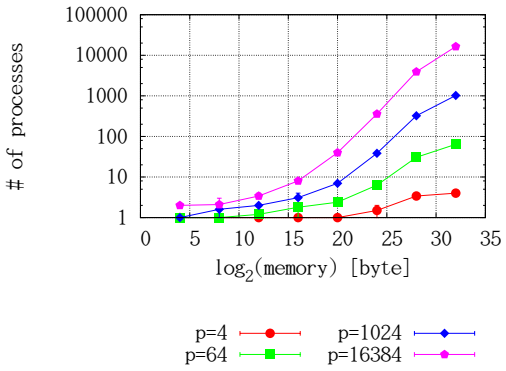

図 8.10 $N(32, process, memory, 16, 1)$.

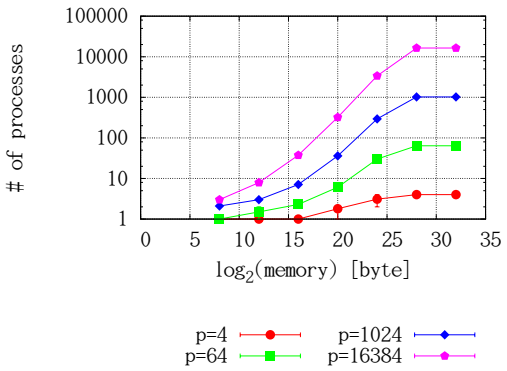

図 8.11 $N(32, process, memory, 256, 1)$.

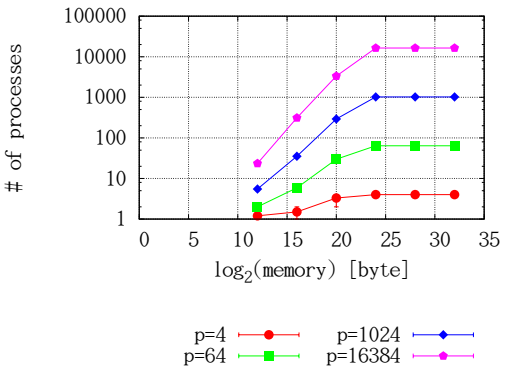

図 8.12 $N(32, process, memory, 4096, 1)$.

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，$N_{\min}(address, process, memory, chunk, align)$，$N_{\max}(address, process, memory, chunk, align)$，$N_{\text{avg}}(address, process, memory, chunk, align)$ と表す．

8.6.2 結果と考察

図 8.9 から図 8.18 までに，シミュレーションの結果を示す．これらのすべてのグラフでは $align = 1$ としている．また，1 個のグラフが 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[⋆4]．各折れ線中の 1 個の点は，$N_{\text{avg}}(address, process, memory, chunk, align)$ 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 8.9 のグラフにおいてもっとも右下の赤い点は，「『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

⋆4 グラフ中では $process$ を p と略記している．

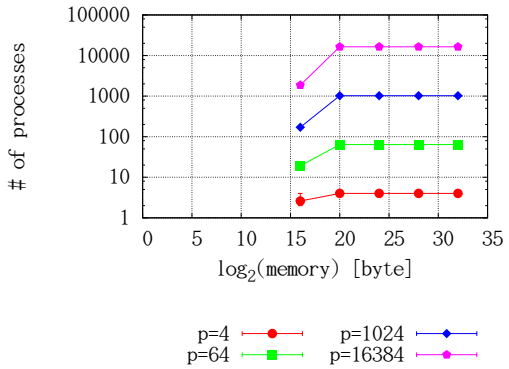

図 8.13 $N(32, process, memory, 65536, 1)$.

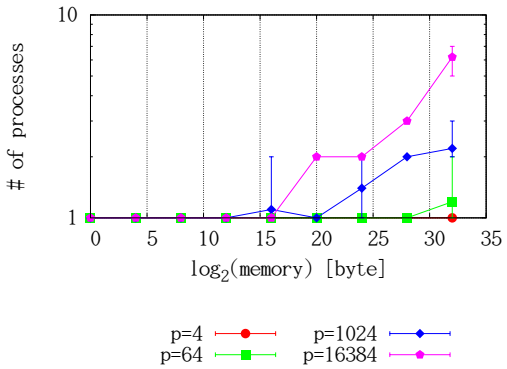

図 8.14 $N(47, process, memory, 1, 1)$.

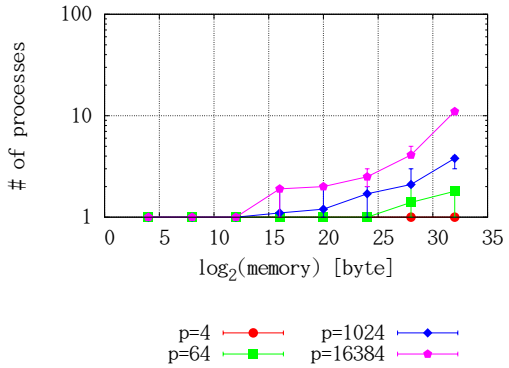

図 8.15 $N(47, process, memory, 16, 1)$.

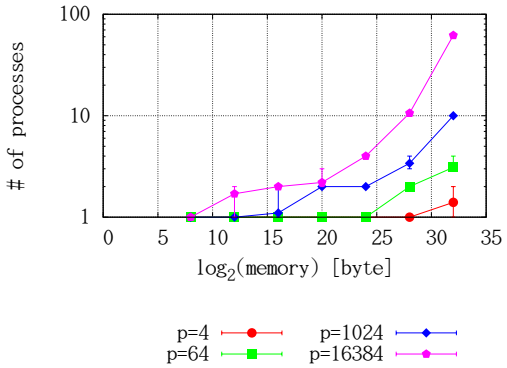

図 8.16 $N(47, process, memory, 256, 1)$.

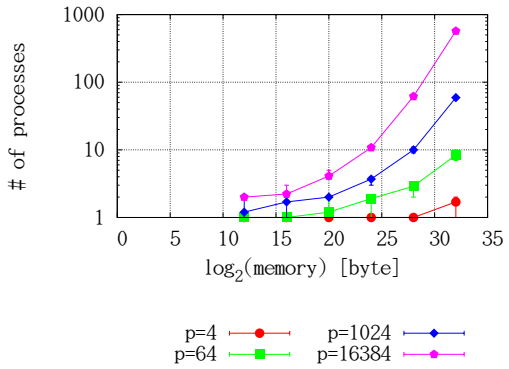

図 8.17 $N(47, process, memory, 4096, 1)$.

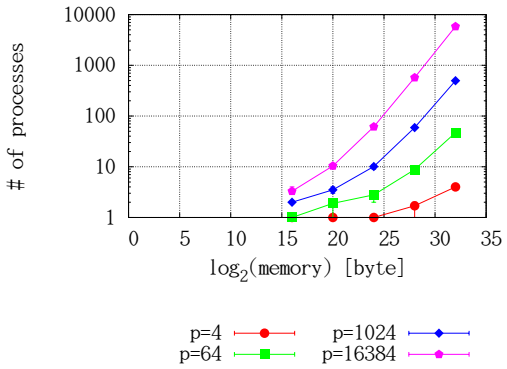

図 8.18 $N(47, process, memory, 65536, 1)$.

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 とおりしか存在せず，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 個のノードへと移動させたとすれば，当然，生成されるプロセス数はかならず 4 個になる．したがって，図 8.9 のグラフにおいてもっとも右下の赤い点の値は 4 であり，エラーバーの下限値も上限値も 4 になっている．なお，**8.4.4** 節で述べたように，各プロセス内では各スレッドの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しているかは問題にならないことに注意する．また，*chunk* の値は *memory* の値以下である必要があるため，図 8.9 から図 8.18 のグラフでは，*chunk* の値が大きくなるにつれて，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，本シミュレーションの結果より以下の事実が読みとれる：

(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もっとも小さく，各プロセスの使用するアドレス領域が離散化するにしたが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，**8.4.2** 節で述べた random-address の戦略が確かに最適であることを裏づけている．なお，random-address の最適性の正確な証明は第 B 章で行う．

第 2 に，図 8.14 と図 8.18 より以下の定量的な事実が読みとれる：

- 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 8.14 におけるもっとも右上の点）．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

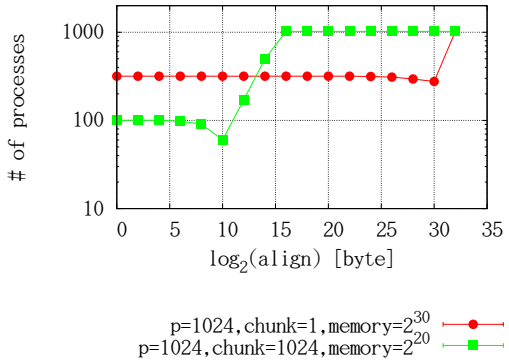

図 8.19 $N(32, 1024, 2^{30}, 1, align)$, $N(32, 1024, 2^{20}, 1024, align)$.

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わかる．

- 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 8.18 におけるもっとも右上の点）．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**ことがわかる．

第 3 に，**8.4.3** 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 8.19 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 8.19 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わかる．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サイズが $memory/chunk = 2^{30}$ であり，$N_{\text{avg}}(32, 1024, 2^{20}, 1024, align)$ では $memory/chunk = 2^{10}$ であることによる．すなわち，**8.4.3** 節で述べたように，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．

8.7 要約：利点と欠点

本章では，透過的なスレッド移動に基づく並列計算の再構成をユーザレベルで実現するための手法として，thread-move のプログラミングモデルの設計と実装について述べた．thread-move の新規性は以下のとおりである：

- 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，iso-address に基づく既存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，アドレス空間のサイズに制限されないスレッド移動手法として random-address を提案している．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ている．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の手法である．
- 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汎用的な手法を提案している．
- 従来のスレッド移動の処理系では，あいまいなままあまり問題とされてこなかったプログラミング制約について緻密に議論している．

並列計算の再構成を thread-move によって実現することの利点は，プログラマは並列計算の再構成を意識することなく，十分な数のスレッドを生成しておくだけで透過的なスレッド移動を実現できる点にある．これに対して，第 1 の欠点は，1 プロセッサあたり複数のスレッドを割り当てることによって性能が低下する点である．第 2 の欠点は，非常に理解しにくいプログラミング制約が存在する点である．たとえば，あるライブラリ関数がプログラミング制約を満たすかどうかは，そのライブラリ関数が $static^p$ と $processheap^p$ をどのように使用しているかの実装を熟読しないかぎり判断できない．第 3 の欠点は，一部の malloc() 関数/free() 関数/realloc() 関数を，処理系の専用の API に書き換えなければならない点である．

次章では，新しいカーネルプリミティブを導入することで，上記の第 2 の欠点と第 3 の欠点を解決する．

第 9 章

真に透過的なスレッド移動を実現するためのカーネルプリミティブ

本章では，half-process-move のプログラミングモデルに基づき，真に透過的なスレッド移動によって並列計算を再構成する方法について述べる．なお，「真に透過的である」とは，「(thread-move に存在するような,) スレッド移動を実現させるための余計なプログラミング制約が存在しない」という意味である．また，その要素技術として，部分的にアドレス空間を共有するプロセスを実現するための汎用的な新しいカーネルプリミティブとして half-process を提案し，その応用可能性，設計，実装について述べる．

9.1 全体像

half-process-move の目的は，thread-move におけるプログラミング制約を完全に撤廃することである．すなわち，グローバル変数の使用に関する制約や，malloc() 関数/realloc() 関数/free() 関数の書き換えなどを撤廃して，図 8.1 に示したプログラミングモデルのもとで，真に透過的なスレッド移動を実現できるようにする．いい換えると，並列計算の再構成に対応していない DMI のプログラムに対して，**DMI_yield() 関数をたった 1 行追加するだけで**，並列計算の再構成を実現できるようにする．

真に透過的なインスタンス移動にとっての要請は，**2.2.5** 節で詳しく議論し，以下の結論を得た：

- 真に透過的なインスタンス移動を実現するためには，各インスタンスが使用するアドレス空間は独立している必要があるため，各インスタンスはスレッドとしてではなくプロセスとして実装される必要がある．
- しかし，インスタンス間でのデータ共有のオーバヘッドを小さくしたり，(処理系の開発者にとっての) データ共有のプログラマビリティを高めるためには，各インスタンスはスレッドとして実装される必要がある[⋆1] ．

[⋆1] なお，本稿では，MPI や DMI などの並列分散プログラミング処理系を利用してユーザプログラムを記述する人間のことを「プログラマ」と呼び，MPI や DMI などの並列分散プログラミング処理系自体を開発する人間のことを「開発者」と

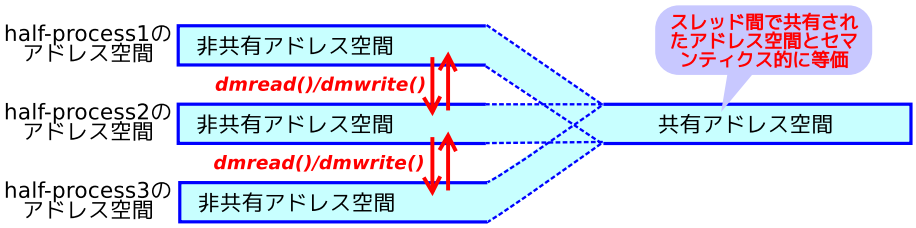

図 9.1 half-process の設計.

要するに，真に透過的なインスタンス移動を実現するためには，インスタンスどうしでアドレス空間を共有したい局面と共有したくない局面が混在する．したがって，真に透過的なインスタンス移動を実現するためには，スレッドとプロセスの「中間」の機能を持つインスタンスが必要となる．そこで，本研究では，**部分的にアドレス空間を共有するプロセス**を実現する新しいカーネルプリミティブとして，**half-process** を提案する．

half-process の概要を図 9.1 に示す．各 half-process のアドレス空間は，非共有アドレス空間と共有アドレス空間から構成される．非共有アドレス空間は各 half-process に固有のアドレス空間で，そのセマンティクスはプロセスのアドレス空間と等価である．各 half-process が mmap() 関数によって確保するメモリ領域やすべての静的変数は，非共有アドレス空間に確保される．一方で，共有アドレス空間はすべての half-process によって共有されており，**そのセマンティクスはスレッド間で共有されているアドレス空間と等価**である．各 half-process が共有アドレス空間にメモリ領域を確保するためには，単に mmap() 関数に MAP_HALFPROC オプションを付けるだけでよい．さらに，プロセス間の複雑なデータ共有を実現しやすくするために，half-process どうしがお互いの非共有アドレス空間をダイレクトメモリアクセスするためのシステムコールも提供する．

なお，本研究における half-process の直接的な目的は，half-process を用いることで真に透過的なスレッド移動を実現することであるが，half-process 自体は，**プロセス間のデータ共有を簡単かつ高速に実現するための汎用的なカーネルプリミティブ**として提案していることを強調しておく．以降では，まず **9.2** 節で half-process とプロセス間共有メモリなどとの相違点を明確化させ，**9.3** 節で half-process の応用可能性について議論したあと，**9.4** 節と **9.5** 節でそれぞれ half-process の設計と実装について述べる．そのうえで，**9.6** 節で真に透過的なスレッド移動に対して half-process をどのように応用させられるかについて述べる．とくに，**9.2** 節から **9.5** 節までの議論は，スレッド移動に限定されない，half-process の汎用的な性質に関する議論である．

9.2 プロセス間通信に関する関連研究

本節では，プロセス間通信に関する既存研究と half-process との相違点について明らかにする．

呼んで区別する．

9.2.1 プロセス間共有メモリ

half-process の共有アドレス空間とプロセス間共有メモリとの相違点について明確化させる．そのため，まず，half-process の共有アドレス空間のセマンティクスを明確化させる．**9.1** 節で述べたように，half-process の共有アドレス空間は，「スレッド間で共有されているアドレス空間とセマンティクス的に等価」であるが，これは，具体的には，以下の 2 つのセマンティクスが満足されることをいう：

セマンティクス I　half-process によって発行された mmap() 関数/munmap() 関数/mprotect() 関数などのメタ操作の結果は，すべての half-process に対して即座に反映されなければならない．いい換えると，ある half-process がメタ操作を発行したとき，そのメタ操作が完了した時点では，そのメタ操作の結果はすべての half-process [*2] によって見えていなければならない．

セマンティクス II　すべての half-process は，同一のアドレスによって同一のデータにアクセスできなければならない．いい換えると，ある half-process がアドレス a によってデータ x にアクセスできるならば，他のすべての half-process もアドレス a によってデータ x にアクセスできなければならない．

プロセス間共有メモリは，上記の 2 つのセマンティクスを満たしていないことを確認する．POSIX のプロセス間共有メモリ[*3] では，shm_open() 関数によってプロセス間共有メモリ m を作成し，ftruncate() 関数によってプロセス間共有メモリ m のサイズを自由に拡張/縮小させることができる．そして，多数のプロセスたちが，MAP_SHARED オプション付きの mmap() 関数を使って，このプロセス間共有メモリ m を各プロセスのアドレス空間にメモリマップすることによって，これらのプロセス間でプロセス間共有メモリを実現することができる．ここで注目すべき事実は，MAP_SHARED オプション付きの mmap() 関数をプロセス p がみずから呼び出さないかぎり，プロセス p はプロセス間共有メモリ m を利用できないという点である．いい換えると，プロセス p が他の多数のプロセスたちとの間でプロセス間共有メモリを実現したいと思った場合，プロセス p が mmap() 関数を呼び出すだけではプロセス間共有メモリは実現できず，何らかの方法で他のすべてのプロセスたちに対して指示を出して，他のすべてのプロセスたちにも mmap() 関数を呼び出してもらわなければならない．このプログラミングインタフェースは非常に不便である．このように，プロセス間共有メモリは，いずれかのプロセスによって発行されたメタ操作が他のプロセスたちには自動的には反映されないという点で，セマンティクス I を満たさない．

さらに，プロセス間共有メモリでは，特別な処理を行わないかぎり，すべてのプロセスが mmap() 関数を発行したとき，プロセス間共有メモリ m が各プロセスのアドレス空間において同一のアドレスにメモリマップされる保証がないという問題もある[*4] ．そして，プロセス間共有メモリ m がメモリマッ

[*2] ここでいう「すべての half-process」とは，「OS 内のすべての half-process」という意味ではなく，「アドレスをお互いに共有することを望んでいる half-process の集合に属するすべての half-process」という意味である．

[*3] 同様の議論は，System V のプロセス間共有メモリに対しても成り立つ．

[*4] なお，ここでいう特別な処理とは，たとえば，メモリマップする前に，各プロセスのアドレス空間上でどのアドレスが空いているかを調べて交渉し，すべてのプロセスにおいて空いていることが保証されているアドレスに対してメモリマップす

プされているアドレスがすべてのプロセスを通じて一致していない場合には，プログラマは，プロセス間共有メモリ m 上のデータへの参照を，ポインタ（＝アドレス）によって管理することができなくなり，かわりに，プロセス間共有メモリ m の先頭アドレスからのオフセットを使って管理しなければならなくなる．これは，グラフ構造などの複雑なポインタ構造を必要とするアプリケーションに対するプログラマビリティを著しく下げてしまう．このように，プロセス間共有メモリはセマンティクスⅡも満たさない．

ただし，プロセス間共有メモリを使ってセマンティクスⅡを満たす簡単な方法は存在する．十分大きなサイズのプロセス間共有メモリ m を作成し，MAP_SHARED オプション付きの mmap() 関数を使ってプロセス間共有メモリ m をメモリマップしたあとで，プロセスたちを fork() 関数によって生成すればよい [86]．fork() 関数によって生成されたプロセスたちは，親プロセスのメモリマップをそのまま引き継ぐため，この方法を使えば，すべてのプロセスにおいてプロセス間共有メモリ m が同一のアドレスにメモリマップされることを保証できる．つまり，セマンティクスⅡは満たされる．しかし，セマンティクスⅠは依然として満たされないため，生成されたプロセスたちが mmap() 関数や munmap() 関数を発行したとしても，それが他のプロセスたちに反映されることはない．したがって，この方法では，初期的に生成した固定的なサイズのプロセス間共有メモリ m が最初から最後まで存在し続けるような処理しか自然には記述できず，プログラミングの柔軟性に欠ける．

要約すると，プロセス間共有メモリのプログラミングインタフェースは，セマンティクスⅠとセマンティクスⅡを満たしておらず，**動的に確保/解放したり拡張/縮小したりするのが非常に不便**である．つまり，プロセス間のデータ共有の手段としては，プロセス間共有メモリは，half-process の共有アドレス空間よりもプログラマビリティが低い．

9.2.2 ダイレクトメモリアクセス

異なるプロセス間のダイレクトメモリアクセスに関しては，Nemesis[32, 30]，Limic2[94]，研究 [70, 108] など，MPI のノード内通信を高速化するための技術として多くの研究が行われてきた．もっとも代表的な方法は，異なるアドレス空間上でデータをコピーするためのシステムコールを新たに実装する方法である [70, 108, 94]．具体的には，(1) コピー元となるアドレス空間のアドレス範囲 s に物理ページを割り当てたあと（pinning と呼ぶ），(2) そのアドレス範囲 s をカーネルアドレス空間にメモリマップし，(3) カーネルアドレス空間から，コピー先となるアドレス空間のアドレス範囲 d に対してデータをコピーする．**9.5.6** 節で述べるように，half-process でもこれと同様の方法を用いており，ダイレクトメモリアクセスの実装方法に関してはとくに新規性はない．なお，(1) における pinning と (2) におけるデータのコピーをオーバラップさせたり [70]，データのコピーを CPU ではなく I/OAT DMA エンジンに担当させて CPU のオフロードを図ったりすることにより [30, 70, 108]，さらなる性能最適化を図る研究も行われている．

これらの研究は，いずれも新たなシステムコールを実装することで，異なるアドレス空間のダイレクトメモリアクセスをカーネルレベルで実現する手法である．これに対して，SMARTMAP[29, 28, 156]

るなどの処理のことをいう．

では，他のプロセスの仮想アドレスを簡単なオフセット計算で求められるようにしたうえで，他のプロセスのアドレス空間にユーザレベルから直接アクセスできるようにすることで，ユーザレベルでの高速なダイレクトメモリアクセスを実現している．しかし，SMARTMAP は，ディマンドページングを行うことなく仮想アドレスを物理ページにリニアマップする，Catamount という特殊な軽量カーネルのうえでのみ動作するシステムであり，SMARTMAP の実装を，Linux カーネルにおける実装に応用させることはできない．

9.3 half-process の応用可能性

half-process の設計と実装の説明に入る前に，half-process の潜在的な応用可能性について考察する．half-process は **9.6** 節で述べるように真に透過的なスレッド移動に応用できるだけでなく，プロセス間のデータ共有を簡単かつ高速に実現するための汎用的なカーネルプリミティブとして，以下のようなさまざまな応用可能性を持っている．

9.3.1 マルチスレッドプログラミングにおけるスレッドアンセーフなライブラリの使用

第 1 の応用可能性は，マルチスレッドプログラミングにおいて，スレッドアンセーフなライブラリの使用を許可できることである．スレッドアンセーフなライブラリは多く，また，仮にスレッドセーフであったとしても，粒度が粗すぎるロックが使われていたり，キャッシュのフォルスシェアリングが発生したりすることが原因で，マルチスレッドから利用すると意図しない性能劣化が起きるようなライブラリも存在する．当然，通常ならば，これらのライブラリをマルチスレッドプログラミングで利用することはできない．ところが，half-process を利用すれば，ライブラリだけは非共有アドレス空間で動作させつつも，ライブラリ以外の部分は共有アドレス空間で動作させるようなことが可能になる．たとえば，スレッドアンセーフなライブラリを利用しつつ，並列グラフ探索を記述することを考える．並列グラフ探索のように複雑なポインタ計算が必要となるアルゴリズムでは，共有されたアドレス空間を使いたい場合が多い．このような場合，half-process を利用することで，ライブラリだけは非共有アドレス空間で動作させつつも，グラフを表現するためのデータ構造は共有アドレス空間に割り当てることができる．

9.3.2 より柔軟なハイブリッドプログラミング

第 2 の応用可能性は，MPI+OpenMP あるいは MPI+pthread よりも柔軟なハイブリッドプログラミングである．通常ならば，複数のインスタンス間でアドレス空間を共有するためには，インスタンスとしてはスレッドを使うしかない．そのため，既存のハイブリッドプログラミングの処理系 [153, 178, 78, 24] では，1 個の MPI プロセス内のインスタンスはスレッドとして実装されているが，この場合には，インスタンス間でアドレス空間が完全に共有されてしまう．よって，このようなハイブリッドプログラミングの処理系では，たとえば，プログラマがスレッドアンセーフなライブラリを使うことは許されない．ところが，そもそも，ハイブリッドプログラミングの目的は，インスタンス間で性能上共有するべきデータを共有することであって，決してすべてのデータを共有することが目的なわけ

ではない．以上の視点から考えると，各インスタンスをスレッドではなく half-process として実装することで，インスタンス間で共有するべきデータと共有するべきではないデータをプログラマが自由に選択できるような，より柔軟なハイブリッドプログラミングを実現できると考えられる．

9.3.3 並列分散プログラミング処理系の開発者の負担減

第 3 の応用可能性は，並列分散プログラミング処理系の開発者の負担減である．以下では，MPI におけるノード内通信の実装と DMI のノード内通信の実装を例にして議論するが，この議論はノード内通信を高性能化しようとしている並列分散プログラミング処理系一般に対して成り立つ議論である．

9.3.3.1 MPI のノード内通信の実装に対する応用

第 1 に，MPI におけるノード内通信の実装について考える．さしあたりハイブリッドプログラミングを無視するならば，フラットな MPI の各インスタンスはプロセスとして実装される．プロセスとして実装される理由はいくつかあるが，その理由の 1 つは，各インスタンスのアドレス空間を独立させることによって，プログラマから見たときの，グローバル変数のとり扱いに関するセマンティクスをわかりやすくするためである．仮に各インスタンスをスレッドとして実装するとすると，グローバル変数が 2 個のスレッド間で共有されるかどうかは，その 2 個のスレッドが同一のプロセスに存在するかどうかに依存する．ところが，フラットな MPI では，プログラマから見て，ある 2 個のスレッドが同一のプロセスに存在するかどうかを自然に知る手段は存在しないため，結果的に，グローバル変数が共有されるかどうかに関するセマンティクスがわかりにくくなってしまう．以上のような理由などにより，フラットな MPI の各インスタンスはスレッドではなくプロセスとして実装されるのが通常である．したがって，これまで MPI の開発者たちは，プロセス間共有メモリや異なるプロセス間のダイレクトメモリアクセス [29, 28, 32, 30, 70, 93, 108, 94] などを用いて，MPI のノード内プロセス間通信を高性能化させるための技術を積極的に研究してきた [31]．ところが，**9.2** 節で述べたように，プロセス間共有メモリには，動的に確保/解放したり，そのサイズを拡張/縮小したりするのが難しいという欠点がある．そこで，MPI のノード内プロセス間通信をプロセス間共有メモリによって実現する既存手法 [108, 39] では，以下のような方法がとられてきた：

(1) 初期的に，各プロセスのペアごとに，（ある程度小さな）固定サイズのプロセス間共有メモリを確保する．つまり，n 個のプロセスが存在するならば，${}_nC_2$ 個のプロセス間共有メモリを確保する．プロセス i とプロセス j のペアのためのプロセス間共有メモリを $m_{i,j}$ と表す．

(2) プロセス i がプロセス j に対してデータを送信するとき，送信するデータサイズ s が $m_{i,j}$ のサイズ以下ならば，$m_{i,j}$ を使う．具体的には，プロセス i がプロセス i 上のデータをいったん $m_{i,j}$ にコピーしたあと，プロセス j が $m_{i,j}$ からデータをコピーする．

(3) 送信するデータサイズ s が $m_{i,j}$ のサイズより大きいならば，いくつかの方法がある．第 1 の方法は，データを $m_{i,j}$ のサイズごとのチャンクに区切ってチャンクごとにパイプライン化して送信する方法である．第 2 の方法は，プロセス間のダイレクトメモリアクセスを利用して，プロセス j がプロセス i のデータを直接 read する方法である．第 3 の方法は，(1) プロセス i が s バイトの新しいプロセス間共有メモリ m' を確保し，(2) プロセス i がプロセス i 上のデータを m'

にコピーし，(3) プロセス i は $m_{i,j}$ を通じて m' の先頭アドレスをプロセス j に教え，(4) プロセス j が m' からデータをコピーする，という方法である．

このように，プロセス間共有メモリを利用する場合，余分なメモリコピーが必要になったり，開発者にとって複雑なプログラミングが必要になったりする．さらに，集合通信におけるノード内通信を最適化しようとするならば，より複雑なプログラミングが要求される [117].

これに対して，仮にプロセス間で half-process の共有アドレス空間を利用できるならば，開発者は，これらのノード内通信を，マルチスレッドプログラミングの要領でより簡単に記述できるようになる．このように，half-process を利用することで，プログラマに対してはフラットな MPI としての独立したアドレス空間（プロセス）を見せつつも，開発者側では共有アドレス空間上でのデータ共有を利用することができ，オーバヘッドが小さくかつプログラマビリティの高いノード内通信を実現することができる．要約すると，half-process を利用することで，**プログラマの視点ではプロセスとして見えていつつも，開発者の視点ではスレッドとして見えているようなインスタンス**を実現できる．

9.3.3.2 DMI のノード内通信の実装に対する応用

第 2 に，DMI の実装について考える．ユーザプログラムから指示された send/receive 操作をほぼそのまま実現すればよいだけの MPI と比較すると，DMI では，キャッシュコヒーレントなグローバルアドレス空間の管理などの高度な仕組みが必要であり，スレッド間でのより複雑なデータ共有が必要とされる．

DMI のスレッド構成を図 4.1 に示す．第 **4** 章で説明したように，DMI では，第 1 に，receiver スレッドが受信したメッセージを FIFO なキューを介して handler スレッドに渡す必要がある．第 2 に，受信したメッセージが read フォルトに対する応答であった場合，handler スレッドは，その応答を処理するときに，受信したデータを各スレッドのローカルアドレス空間に書き込む必要がある．第 3 に，ページテーブルなどのデータ構造はスレッド間で共有されており，すべてのスレッドから適切な排他制御のもとでアクセスできる必要がある．第 4 に，メモリプールは同一プロセス内のすべてのスレッドによって共有されており，あるスレッドがメモリプールにキャッシュしたページは，他のスレッドによってもアクセスできる必要がある．以上のように，DMI におけるスレッド間のデータ共有は非常に複雑であり，アドレス空間が共有されているからこそ記述できているといっても過言ではない．これと同等の仕組みをプロセス間共有メモリで実現するのは非常に困難である．

このような場合でも，half-process を利用すれば，プログラマに対しては独立したアドレス空間（プロセス）を見せつつも，DMI の内部では共有アドレス空間上でのデータ共有を実現することができる．詳しくは **9.6** 節で述べるが，真に透過的なスレッド移動ではこの仕組みを利用する．

9.4 half-process の設計

half-process の設計を図 9.1 に示す．各 half-process のアドレス空間全体は，**非共有アドレス空間**と**共有アドレス空間**から構成されている．非共有アドレス空間は各 half-process にとって固有のアドレス空間であり，そのセマンティクスは通常のプロセスのアドレス空間と等価である．一方で，共有ア

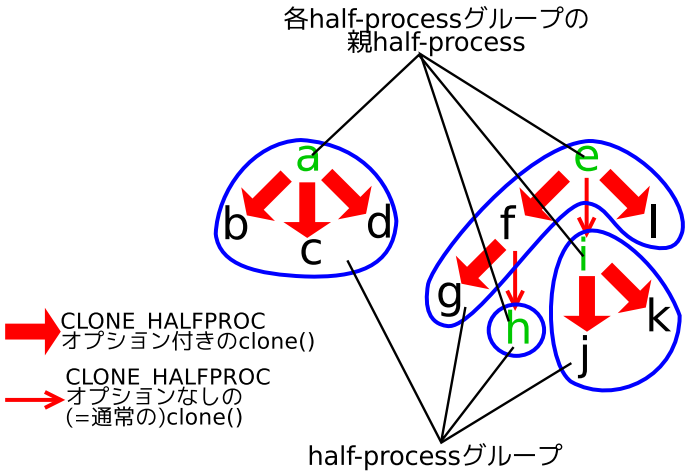

図 9.2 half-process グループの例.

ドレス空間はすべての half-process によって共有されているアドレス空間であり，そのセマンティクスは通常のスレッド間で共有されているアドレス空間と等価で，セマンティクス I とセマンティクス II を満たす．当然，セキュリティ上の理由などから，共有アドレス空間は，OS 内に存在するすべての half-process 間で共有されるわけではなく，ある決まった half-process たちの間でのみ共有される．ここで，ある共有アドレス空間を共有している half-process の集合を，half-process グループと呼ぶことにする．たとえば，図 9.1 の場合には，half-process 1，half-process 2，half-process 3 が half-process グループを構成している．

とくに，ある half-process グループが 1 個の half-process だけから構成されているとき，その half-process は通常のプロセスと等価である．後述するような half-process 特有のシステムコールやオプションを使わないかぎり，half-process が mmap() 関数によって確保するメモリ領域や half-process のすべての静的変数は，すべて非共有アドレス空間に割り当てられる．つまり，デフォルトでは，half-process は非共有アドレス空間しか使用せず，通常のプロセスとして振る舞う．したがって，half-process の設計において鍵となるのは，(1) 共有アドレス空間上にメモリ領域を確保するためにはどうすればよいか，(2) half-process グループはどのように定義できるのか，の 2 点である．

第 1 に，共有アドレス空間上にメモリ領域を確保する方法であるが，非常に簡単で，**mmap() 関数に MAP_HALFPROC オプションを渡すだけ**でよい．この mmap() 関数によって確保されたメモリ領域は，half-process グループに属するすべての half-process によって共有される．

第 2 に，half-process グループを定義する方法であるが，clone() 関数のオプションによって定義する．まず，Linux では，clone() 関数に CLONE_VM オプションを渡すことで，子プロセスと親プロセスとでアドレス空間を共有させることができ，スレッドを生成することができる．また，clone() 関数に CLONE_VM オプションを渡さなければ，子プロセスは親プロセスとは別の新しいアドレス空間を使うようになり，プロセスを生成することができる．これと同様にして，**clone() 関数に**

CLONE_HALFPROC オプションを渡すことで，half-process を生成することができる．正確には，以下のルールによって half-process グループを定義できる：

- CLONE_VM オプション付きの clone() 関数によって生成された half-process は，新しい half-process グループを生成し，その half-process グループの要素となる．（よって，この half-process は通常のプロセスとなる．）
- CLONE_HALFPROC オプション付きで生成された half-process は，その親 half-process が属する half-process グループの要素となる．

さまざまな half-process グループの例を図 9.2 に示す．

さらに，half-process 間でより密なデータ共有を行いやすくするために，同一の half-process グループに属する half-process 間で，**お互いの非共有アドレス空間をダイレクトメモリアクセス**するための手段も導入する．具体的には，dmread(pid_t *pid*, void **ptr*, size_t *size*, void **buf*) 関数によって，プロセス ID が *pid* のプロセスのアドレス領域 [*ptr*, *ptr*+*size*) を，この dmread() 関数を呼び出したプロセスのアドレス領域 [*buf*, *buf*+*size*) にコピーすることができる．同様に，dmwrite(pid_t *pid*, void **ptr*, size_t *size*, void **buf*) 関数によって，この dmwrite() 関数を呼び出したプロセスのアドレス領域 [*buf*, *buf*+*size*) を，プロセス ID が *pid* のプロセスのアドレス領域 [*ptr*, *ptr*+*size*) にコピーすることができる．なお，セキュリティ上の理由から，dmread() 関数/dmwrite() 関数によるダイレクトメモリアクセスは，同一の half-process グループに属する half-process 間のみに制限される．

要約すると，half-process では，共有アドレス空間を利用するか，または非共有アドレス空間のダイレクトメモリアクセスを利用することで，さまざまなノード内通信を簡単に記述することができる．

9.5 half-process のカーネルレベル実装

9.5.1 カーネルレベルで実装する理由

具体的な実装の説明に入る前に，half-process をユーザレベルではなくカーネルレベルで実装する理由について述べる．

第 1 に，dmread() 関数/dmwrite() 関数に関しては，カーネルレベルの実装が必要である．なぜなら，Linux カーネルでは，異なるプロセスのアドレス空間にユーザレベルから直接アクセスすることは許されていないためである．

第 2 に，half-process 間の共有アドレス空間に関しては，以下のようにプロセス間共有メモリを利用することで，ユーザレベルで実装することも可能である：

(1) 共有アドレス空間を表現するために，shm_open() 関数によって十分なサイズのプロセス間共有メモリを確保する．
(2) 各 half-process で発行される mmap() 関数/munmap() 関数/mprotect() 関数などのメタ操作をすべてフックし，それが共有アドレス空間を対象としたメタ操作であれば，すべての half-process に対してそのメタ操作を発行するように指示する．これにより，セマンティクス I を満たすこと

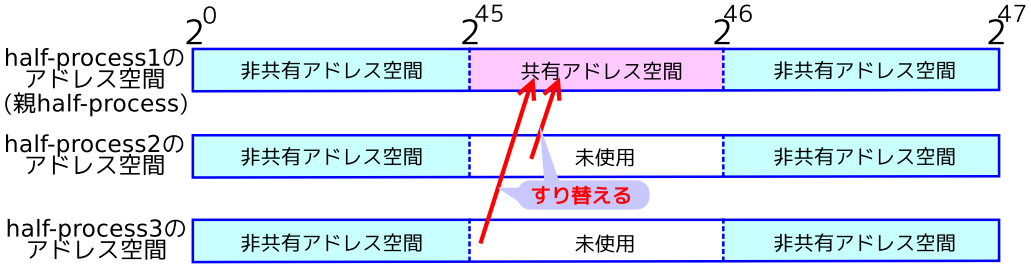

図 9.3 half-process のアドレス空間全体の構成.

ができる.

(3) とくに mmap() 関数に関しては，すべての half-process 上で発行される mmap() 関数が同一のアドレスにメモリ領域を確保するよう，確保するアドレスをうまく調整する．これにより，セマンティクス II を満たすことができる.

しかし，このユーザレベルの実装は，各メタ操作のたびに，half-process の個数に比例したオーバヘッドをともなうという問題がある．なぜなら，各メタ操作のたびに，すべての half-process にメタ操作を発行させるための同期的な通信が必要になるからである．このオーバヘッドは half-process の数が大規模になるほど顕著になるうえ，メタ操作は多くのアプリケーションにおいて頻繁に用いられるプリミティブな操作であるため，メタ操作のオーバヘッドがアプリケーション全体の性能に及ぼす影響は少なくないと考えられる．よって，本研究では，次節で述べるカーネルレベルの実装を採用する.

9.5.2 基本アイディア

本節では，Linux カーネル 2.6 と x86_64 アーキテクチャに基づく half-process のカーネルレベル実装について述べる.

第 1 に，アドレス空間全体を 2 種類のアドレス範囲に分割し，非共有アドレス空間として使用するアドレス範囲と，共有アドレス空間として使用するアドレス範囲を決定する．具体的には，x86_64 アーキテクチャでは，ユーザアドレス空間として使用できるアドレス範囲は $[0, 2^{47})$ なので，アドレス範囲 $[2^{45}, 2^{46})$ を共有アドレス空間として使用し，残りのアドレス範囲 $[0, 2^{45})$ と $[2^{46}, 2^{47})$ を非共有アドレス空間として使用することにする（図 9.3）．なお，ここでは，共有アドレス空間と非共有アドレス空間の両方に対して十分なサイズのアドレス範囲が割り当てられればよく，2^{45} や 2^{46} などの数字自体に明確な根拠があるわけではない．また，単純にアドレス空間全体を前半と後半に 2 分割するのではなく，非共有アドレス空間のアドレス範囲を「飛び地」にしている理由は，先頭のアドレス範囲 $[0, \ldots)$ と末尾のアドレス範囲 $[\ldots, 2^{47})$ は，コード領域やスタック領域などの各 half-process 固有の特別なメモリ領域として使われるため，これらのアドレス範囲を非共有アドレス空間に含めておく必要があるためである.

第 2 に，各 half-process p は，非共有アドレス空間としてはその half-process p 自身の非共有アドレス空間を使用し，共有アドレス空間としてはその half-process p の親 half-process の共有アドレス空間を

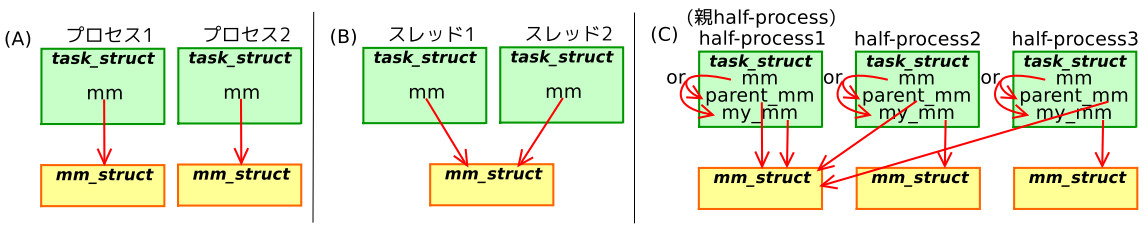

図 9.4 task_struct 構造体と mm_struct 構造体の関係．（A）プロセスの場合，（B）スレッドの場合，（C）half-process の場合．

使用するようにする（図 9.3）．ここで，half-process p の親 half-process とは，half-process p が属する half-process グループのなかでもっとも最初にその half-process グループの要素となった half-process のことを意味する．いい換えると，各 half-process グループに一番最初から存在していた half-process が，その half-process グループの親 half-process である．すなわち，各 half-process グループにはちょうど 1 個の親 half-process が存在している．図 9.2 に親 half-process の例を示す．なお，図 9.3 に示すように，各 half-process p は共有アドレス空間として親 half-process の共有アドレス空間を使うことになるので，親 half-process ではない half-process の共有アドレス空間は使用されないことになる．

以上で述べた内容を実装すれば，各 half-process は，非共有アドレス空間としては各 half-process 固有の非共有アドレス空間を使用し，共有アドレス空間としては，half-process グループにつき 1 個しか存在しない親 half-process の共有アドレス空間を使用することになるため，half-process の目標は達成できる．ここでの実装上の課題は，各 half-process の共有アドレス空間に対して発行されるメモリ操作（メタ操作および通常の read/write アクセス）を，親 half-process の共有アドレス空間に対するメモリ操作にすり替えるにはどうすればよいかである．そこで本研究では，メタ操作をすり替えるための新しい実装手法として**アドレス空間スイッチング**を，通常の read/write アクセスをすり替えるための新しい実装手法として**ページテーブルリダイレクション**を提案する．詳しくは次節で述べるが，アドレス空間スイッチングはセマンティクス I を満たすための手法であり，ページテーブルリダイレクションはセマンティクス II を満たすための手法である．なお，これらのすり替えの対象は half-process の共有アドレス空間に対するメモリ操作だけであって，非共有アドレス空間に対するメモリ操作には何の介入も行わないという点は重要である．したがって，half-process の実装のために施すカーネルの改造は，half-process 特有の機能を使わない通常のプロセスには何の悪影響も及ぼさない[*5] ．

9.5.3 アドレス空間スイッチング

アドレス空間スイッチングは，各 half-process の共有アドレス空間に対するメタ操作を，親 half-process のアドレス空間に対するメタ操作にすり替えるための手法である．

Linux カーネルでは，プロセスとスレッドは task_struct 構造体として表現され，それらのアドレス空間は mm_struct 構造体として表現される [201, 152]．各 task_struct 構造体は mm_struct 構

[*5] ただし，通常のプロセスが利用可能なアドレス範囲は，$[0, 2^{45})$ と $[2^{46}, 2^{47})$ に限定される．

```
some_meta_operation(uint64_t addr, ...) :
  struct task_struct *current := the task_struct running now
  if addr is in a shared address space then
    current→mm := current→parent_mm
    flush a TLB
    load the page table of current→mm to a CPU
  endif
  ... /* the unmodified kernel code of this meta operation */
  if addr is in the shared address space then
    current→mm := current→my_mm
    flush the TLB
    load the page table of current→mm to the CPU
  endif
```

図 9.5 アドレス空間スイッチングのアルゴリズム.

造体型の *mm* メンバを持っており，この *mm* メンバが，その task_struct 構造体が表すプロセスまたはスレッドのアドレス空間を表している．たとえば，プロセスの場合には，図 9.4（A）のように，各 task_struct 構造体の *mm* メンバが異なる mm_struct 構造体を指すことによってプロセスごとに異なるアドレス空間を表現している．また，スレッドの場合には，図 9.4（B）のように，すべての task_struct 構造体の *mm* メンバが同一の mm_struct 構造体を指すことによってスレッド間で共有されたアドレス空間を表現している．また，mm_struct 構造体は，それが表すアドレス空間に対するメタ操作の結果を管理しており，たとえば，どのアドレス範囲がどのような保護属性でマップされているかなどの情報を管理している．

half-process の場合には，デフォルトでは通常のプロセスとして振る舞う必要があるため，各 task_struct 構造体ごとに mm_struct 構造体が存在する．そして，いまの目標は，共有アドレス空間に対するメタ操作が起きた場合にのみ，メタ操作の対象を親 half-process の mm_struct 構造体にすり替えることである．ここでポイントとなるのは，あるメタ操作が発行されたとき，そのメタ操作が適用される対象となる mm_struct 構造体は，その時点でそのメタ操作を発行している task_struct 構造体の *mm* メンバで決定されるという点である．したがって，メタ操作が発行された時点で，task_struct 構造体の *mm* メンバの値を別の mm_struct 構造体にすり替えておけば，そのメタ操作が適用される対象となる mm_struct 構造体を自由に変更することができる．

これを実現するために，第 1 に，図 9.4（C）に示すように，half-process を表現する task_struct 構造体に対して，*my_mm* と *parent_mm* の 2 つのメンバを新たに追加する．ここで，*my_mm* メンバは「その half-process 固有の mm_struct 構造体」を意味し，*parent_mm* メンバは「親 half-process の mm_struct 構造体」を意味し，*mm* メンバは「その時点で発行されているメタ操作の適用対象となる mm_struct 構造体」を意味するように管理する．よって，これらのメンバの初期値は以下のように定める：

- CLONE_HALFPROC オプションを付けない clone() 関数によって生成された half-process では，*my_mm* メンバと *parent_mm* メンバはともに，その half-process の mm_struct 構造体を

指すように初期化される．

- CLONE_HALFPROC オプション付きの clone() 関数によって生成された half-process では，*my_mm* メンバは，その half-process の mm_struct 構造体を指すように初期化される．*parent_mm* メンバは，この clone() 関数を呼び出した half-process の *parent_mm* メンバの値に初期化される．

上記のように各メンバを初期化することで，図 9.2 に示した half-process グループの関係を表現できていることに注意する．

第 2 に，各メタ操作（mmap() 関数，munmap() 関数，mprotect() 関数，msync() 関数，mbind() 関数，ページフォルトを処理する関数など）のカーネルコードを，図 9.5 に示すように修正する：

(1) メタ操作が共有アドレス空間に対するものであるならば，*mm* メンバの値を *parent_mm* メンバの値へと更新する．これにより，メタ操作の対象を親 half-process の共有アドレス空間にすり替えることができる．
(2) TLB をフラッシュしたあと，親 half-process のページテーブルを CPU にロードする．
(3) メタ操作の本体を実行する．
(4) *mm* メンバの値を *my_mm* メンバの値へと戻す．
(5) TLB をフラッシュしたあと，この half-process 自身のページテーブルを CPU にロードする．

この修正においては，メタ操作の本体のカーネルコードを修正する必要はない．このように，**メタ操作の前後で *mm* メンバの値を単にすり替えるだけで，カーネルは，あたかもメタ操作の対象が親 half-process に切り替わったかのように思い込んでメタ操作を実行してくれる**ため，目的を達成することができる．また，アドレス空間スイッチングにおいて必要となるオーバヘッドは変数やレジスタの read/write 数回と TLB のフラッシュ 2 回だけであって，このオーバヘッドは half-process の個数に依存しない．

9.5.4 ページテーブルリダイレクション

ページテーブルリダイレクションは，ページテーブルエントリをリダイレクトすることによって，各 half-process の共有アドレス空間に対する read/write アクセスを，親 half-process の共有アドレス空間に対する read/write アクセスにすり替える手法である．

一般に，x86_64 アーキテクチャでは，4 レベルのページテーブルによって 48 ビットのアドレス空間を表現している [201, 152]．このうち前半 47 ビットがユーザアドレス空間として使用され，後半 47 ビットがカーネルアドレス空間として使用される．各レベルのページテーブルは 512 エントリを保持している．たとえば，第 4 レベル（最上位レベル）ページテーブルの各エントリは，$2^{48}/512 = 2^{39}$ 個のアドレスを管理しており，第 4 レベル（最上位レベル）ページテーブルの第 i エントリ（$0 \leq i < 512$）は，アドレス範囲 $[2^{39} \times i, 2^{39} \times (i+1))$ を管理している．また，Linux カーネルでは，各 mm_struct 構造体に *pgd* メンバが存在していて，この *pgd* メンバが，その mm_struct 構造体が表すアドレス空間に対応する第 4 レベルページテーブルの先頭アドレスを指している．そして，Linux カーネルは，

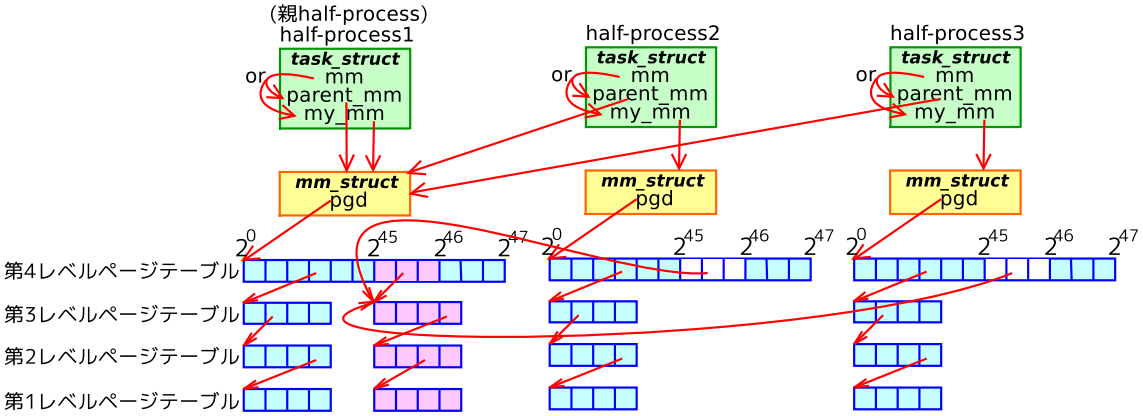

図 9.6　ページテーブルリダイレクションの仕組み.

task_struct 構造体 t を task_struct 構造体 t' にコンテキストスイッチするたびに，$t{\rightarrow}mm{\neq}t'{\rightarrow}mm$ ならば，TLB をフラッシュしたあと，$t'{\rightarrow}mm{\rightarrow}pgd$ の値を CPU の %cr3 レジスタに読み込むことによって，CPU が使用するページテーブルを task_struct 構造体 t' のページテーブルに切り替える．

いまの目標は，図 9.6 に示すように，**各 half-process のアドレス範囲 $[2^{45}, 2^{46})$ に対応するページテーブルエントリを，それに対応する親 half-process のページテーブルエントリにすり替える**ことである．これにより，各 half-process の共有アドレス空間に対するすべての read/write アクセスを，自動的に，親 half-process の共有アドレス空間に対する read/write アクセスにすり替えることができる．しかし，Linux においては，ページテーブルの構造はページフォルトを契機として demand-driven に構築されていくため，ページテーブルエントリをどのようにすり替えればよいかは自明ではない．いま，ある half-process が，共有アドレス空間上のアドレス a でページフォルトを起こしたとし，そのアドレス a に対応する第 4 レベルページテーブルエントリ，第 3 レベルページテーブルエントリ，第 2 レベルページテーブルエントリ，第 1 レベルページテーブルエントリが，それぞれ，i_4，i_3，i_2，i_1 であるとする．また，その half-process の task_struct 構造体を t，親 half-process の task_struct 構造体を t' とおく．このとき，以下の手順でページテーブルエントリのすり替えを実現する：

(1) $t{\rightarrow}mm{\rightarrow}pgd$ を検査することで，第 4 レベルページテーブルエントリが存在するかどうかを調べる．存在しない場合，新たに第 4 ページテーブルエントリを割り当てて，その先頭アドレスを $t{\rightarrow}mm{\rightarrow}pgd$ に書き込む．

(2) $t{\rightarrow}mm{\rightarrow}pgd[i_4]$ を検査することで，第 3 レベルページテーブルエントリが存在するかどうかを調べる．存在しない場合，新たに第 3 ページテーブルエントリを割り当てて，その先頭アドレスを $t'{\rightarrow}mm{\rightarrow}pgd[i_4]$ に書き込んだうえで，$t{\rightarrow}mm{\rightarrow}pgd[i_4]$ に $t'{\rightarrow}mm{\rightarrow}pgd[i_4]$ を書き込む．その結果，この時点で，かならず，$t{\rightarrow}mm{\rightarrow}pgd[i_4]{==}t'{\rightarrow}mm{\rightarrow}pgd[i_4]$ であることが保証される．

(3) $t'{\rightarrow}mm{\rightarrow}pgd[i_4][i_3]$ を検査することで，第 2 レベルページテーブルエントリが存在するかどう

かを調べる．存在しない場合，新たに第 2 ページテーブルエントリを割り当てて，その先頭アドレスを $t'\rightarrow mm\rightarrow pgd[i_4][i_3]$ に書き込む．

(4) $t'\rightarrow mm\rightarrow pgd[i_4][i_3][i_2]$ を検査することで，第 1 レベルページテーブルエントリが存在するかどうかを調べる．存在しない場合，新たに第 1 ページテーブルエントリを割り当てて，その先頭アドレスを $t'\rightarrow mm\rightarrow pgd[i_4][i_3][i_2]$ に書き込む．

以上の手順によって，ページフォルトにともなってページテーブルエントリを demand-driven にすり替えることができる．なお，このページテーブルリダイレクションにおいては TLB のフラッシュは必要ない．なぜなら，値を書き換えられようとしているページテーブルエントリは，この書き換え以前には存在しなかったものなので，そのページテーブルエントリの内容が TLB にキャッシュされている可能性がないからである．よって，ページテーブルリダイレクションにおいて必要となるオーバヘッドは変数の read/write 数回だけであって，このオーバヘッドは half-process の個数に依存しない．

9.5.5 コピーオンライトの高速化

9.5.2 節で述べたように，親 half-process ではない half-process の共有アドレス空間は使用されない．よって，clone() 関数が発行されたとき，共有アドレス空間に対してはコピーオンライトを仕掛ける必要がないため，clone() 関数のオーバヘッドを削減することができる．

9.5.6 ダイレクトメモリアクセス

dmread(pid_t *pid*, void **ptr*, size_t *size*, void **buf*) 関数は以下のように実装することができる [70, 108, 94]：

(1) プロセス ID が *pid* の half-process のアドレス範囲 [*ptr*, *ptr*+*size*) に物理ページを割り当てる．

(2) kmap() 関数によって，プロセス ID が *pid* の half-process のアドレス範囲 [*ptr*, *ptr*+*size*) をカーネルアドレス空間にマップする．

(3) copy_to_user() 関数によって，カーネルアドレス空間から，この dmread() 関数を呼び出しているプロセスのアドレス範囲 [*buf*, *buf*+*size*) にデータをコピーする．

dmwrite() 関数も同様に実現できる．

9.5.7 ユーザレベルのライブラリの実装

以上の実装によって，half-process のカーネルレベル実装は完了するが，half-process のカーネルプリミティブをより便利に利用できるようにするために，いくつかのユーザレベルのライブラリ関数を実装する．

第 1 に，half-process では，glibc の pthread ライブラリを使用できない．pthread_create() 関数/pthread_join() 関数などのスレッド操作だけでなく，pthread_mutex_xxxx() 関数や pthread_cond_xxxx() 関数などの同期操作も使用できない．これは，glibc の pthread ライブラリは，そもそもスレッドを扱うためのライブラリであって，グローバル変数やスタック変数がスレッド間で共有されていることを前提とした実装になっているためである．これに対して，half-process では，グローバル変数やスタック変数が非共有アドレス空間に配置されることになっているため，pthread ラ

イブラリの実装を利用することはできない．そこで，half-process 向けに，グローバル変数やスタック変数を使用しないような実装によって，各 pthread_xxxx() 関数に対応する halfproc_xxxx() 関数をユーザレベルで実装して提供する．たとえば，halfproc_mutex_xxxx() 関数や halfproc_cond_xxxx() 関数は futex() システムコールを使って実装し，halfproc_create() 関数は CLONE_HALFPROC オプション付きの clone() 関数を使って実装する．

第 2 に，half-process の共有アドレス空間は MAP_HALFPROC オプション付きの mmap() 関数で確保できるが，malloc/free/realloc 操作を使えないのは不便である．そこで，half-process の共有アドレス空間向けの malloc/free/realloc 操作として，halfproc_malloc() 関数/halfproc_free() 関数/halfproc_realloc() 関数を実装して提供する．

第 3 に，スレッド移動を行うためには，移動元で非共有アドレス空間に含まれるデータすべてをチェックポイントし，移動先でリスタートさせる必要がある．そのため，非共有アドレス空間をまるごとチェックポイント/リスタートするためのライブラリ関数を実装して提供する．

9.6 真に透過的なスレッド移動への応用

本節では，half-process を利用して，真に透過的なスレッド移動を実現する方法を述べる．実現すべきことは，**9.3.3.2** 節で述べたように，プログラマの視点では DMI のインスタンスがプロセスに見えていつつも，開発者の視点では DMI のインスタンスがスレッドとして見えるようにすることである．

9.6.1 設計

第 1 に，DMI におけるスレッドを half-process に置き換える．これにより，DMI の half-process の静的変数や，DMI の half-process が mmap() 関数/munmap() 関数/mremap() 関数によって確保/解放するアドレス領域は，その half-process の非共有アドレス空間に割り当てられることになる．よって，ユーザプログラムは独立したアドレス空間上で実行されることになるため，thread-move のようなプログラミング制約を設ける必要がなくなる．

第 2 に，ページテーブルやメモリプール上のページの実体など，half-process 間で共有する必要があるデータについては，half-process の共有アドレス空間上に割り当てる．これにより，開発者は，DMI の各インスタンスをスレッドで実装していた場合とほぼ同様の方法で，half-process 間のデータ共有を記述することができる．

第 3 に，half-process の移動時には，移動元から移動先へ，その half-process の非共有アドレス空間をまるごと移動させるだけでよい．このとき，非共有アドレス空間は各 half-process ごとに独立したアドレス空間であるため，thread-move とは異なり，移動先においてアドレス衝突が引き起こされることはない．

9.6.2 実装

前節で述べた設計に基づいて，DMI のスレッドを half-process に置き換える場合に必要な修正は，以下で述べる 4 ヶ所のみである．

第 1 の修正は，pthread_xxxx() 関数を，対応する halfproc_xxxx() 関数に置き換えることであ

る．第 2 の修正は，共有アドレス空間を利用したい malloc() 関数/free() 関数/realloc() 関数を，halfproc_malloc() 関数/halfproc_free() 関数/halfproc_realloc() 関数に置き換えることである．一部の例外をのぞいては，DMI の処理系が使用するメモリ領域はすべて共有アドレス空間に確保すればよいので，単に，すべての malloc() 関数/free() 関数/realloc() 関数を置き換えるだけでよい．第 3 の修正は，スレッドのチェックポイント/リスタートのコードを，half-process のチェックポイント/リスタートのコードに置き換えることである．第 4 の修正は，異なる half-process の非共有アドレス空間に対する read/write やデータコピーを，dmread() 関数/dmwrite() 関数に置き換えることである．たとえば，ある half-process p が DMI_read(*addr*, *size*, *buf*, ...) 関数を呼び出し，この DMI_read() 関数が read フォルトを起こしたとする．この場合，第 **4** 章で述べたように，(1) オーナーに read フォルトが通知され，(2) オーナーが最新ページを転送し，(3) それを receiver half-process [*6] が受信して，(4) さらにそれが handler half-process に渡され，(5) そして handler half-process が最新ページのデータを half-process p の *buf* にデータコピーすることによって DMI_read() 関数が完了する．ここで，handler half-process が half-process p の *buf* にデータコピーする部分は，異なる half-process の非共有アドレス空間へのデータコピーになるため，通常の memcpy() 関数では実現できず，dmwrite() 関数に置き換える必要がある．

以上からわかるように，スレッドを half-process に置き換えるために必要な修正はそれほど多くなく，とくに，処理系の基本構造やアルゴリズム自体を変更する必要はない．いい換えると，half-process では，マルチスレッドプログラミングとほぼ同様の方法で並列分散プログラミング処理系などを記述することができる．

9.7 要約：利点と欠点

本章では，部分的にアドレス空間を共有するプロセスを実現するための汎用的なカーネルプリミティブとして half-process を提案し，それに基づく真に透過的なスレッド移動について述べた．half-process-move の新規性は以下のとおりである：

- プロセス間のデータ共有を簡単かつ高速に実現するためには，プロセス間共有メモリのセマンティクスでは不十分であることを指摘したうえで，スレッド間で共有されるアドレス空間とセマンティクス的に等価な共有アドレス空間を実現するためのカーネルプリミティブとして，half-process を提案している．
- half-process の応用可能性として，真に透過的なスレッド移動だけでなく，マルチスレッドプログラミングにおけるスレッドアンセーフなライブラリのサポート，柔軟なハイブリッドプログラミングの実現，並列分散プログラミング処理系の開発者の負担減などを指摘している．
- half-process の共有アドレス空間を低オーバヘッドで実装する新たな手法として，アドレス空間ス

[*6] **4.1** 節で述べた receiver スレッド，handler スレッド，sweeper スレッド，計算スレッドを，スレッドではなく half-process として実装したものを，それぞれ receiver half-process，handler half-process，sweeper half-process，計算 half-process と呼ぶことにする．

イッチングとページテーブルリダイレクションを提案している．

half-process-move の利点は，余計なプログラミング制約を課すことなく，DMI_yield() 関数をたった 1 行追加するだけで並列計算の再構成を実現できる点である．これに対して，第 1 の欠点は，thread-move と同様に，1 プロセッサに複数の half-process が割り当てられることによって性能が低下する点である．第 2 の欠点は，カーネルの改造が必要になる点である．第 3 の欠点は，**9.6.2** 節で述べたように，スレッドとして実装していれば単なるデータコピーで済んでいた処理を，dmread() 関数/dmwrite() 関数というシステムコールに置き換える必要があるため，非共有アドレス空間の間でのデータコピーが多く要求されるアプリケーションでは，カーネルへのコンテキストスイッチのオーバヘッドが全体の性能を低下させてしまう点である．

第 10 章

評価 II：並列計算の再構成に対する性能とプログラマビリティ

本章では, 第 **7** 章で述べた rescale, 第 **8** 章で述べた thread-move, 第 **9** 章で述べた half-process-move の 3 種類のプログラミングモデルに関して，並列計算の再構成に対する性能とプログラマビリティの評価を行う.

10.1 実験環境

実験環境としては，**6.1** 節で記述した環境を用いた．また，thread-move および half-process-move では，**8.4.1** 節で述べたように，スレッドスケジューリングをどのように行うかが重要である．しかし，本研究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し，再構成が必要になった場合には，その時点で存在するノードに対して，均等な数ずつスレッド/half-process を割り当てるような単純なスレッドスケジューリングを行う．また，**8.4.3** 節で述べたように，thread-move では *align* の値を最適化する余地があるが，本実験では $align = 1$ とした．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一度も発生しなかった.

10.2 各実験の意図

本章では多くの実験の結果と考察を示すが，各実験の意図を事前にまとめておく.

第 1 に，マイクロベンチマークとして，**10.3.1** 節では，half-process におけるプロセス間通信の性能を，既存のさまざまなプロセス間通信の性能と比較する．また，**10.3.2** 節では，アドレス空間スイッチングおよびページテーブルリダイレクションのオーバヘッドを測定する．さらに，**10.3.3** 節では，thread-move と half-process-move におけるスレッド移動/half-process 移動の性能を評価する.

第 2 に，**10.4** 節では，**6.5.1** 節から **6.6.3** 節までで評価した基本的なアプリケーションおよび応用的なアプリケーションを題材として，実際のアプリケーションにおける half-process のオーバヘッドを評価する.

第 3 に，**10.5** 節では，**6.5.1** 節から **6.6.3** 節までで評価した基本的なアプリケーションおよび応用

的なアプリケーションを題材として，thread-move および half-process-move において，1 プロセッサあたり複数のスレッド/half-process を生成することのオーバヘッドを評価する．

第 4 に，**10.6** 節では，**6.5.8** 節から **6.6.3** 節までで評価した N 体問題，ヤコビ法，有限要素法，ページランク計算，同期的な最短路計算を題材として，rescale，thread-move，half-process-move の各プログラミングモデルを使って再構成可能な並列計算を記述した場合のプログラマビリティを比較する．また，利用可能なノード数を動的に増減させたときに並列度がどのように変化するかを調べ，各プログラミングモデルの性能を比較する．

第 5 に，**10.7** 節では，**6.6.4** 節で評価した非同期的な最短路計算を題材として，thread-move を使って利用可能なノード数を動的に増減させたときに並列度がどのように変化するかを評価する．

10.3 マイクロベンチマーク

10.3.1 half-process におけるプロセス間通信のオーバヘッド

10.3.1.1 実験設定

さまざまなプロセス間通信の手段について，2 個の half-process 間でデータをコピーするのに要する時間を測定した．まず，ある 1 個のノード上に 2 個の half-process p と half-process p' を生成したあと，half-process p がデータ d をメモリ上の位置 l に書き込む．ここで時間計測を開始し，もう一方の half-process p' が，プロセス間通信の手段 m を使ってデータ d を読み込んで，データ d のチェックサムを計算するまでの時間を測定した．ここで，メモリ上の位置 l とプロセス間通信の手段 m について，以下の 6 とおりの場合を測定した：

socket l は half-process p の非共有アドレス空間であり，half-process p は TCP ソケットを使って half-process p' に対してデータ d を送信する．

pipe l は half-process p の非共有アドレス空間であり，half-process p は無名パイプを使って half-process p' に対してデータ d を送信する．

shared-space l は half-process p と half-process p' の共有アドレス空間であり，half-process p' は（単に）この共有アドレス空間からデータ d を読み込む．

inter-process l は half-process p と half-process p' の間のプロセス間共有メモリであり，half-process p' は（単に）このプロセス間共有メモリからデータ d を読み込む．

dmread l は half-process p の非共有アドレス空間であり，half-process p' は dmread() 関数によってデータ d を読み込む．

doublecopy l は half-process p の非共有アドレス空間であり，まず half-process p が half-process p' とのプロセス間共有メモリにデータ d を書き込んだあと，half-process p' がそのプロセス間共有メモリからデータ d を読み込む．これは，MVAPICH2 や OpenMPI のノード内通信の手段として用いられている方法である [30, 70]．

また，2 個の half-process をどの CPU 上に生成するかによっても性能が変化する．**6.1** 節で述べた実験環境では，各ノードは Intel Xeon E5530（4 プロセッサ）の CPU を 2 個搭載しているため，(1) 2

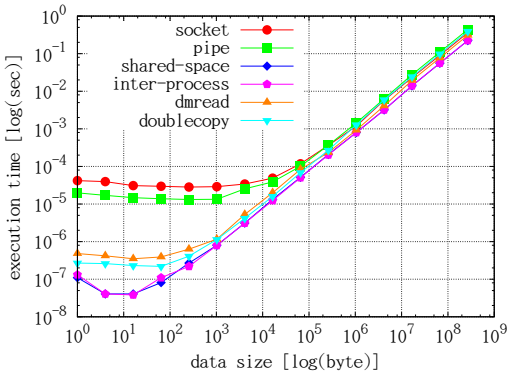

図 10.1　同一 CPU 上の 2 個の half-process 間のデータコピーの性能比較.

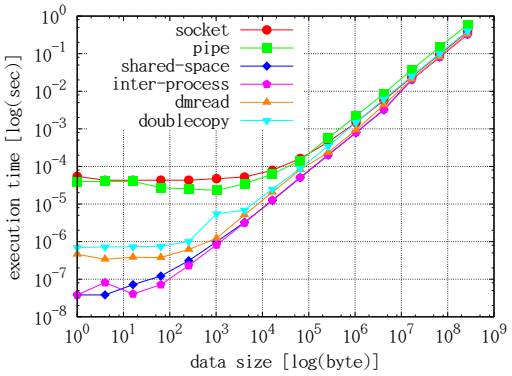

図 10.2　異なる CPU 上の 2 個の half-process 間のデータコピーの性能比較.

個の half-process を同一の CPU 上に配置する場合，(2) 2 個の half-process を異なる CPU 上に配置する場合の 2 とおりに関して調べた.

10.3.1.2　結果と考察

2 個の half-process を同一の CPU 上に配置した場合の結果を図 10.1 に，2 個の half-process を異なる CPU 上に配置した場合の結果を図 10.2 に示す．図 10.1 および図 10.2 より，第 1 に，socket および pipe は shared-space より遅く，たとえばデータサイズが 2^{28} バイトの場合には 1.17〜1.96 倍遅いことがわかる．第 2 に，shared-space と inter-process はほぼ同一の性能を示しているが，これは，両者とも，実際の共有メモリ上のデータを読み出しているという点でまったく同じ処理を行っているためである．第 3 に，データサイズが 2^{28} バイトの場合，図 10.1 では dmread によるダイレクトメモリアクセスは doublecopy よりも 1.22 倍速く，図 10.2 では dmread によるダイレクトメモリアクセスは doublecopy よりも 1.26 倍速い．第 4 に，図 10.2 においては，dmread と shared-space の性能がほぼ等しくなっている．ここで，dmread では half-process p' がデータ d を読み込んでチェックサムを計算する前にカーネル内部でのデータコピーが必要であるのに対して，shared-space では half-process p' は共有アドレス空間から直接データ d を読み込んでチェックサムを計算するだけでよいことをふまえると，shared-space の方が dmread よりも性能がよくなりそうに思われるが，そうならない理由は，本実験では，dmread の性能も shared-space の性能も，異なる CPU 間のメモリアクセス速度に支配されているためである.

10.3.2　アドレス空間スイッチングおよびページテーブルリダイレクションのオーバヘッド

10.3.2.1　実験設定

アドレス空間スイッチングのオーバヘッドを調べるために，非共有アドレス空間に対して 10000000 回の mmap() 関数を呼び出すのに要する時間と，共有アドレス空間に対して 10000000 回の mmap() 関数を呼び出すのに要する時間を測定した．なお，共有アドレス空間に対して mmap() 関数を呼び出し

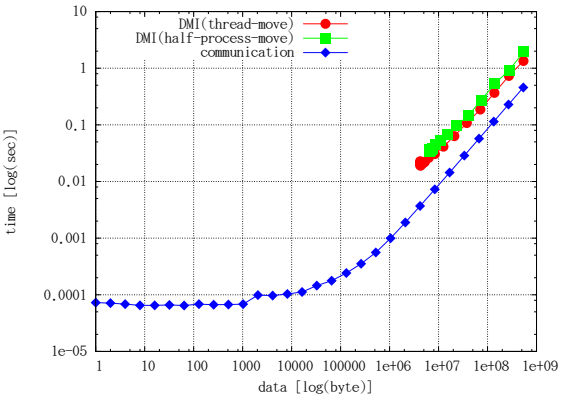

図 10.3　スレッド移動/half-process 移動の実行時間の内訳.

たときのみ，アドレス空間スイッチングが起きることに注意する．

また，ページテーブルリダイレクションのオーバヘッドを調べるために，非共有アドレス空間で連続する 1000000 個のページのページフォルトを処理するのに要する時間と，共有アドレス空間で連続する 1000000 個のページのページフォルトを処理するのに要する時間を測定した．なお，共有アドレス空間のページフォルトを処理するときのみ，ページテーブルリダイレクションが起きることに注意する．

10.3.2.2　結果と考察

非共有アドレス空間に対する 10000000 回の mmap() 関数の呼び出しは 2.531 秒，共有アドレス空間に対する 10000000 回の mmap() 関数の呼び出しは 2.654 秒であり，アドレス空間スイッチングのオーバヘッドは 4.8% であった．また，非共有アドレス空間での 1000000 回のページフォルトの処理は 1.316 秒，共有アドレス空間での 1000000 回のページフォルトの処理は 1.327 秒で，ページテーブルリダイレクションのオーバヘッドは 0.83% であり，このオーバヘッドは十分に小さいといえる．

10.3.3　スレッド移動のオーバヘッド

10.3.3.1　実験設定

スレッド/half-process が使用するデータサイズをさまざまに変化させたとき，2 ノード間でスレッド/half-process を移動させる場合の，thread-move におけるスレッド移動時間（thread-move）と half-process-move における half-process 移動時間（half-process-move）を調べた．また，そのスレッド移動時間/half-process 移動時間に占めるデータ転送の時間（communication）を調べた．

10.3.3.2　結果と考察

thread-move におけるスレッド移動時間，half-process-move における half-process 移動時間，2 ノード間の TCP のレイテンシを図 10.3 に示す．図 10.3 において，thread-move と half-process-move のグラフの横軸は，スレッド移動/half-process 移動において実際に転送されたデータサイズである．デフォルトではスレッド/half-process のスタック領域のサイズを 4 MB としているため，thread-move のグラフの横軸は 4 MB 付近から始まっている．また，half-process-move では，4 MB のスタック領

域に加えて静的変数領域のデータも half-process 移動の対象となるため，half-process-move のグラフの横軸は，6 MB 付近から始まっている．約 512 MB のデータをともなったスレッド移動/half-process 移動が起きる場合，スレッド移動時間/half-process 移動時間のうちデータ転送時間が占める割合は，thread-move の場合に 33.9%，half-process-move の場合に 23.5% であり，残りの時間がスレッドのチェックポイント/リスタートに消費されている．

10.4 実際のアプリケーションにおける half-process のオーバヘッド

10.4.1 実験設定

DMI の各インスタンスをスレッドとして実装する場合と比較すると，half-process として実装する場合には一定のオーバヘッドがともなう．オーバヘッドの要因としては，アドレス空間スイッチングのオーバヘッド，ページテーブルリダイレクションのオーバヘッド，TLB のフラッシュに起因するオーバヘッド，スレッドであれば単なるメモリコピーで済んでいた処理を dmread() 関数/dmwrite() 関数に置き換えることに起因するオーバヘッドなどが考えられる．これらのオーバヘッドを実際のアプリケーションで評価するため，**6.5** 節の評価で用いた NAS Parallel Benchmark の EP，マンデルブロ集合の描画，横ブロック分割による行列行列積，Fox アルゴリズムによる行列行列積，ランダムサンプリングソート，N 体問題，ヤコビ法，有限要素法，ページランク計算，同期的な最短路計算の各アプリケーションを，half-process で実装した DMI で実行し，その実行時間，ウィークスケーラビリティ，128 プロセッサで実行した場合における全体の実行時間と計算実行時間を測定した．

10.4.2 結果と考察

half-process で実装した DMI による実行結果を，DMI（half-process）というラベルで，図 6.11，図 6.12，図 6.13，図 6.16，図 6.17，図 6.18，図 6.20，図 6.21，図 6.22，図 6.23，図 6.24，図 6.25，図 6.26，図 6.27，図 6.28，図 6.29，図 6.30，図 6.31，図 6.32，図 6.33，図 6.34，図 6.36，図 6.38，図 6.39，図 6.42，図 6.44，図 6.46，図 6.43，図 6.45，図 6.47，図 6.52，図 6.54，図 6.56，図 6.53，図 6.55，図 6.57 の各グラフに示す．これらの結果より，NAS Parallel Benchmark の EP，マンデルブロ集合の描画，横ブロック分割の行列行列積，Fox アルゴリズムによる行列行列積，ランダムサンプリングソート，N 体問題，ヤコビ法では，DMI（half-process）の性能は通常の DMI の性能とほぼ等しく，half-process のオーバヘッドは現れていない．これに対して，有限要素法，ページランク計算，同期的な最短路計算では，有意に half-process のオーバヘッドが現れている．詳細なプロファイリングの結果，この 3 種類のアプリケーションにおけるオーバヘッドの主因は，いずれも，各イテレーションにおいて呼び出される rwset_read() 関数の内部で，handler half-process が，計算 half-process の非共有アドレス空間にデータを書き込む際のオーバヘッドにあることがわかった．

以下では，データセット medium0.1 の Web グラフを用いたページランク計算を例にして，このオーバヘッドの原因を説明する．**6.6.2** 節で述べたように，このページランク計算では，外点の値を取得するときに，128 個の各 half-process が自分以外の 127 個の half-process と各 21.5 KB の通信を行うような，ほぼ一様な All-to-all 型の通信が発生する．この外点の値の取得は，read-write-set の

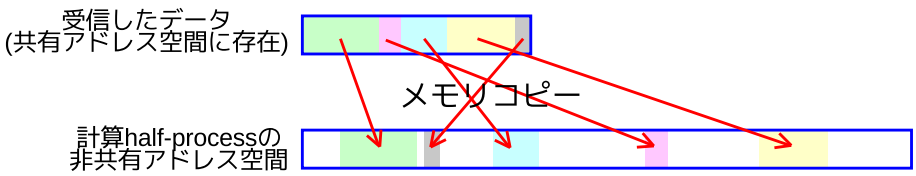

図 10.4 handler half-process が read 応答を処理するときに行われるデータコピー.

rwset_read() 関数によって実現される．そして，**5.3** 節で述べたように，この rwset_read() 関数は，内部で group_read() 関数を呼び出し，この group_read() 関数は，内部で 127 個のページに対して独立に read 要求を発行する．よって，やがてこれら 127 個の read 要求に対する 127 個の read 応答が，それぞれのページのオーナーから独立に返って来ることになり，これら 127 個の read 応答が handler half-process によって 1 個ずつ処理されることになる．

さて，ここで，handler half-process が，ある 1 個の read 応答を処理するときに何が起きるかを考える．一般に，group_read() 関数に起因した read 応答を処理する場合には，handler half-process は，図 10.4 に示すように，そのページのオーナーから送信されてきた各データを，計算 half-process の非共有アドレス空間上の「指示された位置」にコピーする必要がある．このとき，handler half-process が計算 half-process の非共有アドレス空間にデータをコピーするためには，dmwrite() 関数を使用する必要がある．ここで，「指示された位置」が 1 個の連続したアドレス範囲ならば dmwrite() 関数を 1 回呼び出すだけで済むが，「指示された位置」が複数のアドレス範囲に分断されている場合，そのアドレス範囲の個数だけ dmwrite() 関数を呼び出さなければならない．実際には，データセット medium0.1 の Web グラフを用いたページランク計算では，各 read 応答につき，「指示された位置」が 110〜160 個ものアドレス範囲に分断されている．したがって，これらの 110〜160 個の各アドレス範囲に対して dmwrite() 関数でデータをコピーしようとすると，そのつどカーネルへのコンテキストスイッチをはさむこととなり，性能が著しく落ちてしまう．この性能劣化を防ぐため，DMI では，「指示された位置」がある程度多い場合には，handler half-process が dmwrite() 関数を複数回呼び出すことでデータをコピーするのではなく，計算 half-process がみずからデータをコピーするように実装している．いい換えると，いったん handler half-process から計算 half-process に対してユーザレベルでコンテキストスイッチを行い，計算 half-process がみずから read 応答のデータをその計算 half-process の非共有アドレス空間上の「指示された位置」にコピーし，コピーが完了したあとで再び handler half-process に対してユーザレベルでコンテキストスイッチを行うように実装している．この場合，read 応答のデータを共有アドレス空間に配置しておけば，計算 half-process が read 応答のデータをその計算 half-process の非共有アドレス空間にコピーする処理は，通常の memcpy() 関数によって実現できるため，カーネルへのコンテキストスイッチが起きることはない．以上のような最適化により，「指示された位置」が複数のアドレス範囲にまたがるような非定型な有限要素法やグラフ計算に対して，half-process 上の read-write-set の性能を大きく改善できることを確認している．

しかし，以上のように最適化された実装であっても，通常の DMI と比較すると，DMI (half-process)

では無駄な処理がはさまることには変わりない．通常の DMI の場合には，handler スレッドが計算スレッドのアドレス空間に対して直接データをコピーできるのに対して，DMI（half-process）の場合には，handler half-process と計算 half-process との間でのユーザレベルのコンテキストスイッチが必要になってしまう．実際にこのオーバヘッドを測定してみたところ，通常の DMI の場合には，1 個の read 応答の処理は平均 41.0 マイクロ秒で完了するのに対して，DMI（half-process）の場合には，1 個の read 応答の処理は平均 96.8 マイクロ秒を要することがわかった．

さて，1 イテレーションにつき，rwset_read() 関数がすべての計算 half-process から 1 回ずつ呼ばれることになるが，このとき，handler half-process がそれに起因する read 応答をすべて処理するためにどれくらいの時間を要するのかを見積もる．まず，handler half-process は各ノード上に 1 個だけ存在しており，そのノードに届くすべての read 応答をシリアライズして処理する．そして，128 プロセッサで実行する場合には，各ノード上には計算 half-process が 8 個存在しているため，これらの各計算 half-process は 120 個の read 応答を引き起こすことになる（残り 7 個の read 要求はそのノードのメモリプールにキャッシュヒットしてその時点で処理されるため，この 7 個に関しては他のノードから read 応答が届くことはない）．したがって，handler half-process は，$120 \times 8 = 960$ 個の read 応答をシリアライズして処理しなければならないことになるため，すべての read 応答を処理するために要する時間は $960 \times 96.8 = 92.9$ ミリ秒である．一方で，通常の DMI の場合には，handler スレッドがすべての read 応答を処理するために要する時間は $960 \times 41.0 = 39.3$ ミリ秒である．すなわち，DMI の各インスタンスをスレッドとして実装するか half-process で実装するかによって，すべての read 応答を処理する時間に $92.9 - 39.3 = 53.6$ ミリ秒の差が出ると見積もることができる．ここで，図 6.43 を見ると，1 イテレーションに要している時間は通常の DMI が 0.479 秒，DMI（half-process）が 0.663 秒であり，その差は 184 ミリ秒であるから，以上の見積もりから，通常の DMI と DMI（half-process）の性能差の $53.6/184 \times 100 = 29.1\%$ を，read 応答を処理する時間の差として説明づけることができる．

より深い考察は避けるが，その他のプロファイリング結果から，結局のところ，通常の DMI と DMI（half-process）の性能差の大部分は，DMI（half-process）が余分なコンテキストスイッチを行っていることと，（**10.5** 節で述べるように）共有アドレス空間に実装している malloc アルゴリズムの性能が悪いことに起因することがわかっている．しかし，前者の原因に関しては，Linux では異なる非共有アドレス空間を直接アクセスする方法がないことをふまえると，DMI（half-process）において可能なアプローチは，コンテキストスイッチを行うことで計算 half-process がみずからデータをコピーするか，または handler half-process が dmwrite() 関数によってデータをコピーするかのいずれかの方法しかない．すなわち，以上で議論したオーバヘッドは，half-process の設計にとって潜在的に存在してしまうものである．改善案としては，handler half-process を複数用意することによって，これらのオーバヘッドを隠蔽することが考えられる．

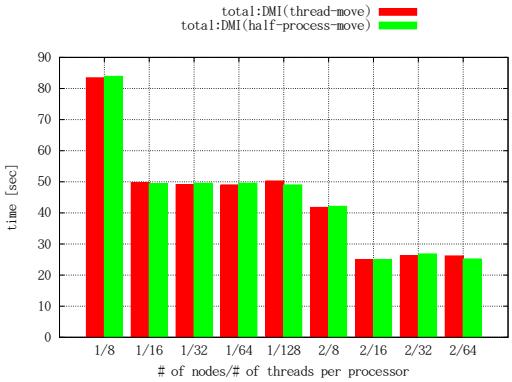

図 10.5　NAS Parallel Benchmark の EP においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

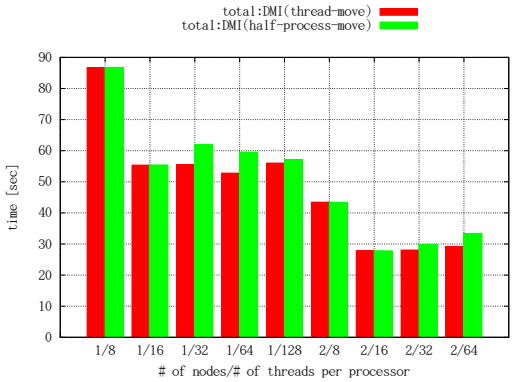

図 10.6　マンデルブロ集合描画においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

10.5　プロセッサ数以上のスレッドを生成することによる性能低下

10.5.1　実験設定

1.3.3.2 節で議論したように，thread-move と half-process-move のプログラミングモデルにおける欠点は，1 プロセッサあたり複数のスレッド/half-process が割り当てられてしまうことに起因する性能劣化である．そこで，1 プロセッサあたり複数のスレッド/half-process を割り当てた場合に，実際のアプリケーションにおいてどの程度性能が劣化するのかを測定した.

この実験では，NAS Parallel Benchmark の EP，マンデルブロ集合の描画，ランダムサンプリングソート，N 体問題，ヤコビ法，有限要素法，ページランク計算，同期的な最短路計算の各アプリケーションに関して，(1) 1 ノードに n 個のスレッド/half-process を割り当てて実行した場合の実行時間（n =8，16，32，64，128）と，(2) 2 ノードを使って各ノードに n 個のスレッド/half-process を割り当てて実行した場合の実行時間（n =8，16，32，64）を測定した．(1) と (2) の違いは，1 ノードを使うだけでは，ページフォルトやそれにともなう通信に起因するオーバヘッドの影響が現れないのに対して，2 ノードを使うとその影響が現れるという点である.

10.5.2　結果と考察

NAS Parallel Benchmark の EP，マンデルブロ集合の描画，ランダムサンプリングソート，N 体問題，ヤコビ法，有限要素法，ページランク計算（データセット medium0.01），ページランク計算（データセット medium0.1），同期的な最短路計算（データセット medium0.01），同期的な最短路計算（データセット medium0.1）の各アプリケーションに関して，プロセッサ数以上のスレッド/half-process を生成した場合の実行時間を，それぞれ，図 10.5，図 10.6，図 10.7，図 10.8，図 10.9，図 10.10，図 10.11，図 10.12，図 10.13，図 10.14 に示す．これらのグラフにおける横軸の m/n は，m 個のノードを使って各ノードに n 個のスレッド/half-process を生成して実行した場合の結果を表している.

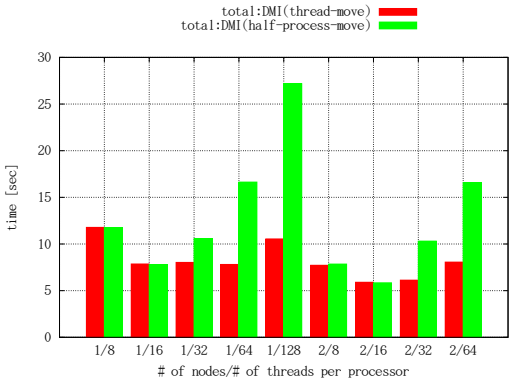

図 10.7 ランダムサンプリングソートにおいてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

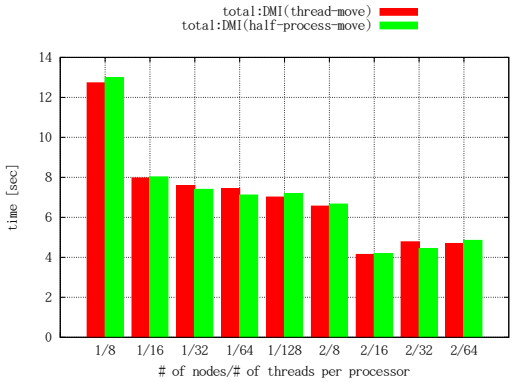

図 10.8 N 体問題においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

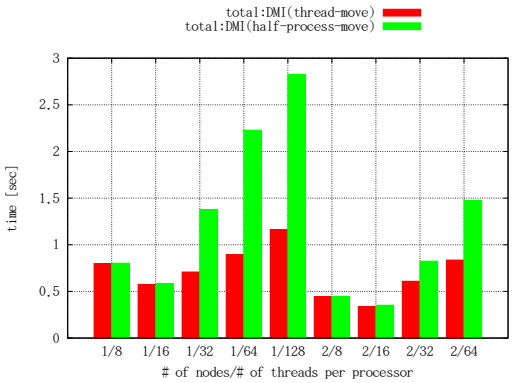

図 10.9 ヤコビ法においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

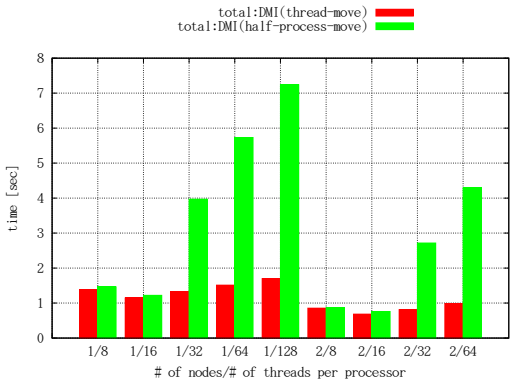

図 10.10 有限要素法においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

第 1 に，1/8 よりも 1/16 の方が実行時間が短くなっている場合が多いのは，各ノード上の物理的なプロセッサ数は 8 であるが，ハイパースレッディングによって論理的なプロセッサ数は 16 になっており，アプリケーションによってはハイパースレッディングによる性能向上が得られているためである．

第 2 に，DMI（thread-move）に関して，各アプリケーションごとに，「1/64 が 1/16 よりどれくらい遅いか」と「2/64 が 2/16 よりどれくらい遅いか」を比較すると，後者の影響の方が大きいことがわかる．たとえば，DMI（thread-move）におけるランダムサンプリングソートでは，1/64 は 1/16 より-0.62% 遅いだけだが，2/64 は 2/16 より 36.6% も遅い．また，DMI（thread-move）におけるページランク計算（データセット medium0.1）では，1/64 は 1/16 より 4.28% 遅いだけだが，2/64 は 2/16 より 45.8% も遅い．これらの事実は，**ノード数が増えてページフォルトやそれにともなう通信が増えるほど，1 プロセッサに割り当てるスレッド数を増やした場合の性能劣化が現れやすい**という傾向を示

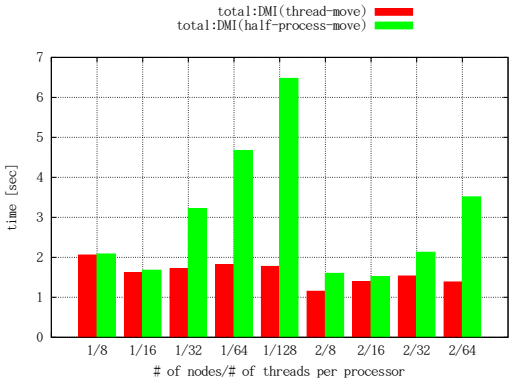

図 10.11 ページランク計算（データセット medium0.01）においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

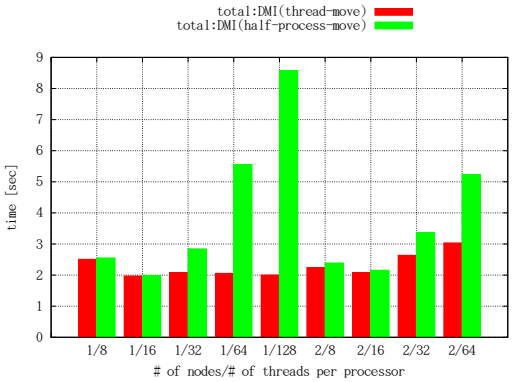

図 10.12 ページランク計算（データセット medium0.1）においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

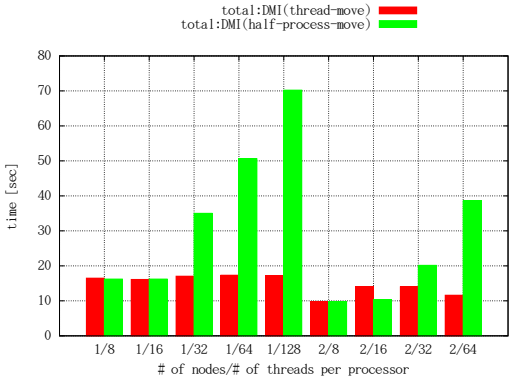

図 10.13 同期的な最短路計算（データセット medium0.01）においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

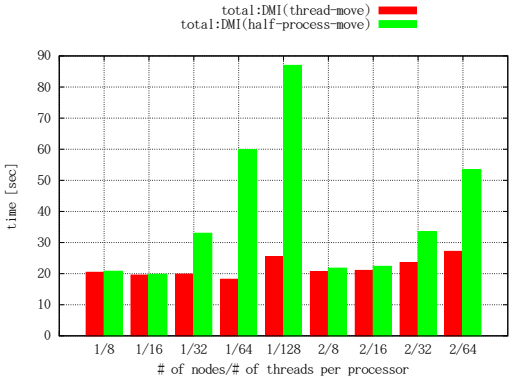

図 10.14 同期的な最短路計算（データセット medium0.1）においてプロセッサ数以上のスレッド/half-process を生成した場合の性能低下.

唆している．このような傾向が出る理由は，現時点の DMI では，各プロセスに対して handler スレッドが 1 個しか存在しないことに関係していると考えられる．handler スレッドに加わる負荷は，ページフォルトやそれにともなう通信が増えるほど，そしてプロセス内のスレッド数が増えるほど大きくなる．さらに，**10.4** 節で述べたように，DMI では handler スレッドの処理がボトルネックになっていることが多く，handler スレッドに加わる負荷が，そのまま全体の性能に影響しやすい．したがって，ノード数が増えてページフォルトやそれにともなう通信が増えるほど，1 プロセッサに割り当てるスレッド数を増やした場合の性能劣化が現れやすくなってしまうのだと考えられる．よって，この性能劣化を改善するための工夫としては，handler スレッドの複数化が考えられる．

第 3 に，**DMI（half-process-move）は，DMI（thread-move）と比較すると，1 プロセッサに割り当てる half-process 数を増やした場合の性能劣化が非常に大きい**ことがわかる．より詳しく観

表 10.1 再構成に対応しないプログラムを再構成に対応させるために必要なプログラムの変更行数 [行].

	N 体問題	ヤコビ法	有限要素法	ページランク計算	同期的な最短路計算
rescale	44	11	187	29	29
thread-move	5	5	62	18	16
half-process-move	1	1	1	1	1

察すると，DMI（half-process-move）の性能劣化は，通信が単純でかつ通信量の少ない NAS Parallel Benchmark の EP，マンデルブロ集合の描画，N 体問題では小さく，通信が複雑で通信量の多いアプリケーションほど大きくなる傾向がある．DMI（half-process-move）における性能劣化の原因の 1 つは，**10.4** 節で述べたように，handler half-process が計算 half-process の非共有アドレス空間に直接データをコピーできないことに起因するオーバヘッドである．ところが，1 ノードで DMI を実行する場合にはページフォルトは発生せずこのオーバヘッドは起きえないことをふまえると，このオーバヘッドだけでは，$1/n$ の場合に n を増やすにしたがって DMI（half-process-move）の性能が劣化する理由を説明づけることはできない．別のプロファイリングから，オーバヘッドのもう 1 つの主要な原因は，現在の DMI（half-process-move）の実装における malloc アルゴリズムの性能の悪さであることがわかった．**9.5.7** 節で述べたように，DMI（half-process-move）では，half-process の共有アドレス空間を対象に malloc/free/realloc 操作を行うためのユーザレベルのライブラリ関数を提供しているが，現在の実装は単純なカーニハン・リッチーの malloc アルゴリズム [35] を使ったものであり，スレッド数が増えた場合には，少なくとも glibc の malloc アルゴリズムよりも大幅に性能が悪くなることを確認している．この問題は，Tcmalloc[8] や Hoard[52] など，マルチスレッド向けの洗練された malloc アルゴリズム [131, 166] を利用することで改善できると考えられる．

10.6 同期的な並列反復計算の再構成

10.6.1 プログラマビリティの比較

10.6.1.1 実験設定

N 体問題，ヤコビ法，有限要素法，ページランク計算，同期的な最短路計算の各アプリケーションに関して，rescale，thread-move，half-process-move の 3 種類のプログラミングモデルを使って再構成可能な並列計算を記述した場合のプログラマビリティについて調べた．具体的には，rescale，thread-move，half-process-move の各プログラミングモデルについて，再構成に対応しないプログラムを再構成に対応させる場合に，プログラムに何行の変更を加える必要があるかを調べた．

10.6.1.2 結果と考察

各プログラミングモデルについて，再構成に対応しないプログラムを再構成に対応させるために必要なプログラムの変更行数を表 10.1 にまとめる．表 10.1 からわかるように，プログラマビリティは，高い順に，half-process-move，thread-move，rescale である．

第 1 に，rescale における変更箇所は，データのチェックポイント/リストアのためのコード，

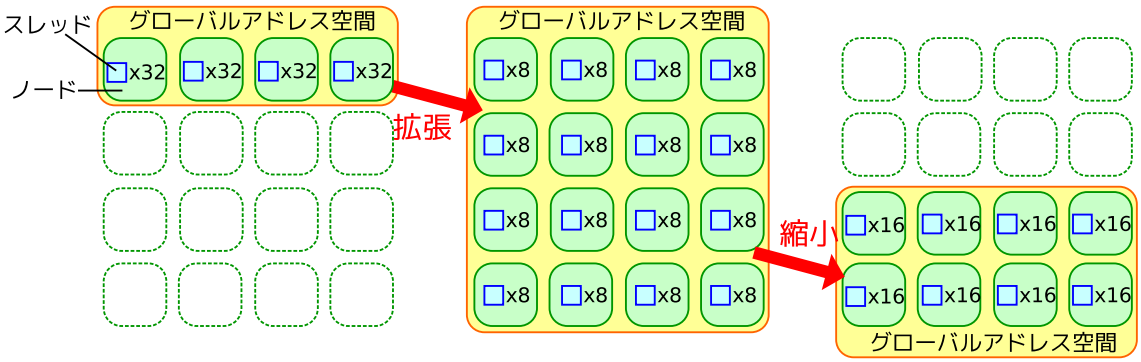

図 10.15　利用可能なノード数を 4 ノード →16 ノード →8 ノードの順に増減させる様子.

DMI_check_reconf() 関数の記述，DMI_judge_reconf() 関数の記述の 3 ヶ所である．このうち，DMI_check_reconf() 関数と DMI_judge_reconf() 関数の記述は機械的に行えるが，どのデータをチェックポイント/リストアすればよいかは，アプリケーションが複雑な場合には自明でないことが多い．たとえば，有限要素法で用いている図 6.37 の BiCGSafe 法では，チェックポイント/リストアする必要のあるデータは 13 個のベクトルと変数 2 個であるが，これらのデータをチェックポイント/リストアするべきであると判断するためには，BiCGSafe 法のアルゴリズムの正確な分析が欠かせない．また，当然，1 個でもデータのチェックポイント/リストアを忘れると正しい実行結果は得られず，デバッグの見通しが悪いといえる．

第 2 に，thread-move における変更箇所は，DMI_yield() 関数の記述，スレッド移動を越えて利用されるメモリ領域に対する malloc() 関数/free() 関数/realloc() 関数を DMI_thread_malloc() 関数/DMI_thread_free() 関数/DMI_thread_realloc() 関数に置き換えることの 2 ヶ所である．変更行数は rescale よりも少ない．また，各 malloc() 関数/free() 関数/realloc() 関数で確保/解放するメモリ領域がスレッド移動を越えて利用されるかどうかを判断する部分をのぞけば，いずれの変更もほぼ機械的に行うことができる．ただし，**8.3** 節で述べたように，thread-move には理解にくいプログラミング制約が存在する．

第 3 に，half-process-move における変更箇所は，**DMI_yield() 関数の記述だけ**である．また，thread-move のようなプログラミング制約も存在せず，再構成可能な並列計算に対するプログラマビリティは非常に高い．

10.6.2　性能の比較

10.6.2.1　実験設定

N 体問題，ヤコビ法，有限要素法，ページランク計算，同期的な最短路計算の各アプリケーションに関して，利用可能なノード数を動的に増減させたときに並列度がどのように変化するかを調べ，rescale，thread-move，half-process-move の 3 種類のプログラミングモデルの性能を比較した．具体的には，図 10.15 に示すように，(1) 初期的にはノード 0 からノード 3 の 4 ノードで実行を開始し（合計 4 ノード，

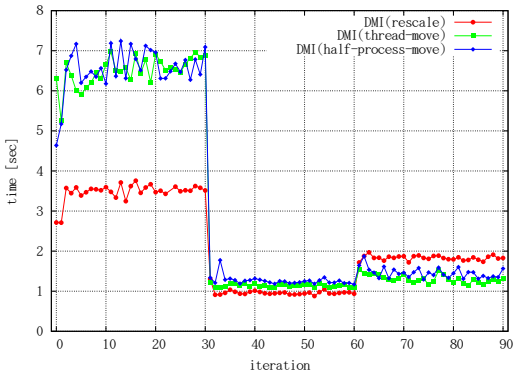

図 10.16　N 体問題を再構成した場合の各イテレーションの実行時間の変化.

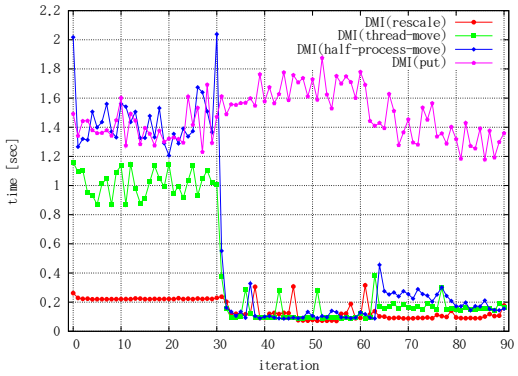

図 10.17　ヤコビ法を再構成した場合の各イテレーションの実行時間の変化.

32 プロセッサ），（2）約第 30 イテレーション*1 の直後にノード 4 からノード 15 の 12 ノードを参加させ（合計 16 ノード，128 プロセッサ），（3）約第 60 イテレーションの直後にノード 0 からノード 8 の 8 ノードを脱退させる（合計 8 ノード，64 プロセッサ）というように，利用可能なノード数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の変化，12 ノード参加時の再構成に要した時間と移動したデータ量，8 ノード脱退時の再構成に要した時間と移動したデータ量について調べた．なお，thread-move および half-process-move では，生成するスレッド/half-process 数は 128 個とした．

なお，**10.2** 節で述べたように，thread-move と half-process で再構成が必要になった場合には，その時点で存在するノードに対して，均等な数ずつスレッド/half-process を割り当てるような単純なスレッドスケジューリングを行う．よって，12 ノード参加時および 8 ノード脱退時には，128 個のスレッド/half-process のうち 120 個のスレッド/half-process が移動することになる．

10.6.2.2　結果

N 体問題，ヤコビ法，有限要素法，ページランク計算（データセット medium0.1），同期的な最短路計算（データセット medium0.1）の各アプリケーションに関して，利用可能なノード数を変化させた場合の各イテレーションの実行時間の変化を，それぞれ図 10.16，図 10.17，図 10.18，図 10.19，図 10.20 に示す．グラフ中の DMI（rescale），DMI（thread-move），DMI（half-process-move）が，それぞれ rescale，thread-move，half-process-move の結果を表す．図 10.17 における DMI（put）については後述する．また，各アプリケーションに関して，12 ノード参加時の再構成に要した時間と再構成に関係したデータ量を，それぞれ図 10.21，図 10.22，図 10.23，図 10.24，図 10.25 に示す．さらに，各アプリケーションに関して，8 ノード脱退時の再構成に要した時間と再構成に関係したデータ量を，それぞれ図 10.26，図 10.27，図 10.28，図 10.29，図 10.30 に示す．図 10.21 から図 10.30 までのグラフでは，data size の縦軸が右であり，checkpoint，restore，migration の縦軸が左である．rescale の data size は，

*1　正確に第 30 イテレーションではなく，第 31 イテレーションや第 32 イテレーションになっている場合もある．これは，DMI における参加/脱退の指示は，コマンドラインからタイミングよく手動で出す必要があり，若干の誤差が生じるためである．

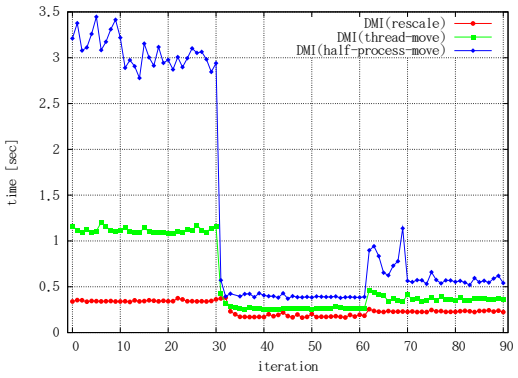

図 10.18　有限要素法を再構成した場合の各イテレーションの実行時間の変化.

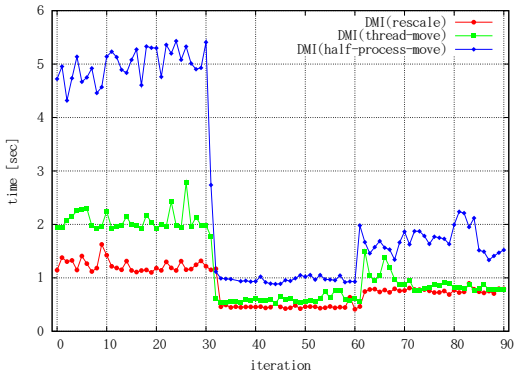

図 10.19　ページランク計算（データセット medium0.1）を再構成した場合の各イテレーションの実行時間の変化.

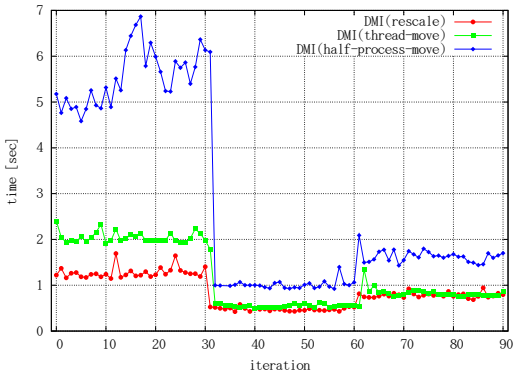

図 10.20　同期的な最短路計算（データセット medium0.1）を再構成した場合の各イテレーションの実行時間の変化.

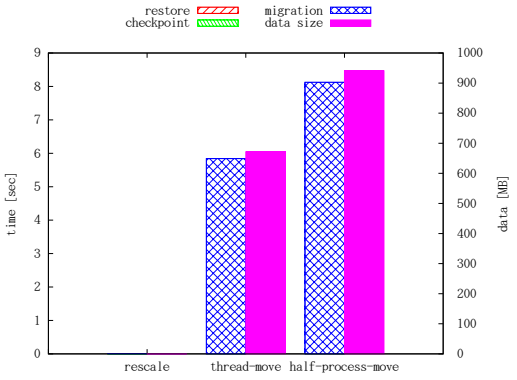

図 10.21　N 体問題について，12 ノード参加時の再構成に要した時間と再構成に関係したデータ量.

rescale においてチェックポイント/リストアされたデータ量を表し，thread-move/half-process-move の data size は，thread-move/half-process-move におけるスレッド移動/half-process 移動で実際に移動されたデータ量を表す．checkpoint，restore，migration は，それぞれ，「rescale においてデータのチェックポイントに要した時間」,「rescale においてデータのリストアに要した時間，および再構成にともなって各スレッドの担当範囲を再計算するのに要した時間」,「thread-move/half-process-move においてスレッド移動/half-process 移動全体に要した時間」を表す．rescale では migration は関係せず，thread-move と half-process-move では checkpoint と restore は関係しない．rescale では checkpoint の時間と restore の時間の合計が再構成に要した全体の時間を表し，thread-move と half-process-move では migration の時間が再構成に要した全体の時間を表す.

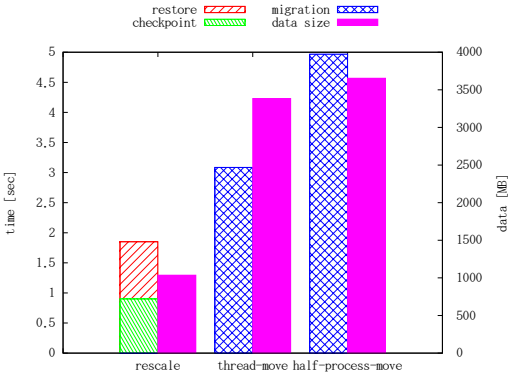

図 10.22　ヤコビ法について，12 ノード参加時の再構成に要した時間と再構成に関係したデータ量.

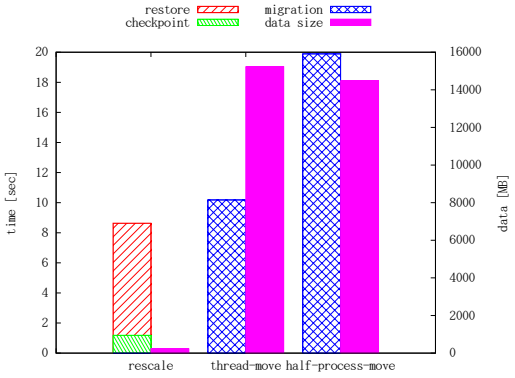

図 10.23　有限要素法について，12 ノード参加時の再構成に要した時間と再構成に関係したデータ量.

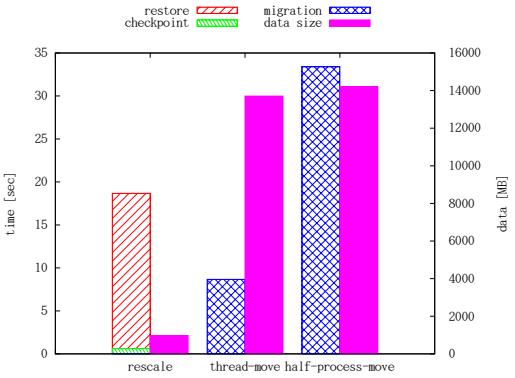

図 10.24　ページランク計算（データセット medium0.1）について，12 ノード参加時の再構成に要した時間と再構成に関係したデータ量.

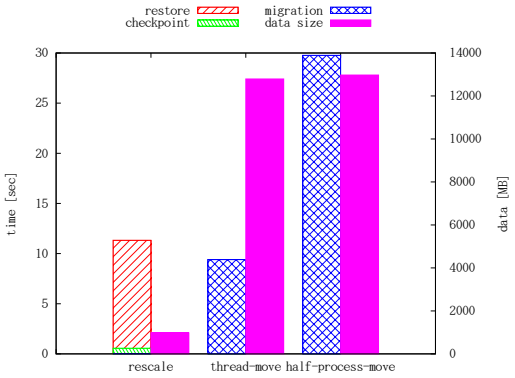

図 10.25　同期的な最短路計算（データセット medium0.1）について，12 ノード参加時の再構成に要した時間と再構成に関係したデータ量.

10.6.2.3　利用可能なノード数の増減に対応した並列度の変化

第 1 に，図 10.16，図 10.17，図 10.18，図 10.19，図 10.20 より，rescale，thread-move，half-process のいずれのプログラミングモデルでも，利用可能なノード数の増減に対応して効果的に並列度を増減させられていることがわかる．とくに，**有限要素法や Web グラフ解析などの非定型で応用的なアプリケーションに対しても効果的に並列度を増減させられている**ことがわかる．

第 2 に，4 ノードで実行している第 0 イテレーション〜第 30 イテレーションまでの期間では，いずれのアプリケーションでも，rescale>thread-move>half-process の性能になっている．この理由は，thread-move と half-process では，1 プロセッサあたり 4 スレッド/half-process が割り当てられており，**10.5** 節で評価したオーバヘッドが加わっているためである．たとえば，図 10.17 に示したヤコビ法では，rescale と比較して，thread-move は 355% 遅く，half-process は 544% 遅い．図 10.19 に示し

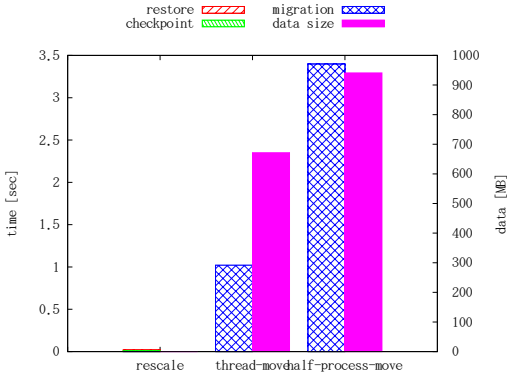

図 10.26　N 体問題について，8 ノード脱退時の再構成に要した時間と再構成に関係したデータ量.

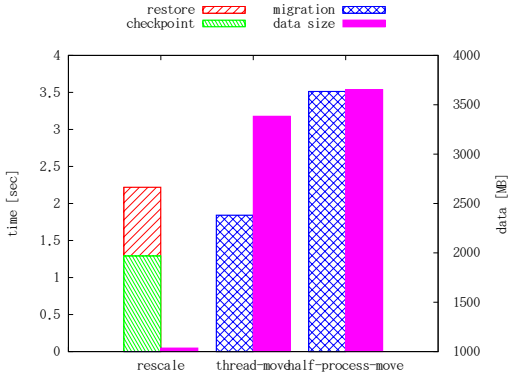

図 10.27　ヤコビ法について，8 ノード脱退時の再構成に要した時間と再構成に関係したデータ量.

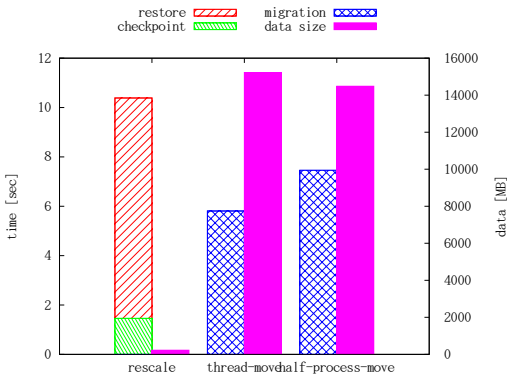

図 10.28　有限要素法について，8 ノード脱退時の再構成に要した時間と再構成に関係したデータ量.

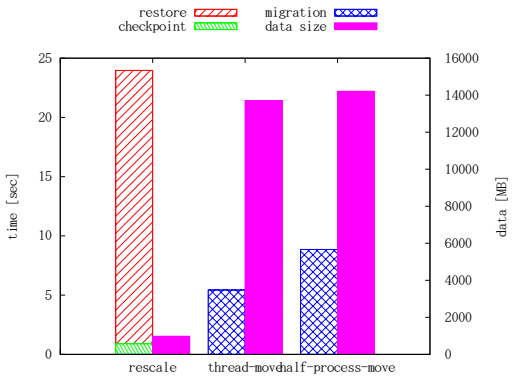

図 10.29　ページランク計算（データセット medium0.1）について，8 ノード脱退時の再構成に要した時間と再構成に関係したデータ量.

たページランク計算（データセット medium0.1）では，rescale と比較して，thread-move は 68.3% 遅く，half-process は 305% 遅い．このように，thread-move と half-process においては，handler スレッド/handler half-process の複数化や malloc アルゴリズムの改良などによって，**1 プロセッサに複数のスレッド/half-process を割り当てる場合のオーバヘッドを削減することがきわめて重要**であるといえる．

第 3 に，16 ノードで実行している第 31 イテレーション〜第 60 イテレーションまでの期間では，N 体問題とヤコビ法では，rescale≈thread-move≈half-process の性能になっており，有限要素法，ページランク計算，同期的な最短路計算では，rescale≈thread-move>half-process の性能になっている．ここで，rescale と thread-move の性能がほぼ等しい理由は，thread-move では 128 個のスレッドを生成しているため，16 ノードで実行される場合には，rescale と同様に，1 プロセッサあたり 1 スレッドが

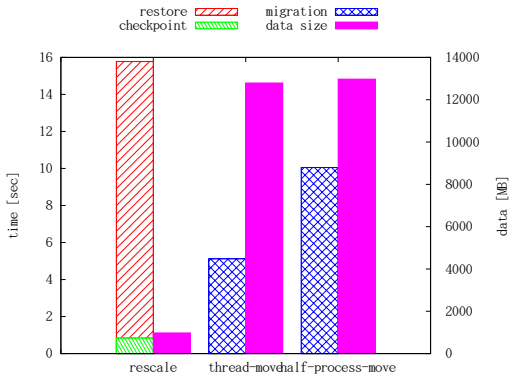

図 10.30　同期的な最短路計算（データセット medium0.1）について，8 ノード脱退時の再構成に要した時間と再構成に関係したデータ量.

割り当てられているためである．また，half-process の性能が，N 体問題とヤコビ法では thread-move とほぼ等しいにもかかわらず，有限要素法，ページランク計算，同期的な最短路計算では thread-move よりも劣るという結果は，**10.4** 節で評価した結果と合致している．

10.6.2.4　再構成に要する時間

図 10.21 から図 10.30 より，再構成に要する時間や再構成にかかわるデータ量について以下のことがわかる．

第 1 に，rescale と thread-move を比較すると，rescale においてチェックポイント/リストアされるデータ量は，thread-move においてスレッド移動で移動されるデータ量よりも非常に少ないものの，再構成に要する時間自体はまちまちで，thread-move の方が速い場合もあることがわかる．まず，rescale においてチェックポイント/リストアされるデータ量が少ない理由は，rescale では，再構成後に各スレッドの担当範囲が変化したときに，各スレッドが新しい担当範囲のデータを入力ファイルから読みなおすことになるため，チェックポイント/リストアする必要があるのは並列計算の実行状態を表すようなデータだけで済むからである．たとえば，ページランク計算を行う場合，各スレッド i は，そのスレッド i が担当するサブグラフの形状を表すデータ（エッジの結合関係やエッジの重み）と，そのスレッド i が担当するサブグラフの各節点の現在の値を必要とする．このうち，サブグラフの各節点の現在の値はページランク計算が進むにつれて変化するため，チェックポイント/リストアする必要がある．これに対して，サブグラフの形状を表すデータはページランク計算が進んでも変化しないため，各スレッドが担当するサブグラフが再構成にともなって変化するたびに入力ファイルから読みなおす方が，プログラムの記述上自然である．このように，rescale では，チェックポイント/リストアするデータ量は少なくなるものの，再構成のたびにデータを入力ファイルから読みなおす必要があるため，restore に要する時間が長くなり，結局，再構成全体に要する時間が thread-move よりも長くなる場合がある[⋆2]．ま

[⋆2] rescale では，再構成のたびにデータを入力ファイルから読みなおすのではなく，それらのデータもグローバルアドレス空

た，rescale における restore では，入力ファイルの読みなおしだけではなく，各スレッドの担当範囲の再計算を行う必要もある．ヤコビ法などのように単純な 1 次元的な領域分割しか行わない場合には，各スレッドの担当範囲の再計算は単純な代数計算だけで済むため，これに要する時間は無視できる．しかし，有限要素法では，領域間オーバラップやフィルインを考慮して領域間の計算負荷が均等化するような非定型で複雑な領域分割を行う必要があるため，各スレッドの担当範囲の再計算の時間も無視できなくなる．実際には，図 10.23 の rescale では領域分割に 3.54 秒を要しており，図 10.28 の rescale では領域分割に 2.85 秒を要している．これに対して，thread-move の場合には，各スレッドの担当範囲が途中で変化することはないため，並列計算の実行に必要なすべてのデータが，実行開始から実行終了まで各スレッドのメモリ上に乗っているようにプログラムが記述される．そして，スレッド移動時には，各スレッドのメモリ上のすべてのデータが移動対象となるため，再構成時に移動されるデータ量が多くなる．たとえば，thread-move では，物理的にいくつのプロセッサが利用可能であるかにかかわらずつねに 128 個のスレッドが立っており，スレッド i のメモリ上には，実行開始から実行終了までサブグラフ i の形状を表すデータとサブグラフ i の各節点の現在の値が乗っていて，スレッド移動時にはこの両方のデータが移動の対象になる．以上を要約すると，rescale においてチェックポイント/リストアされるデータ量と thread-move においてスレッド移動で移動されるデータ量を比較するだけならば，たしかに前者の方が少ない．しかし，rescale では再構成時に入力ファイルからのデータの読みなおしや各スレッドの担当範囲の再計算が必要となるため，再構成全体に要する時間を比較するならば，rescale と thread-move のどちらが速いかはアプリケーションに依存する．

第 2 に，thread-move と half-process-move を比較すると，スレッド移動/half-process 移動にともなって移動されるデータ量は，half-process-move の方がやや多い．この理由は，thread-move では，DMI_thread_mmap() 関数/DMI_thread_munmap() 関数/ DMI_thread_mremap() 関数で確保/解放されたメモリ領域のみが移動対象となるのに対して，half-process-move では，その時点で half-process の非共有アドレス空間に存在するすべてのメモリ領域が移動対象になり，静的変数領域のデータも移動対象になるからである[⋆3] ．また，再構成に要する時間は，thread-move よりも half-process-move の方が長い．ただし，この再構成に要する時間の差は，移動データ量の差だけで説明づけられるものではない．たとえば，図 10.22 のヤコビ法では，half-process-move における移動データ量は thread-move における移動データ量より 7.96% 多いだけであるが，half-process-move の再構成に要する時間は thread-move の再構成に要する時間よりも 60.9% も長い．また，図 10.24 のページランク計算では，half-process-move における移動データ量は thread-move における移動データ量より 3.67% 多いだけであるが，half-process-move の再構成に要する時間は thread-move の再構成に要する時間よりも 285% も長い．この理由は，**10.4** 節や **10.5** 節で観測された half-process のオーバヘッドに起因するものであると思われるが，オーバヘッドの具体的な内訳はまだ分析できていない．

間にチェックポイント/リスタートするようにすれば，NFS へのアクセスを避けられるため，再構成全体に要する時間を短縮できる可能性はある．

[⋆3] ただし，図 10.23 と図 10.28 では，half-process-move における移動データ量が thread-move における移動データ量よりも少なくなっている．この原因は特定できていない．

10.6.2.5 選択的キャッシュ read/write の効果

3.2.3.4 節で述べたように，選択的キャッシュ read/write はアクセスローカリティを柔軟に最適化するための強力な手段であると同時に，再構成にともなって動的にアクセスローカリティが変化する並列計算において，実際のアクセスローカリティにしたがってデータ分散を動的に適応させるための手段でもある．ここでは，ヤコビ法を題材にして，再構成可能な並列計算に対する選択的キャッシュ read/write の効果を確認する．

6.5.9 節で述べたヤコビ法では，各領域は左右 2 つの隣接領域を持ち，それぞれ 514^2 個の ghost 要素を持つ．そして，DMI では，ghost 要素の値の交換はグローバルアドレス空間を介して行われる．具体的には，ページサイズが 514^2×sizeof(double) で十分な数のページを持ったグローバルアドレス空間を確保しておき，各プロセッサ i は各イテレーションにおいて以下の処理を行う：

(1) 各プロセッサ i は，プロセッサ i の左の隣接領域にとっての ghost 要素の値を，グローバルアドレス領域 [514^2×sizeof(double)×$2i$, 514^2×sizeof(double)×$(2i+1)$) に write する．また，プロセッサ i の右の隣接領域にとっての ghost 要素の値を，グローバルアドレス領域 [514^2×sizeof(double)×$(2i+1)$, 514^2×sizeof(double)×$(2i+2)$) に write する．
(2) すべてのプロセッサが同期する．
(3) 各プロセッサ i は，グローバルアドレス領域 [514^2×sizeof(double)×$(2i-1)$, 514^2×sizeof(double)×$2i$) とグローバルアドレス領域 [514^2×sizeof(double)×$(2i+2)$, 514^2×sizeof(double)×$(2i+3)$) を read することで，ghost 要素の値を取得する．
(4) 各プロセッサ i は，担当領域に関して 27 点ステンシル計算を行う．

上記の処理において，(1) における write を EXCLUSIVE モードで発行すれば，実際のアクセスローカリティにしたがってページのオーナーを移動させることができ，再構成にともなってデータ分散を動的に適応させることができる．

図 10.17 における DMI(thread-move)が，thread-move に関して，(1)における write を EXCLUSIVE モードで発行した場合の結果であり，図 10.17 における DMI（put）が，thread-move に関して，(1) における write を PUT モードで発行した場合の結果である．図 10.17 より，PUT モードを使う場合には，ノードを参加させてもイテレーションの実行速度を改善できていないことがわかる．以上の結果より，**再構成にともなって並列度を効果的に増減させるためには，選択的キャッシュ read/write によるアクセスローカリティの最適化が重要な役割を果たす**ことが確認できる．

10.7 非同期的な並列計算の再構成

10.7.1 実験設定

以上で述べた実験は，いずれも同期的な反復計算を対象にしたものである．これに対して，本実験では，**6.6.4** 節で評価した非同期的な最短路計算（データセット medium0.1）を題材にして，thread-move を使ってプロセスを動的に参加/脱退させたとき，内部的にどのような挙動が起きるかを調べた．

thread-move におけるスレッド数は 128 個とした．また，1 ノードあたり 1 プロセスを生成すること

とし，初期的にはノード 0 からノード 3 の 4 ノードで実行し（合計 4 ノード，32 プロセッサ），しばらくしてからノード 4 からノード 15 の 12 ノードを参加させ（合計 16 ノード，128 プロセッサ），さらにしばらくしてからノード 0 からノード 7 の 8 ノードを脱退させた（合計 8 ノード，64 プロセッサ）．

10.7.2 結果と考察

128 個のスレッドの時系列的な挙動を図 10.31 に示す．なお，図 10.31 は，図 6.59 に対してプロセスの参加/脱退を加えたものであるため[*4]，アルゴリズム自体の挙動に関しては，図 6.59 および **6.6.4** 節の説明を参照されたい．図 10.31 では，横軸が時間を表し，縦軸が 128 個のスレッドを表す．また，茶色の長方形（wait）が，図 6.58 に示した最短路計算のアルゴリズムの 5 行目の do_iteration_synchronously() 関数内の先頭のバリアで待機している時間を表し，黄緑色の長方形（yield）が，DMI_yield() 関数の内部で過ごしている時間を表す．その他の色の長方形（iterXXXXX）が 1 イテレーションを表し，各色はそのイテレーションが実行されたプロセスを表している．たとえば凡例の iter17850 は，そのイテレーションがプロセス 17850 で実行されたことを意味している．

図 10.31 より以下のことが読みとれる：

- DMI_yield() 関数の前後で長方形の色が変化しているが，これはスレッドが移動し，そのスレッドを実行しているプロセスが変化したことを意味する．たとえば，スレッド 1 は，時刻 12 秒付近ではピンク色のプロセスでイテレーションを実行しているが，時刻 13〜24 秒に DMI_yield() 関数の内部でスレッド移動が生じ，時刻 25 秒付近では青色のプロセスでイテレーションを実行している．
- 時刻 0〜13 秒の間では 128 個のスレッドが 4 個のプロセスに分散されて実行されたことがわかる．そして，時刻 13 秒付近で 12 ノードの参加が指示され，時刻 15〜29 秒付近では 128 個のスレッドが 16 個のプロセスに分散されて実行されたことがわかる．やがて，時刻 29 秒付近で 8 ノードの脱退が指示され，時刻 32〜57 秒付近では 128 個のスレッドが 8 個のプロセスに分散されて実行されたことがわかる．

図 10.31 の結果は，DMI では，**グローバルアドレス空間モデルで記述された同期をともなわない並列計算に対しても，プロセスを自由なタイミングで参加/脱退させられる**という事実を示している．これは，**DMI のグローバルアドレス空間のコヒーレンシが，プロセスの非同期的な参加/脱退を越えて維持されているからこそ実現できていることであり，著者の知るかぎり，同様のことを実現できた処理系はこれまでに存在しない**．

10.8 要約

本章では，マイクロベンチマーク，基本的なアプリケーション，応用的なアプリケーションを題材にして，rescale，thread-move，half-process-move の 3 種類のプログラミングモデルを使って並列計算の再構成を実現した場合の性能とプログラマビリティを評価した．全体を要約すると以下のとおりである：

[*4] ただし，図 10.31 はデータセット medium0.1 を使っており，図 6.59 はデータセット large0.1 を使っている．

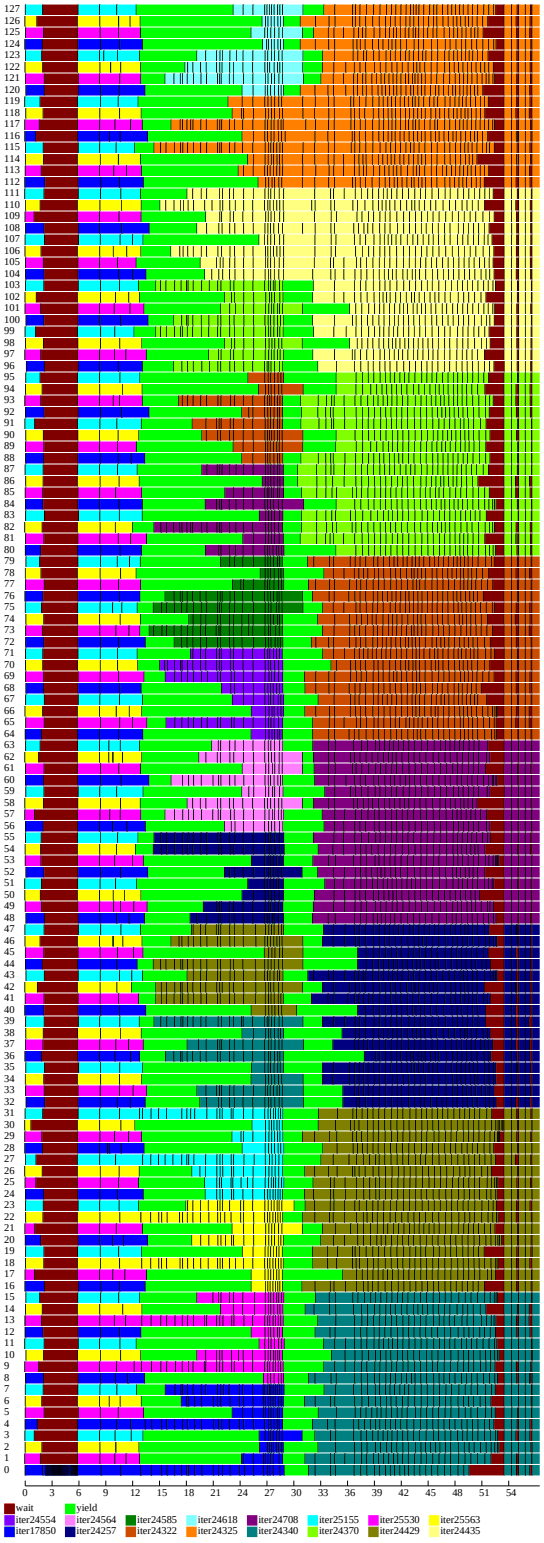

図 10.31　DMI（thread-move）において，非同期的な最短路計算を再構成した場合の各スレッドの振る舞い．

- rescale，thread-move，half-process のどのプログラミングモデルでも，有限要素法や Web グラフ解析などの**非定型で応用的なアプリケーションに対して，利用可能なノード数の増減に対応して並列度を効果的に増減**させることができる．
- **性能は rescale>thread-move>half-process-move** である．とくに，thread-move/half-process-move では，1 プロセッサに複数のスレッド/half-process を割り当てる場合のオーバヘッドが無視できず，オーバヘッドの削減がきわめて重要な課題である．このオーバヘッドは，handler スレッド/handler half-process の処理がボトルネックになっていることに起因しているため，handler スレッド/handler half-process を複数化することによって改善できる可能性がある．
- **プログラマビリティは half-process-move>thread-move>rescale** である．とくに，half-process-move では，DMI_yield() 関数を 1 行追加するだけで，再構成に対応しないプログラムを再構成に対応させることができる．一方で，rescale では，どのデータをチェックポイント/リスタートするべきかを判断するためにはプログラムの内容の正確な分析が必要であり，プログラマビリティは低い．また，そもそも rescale は SPMD 型の反復計算しか記述できないという欠点もある．
- DMI では，グローバルアドレス空間のコヒーレンシがプロセスの非同期的な参加/脱退を越えて維持されているため，**グローバルアドレス空間モデルで記述された同期をともなわない並列計算に対しても，プロセスを自由なタイミングで参加/脱退させる**ことができる．

第 11 章

結論

11.1 まとめ

並列分散アプリケーションの適用領域と利用機会が増大するにつれて，並列分散プログラミング処理系に求められる要請も多様化している．なかでも，本研究では，(1) 非定型な並列計算を性能を落とすことなく簡単に記述できること，(2) 再構成可能な並列計算を簡単に記述できることを目標として，PGAS モデルに基づく並列分散プログラミング処理系 DMI（Distributed Memory Interface）を提案して実装し，評価した．

本研究の第 1 の目標は，非定型な並列科学技術計算を性能を落とすことなく簡単に記述できるようにすることである．並列プログラム開発にとっては，性能と性能最適化の見通しのよさが第一義的に重要である．よって，DMI では，見通しのよい強力な性能最適化によってメッセージパッシングモデルと同等の性能を引き出せるという条件下で，できるかぎりプログラマビリティを高めることを設計方針とした．そこで，本研究ではまず，メッセージパッシングモデル，ローカルビュー型のグローバルアドレス空間モデル，グローバルビュー型のグローバルアドレス空間モデルの 3 種類の並列分散プログラミングモデルを，性能のよさ，性能最適化の自由度と見通しのよさ，非定型な並列計算に対するプログラマビリティ，再構成可能な並列計算に対するプログラマビリティという 4 つの観点から比較して，以下の事実を明らかにした：

- メッセージパッシングモデルよりもグローバルビュー型のグローバルアドレス空間モデルの方が，非定型な並列計算に対するプログラマビリティも再構成可能な並列計算に対するプログラマビリティも高い．
- read/write が内部的に引き起こす通信をわかりやすく強力に制御できるような API を設計しさえすれば，グローバルビュー型のグローバルアドレス空間モデルであっても，メッセージパッシングモデルに匹敵する性能と性能最適化を達成できると考えられる．しかし，著者の知るかぎり，実際にそのような API を設計している PGAS 処理系は存在しない．

上記の 2 点を根拠として，DMI では，グローバルビュー型のグローバルアドレス空間モデルを採用

し，read/write が内部的に引き起こす通信をわかりやすく強力に最適化できるような API を設計することを目標とした．具体的な API としては，任意のコヒーレンシ粒度でグローバルアドレス空間を確保する API，アクセスローカリティを柔軟に最適化するための選択的キャッシュ read/write，非同期 read/write，離散アクセスのグルーピング，ユーザ定義のアトミック命令などを提案した．さらに，非定型な並列計算に関して，グローバルビュー型のグローバルアドレス空間モデルに基づいて並列計算を記述しつつも，内部的にはメッセージパッシングモデルと同等の通信しか起こさないような API として，read-write-set を提案した．

以上のような API を利用して，11 種類のさまざまなアプリケーションを記述してみたところ，DMI の性能とプログラマビリティに関して以下の事実がわかった：

- 有限要素法による応力解析や大規模な Web グラフ解析などの非定型なアプリケーションでは，DMI のプログラム行数は MPI のプログラム行数より有意に短く，DMI のプログラマビリティが高いことを確認できた．
- MPI との性能を比較したところ，有限要素法による応力解析では mpich2>DMI>OpenMPI であり，大規模な Web グラフのページランク計算や最短路計算では DMI>mpich2>OpenMPI であった．さらに，DMI では，PGAS 処理系としての単方向通信の特徴を活かして，非同期的なアルゴリズムによって最短路計算を記述することでさらに性能を改善することができた．さまざまなアプリケーションに対する性能比較の結果を総合すると，DMI は，mpich2 と同等で，OpenMPI よりも高い性能を達成できた．
- DMI は，MPI と比較して Broadcast や Allreduce などの集合通信が遅く，集合通信の最適化が DMI の性能改善にとって重要であることがわかった．

本研究の第 2 の目標は，再構成可能な並列計算を簡単に記述できるようにすることである．そこで DMI では，もっとも基礎となる要素技術として，プロセスが非同期的に参加/脱退できるグローバルアドレス空間のコヒーレンシプロトコルを実装した．そのうえで，再構成可能な並列計算のためのプログラミングモデルとして，rescale，thread-move，half-process-move の 3 種類を提案した．それぞれの特徴は以下のとおりである：

rescale 再構成にともなってスレッドを増減させることによって，つねに 1 プロセッサあたり 1 スレッドが割り当てられるようにするモデルである．そのため性能はよい．しかし，適当なイテレーション数ごとにデータをチェックポイント/リスタートするようにプログラムを記述する必要があり，プログラマビリティは低い．また，記述できるプログラムが，SPMD 型の同期的な反復計算に限定されるという欠点もある．

thread-move プログラマは大量のスレッドを生成しておくだけでよく，あとは処理系が，透過的なスレッド移動によって，それら大量のスレッドを利用可能なノードに動的にマッピングしてくれる．よって，プログラマが再構成を意識しなくてもよいという点ではプログラマビリティは高いが，安全なスレッド移動を実現させるために理解しにくいプログラミング制約が存在する．また，1 プロセッサあたり複数のスレッドを割り当てることによる性能劣化が起きる．な

お，thread-move では，スレッド移動のための要素技術として，アドレス空間のサイズに制限されない新たなスレッド移動の手法として random-address を提案し，その最適性を証明している．

half-process-move スレッドとプロセスの「中間」の機能を持つ新たなカーネルプリミティブとして half-process を導入することで，thread-move におけるプログラミング制約を完全に撤廃し，真に透過的なスレッド移動を実現する．そのためプログラマビリティはきわめて高いが，1 プロセッサあたり複数スレッドを割り当てることによる性能低下は依然として起きる．なお，half-process は，プロセス間のデータ共有を簡単かつ高速に実現するための汎用的なカーネルプリミティブとして設計されており，真に透過的なスレッド移動にかぎらず，柔軟なハイブリッドプログラミングや並列分散プログラミング処理系の開発者の負担減などの応用可能性を持っている．

以上の 3 種類のプログラミングモデルを使って，有限要素法による応力解析や大規模な Web グラフ解析などの実用的な並列反復計算を記述して，その性能とプログラマビリティを評価した結果，以下の事実がわかった：

- いずれのプログラミングモデルでも，非定型で応用的なアプリケーションに対して，利用可能なノード数の増減に対応して並列度を効果的に増減させることができた．
- プログラマビリティは half-process-move>thread-move>rescale だった．とくに，half-process-move では，DMI_yield() 関数を 1 行追加するだけで再構成に対応しないプログラムを再構成に対応させることができた．
- 性能は，rescale>thread-move>half-process-move だった．とくに，thread-move と half-process-move では，1 プロセッサに複数のスレッド/half-process を割り当てた場合のオーバヘッドが無視できず，オーバヘッドの削減がきわめて重要な課題であることがわかった．このオーバヘッドは，handler スレッド/half-process の処理がボトルネックになっていることに起因しているため，handler スレッド/half-process を複数化することによって改善できる可能性がある．

最後に，非同期的なアルゴリズムで記述された Web グラフの最短路計算について，ノードを自由なタイミングで参加/脱退させても並列計算が正しく継続できることを確認した．このように，実行中のノードの同期をともなうことなく，グローバルアドレス空間に対して自由なタイミングでノードを参加/脱退させることができる処理系は，DMI がはじめてである．

11.2 今後の課題：より高生産な並列分散プログラミング処理系の開発

本研究の目的は，見通しのよい強力な性能最適化によってメッセージパッシングモデルと同等の性能を引き出せるという条件下で，できるかぎりプログラマビリティの高い並列分散プログラミング処理系を作ることである．DMI では，グローバルビュー型のグローバルアドレス空間モデルを採用し，グローバルアドレス空間に対する read/write を強力に最適化できる API を設計することで，MPI の性能を

妥協することなくプログラマビリティを高めることに成功した．ところが，DMI を基盤とすることで，**MPI の性能を妥協することなく，さらにプログラマビリティの高い並列分散プログラミング処理系**を設計することが可能であると考えている．

DMI では，たしかにグローバルビュー型のグローバルアドレス空間モデルに基づいてプログラムを記述できるが，グローバルアドレス空間を自由自在に read/write することは想定されていない．かわりに，グローバルアドレス空間上のできるかぎり大きい範囲のデータをいったんローカルアドレス空間に read し，できるかぎりローカルアドレス空間上で計算を行い，ローカルアドレス空間上の計算結果を一気にグローバルアドレス空間に対して write するような記述方法が基本となる．したがって，DMI は，**2.1.4.2** 節で述べたように，たしかに「グローバルインデックスによってグローバルアドレス空間上のデータを read/write できる」という点では便利であるが，計算自体はローカルアドレス空間上で進める必要があるため，「計算はグローバルインデックスでは記述できず，ローカルインデックスによって記述せざるをえない」という点では不便である．端的に言えば，DMI は，グローバルアドレス空間上のデータをとり扱うために，いったんローカルアドレス空間にデータを読み込まなければならないという点で，まだプログラマビリティが低い．理想的には，プログラマは，ローカルアドレス空間をいっさい使うことなく，すべての計算をグローバルアドレス空間のうえで記述できることが望ましい．**2.1.4.2** 節で導入した用語を使うならば，DMI のような「グローバルアドレス空間を極力アクセスしない方法」ではなく，「グローバルアドレス空間を自由にアクセスする方法」で記述できることが望ましい．そこで，現在，「グローバルアドレス空間を自由にアクセスする方法」で記述できるような並列分散プログラミング処理系を，DMI を基盤レイヤとして開発している．

しかし，ここで問題となるのは，**2.1.4.3** 節で述べたように，通常の「グローバルアドレス空間を自由にアクセスする方法」では，性能よく実行するのが難しく，性能最適化の見通しが悪いという点である．第 1 に，性能に関しては，「グローバルアドレス空間を自由にアクセスする方法」で記述されているコードだけからでは，処理系がグローバルアドレス空間へのアクセスをどのように集約すればよいかを判断できない場合が多いため，通信が不必要に細分化されてしまう場合が多く，性能よく実行することが非常に難しい．第 2 に，性能最適化に関しては，グローバルアドレス空間に対してあまりに透過的にアクセスできてしまうため，何を記述したときにどのような通信が内部的に起きるのかを非常に把握しにくく，性能最適化の見通しが悪い．

これらの問題を解決するため，現在開発している並列分散プログラミング処理系では，**グローバルインデックスに基づいて記述できるものの，どの場所でグローバルアドレス空間に対する read/write を発生させるのかはすべてプログラマに明示させる**というアプローチをとる．具体的には，図 11.1 に示すように，グローバルアドレス空間に対するアクセスを行うコードは，sync ブロックと呼ばれるブロックのなかに記述する．そして，sync ブロックの先頭で，その sync ブロックのなかで read/write するグローバルアドレスの集合を記述する．図 11.1 の例では，グローバルアドレス領域 $[srcaddr, srcaddr+1000)$ とグローバルアドレス領域 $[dstaddr, dstaddr+1000)$ を指示している．すると，処理系は，それらのグローバルアドレスの各集合が，その sync ブロックのなかで read されるのか write されるのかを自動的に解析したうえで，以下のような処理を行う DMI のコードを生成する：

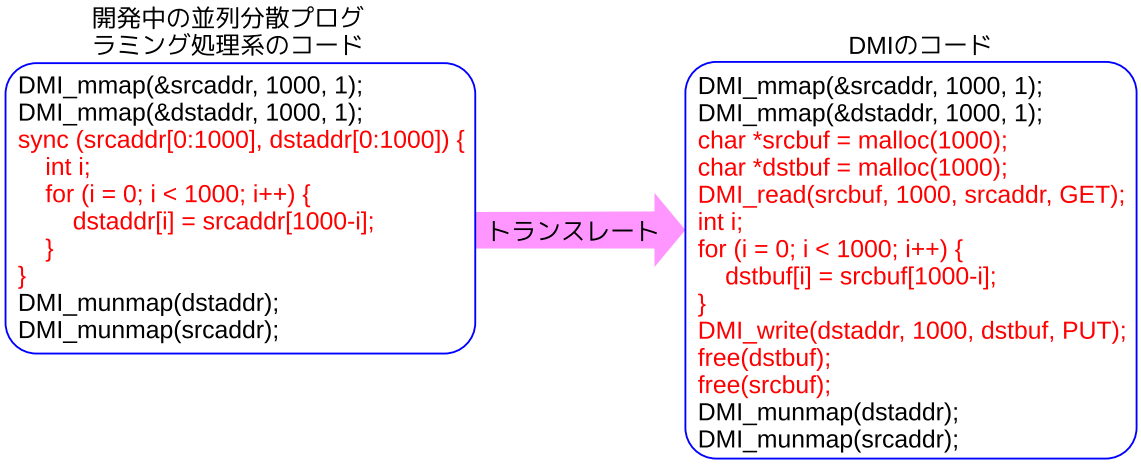

図 11.1　DMI の上位レイヤとして開発中の並列分散プログラミング処理系で記述したプログラム.

(1) read/write されるグローバルアドレスの各集合について，対応するローカルアドレス空間のバッファを用意する.

(2) read されるグローバルアドレスについて，グローバルアドレス空間からローカルアドレス空間のバッファにデータを read する.

(3) sync ブロックのなかのコードを実行する．ただし，sync ブロックのなかに記述されているグローバルアドレス空間への read/write は，対応するローカルアドレス空間のバッファへの read/write にすり替える.

(4) write されるグローバルアドレスについて，ローカルアドレス空間のバッファのデータをグローバルアドレス空間に write する.

このように，この処理系では，プログラマはグローバルインデックスを使ってプログラムを記述でき，ローカルアドレス空間を管理する必要がないため，DMI よりもプログラマビリティが高い．また，sync ブロックのなかで read/write されるグローバルアドレスをすべて明示的に記述しておくと，read されるデータは sync ブロックの先頭で集約的に read され，write されるデータは sync ブロックの最後で集約的に write されるという設計になっているため，プログラマから見て内部的に起きる通信がわかりやすい．さらに，sync ブロックの範囲を伸縮させることによって通信の集約の度合いを自由に変更できるため，性能最適化の見通しもよい.

このように，DMI を基盤として，現在の HPC 分野のデファクトスタンダードである MPI の実行時性能と性能最適化の強力さを妥協することなく，よりプログラマビリティの高い並列分散プログラミング処理系を開発していく必要がある.

参考文献

[1] Amazon EC2 [Online]. http://aws.amazon.com/ec2/.

[2] Google App Engine [Online]. http://code.google.com/intl/appengine/.

[3] Himeno Benchmark [Online]. http://accc.riken.jp/HPC_e/himenobmt_e.html.

[4] Linux Manpages [Online]. http://linuxmanpages.com/.

[5] Memcached [Online]. http://memcached.org/.

[6] Parallel and Distributed Programming 2010 [Online]. http://www.logos.ic.i.u-tokyo.ac.jp/~tau/lecture/parallel_distributed/2010/.

[7] T2K [Online]. http://www.cc.u-tokyo.ac.jp/.

[8] TCMalloc [Online]. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[9] Top500 [Online]. http://www.top500.org/.

[10] TORQUE Resource Manager [Online]. http://www.clusterresources.com/products/torque-resource-manager.php.

[11] TSUBAME2.0 [Online]. http://www.gsic.titech.ac.jp/en.

[12] Windows Azure [Online]. http://www.microsoft.com/windowsazure/.

[13] 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/.

[14] Juan A.Lorenzo, Julio L.Albin, Tomas F.Pena, Francisco F.Rivera, and David E.Singh. An Inspector/Executor Based Strategy to Efficiently Parallelize N-Body Simulation Programs on Shared Memory Systems. *Proceedings of the 6th International Symposium on Parallel and Distributed Computing*, Jul 2007.

[15] Christiana Amza, Alan L.Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Weimin Yu Ramakrishnan Rajamony, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, Vol. 29, No. 2, pp. 18–28, Feb 1996.

[16] Cristiana Amza, Alan Cox, Karthick Rajamani, and Willy Zwaenepoe. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. *Proceedings of the 6th ACM SIGPLAN symposium on Principles and practice of parallel programming*, Vol. 32, No. 7, pp. 90–99, Jul 1997.

[17] Thara Angskun, George Bosilca, Graham E.Fagg, Edgar Gabriel, and Jack J.Dongarra.

Performance analysis of MPI collective operations. *Cluster Computing*, Vol. 10, No. 2, pp. 127–143, Jun 2007.

[18] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System. *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pp. 496–510, 1999.

[19] Gabriel Antoniu and Christian Perez. Using Preemptive Thread Migration to Load-Balance Data-Parallel Applications. *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pp. 117–124, 1999.

[20] A.Petitet, R.C.Whaley, J.Dongarra, and A.Cleary. HPL-A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Technical report, Innovative Computing Laboratory, University of Tennessee Computer Science Department, Sep 2008.

[21] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D.Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, Vol. 53, No. 4, pp. 50–58, Apr 2010.

[22] Rafik A.Salama and Ahmed Sameh. *Potential Performance Improvement of Collective Operations in UPC.* John von Neumann Institute for Computing, 2007.

[23] Fabrizio Baiardi, Gianmarco Dobloni, Paolo Mori, and Laura Ricci. Hive: Implementing a Virtual distributed Shared Memory in Java. *Proceedings of Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pp. 169–172, 2000.

[24] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. *International Journal of High Performance Computing Applications*, Vol. 24, No. 1, pp. 49–57, Feb 2010.

[25] Ayon Basumallik and Rudolf Eigenmann. Towards Automatic Translation of OpenMP to MPI. *Proceedings of the 19th annual International Conference on Supercomputing*, pp. 189–198, 2005.

[26] B.L.Chamberlain, D.Callahan, and H.P.Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, Vol. 21, No. 3, pp. 291–312, Aug 2007.

[27] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, Vol. 37, No. 1, pp. 55–69, Aug 1996.

[28] Ron Brightwell and Kevin Pedretti. Optimizing Multi-core MPI Collectives with SMARTMAP. *2009 International Conference on Parallel Processing Workshops*, pp. 370–

377, Sep 2009.

[29] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–12, Nov 2008.

[30] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stephanie Moreaud. Cache-Efficient, Intranode, Large-Message MPI Communication with MPICH2-Nemesis. *2009 International Conference on Parallel Processing*, pp. 462–469, Sep 2009.

[31] Darius Buntinas, Guillaume Mercier, and William Gropp. Data Transfers between Processes in an SMP System: Performance Study and Application to MPI. *2006 International Conference on Parallel Processing*, pp. 487–496, Aug 2006.

[32] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem. *6th IEEE International Symposium on Cluster Computing and the Grid*, pp. 521–530, May 2006.

[33] Jennifer Burge, Parthasarathy Ranganathan, and Janet L. Wiener. Cost-aware Scheduling for Heterogeneous Enterprise Machines (CASH'EM). *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pp. 481–487, Sep 2007.

[34] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems*, Vol. 25, pp. 599–616, 12 2008.

[35] B.W. カーニハン, D.M. リッチー, 石田晴久. プログラミング言語 C 第 2 版 ANSI 規格準拠. 共立出版, Jun 1989.

[36] Xiao-Chuan Cai and Marcus Sarkis. A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems. *SIAM Journal on Scientific Computing*, Vol. 21, No. 2, pp. 792–797, 9 1999.

[37] David Callahant, Bradford L.Chamberlaint, and Hans P.Zimaj. The Cascade High Productivity Language. *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pp. 52–60, Apr 2004.

[38] C.Carothers and B.Szymanski. Linux Support for Transparent Checkpointing of Multithreaded Programs. *Dr. Dobbs Journal*, Vol. 15, No. 8, pp. 45–60, Aug 2002.

[39] Lei Chai, A. Hartono, and D. K. Panda. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. *2006 IEEE International Conference on In Cluster Computing*, pp. 1–10, Sep 2006.

[40] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C.Hsieh, Deborah A.Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E.Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, Vol. 26, ,

Jun 2008.

[41] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 519–538, Oct 2005.

[42] V. Chaudhary and H.Jiang. Techniques for Migrating Computations on the Grid. *Engineering the Grid: Status and Perspective*, pp. 399–415, Jan 2006.

[43] Barton Christopher, Cascaval Clin, Almasi George, Zheng Yili, Farreras Montse, Chatterje Siddhartha, and Amaral Jose Nelson. Shared memory programming for large scale machines. *ACM SIGPLAN Notices*, Vol. 41, No. 6, pp. 108–117, Jun 2006.

[44] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansenf, Eric Julf, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation*, Vol. 2, pp. 273–286, 2005.

[45] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John Mellor-crummey. 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.

[46] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, Francois Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanty, YiYi Yao, , and Daniel Chavarria-Miranda. 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.

[47] Jonathan Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, Vol. 11, No. 4, pp. 29–41, Jul 2009.

[48] Guojing Cong, George Almasi, and Vijay Saraswat. Fast PGAS Implementation of Distributed Graph Algorithms. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010.

[49] David Cronk, Matthew Haines, and Piyush Mehrotra. Thread Migration in the Presence of Pointers. *Proceedings of the 30th Hawaii International Conference on System Sciences: Software Technology and Architecture*, Vol. 1, pp. 292–302, 1997.

[50] Kaushik Datta1, Dan Bonachea1, and Katherine Yelick1. Titanium Performance and Potential: An NPB Experimental Study. *18th International Workshop on Languages and Compilers for Parallel Computing*, Vol. 4339, pp. 200–214, 2006.

[51] D.Bailey, E.Barszcz, J.Barton, D.Browning, R.Carter, L.Dagum, R.Fatoohi, S.Fineberg, P.Frederickson, T.Lasinski, R.Schreiber, H.Simon, V.Venkatakrishnan, and S.Weeratunga. THE NAS PARALLEL BENCHMARKS. Technical report, RNR-94-007, Mar 1994.

[52] Emery D.Berger, Kathryn S.Mckinley, Robert D.Blumofe, and Paul R.Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*, pp. 117–128, Dec 2000.

[53] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of the 6th conference on Symposium on Opearting Systems Design Implementation*, Vol. 6, , Dec 2004.

[54] Travis Desell, Kaoutar E.Maghraoui, and Carlos A.Varela. Malleable Components for Scalable High Performance Computing. *Proceedings of HPDC'15 Workshop on HPC Grid programming Environments and Components*, pp. 37–44, Jun 2006.

[55] Travis Desell, Kaoutar El Maghraoui, and Carlos A. Varela. Malleable Applications for Scalable High Performance Computing. *Cluster Computing*, Vol. 10, No. 3, pp. 323–337, Sep 2007.

[56] Bozhidar Dimitrov and Vernon Reg. Arachne: a portable threads system supporting migrant threads on heterogeneous network farms. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, pp. 459–469, May 1998.

[57] Cong Du and Xian-He Sun Sun. MPI-Mitten: Enabling Migration Technology in MPI. *IEEE International Symposium on Cluster Computing and the Grid*, pp. 11–18, May 2006.

[58] Cong Du, Xian-He Sun, and Kasidit Chanchio. HPCM: A Pre-compiler Aided Middleware for the Mobility of Legacy Code. *5th IEEE International Conference on Cluster Computing*, pp. 180–187, Dec 2003.

[59] Jason Duell. The design and implementation of Berkeley Lab's linuxcheckpoint/restart. Technical report, Ernest Orlando Lawrence Berkeley NationalLaboratory, Apr 2005.

[60] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-VLM:Remote Memory Paging for Software Distributed Shared Memory. *The 13th International Parallel Processing Symposium*, Apr 1999.

[61] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, Jun 2010.

[62] Tarek El-Ghazawi and Francois Cantonnet. UPC Performance and Potential: A NPB Experimental Study. *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–26, Nov 2002.

[63] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 2.2. Technical report, Message Passing Interface Forum, Sep 2009.

[64] Seiji Fujino, Maki Fujiwara, and Masahiro Yoshida. BiCGSafe Method Based on Minimization of Associate Residual (in Japanese). *Transactions of the Japan Society for Computational*

Engineering and Science, Vol. 8, pp. 145–152, 2006.

[65] Dragan Bosnacki Gerard J. Holzmann. Multi-Core Model Checking with SPIN. *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 220–227, Mar 2007.

[66] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers. *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pp. 1–9, Nov 2005.

[67] G.J.Holzmann, D.Peled, and M.Yannakakis. On Nested DepthFirst Search. *Proceedings of the 2nd SPIN Workshop*, Aug 1996.

[68] G.J.Holzmann, P.Godefroid, and D.Pirottin. Coverage Preserving Reduction Strategies for Reachability Analysis. *Proceedings of the 12th International Symposium on Protocol Specification, Testing, and Verication*, pp. 349–364, Jun 1992.

[69] Etienne Godard, Sanjeev Setia, and Elizabeth L.White. DyRecT: Software support for adaptive parallelism on NOWs. *15th IPDPS Workshops on Parallel and Distributed Processing*, pp. 1144–1151, May 2000.

[70] Brice Goglin. High Throughput Intra-Node MPI Communication with Open-MX. *2009 Parallel, Distributed and Network-based Processing*, pp. 173–180, Feb 2009.

[71] Jan Gotz, Klaus Iglberger, Markus Sturmer, and Ulrich Rude. Direct Numerical Simulation of Particulate Flows on 294912 Processor Cores. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2010.

[72] XcalableMP Specification Working Group. XcalableMP Application Program Interface Version 1. Technical report, Center for Computational Sciences, University of Tsukuba, Nov 2009.

[73] G.Wrzesinska, R.V.van Nieuwport, J.Maassen, and Henri E.Bal. Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid. *Proceedings of 19th International Parallel and Distributed Processing Symposium*, Apr 2005.

[74] Erik Hagersten, Anders Landin, and Seif Haridi. DDM — a Cache-Only Memory Architecture. *Multiprocessor performance measurement and evaluation*, pp. 304–314, 1995.

[75] James H.Anderson and Yong-Jik Kim. Shared-memory Mutual Exclusion: Major Research Trends Since 1986. *Distributed Computing*, Vol. 16, No. 2–3, pp. 75–110, Sep 2003.

[76] Kentaro Hara and Kenjiro Taura. A Global Address Space Framework for Irregular Applications. *High Performance Distributed Computing*, Jun 2010.

[77] Paul Hilfinger, Dan Bonachea, David Gay, Susan Graham, Ben Liblit, Geoff Pike, and Katherine Yelick. Titanium Language Reference Manual. Technical report, University of California at Berkeley, Aug 2006.

[78] Torsten Hoefler, Greg Bronevetsky, Brian Barrett, Bronis R.De Supinski, and Andrew Lumsdaine. Efficient MPI Support for Advanced Hybrid Programming Models. *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface*, pp. 50–61, Sep 2010.

[79] Weiwu Hu, Weisong Shi, Zhimin Tang, and Zhiyu Zhou. JIAJIA: An SVM System Based on a New Cache Coherence Protocol. Technical report, Center of High Performance Computing Institute of Computing Technology Chinese Academy of Sciences, Jan 1998.

[80] Chao Huang, Orion Lawlor, and L.V.Kale. Adaptive MPI. *16th International Workshop on Languages and Compilers for Parallel Computing*, pp. 306–322, Oct 2003.

[81] Chao Huang, Gengbin Zheng, Laxmikant Kale, and Sameer Kumar. Performance Evaluation of Adaptive MPI. *11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 12–21, Mar 2006.

[82] Ting-Lu Huang. Fast and Fair Mutual Exclusion for Shared Memory Systems. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 224–231, 1999.

[83] Wei Huang, Matthew J.Koop, Qi Gao, and Dhabaleswar K.Panda. Virtual Machine Aware Communication Libraries for High Performance Computing. *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pp. 1–12, Nov 2007.

[84] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K.Panda. A Case for High Performance Computing with Virtual Machines. *Proceedings of the 20th annual international conference on Supercomputing*, Jun 2006.

[85] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, Vol. 42, No. 1, pp. 71–87, Jul 1998.

[86] Jason Duell. Pthreads or Processes: Which is Better for Implementing Global Address Space languages?

[87] Gerard J.Holzmann. A Stack-Slicing Algorithm for Multi-Core Model Checking. *Electronic Notes in Theoretical Computer Science*, Vol. 198, No. 1, pp. 3–16, Feb 2008.

[88] Gerard J.Holzmann and Dragan Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, Vol. 33, No. 10, pp. 659–674, Jul 2007.

[89] Hai Jiang and Vipin Chaudhary. Compile/Run-Time Support for Thread Migration. *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pp. 58–66, 2002.

[90] Hai Jiang and Vipin Chaudhary. MigThread: Thread Migration in DSM Systems. *International Conference on Parallel Processing*, p. 581, 2002.

[91] Hai Jiang and Vipin Chaudhary. On Improving Thread Migration: Safety and Performance. *Proceedings of the 9th International Conference on High Performance Computing*, pp. 474–484, 2002.

[92] Hai Jiang and Vipin Chaudhary. Thread Migration/Checkpointing for Type-Unsafe C Programs. *International conference on high performance computing*, Vol. 2913, pp. 469–479, Nov 2003.

[93] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K.Panda. LiMIC: Support for High-Performance MPI Intra-node Communication on Linux Cluster. *2005 International Conference on Parallel Processing*, pp. 184–191, Jun 2005.

[94] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems. *2007 IEEE International Conference on Cluster Computing*, pp. 446–451, Sep 2007.

[95] Matchy J.M.Ma, Cho-Li Wang, and Francis C.M.Lau. Delta Execution: A preemptive Java thread migration mechanism. *Cluster Computing*, Vol. 3, pp. 83–94, 2000.

[96] Theodore Johnson. A Performance Comparison of Fast Distributed Synchronization Algorithms. *International Conference on Parallel Processing*, pp. 258–264, 1994.

[97] Ueng Jyh-Chang, Shieh Ce-Kuen, Mac Su-Cheong, Lai An-Chow, and Liang Tyng-Yue. Multi-Threaded Design for a Software Distributed Shared Memory Systems. *IEICE Transactions on Information and Systems*, Vol. E82-D, No. 12, pp. 1512–1523, Dec 1999.

[98] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vSnoop: Improving TCP Throughput in Virtualized Environments via Acknowledgement Offload. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2010.

[99] G Karypis and K Schloege. ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 3.1. Technical report, Department of Computer Science and Engineering, Army HPC Research Center, Aug 2003.

[100] George Karypis and Vipin Kumar. Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, Vol. 48, No. 1, pp. 96–129, Jan 1998.

[101] John K.Bennett, John B.Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. *Proceedings of the 2nd ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, Vol. 25, No. 3, pp. 168–176, Mar 1990.

[102] Jian Ke and Evan Speight. Tern: Thread Migration in an MPI Runtime Environment. Technical report, Cornell, Nov 2001.

[103] Vijay K.Naik, Samuel P.Midkiff, and Jose E.Moreira. A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems. *1997 ACM/IEEE conference on Supercomputing*,

pp. 1–19, Nov 1997.

[104] Pradeep K.Sinha. *Distributed Operating Systems : Concepts and Design*. IEEE COMPUTER SOCIETY PRESS,IEEE PRESS, Dec 1996.

[105] K.Thitikamol and P.Keleher. Thread migration and communication minimization in DSM systems. *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, Vol. 87, pp. 487–497, 3 1999.

[106] K.Vaidyanathan, S.Narravula, and D.K.Panda. DDSS: A Low-Overhead Distributed Data Sharing Substrate for ClusterBased Data-Centers over Modern Interconnects. *International Conference on High Performance Computing*, Dec 2006.

[107] Oren Laadan and Jason Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. *Proceedings of the USENIX Annual Technical Conference*, Jun 2007.

[108] Ping Lai, Sayantan Sur, and Dhabaleswar K. Panda. Designing Truly One-sided MPI-2 RMA Intra-node Communication on Multi-core Systems. *International Computing Conference*, Vol. 25, pp. 3–14, May 2010.

[109] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, Jul 1978.

[110] Francis Lau, Matchy Ma, Cho li Wang, and Benny Cheung. Cluster Computing with Single Thread Space.

[111] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, Vol. 7, No. 4, pp. 321–359, Nov 1989.

[112] Song Li, Yu Lin, and Michael Walker. Region-based Software Distributed Shared Memory, May 2000.

[113] Shuang Liang, Ranjit Noronha, and Dhabaleswar K.Panda. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. *2005 IEEE International Conference on Cluster Computing*, pp. 1–10, Sep 2005.

[114] Xiaofei Liao, Yifan Yue, Hai Jin, and Haikun Liu. LAOVM: Lightweight Application-Oriented Virtual Machine for Thread Migration. *2009 Eigth IEEE/ACIS International Conference on Computer and Information Science*, pp. 882–887, Jun 2009.

[115] Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pp. 78–85, Jul 2010.

[116] Wai-Hung Liu and Andrew H.Sherman. 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, Apr 1976.

[117] Zhiqiang Liu, Junqiang Song, Kaijun Ren, Fen Xu, and Xiaoling Qu. A Systemic Strategy

for Tuning Intra-node Collective Communication on Multicore Systems. *Proceedings of the 2009 Fourth International Conference on Frontier of Computer Science and Technology*, pp. 14–21, Dec 2009.

[118] Kirk L.Johnson, M.Frans Kaashoek, and Deborah A.Wallach. CRL: High-Performance All-Software Distributed Shared Memory. *Proceedings of the 15th Symposium on Operating Systems Principles*, Vol. 29, No. 5, pp. 213–228, Mar 1995.

[119] Mitchell L.Neilsen and Masaaki Mizuno. A Dag-Based Algorithm for Distributed Mutual Exclusion. *Proceedings of the 11th International Conference on Distributed Computing Systems*, pp. 354–360, May 1991.

[120] Giorgia Lodi, Vittorio Ghini, Fabio Panzieri, and Filippo Carloni. An Object-based Fault-Tolerant Distributed Shared Memory Middleware. Technical report, Department of Computer Science University of Bologna, Jul 2007.

[121] L.Peng, W.F.Wong, M.D.Feng, and C.K.Yuen. SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters. *2nd IEEE International Conference on Cluster Computing*, pp. 243–249, Dec 2000.

[122] Mathieu Luisier. A Parallel Implementation of Electron-Phonon Scattering in Nanoelectronic Devices up to 95k Cores. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2010.

[123] L.Vaquero, L.Rodero-Marino, J.Caceres, and M.Lindner. A Break in the Clouds : Towards a Cloud Definition. *SIGCOMM Computer Communication Review*, pp. 137–150, 2009.

[124] Armbrust M., A.Fox, R.Griffith, A.D.Joseph, R.Katz, A.Konwinski, G.Lee, D.A.Patterson, A.Rabkin, I.Stoica, and M.Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2 2009.

[125] Mamoru Maekawa. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pp. 145–159, May 1985.

[126] Kaoutar El Maghraoui, Travis J.Desell, Boleslaw K.Szymanski, and Carlos A.Varela. Dynamic Malleability in Iterative MPI Applications. *7th IEEE International Symposium on Cluster Computing and the Grid*, pp. 591–598, May 2007.

[127] Kaoutar El Maghraoui, Boleslaw K.Szymanski, and Carlos Varela. An Architecture for Reconfigurable Iterative MPI Applications in Dynamic Environments. *International Conference on Parallel Processing and Applied Mathematics*, Vol. 3911, pp. 258–271, Sep 2006.

[128] Kaoutar El Maghraouia, Travis Desella, Boleslaw K.Szymanskia, James D.Terescob, and Carlos A.Varela. Towards a Middleware Framework for Dynamically Reconfigurable Scietific Computing. *Advances in Parallel Computing*, Vol. 14, pp. 275–301, Jun 2005.

[129] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing.

Proceedings of the 2010 international conference on Management of data, pp. 135–146, 2010.

[130] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec 1995.

[131] Maged M. Michael. Scalable Lock-Free Dynamic Memory Allocation. *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, Vol. 39, No. 6, pp. 35–46, Jun 2004.

[132] Scott Milton. Thread Migration in Distributed Memory Multicomputers. Technical report, Australia National University, 1998.

[133] John M.Mellor-Crummey and Michael L.Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, Vol. 9, No. 1, pp. 21–65, Feb 1991.

[134] Eric Mohr, David A.Kranz, and Robert H.Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, pp. 264–280, Jul 1991.

[135] Frank Mueller. Distributed Shared-Memory Threads: DSM-Threads. *Workshop on Run-Time Systems for Parallel Programming*, pp. 31–40, Apr 1997.

[136] Frank Mueller. On the Design and Implementation of DSM-Threads. *Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 315–324, Jun 1997.

[137] Frank Mueller. Priority Inheritance and Ceilings for Distributed Mutual Exclusion. *IEEE Real-Time Systems Symposium*, pp. 340–349, Dec 1999.

[138] Mohamed Naimi, Michel Trehel, and Andr Arnold. A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal. *Journal of Parallel and Distributed Computing*, Vol. 34, No. 1, pp. 1–13, Apr 1996.

[139] Brian N.Bershad, Matthew J.Zekauskas, and Wayne A.Sawdon. The Midway Distributed Shared Memory System. *Proceedings of the IEEE Compcon Spring 1993*, pp. 528–537, 1993.

[140] Tia Newhall, Dan Amato, and Alexandr Pshenichkin. Reliable Adaptable Network RAM. *Proceedings of IEEE Cluster Conference*, pp. 2–12, Sep 2008.

[141] Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel. Nswap: A Network Swapping Module for Linux Clusters. *Proceedings of Euro-Par'03 International Conference on Parallel and Distributed Computing*, Aug 2003.

[142] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems. *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, Apr 1999.

[143] Jarek Nieplocha, Manojkumar Krishnan, Bruce Palmer, and Vinod Tipparaju. The Global

Arrays User's Manual. Technical report, Pacific Northwest National Laboratory, Jul 2009.

[144] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edo Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, Vol. 20, No. 2, pp. 203–231, 2006.

[145] Jarek Nieplocha, Vinod Tipparaju, Amina Saify, and Dhabaleswar Panda. Protocols and Strategies for Optimizing Performance of Remote Memory Operations on Clusters. *International Parallel and Distributed Processing Symposium*, Vol. 2, pp. 164–173, Apr 2002.

[146] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, Vol. 10, No. 2, pp. 169–189, 1996.

[147] Rob V.van Nieuwpoort, Thilo Kielmann, and Henri E.Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pp. 34–43, Jul 2001.

[148] Nitzan Niv and Assaf Schuster. Transparent Adaptation of Sharing Granularity in MultiView-Based DSM Systems. *Software - Practice and Experience*, Vol. 31, No. 15, pp. 1439–1459, Dec 2001.

[149] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, Vol. 17, No. 2, pp. 1–31, 1998.

[150] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *Proceedings of the 5th symposium on Operating systems design and implementation*, pp. 361–376, Dec 2002.

[151] Scott Pakin and Greg Johnson. Performance Analysis of a User-level Memory Server. *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pp. 249–258, Sep 2007.

[152] Daniel P.Bovet, Marco Cesati, 高橋浩和, 杉田由美子, 清水正明, 高杉昌督, 平松雅巳, 安井隆宏. 詳解 Linux カーネル 第 3 版. オライリー・ジャパン, Feb 2007.

[153] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. *2009 Parallel, Distributed and Network-based Processing*, pp. 427–436, Feb 2009.

[154] Zoran Radovic and Erik Hagersten. Removing the Overhead from Software-Based Shared Memory. *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Nov 2001.

[155] Kerry Raymond. A Tree-Based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems*, Vol. 7, No. 1, pp. 61–77, Feb 1989.

[156] R.Brightwell. Lightweight Kernel Support for Direct Shared Memory Access on a Multi-Core Computer. *Proceedings of the 1st Workshop on Managed Many-Core Systems*, Jun 2008.

[157] Michael R.Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 51–60, 2009.

[158] Michael R.Hines, Jian Wang, and Kartik Gopalan. Distributed Anemone: Transparent Low-Latency Access to Remote Memory. *Proceedings of the International Conference on High Performance Computing*, pp. 509–521, Dec 2006.

[159] Glenn Ricart and Ashok K.Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, Vol. 24, No. 1, pp. 9–17, Jan 1981.

[160] Thomas Roblitz and Frank Mueller. 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.

[161] Eduardo R.Rodrigues, Philippe O.A.Navaux, Jairo Panetta, and Celso L.Mendes. A New Technique for Data Privatization in User-level Threads and its Use in Parallel Application. *Proceedings of the 2010 ACM Symposium on Applied Computing*, Mar 2010.

[162] Hideo Saito and Kenjiro Taura. Locality-aware Connection Management and Rank Assignment for Wide-area MPI. *IEEE International Symposium on Cluster Computing and the Grid 2007*, pp. 249–258, May 2007.

[163] Jose Carlos Sancho, Fabrizio Petrini, Kei Davis, Roberto Gioiosa, and Song Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, Apr 2005.

[164] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Vishal Sahay, and Andrew Lumsdaine. The Lam/Mpi Checkpoint/Restart Framework: System-Initiated Checkpointing. *nternational Journal of High Performance Computing Applications*, Vol. 19, pp. 479–493, 2005.

[165] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A.Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 174–185, Oct 1996.

[166] Scott Schneider, Christos D.Antonopoulos, and Dimitrios S.Nikolopoulos. Scalable Locality-Conscious Multithreaded Memory Allocation. *Proceedings of the 5th international symposium on Memory management*, pp. 84–94, Jun 2006.

[167] Shiwa S.Fu, Nian feng Tzeng, and Zhiyuan Li. Empirical Evaluation of Distributed Mutual Exclusion Algorithms. *International Parallel Processing Symposium*, pp. 255–259, Apr 1997.

[168] Weisong Shi. Heterogeneous Distributed Shared Memory on Wide Area Network. *IEEE TCCA Newsletter*, pp. 71–80, Jan 2001.

[169] Otto Sievert and Henri Casanova. A Simple MPI Process Swapping Architecture for Iterative Applications. *International Journal of High Performance Computing Applications*, Vol. 18, pp. 341–352, Aug 2004.

[170] Dejan S.Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. *ACM Computing Surveys*, Vol. 32, No. 3, pp. 241–299, Sep 2000.

[171] Jimmy Su, Tong Wen, and Katherine Yelick. Compiler and Runtime Support for Scaling Adaptive Mesh Refinement Computations in Titanium. Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley, Jun 2006.

[172] Jimmy Su and Katherine Yelick. Automatic Support for Irregular Computations in a High-Level Language. *19th IEEE International Parallel and Distributed Processing Symposium*, Vol. 1, p. 53b, 2005.

[173] Sathish S.Vadhiyar and Jack J.Dongarra. SRS: A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems. *International Journal of High Performance Applications and Supercomputing*, pp. 291–312, Jun 2003.

[174] Kenjiro Taura. GXP : An Interactive Shell for the Grid Environment. *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp. 59–67, Apr 2004.

[175] Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix:a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 216–229, 2003.

[176] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, Vol. 19, No. 1, pp. 49–66, Feb 2005.

[177] V. Tipparaju, M.Krishnan, J.Nieplocha, G.Santhanaraman, and D.K.Panda. Exploiting Nonblocking Remote Memory Access Communication in Scientific Benchmarks on Clusters. *Proceedings of the International Conference on High Performance Computing*, pp. 248–258, 2003.

[178] Francois Trahay, Elisabeth Brunet, and Alexandre Denis. An Analysis of the Impact of Multi-Threading on Communication Performance. *2009 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–7, May 2009.

[179] U.Kang, Charalampos E.Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System. *9th IEEE International Conference on Data Mining*, pp. 229–238, Dec 2009.

[180] Geoffroy Vallee, Renaud Lottiaux, David Margery, and Christine Morin. Ghost Process: a Sound Basis to Implement Process Duplication, Migration and Checkpoint/Restart in Linux Clusters. *The 4th International Symposium on Parallel and Distributed Computing*, pp.

97–104, Jul 2005.

[181] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L.Scott. Proactive process-level live migration in HPC environments. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–12, Nov 2008.

[182] Nan Wang, Xuhui Liu, Jin He, Jizhong Han, Lisheng Zhang, and Zhiyong Xu. Collaborative Memory Pool in Cluster System. *2007 International Conference on Parallel Processing*, pp. 17–24, Sep 2007.

[183] Boris Weissman, Benedict Gomes, Jurgen W.Quittek, and Michael Holtkamp. Efficient Fine-grain Thread Migration with Active Threads. *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, p. 410, 1998.

[184] Paul Werstein, Xiangfei Jia, and Zhiyi Huang. A Remote Memory Swapping System for Cluster Computers. *Eighth International Conference on Parallel and Distributed Computing*, pp. 75–81, Dec 2007.

[185] Robert W.Numrich, John Reid, and Kieun Kim. 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.

[186] Gosia Wrzesinska, Jason Maassen, and Henri E.Bal. Self-adaptive Applications on the Grid. *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 121–129, Mar 2007.

[187] Jae-Heon Yang and James H.Anderson. A Fast, Scalable Mutual Exclusion Algorithm. *Distributed Computing*, Vol. 9, No. 1, pp. 51–60, Aug 1994.

[188] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. *ACM 1998 Workshop on Java for High-Performance Network Computing*, Vol. 10, No. 11–13, pp. 825–836, Feb 1998.

[189] Jaeheung Yeo, Heon Y.Yeom, Taesoon Park, and Heon Y.Yeom Taesoon Park. An Asynchronous Protocol for Release Consistent Distributed Shared Memory Systems. *Proceedings of the 2000 ACM symposium on Applied computing*, 2000.

[190] Sangho Yi, Derrick Kondo, and Artur Andrzejak. Reducing Costs of Spot Instances via Checkpointing in the Amazon Elastic Compute Cloud. *IEEE 3rd International Conference on Cloud Computing*, pp. 236–243, Jul 2010.

[191] Lamia Youseff and Rich Wolski. Vshmem: Shared-Memory OS-Support for Multicore-based HPC systems. Technical report, Department of Computer Science, University of California, Oct 2009.

[192] Wenzhang Zhu, Cho-Li Wang, and Francis C.M.Lau. Lightweight Transparent Java Thread Migration for Distributed JVM. *International Conference on Parallel Processing*, p. 465, Oct

2003.

[193] Wenzhang Zhu, Cho-Li Wang, and Lau F.C.M. JESSICA2: a distributed Java Virtual Machine with transparent thread migration support. *Fourth IEEE International Conference on Cluster Computing*, pp. 381–388, 2002.

[194] Wenzhang Zhu and Submitted Wenzhang Zhu. JESSICA2: A distributed Java virtual machine with transparent thread migration support. *IEEE International Conference on Cluster Computing*, Sep 2002.

[195] 吉富翔太, 斎藤秀雄, 田浦健次朗, 近山隆. 自動取得したネットワーク構成情報に基づく MPI 集合通信アルゴリズム. 情報処理学会研究報告ハイパフォーマンスコンピューティング, pp. 7–12, Aug 2008.

[196] 原健太朗. 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. 東京大学 卒業論文, Feb 2009.

[197] 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.

[198] 原健太朗, 田浦健次朗, 近山隆. DMI：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース. 情報処理学会論文誌（プログラミング）, Vol. 3, No. 1, pp. 1–40, Mar 2010.

[199] 原健太朗, 中島潤, 田浦健次朗. 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. 情報処理学会論文誌（プログラミング）, Vol. 4, No. 1, pp. 1–40, 2011.

[200] 金田康正. 並列数値処理—高速化と性能向上のために. コロナ社, Apr 2010.

[201] 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.

[202] 田浦健次朗. Phoenix: 動的な資源の増減をサポートする並列計算プラットフォーム. 情報処理学会研究報告ハイパフォーマンスコンピューティング, pp. 135–140, Aug 2001.

[203] 緑川博子, 黒川原佳, 姫野龍太郎. 遠隔メモリを利用する分散大容量メモリシステム DLM の設計と 10GbEthernet における初期性能評価. 情報処理学会論文誌コンピューティングシステム, Vol. 1, No. 3, pp. 136–157, Dec 2008.

[204] 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.

[205] 山本和典, 石川裕. テラスケールコンピューティングのための遠隔スワップシステム Teramem. *Symposium on Advanced Computing Systems and Infrastructures 2009*, May 2009.

発表文献

論文誌

- 原健太朗，中島潤，田浦健次朗．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．情報処理学会論文誌（プログラミング）．Vol.4, No.1．pp.1-40．2011/3
- 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.1-40．2010/3

国際発表（査読あり）

- Kentaro Hara, Kenjiro Taura. A Global Address Space Framework for Irregular Applications. 2010 International ACM Symposium on High Performance Distributed Computing（HPDC2010）. pp.296-299. 2010/6

国際発表（投稿中）

- Kentaro Hara, Kenjiro Taura. Parallel Computational Reconfiguration Based on a PGAS Model. 2011 International ACM Symposium on High Performance Distributed Computing（HPDC2011）. 12 pages（投稿中）.

国内発表（査読あり）

- 原健太朗，塩谷亮太，田浦健次朗．メモリアクセス最適化を適用した汎用プロセッサと Cell の性能比較．先進的計算基盤シンポジウム（SACSIS 2008）．pp.157-166．2008/6

国内発表（査読なし）

- 原健太朗，中島潤，田浦健次朗．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．SWoPP2010．2010/8
- 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．SWoPP2009．2009/8

- 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 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成果報告）．第 9 回 PC クラスタシンポジウム．2009/12
- 原健太朗．ホモロジー検索の並列最適化（クラスタシステム上の並列プログラミングコンテスト非数値計算部門成果報告）．PC クラスタワークショップ in 広島．2009/5

受賞

- 2009 年度 CS 領域奨励賞（プログラミング研究会）．2011/1
- 2009 年 学士卒業論文 東京大学工学部長賞．2009/3

プログラミングコンテスト

- 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 第 2 位．2009/10
- SACSIS2009 併設企画 クラスタシステム上の並列プログラミングコンテスト非数値計算部門 第 1 位．2009/5

謝辞

本研究を進めるにあたって，指導教員である田浦健次朗准教授には，日々緻密なご指導をいただき，DMI の設計，実装，論文執筆にいたるまでたくさんのアドバイスをいただきました．また，国際学会に同行させていただいたり，ごいっしょに帰宅させていただいたり，日々の研究室の業務に協力させていただいたりすることも多く，先生には，研究にかぎらずあらゆる面で相談や議論に乗っていただきました．先生との議論を通じて，さまざまなものごとに対して筋のいい考え方を学ばせていただくことができました．卒論生として配属されてからの 3 年間，手厚いご指導をいただきましたことに心より感謝を申し上げます．

近山隆教授には，発表練習などを通じて，DMI の設計や実装に関する数多くの的確なアドバイスをいただいたほか，研究室の行事日程や備品の調整などでお世話になりました．横山大作助教，ポスドクの柴田剛志さんには，機会あるごとに DMI の設計や実装面での有用なアイディアを提供していただきました．また，M2 の中島潤さん，B4 の藤澤徹さんには，具体的なコーディングにいたるまで何度も相談に乗っていただき，抱えていた疑問をいくつも解決することができました．M1 の加辺友也さんには，クラスタ環境のシステム管理など研究室の仕事で困っているときにはいつも助けていただきました．B4 の堀内美希さんには，研究室のコミュニティを成り立たせるために何度も相談に乗っていただきました．研究室での生活を送るうえで途方にくれてしまうことも何度かありましたが，それでも精神面を保つことができたのは，M2，M1，B4 のみなさんが理解してくれ，そしていつもあたたかく支えてくれたからこそだと本当に痛感しています．秘書の黒田善子さん，荒井美佐緒さんには，出張管理，研究室の備品管理などで大変お世話になりました．学生の負担を減らすためにいつもさりげなくお仕事をこなしてくださっていた黒田さんの姿をいまでも尊敬しております．

この場を借りて，本研究を支えてくださったみなさまに厚くお礼申し上げます．

付録 A

グローバルアドレス空間のコヒーレンシプロトコルのアルゴリズム

ページ *page* に関するコヒーレンシプロトコルを以下に記述する．このプロトコルは，任意の 2 プロセス間が FIFO な通信路で結ばれていることを仮定している．以下の疑似コードにおける *REQ_READ.src* は，*REQ_READ* というメッセージに関連づけられた *src* というデータを意味する．my_rank は各プロセスのランクを示す．また，オーナーからのメッセージのみを順序制御するため，各プロセスは，*seq*≥ 0 なる *seq* をデータに持つメッセージに関しては *seq* の昇順に順序制御してメッセージを受信するものとし，*seq*$= -1$ のメッセージは任意の時点ですぐに受信するものとする．

structure for page {*state, owner, probable, valid, buffer, state_array, seq_array*}
structure for msg {*src, mode, state, buffer, state_array, seq_array*}

001: initialization of *page* whose owner is *owner* at the initial state:
002: **if** *owner* == my_rank **then**
003: *page.state* := DOWN_VALID
004: *page.owner* := TRUE
005: **for each** *rank* in the system **do**
006: *state_array*[*rank*] := INVALID
007: *seq_array*[*rank*] := 0
008: **endfor**
009: *state_array*[my_rank] := DOWN_VALID
010: *page.valid* := 1
011: **else**
012: *page.state* := INVALID
013: *page.owner* := FALSE

```
014:    endif
015:    page.probable := owner
016:
017: read operation for page with buffer and mode:
018:    lock page
019:    if page.state == INVALID
020:       || (page.state == DOWN_VALID && mode == READ_UPDATE)
021:       || (page.state == UP_VALID && (mode == READ_INVALIDATE
022:       || mode == READ_ONCE)) then
023:       REQ_READ.seq := -1
024:       REQ_READ.src := my_rank
025:       REQ_READ.mode := mode
026:       send REQ_READ to page.probable
027:       unlock page
028:       wait for ACK_READ to be received
029:       lock page
030:       page.probable := ACK_READ.src
031:       if ACK_READ.buffer != NIL then
032:          page.buffer := ACK_READ.buffer
033:       endif
034:       page.state := ACK_READ.state
035:    endif
036:    buffer := page.buffer
037:    unlock page
038:
039: when REQ_READ for page is received:
040:    lock page
041:    if page.owner == TRUE then
042:       if page.state_array[REQ_READ.src] == INVALID then
043:          ACK_READ.buffer := page.buffer
044:          if REQ_READ.mode == READ_INVALIDATE then
045:             page.state_array[REQ_READ.src] := DOWN_VALID
046:             page.valid := page.valid + 1
047:          elseif REQ_READ.mode == READ_UPDATE then
048:             page.state_array[REQ_READ.src] := UP_VALID
049:             page.valid := page.valid + 1
```

```
050:          endif
051:       else
052:          ACK_READ.buffer := NIL
053:          if page.state_array[REQ_READ.src] == UP_VALID
054:             && (REQ_READ.mode == READ_INVALIDATE
055:             || REQ_READ.mode == READ_ONCE) then
056:             page.state_array[REQ_READ.src] := DOWN_VALID
057:          elseif page.state_array[REQ_READ.src] == DOWN_VALID
058:                && REQ_READ.mode == READ_UPDATE then
059:             page.state_array[REQ_READ.src] := UP_VALID
060:          endif
061:       endif
062:       ACK_READ.state := page.state_array[REQ_READ.src]
063:       ACK_READ.seq := page.seq_array[REQ_READ.src]
064:       page.seq_array[REQ_READ.src] := page.seq_array[REQ_READ.src]+1
065:       ACK_READ.src := my_rank
066:       send ACK_READ to REQ_READ.src
067:    else
068:       send REQ_READ to page.probable
069:    endif
070:    unlock page
071:
072: write operation for page with buffer and mode:
073:    lock page
074:    if page.owner == TRUE && page.valid == 1 then
075:       page.buffer := buffer
076:    elseif mode == WRITE_REMOTE then
077:       REQ_WRITE.seq := -1
078:       REQ_WRITE.src := my_rank
079:       REQ_WRITE.buffer := buffer
080:       send REQ_WRITE to page.probable
081:       unlock page
082:       wait for ACK_WRITE to be received
083:       lock page
084:       page.probable := ACK_WRITE.src
085:    elseif mode == WRITE_LOCAL then
```

```
086:        if page.owner == FALSE then
087:            REQ_STEAL.seq := −1
088:            REQ_STEAL.src := my_rank
089:            send REQ_STEAL to page.probable
090:            unlock page
091:            wait for ACK_STEAL to be received
092:            lock page
093:            page.probable := my_rank
094:            if ACK_STEAL.buffer != NIL then
095:                page.buffer := ACK_STEAL.buffer
096:                page.state := DOWN_VALID
097:            endif
098:            page.state_array := ACK_STEAL.state_array
099:            page.seq_array := ACK_STEAL.seq_array
100:            page.valid := ACK_STEAL.valid
101:            page.owner := TRUE
102:        endif
103:        page.buffer := buffer
104:        update_and_invalidate(page)
105:    endif
106:    unlock page
107:
108: when REQ_WRITE for page is received:
109:    lock page
110:    if page.owner == TRUE then
111:        page.buffer := REQ_WRITE.buffer
112:        update_and_invalidate(page)
113:        ACK_WRITE.seq := page.seq_array[REQ_WRITE.src]
114:        page.seq_array[REQ_READ.src] := page.seq_array[REQ_READ.src]+1
115:        ACK_WRITE.src := my_rank
116:        send ACK_WRITE to REQ_WRITE.src
117:    else
118:        send REQ_WRITE to page.probable
119:    endif
120:    unlock page
121:
```

```
122: update_and_invalidate(page):
123:     if page.valid != 1 then
124:         for each rank s.t. rank != my_rank
125:                 && page.state_array[rank] == DOWN_VALID do
126:             page.state_array[rank] := INVALID
127:             page.valid := page.valid −1
128:             REQ_INVALIDATE.seq := page.seq_array[rank]
129:             page.seq_array[rank] := page.seq_array[rank] + 1
130:             REQ_INVALIDATE.src := my_rank
131:             send REQ_INVALIDATE to rank
132:         endfor
133:         for each rank s.t. rank != my_rank
134:                 && page.state_array[rank] == UP_VALID do
135:             REQ_VALIDATE.seq := page.seq_array[rank]
136:             page.seq_array[rank] := page.seq_array[rank] + 1
137:             REQ_VALIDATE.src := my_rank
138:             REQ_VALIDATE.buffer := page.buffer
139:             send REQ_VALIDATE to rank
140:         endfor
141:         page.owner := FALSE
142:         unlock page
143:         wait for all ACK_VALIDATE to be received
144:         wait for all ACK_INVALIDATE to be received
145:         lock page
146:         page.probable := my_rank
147:         page.owner := TRUE
148:     endif
149:
150: when REQ_VALIDATE for page is received:
151:     lock page
152:     page.probable := REQ_VALIDATE.src
153:     page.buffer := REQ_VALIDATE.buffer
154:     ACK_VALIDATE.seq := −1
155:     ACK_VALIDATE.src := my_rank
156:     send ACK_VALIDATE to REQ_VALIDATE.src
157:     unlock page
```

```
158:
159: when REQ_INVALIDATE for page is received:
160:     lock page
161:     page.probable := REQ_INVALIDATE.src
162:     page.state := INVALID
163:     ACK_INVALIDATE.seq := −1
164:     ACK_INVALIDATE.src := my_rank
165:     send ACK_INVALIDATE to REQ_INVALIDATE.src
166:     unlock page
167:
168: when REQ_STEAL for page is received:
169:     lock page
170:     if page.owner == TRUE then
171:         page.owner := FALSE
172:         REQ_CHANGE.seq := page.seq_array[my_rank]
173:         page.seq_array[my_rank] := page.seq_array[my_rank] + 1
174:         REQ_CHANGE.src := REQ_STEAL.src
175:         send REQ_CHANGE to my_rank
176:         if page.state_array[REQ_STEAL.src] == INVALID then
177:             ACK_STEAL.buffer := page.buffer
178:             page.state_array[REQ_STEAL.src] := DOWN_VALID
179:             page.valid := page.valid + 1
180:         else
181:             ACK_STEAL.buffer := NIL
182:         endif
183:         ACK_STEAL.seq := page.seq_array[REQ_STEAL.src]
184:         page.seq_array[REQ_STEAL.src] := page.seq_array[REQ_STEAL.src] + 1
185:         ACK_STEAL.src := my_rank
186:         ACK_STEAL.state_array := page.state_array
187:         ACK_STEAL.seq_array := page.seq_array
188:         ACK_STEAL.valid := page.valid
189:         send ACK_STEAL to REQ_STEAL.src
190:     else
191:         send REQ_STEAL to page.probable
192:     endif
193:     unlock page
```

```
194:
195: when REQ_CHANGE for page is received:
196:     lock page
197:     page.probable := REQ_CHANGE.src
198:     unlock page
199:
200: sweep operation for page:
201:     lock page
202:     if page.state == DOWN_VALID || page.state == UP_VALID then
203:         REQ_SWEEP.seq := −1
204:         REQ_SWEEP.src := my_rank
205:         send REQ_SWEEP to page.probable
206:         unlock page
207:         wait for ACK_SWEEP to be received
208:         lock page
209:         page.probable := ACK_SWEEP.src
210:         page.state := INVALID
211:     endif
212:     unlock page
213:
214: when REQ_SWEEP for page is received:
215:     lock page
216:     if page.owner == TRUE then
217:         if page.state_array[REQ_SWEEP.src] == UP_VALID
218:             || page.state_array[REQ_SWEEP.src] == DOWN_VALID then
219:             page.state_array[REQ_SWEEP.src] := INVALID
220:             page.valid := page.valid −1
221:         endif
222:         ACK_SWEEP.seq := page.seq_array[REQ_SWEEP.src]
223:         page.seq_array[REQ_SWEEP.src] := page.seq_array[REQ_SWEEP.src] + 1
224:         ACK_SWEEP.src := my_rank
225:         send ACK_SWEEP to REQ_SWEEP.src
226:         if REQ_SWEEP.src == my_rank then
227:             rank := select new owner except my_rank
228:             page.owner := FALSE
229:             REQ_CHANGE.seq := page.seq_array[my_rank]
```

```
230:          page.seq_array[my_rank] := page.seq_array[my_rank]+1
231:          REQ_CHANGE.src := rank
232:          send REQ_CHANGE to my_rank
233:          if page.state_array[rank] == INVALID then
234:             REQ_DELEGATE.buffer := page.buffer
235:             page.state_array[rank] := DOWN_VALID
236:             page.valid := page.valid+1
237:          else
238:             REQ_DELEGATE.buffer := NIL
239:          endif
240:          REQ_DELEGATE.seq := page.seq_array[rank]
241:          page.seq_array[rank] := page.seq_array[rank]+1
242:          REQ_DELEGATE.src := my_rank
243:          REQ_DELEGATE.state_array := page.state_array
244:          REQ_DELEGATE.seq_array := page.seq_array
245:          REQ_DELEGATE.valid := page.valid
246:          send REQ_DELEGATE to rank
247:       endif
248:    else
249:          send REQ_SWEEP to page.probable
250:    endif
251:    unlock page
252:
253: when REQ_DELEGATE for page is received:
254:    lock page
255:    if REQ_DELEGATE.buffer != NIL then
256:       page.buffer := REQ_DELEGATE.buffer
257:       page.state := DOWN_VALID
258:    endif
259:    page.state_array := REQ_DELEGATE.state_array
260:    page.seq_array := REQ_DELEGATE.seq_array
261:    page.valid := REQ_DELEGATE.valid
262:    page.owner := TRUE
263:    unlock page
```

付録 B

random-address の最適性の証明

2.1 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ひとつが，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 $(0 \leq i \leq m-1)$ 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 にしたが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 にしたが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：

$$C^{\sigma_i}(A) = \{\{\sigma_i(\underline{a_i}), \sigma_i(\underline{a_i+1}), \ldots, \sigma_i(\underline{a_i+b_i-1})\} \mid 0 \leq a_i \leq n-1, 0 \leq b_i \leq n\}. \quad (2.1)$$

$C^{\sigma_i}(A)$ 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 にしたが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 にしたが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，$C^{\sigma_i}(A)$

[⋆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 (2.1) 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，$1 \leq b_i \leq n-1$ 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．

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 にしたが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 にしたが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 にしたがって（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 にしたがって（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 にしたが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ひとつは，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 にしたが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み合わせ $(\sigma_0, \sigma_1, \ldots, \sigma_{m-1})$ のうち（各 σ_i の選び方は $n!$ とおり存在するから，組み合わせは全部で $n!^m$ とおり存在する),「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 にしたが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み合わせ $(\sigma_0, \sigma_1, \ldots, \sigma_{m-1})$ 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，かつその場合にかぎられる．

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

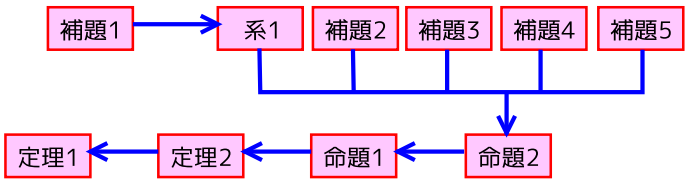

図 2.1 命題や補題の論理関係（$A \to B$ は，証明において A から B を導くことを意味する）.

なる．次節では，定理 2 の証明を行う．

2.2 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 2.1 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 にしたがって一様にアドレスを使用し，スレッド y は置換 σ_y にしたがって一様にアドレスを使用するとする．このとき，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のは，$\sigma_x = \sigma_y$ の場合であり，かつその場合にかぎられる．

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 $P(A)$ のなかで，サイズが 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 $P_i(A)$ の任意の要素はサイズ 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 $C^{\sigma_y}(A)$ のなかで，サイズが 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，

$$M(C; \ldots, S_i \cup S_j, \ldots) = M(C; \ldots, S_i, \ldots) + M(C; \ldots, S_j, \ldots)$$

$$-M(C;\ldots,S_i,S_j,\ldots) \tag{2.2}$$

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,

$$\underline{i} = i \tag{2.3}$$

が成り立つ．また，任意の整数 i，j に対して,

$$\underline{i+\underline{j}} = \underline{i+j} \tag{2.4}$$

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 2.2（A））.
以上の定義より，任意の整数 i_0，i_1，...，i_{k-1} に対して,

$$\begin{aligned} & i_0 \preceq i_1 \preceq \ldots \preceq i_{k-1} \preceq \\ \Longleftrightarrow & \underline{i_1 - i_0} + \underline{i_2 - i_1} + \ldots + \underline{i_{k-1} - i_{k-2}} + \underline{i_0 - i_{k-1}} = n \end{aligned} \tag{2.5}$$

が成り立つ．また，$i_0 \preceq i_1 \preceq i_2 \preceq$ ならば,

$$\underline{i_1 - i_0} + \underline{i_2 - i_1} = \underline{i_2 - i_0} \tag{2.6}$$

が成り立つ.
ここで，サイズが 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 2.2 のような円周上の点集合として図示することにする.

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 にしたがって「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 $S_y \in C^{\sigma_y}(A)$ のすべての組み合わせ (S_x, S_y)（全部で $|C^{\sigma_x}(A)| \times |C^{\sigma_y}(A)|$ 個ある）のなかで，$S_x \cap S_y \neq \emptyset$ を満たすような組み合わせ (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 のなかで，『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 $S_y \in C^{\sigma_y}(A)$ のすべての組み合わせ (S_x, S_y)（全部で $|C^{\sigma_x}(A)| \times |C^{\sigma_y}(A)|$ 個ある）のなかで，$S_x \cap S_y \neq \emptyset$ を満たすような組み合わせ (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 $S_x \in P(A)$ のなかで，『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 のサイズ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 $S_x \in P_{b_x}(A)$ のなかで，『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 $S_x \in P_{b_x}(A)$ のなかで，$M(C^{\sigma_y}(A); S_x)$ 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 をわ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 とおりのみである．
- $b_x = n$ のとき，$\{0, 1, \ldots, n-1\}$ の 1 とおりのみである．
- $1 \leq b_x \leq n-1$ のとき，以下の T^0，T^1，…，T^{n-1} の合計 n とおりのみである：

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

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

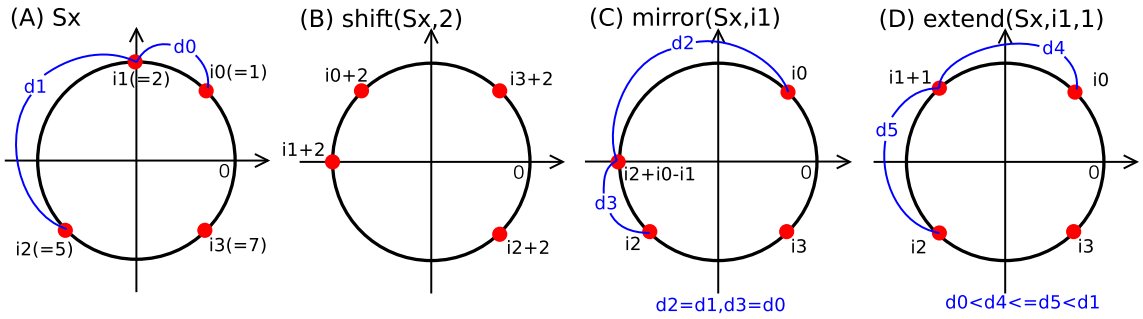

図 2.2　アドレス集合 S_x と各写像の具体例．（A）S_x，（B）$shift(S_x, s)$，（C）$mirror(S_x, i_\alpha)$，（D）$extend(S_x, i_\alpha, s)$．

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 2.2（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 2.2（B）（C）（D）に示す．直観的には，図 2.2 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 (2.1) 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 (2.2) を用いて補題 2 の左辺を計算すると，

$$\begin{aligned} M(C^{\sigma_y}(A); S_x) &= M(C^{\sigma_y}(A); (S_x \backslash \{\sigma_y(i_\alpha)\}) \cup \{\sigma_y(i_\alpha)\}) \\ &= M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) \\ &\quad - M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_\alpha)\}) \end{aligned} \tag{2.7}$$

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，

$$\begin{aligned} D_3 &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1}), \sigma_y(i_{\alpha+1})\}, \{\sigma_y(i_\alpha)\}) \\ &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\} \cup \{\sigma_y(i_{\alpha+1})\}, \{\sigma_y(i_\alpha)\}) \\ &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\ &\quad - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \quad (\because \text{式 (2.2)}) \end{aligned} \tag{2.8}$$

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，

$$M(C^{\sigma_y}(A); S_x) = D_1 + D_2 - (D_4 + D_5 - D_6) \tag{2.9}$$

となる．

同様にして，式 (2.2) を用いて補題 2 の右辺を計算すると，

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

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，

$$D'_3 = M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\})$$

$$
\begin{aligned}
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1}), \sigma_y(i_{\alpha+1})\}, \{\sigma_y(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})\}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\} \cup \{\sigma_y(i_{\alpha+1})\}, \{\sigma_y(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})\}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})\}) \\
&\quad + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})\}, \{\sigma_y(i_{\alpha+1})\}) \\
&\quad - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})\}, \{\sigma_y(i_{\alpha+1})\}) \quad (\because \text{式 (2.2)})
\end{aligned}
$$

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，

$$M(C^{\sigma_y}(A); mirror(S_x, i_\alpha)) = D'_1 + D'_2 - (D'_4 + D'_5 - D'_6) \tag{2.10}$$

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，

$$D_1 = M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) = D'_1, \tag{2.11}$$

$$
\begin{aligned}
D_2 &= M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) \\
&= M(C^{\sigma_y}(A); shift(\{\sigma_y(i_\alpha)\}, \underline{i_{\alpha-1}+i_{\alpha+1}-2i_\alpha})) \quad (\because \text{補題 1}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})\}) \\
&= D'_2,
\end{aligned} \tag{2.12}
$$

$$
\begin{aligned}
D_4 &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) \\
&= M(C^{\sigma_y}(A); shift(\{\sigma_y(i_{\alpha-1})\}, \underline{i_{\alpha+1}-i_\alpha}), shift(\{\sigma_y(i_\alpha)\}, \underline{i_{\alpha+1}-i_\alpha})) \quad (\because \text{補題 1}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})\}, \{\sigma_y(i_{\alpha+1})\}) \\
&= D'_5,
\end{aligned} \tag{2.13}
$$

$$
\begin{aligned}
D_5 &= M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\
&= M(C^{\sigma_y}(A); shift(\{\sigma_y(i_\alpha)\}, \underline{i_{\alpha-1}-i_\alpha}), shift(\{\sigma_y(i_{\alpha+1})\}, \underline{i_{\alpha-1}-i_\alpha})) \quad (\because \text{補題 1}) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})\}) \\
&= D'_4
\end{aligned} \tag{2.14}
$$

となる．次に，D_6 と D'_6 の大小を考える．まず，

$$
\begin{aligned}
&\underline{(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})-i_{\alpha-1}} + \underline{i_{\alpha+1}-(\underline{i_{\alpha-1}+i_{\alpha+1}-i_\alpha})} + \underline{i_{\alpha-1}-i_{\alpha+1}} \\
&= \underline{i_{\alpha+1}-i_\alpha} + \underline{i_\alpha - i_{\alpha-1}} + \underline{i_{\alpha-1}-i_{\alpha+1}} \quad (\because \text{式 (2.4)}) \\
&= n \quad (\because \text{仮定より } i_{\alpha-1} \preceq i_\alpha \preceq i_{\alpha+1} \preceq \text{と式 (2.5)})
\end{aligned}
$$

であるから，式 (2.5) 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，

$$D_6 = M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\})$$

$$= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_{\alpha-1} + i_{\alpha+1} - i_\alpha})\}, \{\sigma_y(i_{\alpha+1})\})$$
$$= D'_6 \tag{2.15}$$

となる．

式 (2.9)(2.10)(2.11)(2.12)(2.13)(2.14)(2.15) より，

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

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 (2.7)(2.8) より，

$$\begin{aligned}
& M(C^{\sigma_y}(A); S_x) \\
= \; & M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) \\
& - M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_\alpha)\}) \\
= \; & M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) \\
& - (M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\
& - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}))
\end{aligned} \tag{2.16}$$

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，

$$\begin{aligned}
& M(C^{\sigma_y}(A); \mathit{extend}(S_x, i_\alpha, s)) \\
= \; & M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\} \cup \{\sigma_y(\underline{i_\alpha + s})\}) \\
= \; & M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}) \\
& - M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}, \{\sigma_y(\underline{i_\alpha + s})\}) \quad (\because \text{式 (2.2)}) \\
= \; & M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}) \\
& - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1}), \sigma_y(i_{\alpha+1})\}, \{\sigma_y(\underline{i_\alpha + s})\}) \\
& \quad (\because i_{\alpha-1} \preceq \underline{i_\alpha + s} \preceq i_{\alpha+1} \preceq \text{かつ } i_{\alpha-1} \preceq i_\alpha \preceq i_{\alpha+1} \preceq) \\
= \; & M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}) \\
& - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\} \cup \{\sigma_y(i_{\alpha+1})\}, \{\sigma_y(\underline{i_\alpha + s})\}) \\
= \; & M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}) \\
& - (M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_\alpha + s})\}) \\
& + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}, \{\sigma_y(i_{\alpha+1})\}) \\
& - M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_\alpha + s})\}, \{\sigma_y(i_{\alpha+1})\})) \quad (\because \text{式 (2.2)})
\end{aligned} \tag{2.17}$$

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，

$$D_1 = M(C^{\sigma_y}(A); S_x \backslash \{\sigma_y(i_\alpha)\}) = D'_1, \tag{2.18}$$
$$\begin{aligned}
D_2 &= M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}) = M(C^{\sigma_y}(A); \mathit{shift}(\{\sigma_y(i_\alpha)\}, s)) \\
&= M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}) = D'_2,
\end{aligned} \tag{2.19}$$

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 (2.15) と同様にして，

$$\begin{aligned} D_5 &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\ &= M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_\alpha + s})\}, \{\sigma_y(i_{\alpha+1})\}) = D_5' \end{aligned} \tag{2.20}$$

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 $C_{b_y}^{\sigma_y}(A) \cap C_{b_y'}^{\sigma_y}(A) = \emptyset$ であり，$C^{\sigma_y}(A) = \bigsqcup_{b_y=0}^{n} C_{b_y}^{\sigma_y}(A)$ が成り立つから，任意の集合 S_x に対して，

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

が成り立つことに着目する．したがって，

$$\begin{aligned} & D_3 + D_4 \\ =\ & M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) + M(C^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\ =\ & \sum_{b_y=0}^{n} M(C_{b_y}^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) + \sum_{b_y=0}^{n} M(C_{b_y}^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \\ =\ & \sum_{b_y=0}^{n} \left(M(C_{b_y}^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(i_\alpha)\}) + M(C_{b_y}^{\sigma_y}(A); \{\sigma_y(i_\alpha)\}, \{\sigma_y(i_{\alpha+1})\}) \right), \\ & D_3' + D_4' \\ =\ & M(C^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_\alpha + s})\}) + M(C^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}, \{\sigma_y(i_{\alpha+1})\}) \\ =\ & \sum_{b_y=0}^{n} M(C_{b_y}^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_\alpha + s})\}) \\ & + \sum_{b_y=0}^{n} M(C_{b_y}^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}, \{\sigma_y(i_{\alpha+1})\}) \\ =\ & \sum_{b_y=0}^{n} \Bigg(M(C_{b_y}^{\sigma_y}(A); \{\sigma_y(i_{\alpha-1})\}, \{\sigma_y(\underline{i_\alpha + s})\}) \\ & + M(C_{b_y}^{\sigma_y}(A); \{\sigma_y(\underline{i_\alpha + s})\}, \{\sigma_y(i_{\alpha+1})\}) \Bigg) \end{aligned}$$

と展開できる．ここで，

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

とおくと，D_3+D_4 と $D'_3+D'_4$ は，

$$D_3+D_4=\sum_{b_y=0}^{n}(d_3(b_y)+d_4(b_y)), \tag{2.21}$$

$$D'_3+D'_4=\sum_{b_y=0}^{n}(d'_3(b_y)+d'_4(b_y)) \tag{2.22}$$

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，

$$d_3(b_y)=\underline{b_y-\underline{(i_\alpha-i_{\alpha-1})}}=b_y-\underline{i_\alpha-i_{\alpha-1}}\quad(\because \text{式 (2.3)(2.4)})$$

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：

$$d_3(b_y)=\begin{cases}0 & \text{if } 0\le b_y\le \underline{i_\alpha-i_{\alpha-1}}\\ b_y-(\underline{i_\alpha-i_{\alpha-1}}) & \text{if } \underline{i_\alpha-i_{\alpha-1}}<b_y\le n-1\\ 1 & \text{if } b_y=n\end{cases} \tag{2.23}$$

$$d_4(b_y)=\begin{cases}0 & \text{if } 0\le b_y\le \underline{i_{\alpha+1}-i_\alpha}\\ b_y-(\underline{i_{\alpha+1}-i_\alpha}) & \text{if } \underline{i_{\alpha+1}-i_\alpha}<b_y\le n-1\\ 1 & \text{if } b_y=n\end{cases} \tag{2.24}$$

$$d'_3(b_y)=\begin{cases}0 & \text{if } 0\le b_y\le \underline{(i_\alpha+s)-i_{\alpha-1}}\\ b_y-(\underline{(i_\alpha+s)-i_{\alpha-1}}) & \text{if } \underline{(i_\alpha+s)-i_{\alpha-1}}<b_y\le n-1\\ 1 & \text{if } b_y=n\end{cases} \tag{2.25}$$

$$d'_4(b_y)=\begin{cases}0 & \text{if } 0\le b_y\le \underline{i_{\alpha+1}-(i_\alpha+s)}\\ b_y-(\underline{i_{\alpha+1}-(i_\alpha+s)}) & \text{if } \underline{i_{\alpha+1}-(i_\alpha+s)}<b_y\le n-1\\ 1 & \text{if } b_y=n\end{cases} \tag{2.26}$$

となる．

ここで，式 (2.23)(2.24)(2.25)(2.26) 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，

$$\begin{aligned}
0 &< \underline{i_\alpha - i_{\alpha-1}} \quad (\because \text{仮定より } i_{\alpha-1} \prec i_\alpha \prec) \\
&< \underline{(i_\alpha + s) - i_{\alpha-1}} \quad (\because \text{仮定より } i_{\alpha-1} \prec i_\alpha \prec \underline{i_\alpha + s} \prec) \\
&\leq \underline{i_{\alpha+1} - (i_\alpha + s)} \quad (\because \text{仮定そのまま}) \\
&< \underline{i_{\alpha+1} - i_\alpha} \quad (\because \text{仮定より } i_\alpha \prec \underline{i_\alpha + s} \prec i_{\alpha+1} \prec) \\
&< n \quad (\because \text{明らか})
\end{aligned} \tag{2.27}$$

が成り立つ．図 2.2（D）を見ると，この大小関係が図形的に理解できる．

以上を踏まえて，$d_3(b_y) + d_4(b_y)$ 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) $0 \leq b_y \leq \underline{i_\alpha - i_{\alpha-1}}$ のとき．

$$(d_3(b_y) + d_4(b_y)) - (d'_3(b_y) + d'_4(b_y)) = (0 + 0) - (0 + 0) = 0 \tag{2.28}$$

である．

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

$$(d_3(b_y) + d_4(b_y)) - (d'_3(b_y) + d'_4(b_y)) = ((b_y - (\underline{i_\alpha - i_{\alpha-1}})) + 0) - (0 + 0) > 0 \tag{2.29}$$

である．

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

$$\begin{aligned}
&(d_3(b_y) + d_4(b_y)) - (d'_3(b_y) + d'_4(b_y)) \\
&= ((b_y - (\underline{i_\alpha - i_{\alpha-1}})) + 0) - ((b_y - (\underline{(i_\alpha + s) - i_{\alpha-1}})) + 0) \\
&= (\underline{(i_\alpha + s) - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \\
&= (\underline{(i_\alpha + s) - i_\alpha}) + (\underline{i_\alpha - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \quad (\because i_{\alpha-1} \preceq i_\alpha \preceq \underline{i_\alpha + s} \preceq \text{と式 (2.6)}) \\
&= \underline{(i_\alpha + s) - i_\alpha} \\
&= \underline{s} \\
&> 0
\end{aligned} \tag{2.30}$$

である．

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

$$\begin{aligned}
&(d_3(b_y) + d_4(b_y)) - (d'_3(b_y) + d'_4(b_y)) \\
&= ((b_y - (\underline{i_\alpha - i_{\alpha-1}})) + 0) - ((b_y - (\underline{(i_\alpha + s) - i_{\alpha-1}})) + (b_y - (\underline{i_{\alpha+1} - (i_\alpha + s)}))) \\
&= -b_y + (\underline{i_{\alpha+1} - (i_\alpha + s)}) + (\underline{(i_\alpha + s) - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \\
&= -b_y + (\underline{i_{\alpha+1} - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \quad (\because i_{\alpha-1} \preceq \underline{i_\alpha + s} \preceq i_{\alpha+1} \preceq \text{と式 (2.6)}) \\
&= -b_y + (\underline{i_{\alpha+1} - i_\alpha}) + (\underline{i_\alpha - i_{\alpha-1}}) - (\underline{i_\alpha - i_{\alpha-1}}) \quad (\because i_{\alpha-1} \preceq i_\alpha \preceq i_{\alpha+1} \preceq \text{と式 (2.6)}) \\
&= -b_y + (\underline{i_{\alpha+1} - i_\alpha})
\end{aligned}$$

$$\geq 0 \tag{2.31}$$

である．

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

$$\begin{aligned}
& (d_3(b_y) + d_4(b_y)) - (d'_3(b_y) + d'_4(b_y)) \\
= \ & ((b_y - (\underline{i_\alpha - i_{\alpha-1}})) + (b_y - (\underline{i_{\alpha+1} - i_\alpha}))) \\
& -((b_y - (\underline{(i_\alpha + s) - i_{\alpha-1}})) + (b_y - (\underline{i_{\alpha+1} - (i_\alpha + s)}))) \\
= \ & ((\underline{i_{\alpha+1} - (i_\alpha + s)}) + (\underline{(i_\alpha + s) - i_{\alpha-1}})) - ((\underline{i_{\alpha+1} - i_\alpha}) + (\underline{i_\alpha - i_{\alpha-1}})) \\
= \ & (\underline{i_{\alpha+1} - i_{\alpha-1}}) - (\underline{i_{\alpha+1} - i_{\alpha-1}}) \quad (\because i_{\alpha-1} \preceq i_\alpha \preceq \underline{i_\alpha + s} \preceq i_{\alpha+1} \preceq \text{と式 (2.6)}) \\
= \ & 0
\end{aligned} \tag{2.32}$$

である．

（vi） $b_y = n$ のとき．

$$(d_3(b_y) + d_4(b_y)) - (d'_3(b_y) + d'_4(b_y)) = (1+1) - (1+1) = 0 \tag{2.33}$$

である．

以上の式 (2.21)(2.22)(2.28)(2.29)(2.30)(2.31)(2.32)(2.33) より，

$$D_3 + D_4 = \sum_{b_y=0}^{n} (d_3(b_y) + d_4(b_y)) > \sum_{b_y=0}^{n} (d'_3(b_y) + d'_4(b_y)) = D'_3 + D'_4 \tag{2.34}$$

が成り立つ．なお，式 (2.34) における大小関係が $\geq$ ではなく $>$ になるのは，式 (2.29) において $>$ が入っており，かつ，式 (2.27) より，式 (2.29) を満たす b_y が少なくとも 1 個存在するためである[*2] ．

式 (2.16)(2.17)(2.18)(2.19)(2.21)(2.34) より，

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

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，$i_0 \prec i_1 \prec \ldots \prec i_{b_x-1} \prec$ なる整数 i_0，i_1，…，i_{b_x-1} を用いて，

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

[*2] 式 (2.30) でも $>$ が入っているが，式 (2.27) より，式 (2.30) を満たす b_y は存在しない可能性がある．

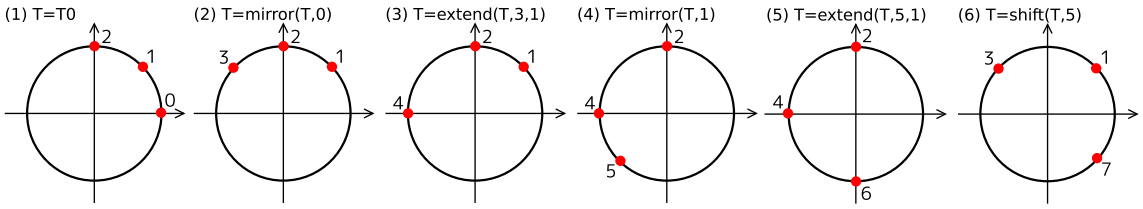

図 2.3 $T^0 = \{\sigma_y(0), \sigma_y(1), \sigma_y(2)\}$ に対して操作 O を適用することで，$S_x = \{\sigma_y(1), \sigma_y(3), \sigma_y(7)\}$ を得るまでの手続き．

と表すことができる（図 2.2(A)）．このとき，すべての j $(0 \le j \le b_x - 1)$ に関して，$\underline{i_k - i_{k-1}} \ge \underline{i_j - i_{j-1}}$ を満たすような k $(0 \le k \le b_x - 1)$ 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，

$$\forall j \ (0 \le j \le b_x - 1) : \underline{i_\alpha - i_{\alpha-1}} \ge \underline{i_j - i_{j-1}} \tag{2.35}$$

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：

$v := b_x - 1$
$T := T^0$
$k := 0$
while $k < b_x - 1$ **do**
  $T := mirror(T, k)$ /* step1 */
  $v := \underline{v + 1}$ /* step2 */
  if $\underline{i_{\alpha+k+1} - i_{\alpha+k} - 1} \ne 0$ **then**
    $T := extend(T, v, \underline{i_{\alpha+k+1} - i_{\alpha+k} - 1})$ /* step3 */
  endif
  $v := \underline{v + i_{\alpha+k+1} - i_{\alpha+k} - 1}$ /* step4 */
  $k := k + 1$
endwhile
$T := shift(T, i_\alpha - (b_x - 1))$ /* 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 2.3 に示す．いまの場合，$i_0 = 1$，$i_1 = 3$，$i_2 = 7$ に関して，$\underline{i_0 - i_2} \ge \underline{i_2 - i_1}$，$\underline{i_0 - i_2} \ge \underline{i_1 - i_0}$ 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する：

$$v = \underline{b_x - 1 + i_{\alpha+k} - i_\alpha}, \tag{2.36}$$

$$T = \{\sigma_y(k), \sigma_y(k+1), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + i_{\alpha+1} - i_\alpha}), \ldots, \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha})\}. \tag{2.37}$$

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 (2.36)(2.37) が成り立つ．

次に，k の場合にループ不変条件 (2.36)(2.37) 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 (2.36)(2.37) が成り立つことをいう．

第 1 に，式 (2.36) 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，

$$v = \underline{\underline{b_x - 1 + i_{\alpha+k} - i_\alpha} + 1} = \underline{b_x + i_{\alpha+k} - i_\alpha} \quad (\because \text{式 (2.4)}) \tag{2.38}$$

に変化する．さらに step4 の操作によって，v は，

$$v = \underline{\underline{b_x + i_{\alpha+k} - i_\alpha} + i_{\alpha+k+1} - i_{\alpha+k} - 1} = \underline{b_x - 1 + i_{\alpha+k+1} - i_\alpha} \quad (\because \text{式 (2.4)})$$

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 (2.36) がループ不変条件であることが示された．

第 2 に，式 (2.37) 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，

$$\begin{aligned} T &= \{\sigma_y(k+1), \sigma_y(k+2), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + i_{\alpha+1} - i_\alpha}), \ldots, \\ &\quad \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha}), \sigma_y(\underline{\underline{b_x - 1 + i_{\alpha+k} - i_\alpha} + (k+1) - k})\} \\ &= \{\sigma_y(k+1), \sigma_y(k+2), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + i_{\alpha+1} - i_\alpha}), \ldots, \\ &\quad \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha}), \sigma_y(\underline{b_x + i_{\alpha+k} - i_\alpha})\} \end{aligned} \tag{2.39}$$

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，

$$\sigma_y(\tilde{i}_\alpha) \in T, \tag{2.40}$$
$$\tilde{i}_\alpha \prec \underline{\tilde{i}_\alpha + s} \prec \tilde{i}_{\alpha+1} \prec, \tag{2.41}$$
$$\underline{(\tilde{i}_\alpha + s) - \tilde{i}_{\alpha-1}} \leq \underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} \tag{2.42}$$

の 3 条件が満足されねばならない．そこで，式 (2.39) に対して step3 の操作を適用する時点で確かに式 (2.40)(2.41)(2.42) が満足されていることを確認する．

式 (2.39) に対して step3 の操作を適用する時点では，式 (2.38) 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 (2.40) 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 (2.40) は成り立つ．

(II) 式 (2.41) 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，

$$\begin{aligned}
\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} &= \underline{(k+1) - ((b_x + i_{\alpha+k} - i_\alpha) + (i_{\alpha+k+1} - i_{\alpha+k} - 1))} \\
&= \underline{(k+1) - (b_x - 1) - (i_{\alpha+k+1} - i_{\alpha+k}) + (i_{\alpha+k} - i_\alpha)} \quad (\because \text{式 (2.4)}) \\
&= \underline{(k+1) - (b_x - 1) - (i_{\alpha+k+1} - i_\alpha)} \\
&\qquad (\because i_\alpha \preceq i_{\alpha+k} \preceq i_{\alpha+k+1} \preceq \text{と式 (2.6)}) \\
&= \underline{n + (k+1) - (b_x - 1) - (i_{\alpha+k+1} - i_\alpha)}
\end{aligned} \tag{2.43}$$

が得られる．ここで式 (2.43) がとりうる値の範囲を考えると，

$$\begin{aligned}
n &> n + (k+1) - (b_x - 1) - (\underline{i_{\alpha+k+1} - i_\alpha}) \quad (\because k \leq b_x - 2 \text{ および } i_{\alpha+k+1} \neq i_\alpha) \\
&\geq n - (\underline{i_{\alpha+b_x-1} - i_{\alpha+k+1}}) - (\underline{i_{\alpha+k+1} - i_\alpha}) \\
&\qquad (\because i_{\alpha+k+1} \prec i_{\alpha+k+2} \prec \ldots \prec i_{\alpha+b_x-1} \prec \text{より} \\
&\quad \underline{i_{\alpha+b_x-1} - i_{\alpha+k+1}} \geq (b_x - 1) - (k+1)) \\
&= n - (\underline{i_{\alpha+b_x-1} - i_\alpha}) \quad (\because i_\alpha \preceq i_{\alpha+k+1} \preceq i_{\alpha+b_x-1} \preceq \text{と式 (2.6)}) \\
&= n - (\underline{i_{\alpha-1} - i_\alpha}) \quad (\because \mathrm{mod}(\alpha + b_x - 1, b_x) = \mathrm{mod}(\alpha - 1, b_x)) \\
&= \underline{i_\alpha - i_{\alpha-1}} \quad (\because i_{\alpha-1} \preceq i_\alpha \preceq \text{と式 (2.5)}) \\
&> 0 \quad (\because i_\alpha \neq i_{\alpha-1})
\end{aligned} \tag{2.44}$$

が成り立つ．よって，式 (2.3)(2.43)(2.44) より，

$$\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} = n + (k+1) - (b_x - 1) - (\underline{i_{\alpha+k+1} - i_\alpha}) \tag{2.45}$$

が成り立つ．

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

$$\begin{aligned}\underline{\tilde{i}_{\alpha+1} - \tilde{i}_\alpha} &= \underline{(k+1) - (b_x + i_{\alpha+k} - i_\alpha)} \\ &= \underline{k - (b_x - 1) - (i_{\alpha+k} - i_\alpha)} \quad (\because \text{式 (2.4)}) \\ &= \underline{n + k - (b_x - 1) - (i_{\alpha+k} - i_\alpha)}\end{aligned} \tag{2.46}$$

となる．この式 (2.46) は，式 (2.43) における $k+1$ を k に置き換えたものであるから，式 (2.44) を導いたときの議論と同様にして，

$$n > n + k - (b_x - 1) - (\underline{i_{\alpha+k} - i_\alpha}) > 0 \tag{2.47}$$

がいえる．よって，式 (2.3)(2.46)(2.47) より，

$$\underline{\tilde{i}_{\alpha+1} - \tilde{i}_\alpha} = n + k - (b_x - 1) - (\underline{i_{\alpha+k} - i_\alpha}) \tag{2.48}$$

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，

$$\begin{aligned}&\underline{(\tilde{i}_\alpha + s) - \tilde{i}_\alpha} + \underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} + \underline{\tilde{i}_\alpha - \tilde{i}_{\alpha+1}} \\ =\ & \underline{s} + \underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} + (n - \underline{\tilde{i}_{\alpha+1}} - \tilde{i}_\alpha) \quad (\because \text{式 (2.4)},\ i_\alpha \preceq i_{\alpha+1} \preceq \text{と式 (2.5)}) \\ =\ & (\underline{i_{\alpha+k+1} - i_{\alpha+k} - 1}) + (n + (k+1) - (b_x - 1) - (\underline{i_{\alpha+k+1} - i_\alpha})) \\ & + (n - (n + k - (b_x - 1) - (\underline{i_{\alpha+k} - i_\alpha}))) \quad (\because \text{式 (2.45)(2.48)}) \\ =\ & ((\underline{i_{\alpha+k+1} - i_{\alpha+k} - 1}) + 1) + (n - (\underline{i_{\alpha+k+1} - i_\alpha})) + (\underline{i_{\alpha+k} - i_\alpha}) \\ =\ & ((\underline{i_{\alpha+k+1} - i_{\alpha+k}}) - 1 + 1) + (\underline{i_\alpha - i_{\alpha+k+1}}) + (\underline{i_{\alpha+k} - i_\alpha}) \\ & \quad (\because i_{\alpha+k+1} \neq i_{\alpha+k},\ i_\alpha \preceq i_{\alpha+k+1} \preceq \text{と式 (2.5)}) \\ =\ & n \quad (\because i_\alpha \preceq i_{\alpha+k} \preceq i_{\alpha+k+1} \preceq \text{と式 (2.5)})\end{aligned}$$

となる．したがって，式 (2.5) 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 (2.44)(2.45) より $\tilde{i}_{\alpha+1} \neq \underline{\tilde{i}_\alpha + s}$ であり，式 (2.47)(2.48) 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 (2.41) が成り立つことが確認できた．

(Ⅲ) 式 (2.42) が成り立つことを確認する．ここで，式 (2.42) の左辺である $\underline{(\tilde{i}_\alpha + s) - \tilde{i}_{\alpha-1}}$ を計算すると，

$$\begin{aligned}\underline{(\tilde{i}_\alpha + s) - \tilde{i}_{\alpha-1}} &= \underline{((b_x + i_{\alpha+k} - i_\alpha) + (i_{\alpha+k+1} - i_{\alpha+k} - 1)) - (b_x - 1 + i_{\alpha+k} - i_\alpha)} \\ &= \underline{i_{\alpha+k+1} - i_{\alpha+k}} \quad (\because \text{式 (2.4)})\end{aligned} \tag{2.49}$$

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 (2.44)(2.45) より，

$$\underline{\tilde{i}_{\alpha+1} - (\tilde{i}_\alpha + s)} = n + (k+1) - (b_x - 1) - (\underline{i_{\alpha+k+1} - i_\alpha}) \geq \underline{i_\alpha - i_{\alpha-1}} \tag{2.50}$$

が成り立つ．よって，式 (2.35)(2.49)(2.50) 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 (2.42) が成り立つことが確認できた．

以上の（I）（II）（III）より，式 (2.39) に対して step3 の操作を適用する時点で，確かに式 (2.40)(2.41)(2.42) が満足されていることを確認できた．

そこで，式 (2.39) に対して step3 の操作を適用すると，T は，

$$\begin{aligned} T &= \{\sigma_y(k+1), \sigma_y(k+2), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + (i_{\alpha+1} - i_\alpha)}), \ldots, \\ &\quad \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha}), \sigma_y(\underline{\underline{b_x + i_{\alpha+k} - i_\alpha} + i_{\alpha+k+1} - i_{\alpha+k} - 1})\} \\ &= \{\sigma_y(k+1), \sigma_y(k+2), \ldots, \sigma_y(b_x - 1), \sigma_y(\underline{b_x - 1 + (i_{\alpha+1} - i_\alpha)}), \ldots, \\ &\quad \sigma_y(\underline{b_x - 1 + i_{\alpha+k} - i_\alpha}), \sigma_y(\underline{b_x - 1 + i_{\alpha+k+1} - i_\alpha})\} \quad (\because \text{式 (2.4)}) \end{aligned}$$

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 (2.37) 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，

$$\begin{aligned} T &= \{\sigma_y(\underline{b_x - 1 + i_\alpha - (b_x - 1)}), \sigma_y(\underline{\underline{b_x - 1 + i_{\alpha+1} - i_\alpha} + i_\alpha - (b_x - 1)}), \ldots, \\ &\quad \sigma_y(\underline{\underline{b_x - 1 + i_{\alpha+b_x-1} - i_\alpha} + i_\alpha - (b_x - 1)})\} \\ &= \{\sigma_y(\underline{i_\alpha}), \sigma_y(\underline{i_{\alpha+1}}), \ldots, \sigma_y(\underline{i_{\alpha+b_x-1}})\} \quad (\because \text{式 (2.4)}) \\ &= \{\sigma_y(i_0), \sigma_y(i_1), \ldots, \sigma_y(i_{b_x-1})\} \\ &= S_x \end{aligned}$$

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 $C_{b_x}^{\sigma_y}(A)$ 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 $C_{b_x}^{\sigma_y}(A)$ 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 にしたが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，

$$\begin{aligned} & p(E(x_0, x_1, \ldots, x_{m-1}, x_m)) \\ = \; & p(E(x_0, x_1, \ldots, x_{m-1}))p(E(x_0, x_1, \ldots, x_{m-1})| \qquad (2.51) \\ & (E(x_m, x_0) \wedge E(x_m, x_1) \wedge \ldots \wedge E(x_m, x_{m-1}))) \qquad (2.52) \end{aligned}$$

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 (2.52) を計算すると，

$$\begin{aligned} & p(E(x_0, x_1, \ldots, x_{m-1}, x_m)) \\ = \; & p(E(x_0, x_1, \ldots, x_{m-1}))p(E(x_m, x_0) \wedge E(x_m, x_1) \wedge \ldots \wedge E(x_m, x_{m-1})) \\ = \; & p(E(x_0, x_1, \ldots, x_{m-1}))p(E(x_m, x_0))p(E(x_m, x_1)) \ldots p(E(x_m, x_{m-1})) \qquad (2.53) \end{aligned}$$

となる．式 (2.53) 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 (2.53) 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 (2.53) 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． ∎

2.3 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 にしたが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 にしたが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[⋆3] ．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 (2.16)(2.17)(2.18)(2.19)(2.21)(2.21)(2.22)(2.28)(2.29)(2.30)(2.31)(2.32)(2.33) より，

$$\begin{aligned} & M(C^{\sigma_y}(A); extend(S_x, i_\alpha, s)) - M(C^{\sigma_y}(A); S_x) \\ = & \sum_{b_y=\underline{i_\alpha - i_{\alpha-1}+1}}^{\underline{(i_\alpha+s)-i_{\alpha-1}}} (b_y - (\underline{i_\alpha - i_{\alpha-1}})) + \sum_{b_y=\underline{(i_\alpha+s)-i_{\alpha-1}+1}}^{\underline{i_{\alpha+1}-(i_\alpha+s)}} (\underline{s}) \\ & + \sum_{b_y=\underline{i_{\alpha+1}-(i_\alpha+s)+1}}^{\underline{i_{\alpha+1}-i_\alpha}} (-b_y + (\underline{i_{\alpha+1} - i_\alpha})) \end{aligned} \tag{2.54}$$

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，

$$\begin{aligned} & M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); S_x^1) \\ = & (M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); T^0)) - (M(C^{\sigma_y}(A); S_x^1) - M(C^{\sigma_y}(A); T^0)) \end{aligned} \tag{2.55}$$

と表せる．式 (2.55) 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 (2.54) 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 (2.54) を計算し，その総和を求めることで得られる．このように，

[⋆3] 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．

$M(C^{\sigma_y}(A); S_x^0) - M(C^{\sigma_y}(A); S_x^1)$ 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．