情報処理学会論文誌　プログラミング　Vol. 3　No. 1　1–40 (Mar. 2010)

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

原　健太朗[†1]　田浦　健次朗[†1]　近山　隆[†2]

計算環境の大規模化と並列分散アプリケーションの高度化が加速している現在，大規模な並列分散環境をサポート可能な並列分散処理系への要請が高まっている．大規模環境をサポートするうえでは，計算資源の動的な参加/脱退を越えて 1 つの並列計算を継続実行できるような枠組みが欠かせない．特に，より多様で広範な並列分散アプリケーションを支援するためには，クライアント・サーバ方式のように計算資源どうしが疎に結び付いて動作するような枠組みではなく，多数の計算資源がもっと密に協調して動作するようなアプリケーション領域に対しても，動的な参加/脱退を柔軟にサポートできるような枠組みが求められている．以上をふまえて本研究では，動的な参加/脱退をサポートする大規模分散共有メモリの処理系として DMI（Distributed Memory Interface）を提案して実装し，評価する．DMI では，動的な参加/脱退を可能とするコンシステンシプロトコルを新たに提案するとともに，動的な参加/脱退を実現するプログラムを容易に記述可能な pthread 型の柔軟なプログラミングインタフェースを整備する．さらに，DMI では，ユーザ指定の任意のサイズによるコンシステンシ維持，非同期 read/write，マルチモード read/write など，分散共有メモリの性能を改善するための明示的で細粒度な最適化手段を提供する．評価の結果，DMI は，ヤコビ反復法による熱伝導方程式の求解のように，単純なクライアント・サーバ方式では記述できず，多数の計算資源が密に協調しながら動作するアプリケーションに対しても，計算資源の参加/脱退を越えた計算の継続実行をサポートし，参加/脱退に相応して動的に並列度を増減できることを確認した．また，pthread プログラムとの記述上の類似性や最適化手段の有効性も確認した．

DMI: A Large Distributed Shared Memory Interface Supporting Dynamically Joining/Leaving Computational Resources

Kentaro Hara,[†1] Kenjiro Taura[†1]
and Takashi Chikayama[†2]

With increasing scale of computational environments and greater sophistication of parallel and distributed applications, there have been increasing demands for parallel and distributed frameworks supporting large scale computational environments. In large scale environments, frameworks are required which can execute one parallel computation continuously beyond dynamic joining/leaving of computational resources. Particularly it is not enough for frameworks to support only loosely coupled applications using a client-server model. Instead, frameworks should accommodate broader range of applications in which many resources can work in a tightly coupled manner. With these backgrounds, we propose, implement and evaluate Distributed Memory Interface, or DMI, a large distributed shared memory framework supporting dynamic joining/leaving of computational resources. DMI not only implements an original protocol for the memory consistency which enables dynamic joining/leaving, DMI also provides pthread-like flexible programming interfaces with which we can easily develop programs supporting dynamic joining/leaving. Furthermore DMI provides explicit and fine-grained optimization methods to improve a performance of a distributed shared memory, for example, consistency maintenance based on a user-specified granularity, asynchronous read/write and multi-mode read/write. Our performance evaluation confirmed that DMI can support a continuous computation beyond dynamic joining/leaving of resources and can dynamically increase/decrease the effective speedup according to the joining/leaving, even for tightly coupled applications which cannot be developed in a simple client-server model, such as Jacobi method for solving a partial differential equation. We also confirmed an expressive similarity with pthread programs and an effectiveness of proposed optimization methods.

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

1. 序　　論

1.1 背景と目的

近年では，情報産業に限らず，気象予測や衝突解析などの産業分野や実社会における並列分散アプリケーションの重要性が高まっている．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計算環境の技術革新にともない，適用可能な並列分散アプリケーションの領域や規模は飛躍的に拡大しており，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，計算資源の動的な参加/脱退のサポートは重要な要請の 1 つである．計算資源の動的な参加/脱退に対しては，参加/脱退を通じて動的にロードバランシングを図りたいというアプリケーション面からの要求と，計算資源は個人のものではないため，クラスタ環境の運用ポリシや課金制度などの都合上，利用中の計算資源を必要に応じて参加/脱退させなければならないという資源面からの要求がある．したがって，長大な計算時間を要する並列計算を実行中に，その計算を継続した状態で動的に計算資源を参加/脱退させたり，さらには参加/脱退を通じて計算環境をマイグレーションしたりできるような枠組みが求められている[44]．このような枠組みの代表例としてはクライアント・サーバ方式があるが，クライアント・サーバ方式に基づく単純なプログラム記述では，特定の計算資源に負荷が集中するためスケーラブルな処理は実現しにくく，このように計算資源どうしが疎に結び付いて動作するモデルの上で効率的に実行可能なアプリケーション領域は限定される．よって，より多様で高度なアプリケーションを支援するためには，多数の計算資源がもっと密に協調して動作するようなアプリケーション領域に対しても，動的な参加/脱退を柔軟にサポートできるような処理系が必要とされている[49]．さらに，動的な参加/脱退をサポートするうえでは，ユーザが動的な参加/脱退に対応したプログラムを容易に記述できるようなインタフェースの整備が欠かせない．

本研究では，分散共有メモリが動的な参加/脱退をサポートするうえで優れたプログラミングモデルであることに着眼し，多数の計算資源が密に協調して動作するアプリケーション領域に対しても動的な参加/脱退をサポート可能な分散共有メモリベース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．特に，DMI では，サーバのような固定的な計算資源を設置することなく動的な参加/脱退を実現できるようなコンシステンシプロトコルを新たに提案する．また，分散共有メモリベースの処理系を構築するという観点から，さらなる要請として，

（I） スレッドプログラミングとの類似性

（II） 分散共有メモリの性能を向上させるための明示的で細粒度な最適化手段の提供

（III） 多数のノードの遠隔メモリを集めた大規模メモリの実現

（IV） 並列分散ミドルウェア基盤としての柔軟で汎用的なインタフェースの整備

という 4 つの要請に焦点を当てた設計を施し，より多様な並列分散アプリケーションを支援できる処理系を構築する．

（I）に関して，近年の CPU の設計思想はマルチコア化を指向しており，共有メモリ環境上でのスレッドプログラミング，特に pthread プログラミングはいっそう身近なものになっている．そしてそれらの多くは，コア数やメモリ量などの資源面において，より大規模で強力な並列環境を求めている．しかし，大規模並列化のためのアプローチとしてはプログラムの分散化が考えられるものの，スレッドによる並列化は達成されていても分散化には至っていないアプリケーションが多い．この原因は，スレッドプログラミングと分散プログラミングとの間の飛躍の大きさにある．分散共有メモリが，分散環境上で仮想的な共有メモリ環境をシミュレートしているとはいえ，既存の分散共有メモリシステムにおけるインタフェースの細部を観察すると，SPMD 型のプログラミングスタイルや同期操作の記述方法などの点において，分散プログラミング特有の形態を採用しているシステムが多い．そのため，スレッドプログラムをこれらの分散共有メモリ上のプログラムに移植するには，シンタックスとセマンティクスの両面において，論理的な思考をともなう変換作業が要求されてしまう．以上をふまえて，DMI では，pthread プログラムに対してほぼ機械的な変換作業を施すことでプログラムが得られるようなインタフェース設計を行うことで，マルチコア並列プログラムを分散化させる際の敷居を下げることを目標とする．特に，排他制御の実現方式に関して，従来の多くの分散共有メモリがメッセージパッシングベースのアルゴリズムを採用しているのに対して，DMI では，fetch-and-store と compare-and-swap に基づく共有メモリベースのアルゴリズムを採用する．

（II）に関して，分散共有メモリはメッセージパッシングよりも抽象度の高いプログラミングモデルであるため，プログラム記述が容易である一方で，コンシステンシ管理などのオーバヘッドをともなうため台数効果が得られにくい．そこで DMI では，ユーザの指定した任意のサイズを粒度とするコンシステンシ維持，マルチモード read/write，非同期 read/write など，明示的で細粒度なアプリケーションの最適化手段を提供する．これにより，ユーザは，分散共有メモリの容易なプログラム記述や高抽象度な機能を享受しつつも，インクリメンタ

ルなチューニングによってアプリケーションの性能を改善できる．

（Ⅲ）に関して，大規模メモリは，モデル検査[16),17)]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アプリケーションをはじめとして，解ける問題の規模が利用可能なメモリ量によって制限されるようなアプリケーションにとって特に重要である．近年では，ネットワーク技術の向上により，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遠隔メモリへのアクセス時間の方が高速になっているため，リモートページングによって大規模メモリを実現する遠隔スワップシステムが出現している[10),50),52)]．DMI では，分散共有メモリに対して遠隔スワップシステムとしての機能を付加することで，CPU の意味でのスケラービリティだけではなく，メモリの意味でのスケーラビリティも達成できる処理系を構築する．

（Ⅳ）に関して，既存の並列分散処理系の中には，取り扱う処理の対象を特定の問題領域や抽象度の高いプログラミングモデルに特化することによって，処理系が想定する範囲内の分散処理であれば，より簡易な記述で効率的に実行できることを狙う処理系もある．しかし，できるだけ多様なアプリケーションの開発基盤となりうるような並列分散処理系を構築するためには，ユーザプログラムにおける記述作業の容易さを追求するよりも，ユーザプログラムに対して幅広い自由度を与えるような汎用的なインタフェース設計が重要である．したがって，DMI では，エンドユーザのための並列分散処理系というよりも，むしろ並列分散ミドルウェア基盤としての並列分散処理系を目標とし，OS のメモリ管理機構と同様の機能を分散環境上にユーザレベルで実装することなどを通じて，柔軟性，汎用性，機能拡張性に富んだインタフェースを整備する．

1.2　本研究の貢献

本研究の貢献は以下のとおりである：

- 分散共有メモリが計算資源の動的な参加/脱退に適したプログラミングモデルであることに着眼し，サーバのような固定的な計算資源を設けることなく，計算資源の動的な参加/脱退を実現できるコンシステンシプロトコルを新たに提案して実装している．
- 動的なスレッド生成/破棄が可能な pthread 型のプログラミングスタイルを採用することで，シンタックスとセマンティクスの両面においてマルチコア並列プログラミングとの対応性に優れ，かつ計算資源の動的な参加/脱退に対応したユーザプログラムを容易に記述できるようなインタフェースを提供している．
- ユーザの指定する任意のサイズを粒度とするコンシステンシ維持，非同期 read/write，マルチモード read/write など，ユーザプログラムに対して明示的で細粒度な最適化手段を提供している．

分散共有メモリは過去 20 年以上にわたって多数の実装が試されているが，多数の計算資源が密に協調して動作するようなアプリケーションに対しても計算資源の動的な参加/脱退をサポートし，計算環境の動的なマイグレーションを達成した研究事例は，我々の知る限りでは存在しない．

1.3　本稿の構成

2 章では，複数の並列分散プログラミングモデルの比較を通じて，DMI が，計算資源の参加/脱退をサポートするためのプログラミングモデルとして分散共有メモリを採用した根拠を述べる．3 章では，DMI の設計コンセプトや機能について述べる．4 章では，DMI のプログラミングインタフェースを紹介し，計算資源の参加/脱退に対応したプログラム記述例を示す．5 章では，実装について説明する．6 章では，マイクロベンチマークとアプリケーションベンチマークを用いた性能評価を行う．7 章では，関連する既存技術を紹介し DMI との比較を行う．8 章では，本稿の結論および今後の課題について述べる．

なお，本稿においては次のように用語を区別する．物理的な共有メモリを備えた通常の共有メモリ環境のことを「共有メモリ環境」，分散環境上に仮想的な共有メモリを構築するプログラミングモデルまたはそのシステムを「分散共有メモリ（システム）」，分散共有メモリによって構築される仮想的な共有メモリを「仮想共有メモリ」，特に DMI が構築する仮想共有メモリを「DMI 仮想共有メモリ」，DMI 仮想共有メモリ上に確保されるメモリを「DMI 仮想メモリ」と呼ぶ．

2.　並列分散プログラミングモデル

現在，並列分散プログラミングモデルとして代表的なものには，MPI[2)] のようなメッセージパッシング，TreadMarks[4)] や UPC[8)] のような分散共有メモリ，Java RMI[1)] や gluepy[48)] のような分散オブジェクト/RPC，OpenMP[3)] のようなコンパイラによる並列化技法などがある．本研究では，並列分散ミドルウェア基盤の構築を 1 つの目標として据えているため，抽象度の高いプログラミング形態や特定の問題領域に特化しない，汎用的なプログラミングモデルを基盤として採用するのが望ましい．したがって，以降ではメッセージパッシングと分散共有メモリに焦点を絞り，スケーラビリティ，記述力，ノードの動的な参加/脱退への適応力などの観点からその特徴を分析する．

2.1　メッセージパッシング

メッセージパッシングでは，系内の各ノードに対して一意なランク（名前）が与えられ，

ユーザプログラムではランクを用いたデータの送受信を記述することで，コネクション接続やトポロジ情報などを意識することなく，任意のノード間通信や集合通信を実現できる．メッセージパッシングの基本操作はランクを明示的に使用したデータ通信であるため，ユーザプログラム側で全ノードとランクの対応関係を把握し，データの所在を明示的に管理する必要がある．

メッセージパッシングの利点は，スケーラビリティの良さである．データの所在や通信形態がユーザプログラム側で管理されるため，処理系側で行うべきことは，ユーザプログラムによって指示されたデータ通信を，下層ハードウェアの提供するインタフェースに従って効率的に実現することに尽きる．よって，ユーザプログラムにおける記述（send/recv）が下層ハードウェアで実際に発生する操作（send/recv）にそのまま対応するため，ユーザプログラムにとって本来必要とされる通信以外は発生せず，通信に無駄が生じない[44]．また，データの所在管理や通信形態の決定に関してユーザプログラム側に自由度があるため，明示的なチューニングが行いやすく性能を引き出しやすい．さらに，gather や broadcast などの集合通信がサポートされており，処理系によって最適化されたトポロジ上の通信が実現できることも，メッセージパッシングのスケーラビリティを支える重要な要素となっている[11),40),46)]．

一方で，メッセージパッシングの第 1 の欠点はプログラミングの負担の大きさである．ユーザプログラム側でデータの所在や通信形態を管理しなければならないため，データの流れが比較的単純な並列計算などは容易に記述できる一方で，動的で不規則なデータ構造を取り扱うような非定型な処理は非常に記述しにくい．例としては，共有メモリ環境上でポインタを複雑に書き換えるような処理，たとえば節数が動的に変化するようなグラフ構造を取り扱う処理などを，メッセージパッシングで記述することは難しい．なかには，Phoenix[44),49)] のように，通信先などの管理をユーザプログラム側から隠蔽する手法も試みられているが，小さくないオーバヘッドがあり，ユーザプログラム側からそのオーバヘッドを予測しにくいという問題がある．また，メッセージパッシングは共有メモリ環境上のプログラミングとは本質的にモデルが異なるため，既存の共有メモリ環境上の並列プログラムを翻訳する際にはアルゴリズムレベルからの再検討が要求される．

第 2 の欠点として，メッセージパッシングはノードの動的な参加/脱退に適していない．まず，ノードの参加/脱退時におけるランクの割当てをどう行うべきかが問題である．たとえば，$0, \ldots i, \ldots N-1$ のランクを持つノードたちが協調して動作している状況で，ランク i のノードが単純に脱退するとランクの連続性が崩れてしまう[44]．さらに，ノードの参加/脱退イベントをユーザプログラム側にどうハンドリングさせるかも大きな問題である．メッセージパッシングではランクを明示的に使用したデータ通信を記述しなければならないため，ノードの参加/脱退イベントが発生した場合には，それをユーザプログラム側でハンドリングして，全ノードとランクとの対応関係を更新するためのコーディングが何らか必要になると考えられるが，その記述は相当に煩雑化すると予想される．実際，MPI2 が動的なプロセス生成をサポートしているが記述は複雑である[2)]．このように，メッセージパッシングはノードの動的な参加/脱退を記述しにくいモデルであるが，その主因は，データの所在管理をユーザプログラム側で行わなければならない点にある．

2.2　分散共有メモリ

2.2.1　特　　徴

分散共有メモリとは，物理的には分散した計算資源上に仮想的な共有メモリを構築することによって，あたかも共有メモリ環境上の並列プログラミングと同様の read/write ベースのインタフェースで分散プログラムを記述可能とするプログラミングモデルである．分散共有メモリにおける通信媒体は処理系によって管理される仮想共有メモリであり，ユーザプログラム側では，この仮想共有メモリに対して read/write を発行することで，所望のデータを操作するために必要なノード間通信が処理系によって実現される．すなわち，分散共有メモリは，メッセージパッシングとは異なり，データの所在がユーザプログラム側ではなく処理系側で管理されるモデルである．

分散共有メモリの利点としては，まず第 1 に記述力の高さがある．分散共有メモリでは，共有メモリ環境上の並列プログラムと同様の read/write ベースの記述が可能なため，ユーザは，各ノード上のデータ配置や煩雑なメッセージ通信を意識することなく並列アルゴリズムの開発に専念できる[4)]．また，メッセージパッシングと比較して，既存のマルチコア並列プログラムを分散化させる際の負担も小さい．第 2 の利点として，ノードの動的な参加/脱退に対する適応力が高い．分散共有メモリではデータの所在管理が処理系によって行われており，ユーザプログラムにおけるデータ操作は，ノードが何台参加しているかにかかわらず，全ノードにとって共通の仮想共有メモリを介して実現されるため，ユーザプログラム側でのデータの所在管理が必要ない．よって，系内にどのノードが存在しているかに関する情報がユーザプログラム側では不要なため[47)]，ノードの動的な参加/脱退が容易に記述できる．第 3 の利点として，応用範囲の広さがある．たとえば，リモートページングやプロセスマイグレーションなどの技術が，分散共有メモリをベースとして実現されている[21)]．これらの各種応用は，分散共有メモリの並列分散ミドルウェア基盤としての強力さを示してい

る．以上の観察から，ノードの動的な参加/脱退を実現する並列分散ミドルウェア基盤の構築を目指す本研究では，分散共有メモリをベースとするのが適当であると判断した．

しかし，一方で，分散共有メモリの欠点はパフォーマンスの引き出しにくさにある．分散共有メモリは，実際に内部的に発生するデータの送受信をユーザプログラムに対して隠蔽し，それを read/write ベースのインタフェースとして見せかけるモデルであり，その抽象度の高さゆえに，メッセージパッシングと比較すると性能面で劣ってしまう[7),21)]．性能劣化の具体的な要因としては，ユーザプログラム側の操作（read/write）と処理系側で実際に起こる動作（send/recv）が対応しないために，ユーザプログラムから処理系の挙動が把握しにくく明示的なチューニングが施しにくい点，ユーザプログラムにとって本来必要な通信パターンが何なのかを処理系側で把握できないために，無駄なメッセージ通信が多量に発生する可能性が高い点，集合通信の最適化が行いにくい点[11)]などが指摘できる．

2.2.2　処理系のアプローチ

以上の事実を背景として，従来の分散共有メモリの研究では，できるだけ通信量を減らして性能を向上させるための多種多様なアプローチが，ハードウェアとソフトウェアの両面から考案されてきた．通信量を減らすための鍵は，極力 demand driven なデータ転送を行うことにある．以下では，ソフトウェア分散共有メモリが demand driven なデータ転送を実現するうえでの重要な設計項目として，コンシステンシモデル，コンシステンシプロトコルのデザイン，コンシステンシ維持の単位の 3 点について取り上げる．

第 1 に，コンシステンシモデルについて考える．分散共有メモリのコンシステンシモデルに対するアプローチとしては，Sequential Consistency，Weak Consistency，Eager/Lazy Release Consistency，Entry Consistency などが代表的であり，この順にコンシステンシ制約が緩和される．制約が緩和されるほど無駄な通信を抑制できて性能が向上するため，Sequential Consistency の IVY[23)]，Eager Release Consistency の Munin[19)]，Lazy Release Consistency の TreadMarks，Entry Consistency の Midway[36)] など，従来の分散共有メモリシステムではコンシステンシモデルの緩和が積極的に試されてきた[51)]．しかし，コンシステンシモデルの緩和はプログラミングの容易さ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プログラムの直観的な動作が掴みにくくなるうえ，共有メモリ環境上のプログラムからの飛躍も大きくなる．そのため，コンシステンシモデルの緩和は分散共有メモリとしての良さを失っているという見解もある[21),42)]．一方で，このトレードオフを相殺するために，複数のコンシステンシモデルをサポートする分散共有メモリシステムも提案されている[5),33)]．このようなシステムでは，初期的には容易な Sequential Consistency で開発し，段階的に緩和型コンシステンシへとチューニングしていくようなインクリメンタルな開発が可能となる．

第 2 に，コンシステンシプロトコルのデザインについて考える．コンシステンシモデルを実現するためのプロトコル設計に関しては多様なデザイン項目が存在し，それぞれの分散共有メモリシステムの設計コンセプトに合致したものが選択される．たとえば以下のようなデザイン項目がある：

共有データへの読み書き　ある共有データに対して，同時に何ノードの read/write を認めるかに応じて，分散共有メモリのプロトコルは，Single Writer/Single Reader 型，Single Writer/Multiple Reader 型，Multiple Writer/Multiple Reader 型に分類できる．複数ノードによる read/write の並列性を高めるうえでは Multiple Writer/Multiple Reader 型が最も望ましいが，プロトコルが複雑化してコンシステンシ管理のオーバヘッドが多大になるため，Single Writer/Multiple Reader 型を採用するシステムが多い．

共有データ更新時の挙動　Single Writer/Multiple Reader 型のプロトコルでは，read を発行したノードに対してデータをキャッシュするが，write によるデータ更新時に，これらのキャッシュを無効化する invalidate 型のプロトコルと，キャッシュを保持するノードに対して最新データを送りつけることでキャッシュをつねに最新状態に保つ update 型のプロトコルが存在する．一般には，write された結果が read される確率が高い場合には update 型，その確率が低い場合には invalidate 型のプロトコルが適しているが，多くのアプリケーションでは確率が低い傾向にあるため，ほとんどの分散共有メモリシステムでは invalidate 型のプロトコルが採用されている．

オーナーの所在管理　一般に，分散共有メモリのプロトコルでは，コンシステンシを維持する単位となる共有データごとに，その共有データに関するさまざまな情報を管理するオーナーが存在する．そして，アクセスフォルトが発生した場合には，オーナーに通知が送られ，オーナーによってコンシステンシ維持のための処理が行われる結果，アクセスフォルトが解決されるような原理になっている．したがって，各ノードはアクセスフォルト発生時などにオーナーに対して通知を送る必要があるため，何らかオーナーの所在に関する情報を知っている必要があるが，その管理技法には，オーナー固定型，ホーム問合せ型，オーナー追跡型のアプローチが存在する[21)]．オーナー固定型では，プログラム開始から終了までオーナーを特定のノードに固定する．ホーム問合せ型では，オーナーを動的に変化させるが，オーナーの位置をつねに把握するホームと呼ばれるノードを固定的に設置し，オーナーを見失った場合にはホームに問い合わせることで

オーナーの位置を解決する．オーナー追跡型では，オーナーを動的に変化させるがホームノードを設置せず，代わりに各ノードに probable owner という情報を持たせる．各ノードの probable owner は真のオーナーを参照しているとは限らないが，全ノードを通じた probable owner の参照関係が，任意のノードが必ず真のオーナーに到達可能なグラフ（以下，このグラフをオーナー追跡グラフと呼ぶ）を形成するように管理される[23]．よって，オーナー追跡型では，アクセスフォルト時のリクエストなどは，各ノードが知っている probable owner の方向へリクエストをフォワーディングすることで，やがてオーナーにリクエストを到達させることができる．オーナー追跡型は，ホームのような固定的なノードを必要としないため動的環境との相性が良い．ただし，どのオーナーの管理技法が優れているかに関しては，各分散共有メモリシステムの設計コンセプトやコンシステンシモデルに依存する部分が大きく，特定ノードへの負荷分散を回避する目的でホーム問合せ型やオーナー追跡型を実装するシステムもあれば，通信の複雑化を避ける意味でオーナー固定型を採用するシステムもある．

第 3 に，コンシステンシ維持の単位について考える．コンシステンシを維持する単位に関するアプローチとしては，page-based，object-based，region-based な手法が代表的である：

page-based　OS のページサイズをコンシステンシ維持の単位とし，OS のメモリ保護違反機構を利用してアクセスフォルトを検出してコンシステンシプロトコルを走らせる．この手法の利点は，ローカルメモリへの操作と同様の記述で仮想共有メモリへのアクセス操作を記述できる点，妥当な仮想共有メモリへのアクセスに対しては通常のローカルメモリへのアクセスと等価であるためオーバヘッドが小さい点などである．その反面，OS の機構に依存するためシステムの可搬性が損なわれる点，コンシステンシ維持の単位がページサイズ（の整数倍）に固定されるためユーザプログラムの振舞いに合致した粒度でのコンシステンシ維持が不可能な点，それゆえアクセスフォルトの頻発やフォルスシェアリングを引き起こしやすい点などが欠点である[29),42)]．処理系としては，IVY，TreadMarks，DSM-Threads[32),33),39)]，SMS[51)] などで採用されている．

object-based　そのシステムが基盤とする言語上で定義されるオブジェクトをコンシステンシ維持の単位とし，各オブジェクトに対してユーザレベルでアクセスフォルトの検査が行われる．この手法の利点は，ユーザプログラムの指定する任意の粒度のオブジェクトでコンシステンシが維持できるためフォルスシェアリングが発生しにくい点，ソフトウェア的なコンシステンシ管理を行うので OS への依存性を排除できる点などである[29),42)]．その反面，妥当なアクセスに対しても逐一ソフトウェア的な検査が入るためオーバヘッドが大きい点や，オブジェクトという単位が必ずしも適切ではないアプリケーションも存在し，仮想共有メモリをバイト列の連続領域として提供する page-based なアプローチよりも汎用性に劣る点などが欠点といえる[4)]．処理系としては，ObFT-DSM[27),29)] などで採用されている．

region-based　region と呼ばれる任意サイズの連続領域をコンシステンシ維持の単位とし，アクセスフォルトの管理は，各 region に対してユーザレベルで行うか，もしくは各 region をローカルなメモリにマップすることで OS に依頼する．この手法の利点は，region のサイズをユーザプログラムから任意に動的に指定できるため，page-based のように仮想共有メモリを連続領域として提供しつつも，page-based なアプローチで問題となっていたアクセスフォルトの多発やフォルスシェアリングの問題を回避している点である．この手法は page-based と object-based のハイブリッド型のアプローチといえる．処理系としては，CRL[25)]，HIVE[5)]，研究 24) などで採用されている．

3. コンセプト

3.1 対象とする分散処理

DMI の最大の目的はノードの動的な参加/脱退のサポートであるため，ノードの動的な参加/脱退に相応して動的に並列度が増加/減少する処理でなければ，わざわざノードを参加/脱退させる意味がない．よって，DMI が対象とすべき分散処理は，数台程度までしかスケールしないような処理ではなく，ある程度のスケーラビリティを発揮できる処理である．したがって，DMI は，細粒度なデータアクセスが頻繁に干渉するような処理ではなく，データアクセスがある程度粗粒度でアクセス干渉の少ないような処理を対象として設計する．当然，このようにデータアクセスが粗粒度で干渉の少ない処理は，分散共有メモリにおけるプログラミングの容易さを持ち出さなくても，メッセージパッシングで十分記述できる場合が多いが，本研究があえて分散共有メモリをベースとする理由は，2.2.1 項で述べたように，分散共有メモリではデータの送受信が仮想共有メモリへのアクセスに抽象化されるためノードの参加/脱退を容易に記述できるからである，という点を強調したい．また，動的な参加/脱退が本当に必要となるのはマルチクラスタ環境などの大規模な WAN 環境であると考えられるが，さしあたって現状の DMI は，ネットワーク的に均質な単一クラスタ内の LAN 環境を想定して設計する．

3.2 大規模分散共有メモリの構成

DMI が実現する大規模分散共有メモリの構成を**図 1** に示す．DMI では，各ノードが一定

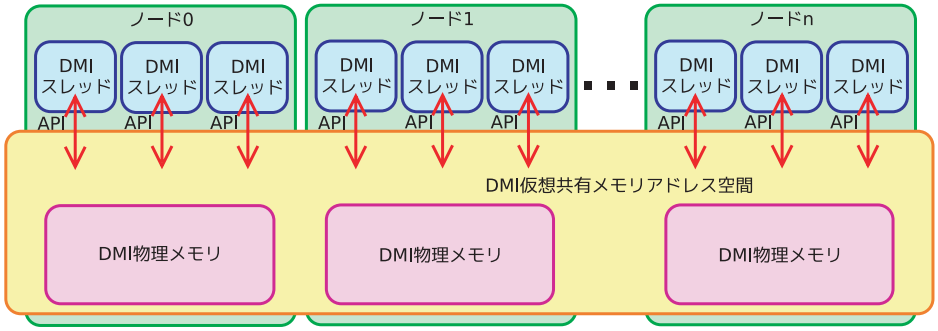

図 1　DMI における大規模分散共有メモリの構成
Fig. 1　System components of DMI.

量のメモリを DMI 物理メモリとして DMI に提供する．DMI は，これら各ノードの DMI 物理メモリをメモリリソースとして扱い，ページテーブルやアドレス空間記述テーブルなどの OS のメモリ管理機構と同様の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 DMI 仮想共有メモリを構築する．

DMI では，pthread 型のプログラミングスタイルを採用しており，プログラム開始時に `DMI_main(...)` が実行され，以降，ユーザプログラムが `DMI_create(...)` を適宜呼び出すことによって，ユーザが指定するノード上に DMI スレッドが生成されていくという形態をとる．DMI スレッドは同一ノード上に任意個生成可能である．したがって，DMI の各ノードは，図 1 に示すように複数の DMI スレッドが DMI 物理メモリを共有する構造となる．そのため，各ノードの DMI 物理メモリが複数の DMI スレッドの “共有キャッシュ” の役割を果たすことが期待でき，マルチコアレベルの並列性と分散レベルの並列性を統合的に活用できる設計になっている．

また，任意の DMI スレッドは，DMI 仮想共有メモリに対して read/write を発行することで，透過的に全ノードが提供する DMI 物理メモリを利用することができる．したがって，DMI は並列分散プログラミング環境としての分散共有メモリとしての機能と同時に，メモリの意味でのスケーラビリティを実現する遠隔スワップシステムとしての機能も兼ね備えている．このような遠隔スワップシステムにおいては，リモートページングを繰り返すうちに DMI 物理メモリの使用量が飽和してしまう場合があるため，DMI では，適宜ページアウトを行うためのページ置換アルゴリズムを実装している．ページ置換アルゴリズムの実装については 5.5 節で述べる．

DMI の処理系はすべてユーザレベルで実装されており，OS やコンパイラにはいっさい手を加えていないため移植性が高い．

3.3　メモリモデル

DMI では，図 1 のように，DMI スレッドが使用するメモリ空間と DMI 仮想共有メモリのメモリ空間は明確に分離されており，DMI スレッドからは関数呼び出しを通じて，DMI 仮想共有メモリへの read/write やメモリ確保/解放などの各種メモリ操作を発行する形態をとる．

read に関しては，`DMI_read(addr, size, buf, mode)` を呼び出すことで，DMI 仮想共有メモリにおけるアドレス `addr` から `size` バイトを，DMI スレッドのメモリ空間におけるアドレス `buf` に読み込むことができる．write に関しては，`DMI_write(addr, size, buf, mode)` を呼び出すことで，DMI スレッドのメモリ空間におけるアドレス `buf` から `size` バイトを，DMI 仮想共有メモリにおけるアドレス `addr` に書き込むことができる．メモリ確保に関しては，`DMI_mmap(page_size, page_num)` によって，ページサイズが `page_size` のページを `page_num` 個確保することができる．ここで，ページとは DMI がコンシステンシを維持するうえでの単位のことであり，よって，ページサイズとはコンシステンシを維持する単位のサイズのことである．このように DMI では，ページテーブルなどのメモリ管理機構をユーザレベルで自前で管理することによって任意のページサイズによるコンシステンシ管理を可能とし，柔軟な region-based なアプローチを実現する．この仕組みにより，OS のメモリ保護違反機構を利用する多くの分散共有メモリでは OS のページサイズ（の整数倍）の単位でしかコンシステンシを維持できないのに対して，DMI では，ユーザプログラムの振舞いに合致した任意のページサイズのメモリを確保できる．たとえば，巨大な行列行列積をブロック分割によって並列に行いたい場合には，各行列ブロックのサイズをページサイズに指定して行列用の DMI 仮想メモリを割り当てればよい．このように DMI では，ページサイズが固定されている page-based な分散共有メモリと比較すると，ページフォルトの回数を大幅に抑制できるため，データ通信が不必要に細分化されることがなく通信上のオーバヘッドが小さい．

なお，`DMI_read(...)/DMI_write(...)` で指定するアドレスはページ境界にアラインされている必要はなく，複数ページにまたがる領域を指定することも可能である．

3.4　コンシステンシ管理

DMI では，コンシステンシモデルとして，ページをコンシステンシ維持の単位とし

た Sequential Consistency を採用している．よって，DMI_read(...)/DMI_write(...) で複数ページにまたがる領域を指定した場合には，その呼び出しがページ単位の “小 DMI_read(...)”/“小 DMI_write(...)” に分割され，それらに関する Sequential Consistency が保証される．したがって，複数ページにまたがって DMI_read(...)/DMI_write(...) を呼び出す場合には，必要に応じて排他制御を行う必要がある．DMI が Sequential Consistency という強いコンシステンシモデルを選択した理由は，ページサイズを任意に指定可能とすることでコンシステンシの強度に由来する性能劣化をある程度補えると考え，論理的な直観性を優先させたためである．

次に，コンシステンシプロトコルについて考える．ノードの脱退を実現するためには，そのノードが DMI 物理メモリ内に保有しているページを脱退前に他ノードに対して追い出す必要があることや，ページ置換時にはページを追い出す必要があることをふまえると，DMI にはページを追い出すためのプロトコルが定義される必要がある．また，DMI のような動的環境下ではホームノードのような固定的なノードを設置できない．よって，DMI においては，固定的なノードを設置することなくページフォルトのハンドリングとページの追い出しを実現するプロトコルが必要である．このうちページフォルトのハンドリングに関しては，2.2.2 項で述べたオーナー追跡グラフを用いるアプローチで解決できる．しかし，ページの追い出しに関しては，我々の把握する限りでは，従来の研究で提案されているプロトコルは，どれも各ページに対して固定的な帰属ノードを設置することを前提としており[10),12),20),50)]，DMI に適用することはできない．そこで DMI では，5.3 節に述べるような，固定的なノードを設置することなくページフォルトのハンドリングとページの追い出しを実現するプロトコルを新たに提案して実装する．

3.5　関数呼び出し型の read/write

3.5.1　利点と欠点

DMI では DMI_read(...)/DMI_write(...) による関数呼び出し型のインタフェースを基本としているが，この方式は，既存の多くの分散共有メモリが採用している OS のメモリ保護違反機構を利用する方式と比較して，以下のような利点と欠点がある．

第 1 の利点は，read/write を関数呼び出し型とすることで，DMI_read(...)/DMI_write(...) に引数としてさまざまな情報を与えることが可能になり，ユーザプログラムに対して明示的なチューニングの自由度を与えるような柔軟な read/write のインタフェースを提供できる点である．その一例として，後述するマルチモード read/write や非同期 read/write がある．このようなインタフェース設計の下では，ユーザは，初期的な開発段階ではごく普通の DMI_read(...)/DMI_write(...) を記述しておき，性能改善が必要になった段階で細粒度なチューニングを試すことができるため，DMI では，段階的にプログラムをチューニングしていくようなインクリメンタルな開発が行いやすい．第 2 の利点は，関数呼び出し型とすることで，ユーザレベルによるメモリ管理が可能になる点があげられる．これにより，先述のように任意のページサイズをサポートすることで効率的なデータ通信が可能となるほか，OS のアドレッシング範囲にとらわれないリモートページングも実現できる．従来の遠隔スワップシステムの多くは，OS のメモリ保護違反機構を利用するために 64 ビット OS を前提としていたのに対して，DMI では 32 ビット OS を多数連結することで大容量の DMI 仮想共有メモリを構築することが可能である．また，ユーザレベルで実装しているため，さらなる機能拡張に対する自由度も高い．第 3 の利点は，ユーザプログラムのメモリ空間と DMI 仮想共有メモリのメモリ空間が明確に分離されるため，ローカルとリモートを明確に意識したプログラミングが強制され，結果的に効率的なプログラムが開発されやすい傾向がある点があげられる．

一方で，欠点としては，第 1 に，関数呼び出しを通じてしか DMI 仮想共有メモリにアクセスできないため，ユーザプログラムの記述が面倒になる点があげられる．しかし，DMI では Sequential Consistency を保証していることもあり，確かに作業的には面倒であるが，プログラミングが論理的に難解なわけではない．第 2 の欠点としては，ユーザプログラムのメモリ空間と DMI 仮想共有メモリのメモリ空間を分離しているために，DMI 仮想共有メモリにアクセスするたびにメモリコピーが発生してしまう点がある．第 3 の欠点としては，OS のメモリ保護違反機構を利用する場合には，ページフォルトが発生しない場合には，仮想共有メモリへのアクセスがローカルメモリへのアクセスと完全に同じオーバヘッドで高速に実現できるのに対して，DMI では，ページフォルトが発生しないアクセスに対しても逐一ユーザレベルでの検査が入るため，オーバヘッドが大きいという問題がある．

3.5.2　マルチモード read/write

並列分散プログラムを最適化するうえでは，データの所在を把握して適切に管理することがきわめて重要である．したがって，DMI においても，ユーザプログラム側からデータの所在管理を明示的に行えるようなインタフェースを提供するのが望ましい．しかし，2.2.1 項で述べたように，分散共有メモリの本質は，データの所在管理がユーザプログラム側ではなく処理系側で行われる点にあり，この性質こそがノードの参加/脱退を容易に記述可能としていることをふまえると，あからさまにユーザプログラム側からデータの物理的な所在を管理できるインタフェースを提供することはできない．そこで，DMI では，処理系が

DMI_read(...)/DMI_write(...) を実行する際にどのようにデータを read/write するのかを，ユーザプログラム側から指示可能とするインタフェースを提供する．

まず，データを read する際の状況としては，

- 今行おうとしている 1 回の read だけが最新ページを読めればよく，今後しばらくは read しないので，最新ページを自ノードにキャッシュする必要がない状況
- それなりに read を繰り返すので，read した最新ページは自ノードにキャッシュしておきたいが，ページが write される頻度と比較して read する頻度が低くないため，ページが write される際には自ノードのキャッシュが無効化されてもよい状況
- ページが write される頻度と比較して read する頻度が高いため，自ノードのキャッシュをつねに最新ページに保っておきたい状況

が考えられる．一方，データを write する際の状況としては，

- オーナー権を持つノードに書き込みたいデータを送信することで，オーナー権を持つノードに write してもらいたい状況
- まず自ノードに最新ページとオーナー権を移動したうえで，自ノードで write したい状況

が考えられる．そして，実際にどう read/write するのが最適であるかは，アプリケーションの各局面や各データに大きく依存すると考えられる．以上をふまえ，DMI では，処理系がどう read/write を実行するのかを，DMI_read(...)/DMI_write(...) の粒度で指示可能とすることで，細粒度で明示的なアプリケーションの最適化を可能とする．従来の分散共有メモリでは，処理系がどう read/write するかはプロトコル単位で固定されていることが多く，DMI のように，アプリケーションの挙動に合致した細粒度なチューニングを施すことはできない．マルチモード read/write の実装に関しては 5.2 節で述べる．

3.5.3　非同期 read/write

DMI では，DMI_read(...)/DMI_write(...) などの同期モードの関数に対して，それに対応する非同期モードの関数を提供している．ユーザは，非同期モードの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でき，ネットワーク通信をともないうる DMI_read(...)/DMI_write(...) とローカルな計算をオーバラップさせることができる．

3.6　排 他 制 御

分散共有メモリを構築するにあたっては，排他制御の機能が欠かせない．緩和型のコンシステンシモデルを採用する場合には，排他制御は仮想共有メモリのコンシステンシ管理の一部として実現されるが，DMI のように Sequential Consistency を採用する場合には，排他制御の実現方式と仮想共有メモリのコンシステンシ管理は別の問題であるため，排他制御をどう実現すべきかのアルゴリズムを検討する余地がある．排他制御のアルゴリズムには，大きく分類して，分散環境におけるメッセージパッシングベース（send/recv）のアルゴリズムと，共有メモリ環境における共有メモリベース（read/write）のアルゴリズムが存在する．従来のほとんどの分散共有メモリはメッセージパッシングベースの排他制御を実装しているが，DMI では，分散共有メモリで作り出した仮想共有メモリを利用して共有メモリベースの排他制御を実装する．以下ではこの根拠を述べる．

3.6.1　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分散アルゴリズムは，permission-based なアルゴリズムと token-based なアルゴリズムに大別できる[26),41)]．

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するためには，適当なノード集合に対して要求を送信し，そのうち一定数のノードたちから許可通知を受け取る必要がある．permission-based なアルゴリズムの例としては，ノード数を N としたとき，各クリティカルセクションあたり，$O(3(N-1))$ のメッセージ数を要する Lamport のアルゴリズム[22)]，$O(2(N-1))$ のメッセージ数を要する Ricart Agrawala のアルゴリズム[38)]，$O(\sqrt{N})$ のメッセージ数を要する Maekawa のアルゴリズム[28)] などがある．

一方，token-based なアルゴリズムでは，系内に token が 1 個だけ存在し，token を所有するノードだけがクリティカルセクションに突入する権利を持つ．よって，クリティカルセクションに突入するためには，token を所有するノードに向けて token 要求を送信し，token を所有するノードに token を譲ってもらう必要がある．token-based なアルゴリズムの例としては，平均メッセージ数が $O(\log N)$ の Naimi らのアルゴリズム[35)]，平均メッセージ数は $O(\log N)$ 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[37)]，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[26)] などがある．一般に，token-based なアルゴリズムは permission-based なアルゴリズムと比較して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，ノード数やコンテンションの程度に大きく依存することが指摘されている[18),41)]．分散共有メモリでは，たとえば DSM-Threads が token-based なアルゴリズムを用いて排他制御を実現している．

また，分散アルゴリズムとはいえないが，SMS のように，特定のノードにロックを管理させるような centralized なロックマネージャ方式を採用する分散共有メモリもある．ロックマネージャ方式では，各ノードは，ロックマネージャに対してロック要求を送信し，その

応答を受け取ることでクリティカルセクションに突入できる．

以上をふまえ，DMI にとってメッセージパッシングベースの排他制御が不適であると考える理由は，次の 2 点である．

第 1 の理由は，メッセージパッシングベースの同期機構における階層関係は，共有メモリ環境の同期機構における階層関係と一致しない点である．まず共有メモリ環境の同期機構を観察すると，共有メモリ環境における最も基礎的な命令である read と write と，プロセッサによって提供される fetch-and-store や compare-and-swap などの read-modify-write のアトミック命令を上手に組み合わせることで，排他制御変数や条件変数が実現されるという階層関係になっている（**図 2** (A)）．ここで重要な点は，共有メモリ環境上のプログラムはこの階層関係を前提として設計されるという点である．たとえば，共有メモリ環境においては，lock-free や wait-free なデータ構造が数多く提案されている．これらのデータ構造は，排他制御変数を使った重いロック操作を回避するために，compare-and-swap を用いてデータ構造に高速にアクセスすることを可能にしているが，この考え方の根本には，「排他制御変数は compare-and-swap よりも重い」という前提が存在している．そのほか，効率化のために，排他制御変数の代わりに compare-and-swap を利用する共有メモリ環境上のプログラムは多い．したがって，共有メモリ環境上のプログラムとのセマンティクス的な対応性を確保した分散共有メモリを構築するうえでは，共有メモリ環境の同期機構の階層関係もそのまま反映させることが重要であるといえる．さて，この観点で分析すると，ロックマネージャや token によるメッセージパッシングベースの排他制御では，この階層関係を実現できないことが分かる．なぜなら，ロックマネージャや token が直接的に実現するのは排他制御変数や条件変数であって，read-modify-write ではないからである（図 2 (B)）．当然，これら排他制御変数と条件変数を組み合わせることで，read-modify-write と “機能的に” 同等の命令を作ることは可能であるが，これは共有メモリ環境の同期機構の階層関係と対応する設計ではない．よって，メッセージパッシングベースで排他制御変数や条件変数を実装するような設計では，wait-free なデータ構造などの，高度に効率的な共有メモリベースのアプリケーションをサポートすることは難しい．

第 2 の理由は，メッセージパッシングベースのアルゴリズムでは，仮想共有メモリのコンシステンシ管理とは別に，排他制御用のデータを管理するためのコンシステンシ管理が必要になる点である．たとえば，token-based なアルゴリズムの場合には，まず token を所有するノードに向けて token 要求を送信するが，token を所有するノードは時々刻々変化するため，オーナー追跡グラフのような要領で token 要求をフォワーディングする仕組みが必要

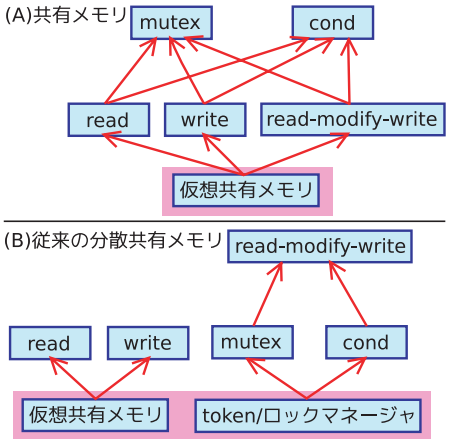

図 2　同期機構の階層関係（(A) 共有メモリベースの同期機構，(B) メッセージパッシングベースの同期機構）
Fig. 2　Hierarchy of synchronization mechanisms ((A) Synchronization based on a shared memory model, (B) Synchronization based on a message passing model).

である．また，多くの token-based なアルゴリズムでは，token 要求を受信した時点ですぐに token を譲れない場合には，その token 要求をキュー構造に入れて管理するなどの作業が必要になる[26),34),35),37)]．このように，token を使って排他制御を実現する場合には，仮想共有メモリのためのコンシステンシ管理とは別に，token のためのコンシステンシ管理が必要になる．そして，これは，分散共有メモリに対して何らか新たな機能を実装しようとする際には，仮想共有メモリと token の両方のコンシステンシプロトコルを設計しなければならないということを意味する．たとえば DMI の場合には，ノードの参加/脱退への対応やページの追い出しの機能を追加しようとする際に，仮想共有メモリと token の両方のプロトコルを設計する必要が生じる．このような設計は冗長であり，システム開発の観点から見てもスケーラブルではない．以上の観察により，Sequential Consistency を採用する分散共有メモリの実装においては，コンシステンシ管理の対象は仮想共有メモリに一本化し，仮想共有メモリを利用した共有メモリベースのアルゴリズムによって排他制御変数や条件変数を実装する方が優れているといえる．

3.6.2　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，read/write のほかに，read-modify-write として何を前提として設計されているかによって分類できる．具体的には，read/write/fetch-and-

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

図 3　共有メモリベースの排他制御におけるプロセスモデル
Fig. 3　A process model of mutual exclusion based on a shared memory model.

store/compare-and-swap を用いるアルゴリズム，read/write/fetch-and-store を用いるアルゴリズム，read/write のみを用いるアルゴリズムなどが研究されている[13]．これらのアルゴリズムは，実行環境としてキャッシュコヒーレントなマルチコア共有メモリ型マシンが主に想定されており，プロセスローカルでない変数へのアクセス回数（= リモートキャッシュへのアクセス回数）が，アルゴリズム評価の指標とされる場合が多い[13),15),45)]．たとえば，read/write/fetch-and-store/compare-and-swap を用いる MCS アルゴリズム[31)] では，各クリティカルセクションあたり，プロセスローカルでない変数へのアクセス回数が $O(1)$ であり，read/write のみを用いる Yang Anderson のアルゴリズム[45)] では，プロセス数 N に対して，プロセスローカルでない変数へのアクセス回数が $O(\log N)$ である．

一般に，共有メモリベースの排他制御に関する研究では，各プロセスが独立に**図 3** の構造を実行するようなプロセスモデルを考え，Entry Section と Exit Section をどうデザインするかが論じられる[13),15),45)]．具体例として，この構造に沿った MCS アルゴリズムを**図 4** に示す[13)]．MCS アルゴリズムの説明は本稿の意図ではないので省くが，図 4 を見ると，待機のスピンがプロセスローカルな変数で行われるため，プロセスローカルでない変数へのアクセス回数が $O(1)$ であることが分かる．

このように，強力な read-modify-write を使用する方が効率的な排他制御アルゴリズムを実現できるため，DMI では，fetch-and-store と compare-and-swap を実装し，それを基盤とする共有メモリベースの排他制御を実装する．

3.7　pthread 型のプログラミングスタイル

3.7.1　ノードの動的な参加/脱退に対する記述力

DMI では，ノードの動的な参加/脱退を容易に記述可能とするため，従来の多くのメッセージパッシングシステムや分散共有メモリシステムが採用している SPMD 型のプログラミングスタイルではなく，動的なスレッドの生成/破棄を記述できる pthread 型のプログラミングスタイルを採用する．SPMD 型は，メッセージパッシングであれば集合通信，分散共有メモリであればバリアなどの集団操作が定義でき，それらの集団操作を処理系によって最適化できるなどの利点を持つ．しかし，その反面 SPMD 型は，時系列的に “全員” がどう動作するかが静的に明確になっている並列計算などのアプリケーションの記述に特化したプログラミングスタイルであり，“全員” の時系列的な挙動が動的に決定されるような非定型な処理は非常に記述しにくい．また，そもそもノードの動的な参加/脱退を実現するには，ユーザプログラムに対して “全員” という概念を隠蔽する必要があるため，ノードの動的な参加/脱退をサポートするうえでは SPMD 型は適していない．これに対して，動的なスレッドの生成/破棄を記述可能な pthread 型は，SPMD 型よりも汎用的なプログラミングスタイルであり，動的な参加/脱退に応じた動的な並列度変化をプログラムとして容易に記述可能である．したがって，DMI では，pthread 型のプログラミングスタイルを採用す

```
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();
    p->next = NULL;
    pred = fetch_and_store(&tail, p);
    if(pred != NULL) {
      p->locked = 1;
      pred->next = p;
      while(p->locked == 1); /* spin */
    }
    CriticalSection();
    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;
    }
  }
}
```

図 4　MCS アルゴリズム
Fig. 4　The MCS algorithm.

ることで，ノードの動的な参加/脱退を容易に記述できるインタフェースを提供する．

3.7.2　pthread プログラムとの対応性

DMI では，マルチコア上の並列プログラムを分散化させる際の敷居を下げるため，pthread プログラムに対してほぼ機械的な変換作業を施すことで DMI のプログラムが得られるようなインタフェース設計を目標としている．そのためには，スレッド生成/破棄の操作から同期操作に至るまで，共有メモリ環境上の pthread プログラムとシンタックス的・セマンティクス的な対応性を重視した設計を施す必要がある．

たとえば，従来の分散共有メモリにおける排他制御のインタフェースを観察すると，id（整数値）をロック関数/アンロック関数に渡すことで排他制御が実現されるような仕組みになっているものが多い[4),51)]．一方，pthread における排他制御のインタフェースは，排他制御変数のアドレスを指定して初期化関数を呼び出した後，そのアドレスをロック関数/アンロック関数に渡すことで排他制御が実現されるインタフェースになっている．つまり，多くの分散共有メモリシステムでは，pthread における「ユーザプログラムが与えた共有メモリアドレスの上で排他制御が実現される」という性質が反映されていない．一見これは瑣末な問題に見えるが，共有メモリ環境上のプログラムと分散共有メモリ環境上のプログラムとのセマンティクス的な対応を論じるうえでは重要であり，事実，この相違が pthread プログラムを分散共有メモリ上のプログラムに変換するうえでの障害になることを確認している．

しかし，共有メモリベースの排他制御を採用する DMI において，図 3 に示すモデルに基づいて研究されてきた排他制御アルゴリズムを，どうすれば pthread 型のインタフェースとして提供できるかは自明ではない．なぜなら，pthread では，ロック関数とアンロック関数は別の関数として定義されており，どちらも共有メモリアドレス 1 個を引数にとって動作するインタフェースとなっていて，ある排他制御変数に関してロック関数を呼び出すスレッドとアンロック関数を呼び出すスレッドが異なってもかまわないのに対して，図 3 のモデルでは，ロックするプロセスとアンロックするプロセスが同一であることが前提とされているためである．言い換えると，図 3 のモデルで研究される排他制御アルゴリズムは，Entry Section と Exit Section で共通の変数が利用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 型のロック関数として切り出し，Exit Section の中身を pthread 型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 の MCS アルゴリズムでは，`node` という変数が同一プロセスの Entry Section と Exit Section を通して利用できることが，アルゴリズムを成立させるうえでの重要なポイントになっており，Entry Section と Exit Section を単純に別の関数に分離することはできない．この問題の安直な解決法としては，**図 5** に示すように，Entry Section と Exit Section を通じて利用する変数をロック関数の最初でヒープ領域に malloc し，ロック関数を抜ける直前にそのアドレスを排他制御変数の中に保存し，アンロック関数の最後で free する方法が考えられる．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 が必要となるため重い．5.6 節では，この問題の解決法も含めて，pthread 型のインタフェースに従った共有メモリベースの排他制御の実装について述べる．

```
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;
}

void unlock(struct mutex_t *mutex) {
  struct vars_t *vars;

  vars = mutex->vars;
  ExitSection();
  free(vars);
}
```

図 5　Entry Section/Exit Section を pthread 型のロック関数/アンロック関数に分離する方法
Fig. 5　Separating an Entry Section/Exit Section into a pthread-like lock/unlock function.

4. プログラミングインタフェース

DMI は，分散共有メモリとしてのメモリ操作を行う API 以外に，pthread 型のプログラミングスタイルでノードの動的な参加/脱退を容易に記述可能とするための各種 API を提供する．API の一覧を付録 A.2 に示す．また，ノードが動的に参加/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の記述例を**図 6** に示す．

本章では，図 6 のプログラムを例にして DMI の実行形態を説明する．第 1 に，このプログラムを libdmi.so と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

```
01: #include "dmi_api.h"
02:
03: typedef struct targ_t {
04:   DMI_mutex_t mutex;
05:   int32_t value;
06:   int64_t flag_addr;
07: }targ_t;
08: #define MUTEX(targ_addr) ((int64_t)&(((targ_t*)targ_addr)->mutex))
09: #define VALUE(targ_addr) ((int64_t)&(((targ_t*)targ_addr)->value))
10: #define FLAG_ADDR(targ_addr) ((int64_t)&(((targ_t*)targ_addr)->flag_addr))
11:
12: void DMI_main(int argc,char **argv) {  /* 最初に起動される関数 */
13:   DMI_node_t node; DMI_thread_t handle[256][32];
14:   int32_t i,value,my_rank; int64_t targ_addr,flag_addr;
15:
16:   DMI_rank(&my_rank); /* 自分のランクを知る */
17:   DMI_mmap(&targ_addr,sizeof(targ_t),1,NULL);  /* DMI スレッドの引数に渡すメモリ領域確保 */
18:   DMI_mmap(&flag_addr,sizeof(int32_t),256,NULL);  /* 終了通知に使うメモリ領域確保 */
19:   DMI_mutex_init(MUTEX(targ_addr));
20:   value = 0;
21:   DMI_write(VALUE(targ_addr),sizeof(int32_t),&value,DMI_LOCAL_WRITE,NULL);
22:   DMI_write(FLAG_ADDR(targ_addr),sizeof(int64_t),&flag_addr,DMI_LOCAL_WRITE,NULL);
23:
24:   while(1) {
25:     DMI_poll(&node);  /* 参加/脱退宣言するノードをポーリング */
26:     if(node.state == DMI_OPEN) {  /* 参加宣言しているノードの場合 */
27:       DMI_welcome(node.rank);  /* 参加させる */
28:       value = 0;
29:       DMI_write(flag_addr + node.rank * sizeof(int32_t),sizeof(int32_t),&value,DMI_REMOTE_WRITE,NULL);
30:       for(i = 0; i < node.core; i++) {  /* コア数だけ DMI スレッド生成 */
31:         DMI_create(&handle[node.rank][i],node.rank,targ_addr,NULL);
32:       }
33:     } else if (node.state == DMI_CLOSE) {  /* 脱退宣言しているノードの場合 */
34:       value = 1;
35:       DMI_write(flag_addr + node.rank * sizeof(int32_t),sizeof(int32_t),&value,DMI_REMOTE_WRITE,NULL);
                                                                                  /* 終了通知を送る */
36:       for(i = 0; i < node.core; i++) {  /* コア数だけ DMI スレッド回収 */
37:         DMI_join(handle[node.rank][i],NULL,NULL);
38:       }
39:       DMI_goodbye(node.rank); /* 脱退させる */
40:       if(node.rank == my_rank) break;
41:     }
42:   }
43:
44:   DMI_read(VALUE(targ_addr),sizeof(int32_t),&value,DMI_INVALIDATE_READ,NULL);
45:   outn("value is 0",value);
46:   DMI_mutex_destroy(MUTEX(targ_addr));
47:   DMI_munmap(flag_addr,NULL);  /* メモリ解放 */
48:   DMI_munmap(targ_addr,NULL);  /* メモリ解放 */
49:   return;
50: }
51:
52: int64_t DMI_thread(int64_t targ_addr) {  /* DMI スレッドとして起動される関数 */
53:   int32_t value,my_rank; int64_t flag_addr;
54:
55:   DMI_rank(&my_rank);
56:   DMI_read(FLAG_ADDR(targ_addr),sizeof(int64_t),&flag_addr,DMI_ONCE_READ,NULL);
57:   while(1) {
58:     DMI_read(flag_addr + my_rank * sizeof(int32_t),sizeof(int32_t),&value,DMI_UPDATE_READ,NULL);
59:     if(value == 1) break;  /* 終了通知が届いていたら DMI スレッドを終了させる */
60:     DMI_mutex_lock(MUTEX(targ_addr));  /* ロック */
61:     DMI_read(VALUE(targ_addr),sizeof(int32_t),&value,DMI_INVALIDATE_READ,NULL);  /* カウンタ変数を read */
62:     value++;  /* カウンタ変数をインクリメント */
63:     DMI_write(VALUE(targ_addr),sizeof(int32_t),&value,DMI_REMOTE_WRITE,NULL);  /* カウンタ変数へ write */
64:     DMI_mutex_unlock(MUTEX(targ_addr));  /* アンロック */
65:   }
66:   return DMI_NULL;
67: }
```

図 6　ノードが参加/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の記述例（API の詳細は付録 A.2 を参照）

Fig. 6　An example program which exclusively increments one counter variable with nodes joining/leaving.

る．第 2 に，ノード A で「`./a.out`」として実行すると，ノード A 上で `DMI_main(...)` が走り始める（12 行目）．第 3 に，別のノード X で「`./a.out -i` ノード A のホスト名」として実行すると，DMI システムにノード X の参加宣言が通知される．この参加宣言は，DMI システムに生じる参加/脱退宣言をポーリングしている `DMI_poll(node)` によって拾われ，ノード X の情報が引数の `node` 変数に格納される（25 行目）．この `node` 変数はノード X に関するさまざまな情報を持つ．`node.rank` にはノード X に対して DMI システムが与えた一意なランクが格納されており，このランクを指定して `DMI_welcome(...)` を呼ぶことでノード X の参加が完了する（27 行目）．その後，ランクを指定して `DMI_create(...)` を呼ぶと，そのノード上に任意個の DMI スレッドを生成でき（31 行目），これら DMI スレッドは `DMI_thread(...)` として走り始める（52 行目）．以降，同様にして「`./a.out -i` すでに実行中のホスト名」として実行することで任意個のノードの参加を実現できる．第 4 に，ノード X を脱退させる際には，ノード X 上で Ctrl+C を叩くなどして SIGINT 割込みを行う．するとノード X の脱退宣言が DMI システムに通知され，やがて `DMI_poll(...)` に拾われる（25 行目）．ノード X 上に生成した DMI スレッドを回収するなどした後（37 行目），ランクを指定して `DMI_goodbye(...)` を呼ぶとノード X の脱退が完了する（39 行目）．すなわち，参加/脱退が実現されるのは，`./a.out` を実行したり SIGINT 割込みを行ったりした時点ではなく，`DMI_welcome(...)`/`DMI_goodbye(...)` が呼ばれた時点である．このような仕様により，アプリケーションにとって適切なタイミングで適切な処理を行った後で参加/脱退が行われるようにプログラムを記述できる．なお，GXP [43] などの並列シェルと組み合わせることで，多数のノードの一括参加/脱退を容易に実現できる．

5. 実　　装

5.1 ノードの構成要素

DMI における各ノードの構成要素は，recver スレッド，handler スレッド，sweeper スレッドと複数の DMI スレッドであり，それぞれ以下の役割を果たす：

recver スレッド　recver スレッドは，下位の TCP レイヤーからメッセージを受信しては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，メッセージキューからメッセージを取り出し，そのメッセージを解釈して，必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後，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

取り出してから次にメッセージキューを覗くまでの間に，他ノードとのメッセージ通信を介するような，有限時間で終了するかどうかが保証できない処理は行わない．これにより，ノード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目も持つ．なお，recver スレッドと handler スレッドを分けているのは，TCP 受信バッファの飽和によるデッドロックを防止するためである．

sweeper スレッド　sweeper スレッドは，そのノードの DMI 物理メモリの使用状況をつねに監視する．DMI では，各ノードが提供する DMI 物理メモリのサイズは各ノードの起動時に指定可能であり，DMI 物理メモリの使用量がその指定量を超過した場合には，sweeper スレッドが適宜ページの追い出し操作を行うことでメモリオーバフローを防ぐ仕組みになっている．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ような機能を果たす．

DMI スレッド　ユーザプログラムが `DMI_create(...)` 関数を適宜呼び出すことによって生成されるスレッドで，各ノードに任意個生成できる．

5.2　マルチモード read/write

5.2.1　コンシステンシプロトコルの概要

マルチモード read/write の説明の準備として，DMI のプロトコルの概要をまとめる：

- Single Writer/Multiple Reader 型のプロトコルで，ページを単位とした Sequential Consistency を維持する．
- 各ページに対してオーナー権を有するノードがちょうど 1 個存在し，ページフォルトのハンドリングなどはすべてオーナーによってシリアライズされて処理される．
- オーナーは動的に遷移しうるが，オーナー追跡グラフが適切に管理されており，各ノードで発行されたページフォルトの通知などは，オーナー追跡グラフに沿ってフォワーディングされることでやがてオーナーに受理され処理される．
- 各ノードの各ページの状態は，INVALID，DOWN_VALID，UP_VALID の 3 状態で管理される．INVALID は最新ページを持っていない状態，DOWN_VALID と UP_VALID は最新ページを持っている状態である．オーナーは全ノードにおけるページの状態を把握しており，write によってオーナーがページに対する更新作業を行う際には，DOWN_VALID なノードたちに対してはオーナーから invalidate 要求が送信されて INVALID に遷移するが，UP_VALID なノードたちに対しては最新ページが付与された update 要求が送信されて UP_VALID なままであり続ける．

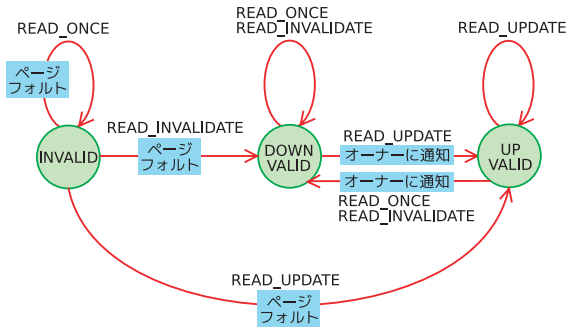

図 7　マルチモード read における状態遷移図
Fig. 7　A state transition diagram of multi-mode read.

5.2.2　マルチモード read

上記のように DMI では，DOWN_VALID と UP_VALID を区別することで，invalidate 型と update 型のハイブリッド型のプロトコルを実現するが，どう invalidate 型と update 型をハイブリッドさせるかを，`DMI_read(...)` のモードとして指定することができる．具体的には，`DMI_read(...)` を呼び出す際に，READ_ONCE，READ_INVALIDATE，READ_UPDATE の 3 種類のモードを指定可能であり，その時点でのそのノードのページの状態に対応して，**図 7** に示すような状態遷移が引き起こされる．状態遷移の具体例をあげる：

- ノード i において，ページが INVALID な状態で READ_UPDATE モードの `DMI_read(...)` が発行された場合，オーナーに read フォルトが送信され，オーナーにおいてノード i が update 型のノードとして登録される．その後，オーナーからノード i に対して最新ページの転送が行われ，ノード i のページの状態は UP_VALID に遷移する．
- ノード i において，ページが UP_VALID な状態で READ_ONCE モードの `DMI_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 型のノードに登録されていたノード i が invalidate 型のノードとして登録し直される．その後，オーナーからノード i に対して通知が送信され，ノード i のページの状態が DOWN_VALID に遷移する．なお，このとき INVALID ではなく DOWN_VALID に遷移させるのは，無理やり INVALID に遷移させることに利点がないためである．

このように DMI では，invalidate 型と update 型を各 `DMI_read(...)` の粒度で自由に

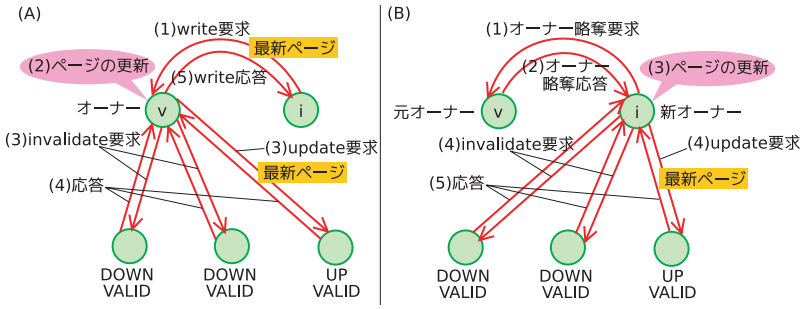

図 8　マルチモード write の挙動（(A) WRITE_REMOTE，(B) WRITE_LOCAL）
Fig. 8　Behavior of multi-mode write ((A) WRITE_REMOTE, (B) WRITE_LOCAL).

ハイブリッドさせることができるプロトコルを実装する．

5.2.3　マルチモード write

DMI における write は，オーナーがページの更新作業を行った後，DOWN_VALID なノードに対しては invalidate 要求を送信し，UP_VALID なノードに対しては最新ページを付与した update 要求を送信し，それらに対する応答を回収することでコンシステンシを維持するが，この際，write を発行したノードにオーナー権を移動するか移動しないかを，`DMI_write(...)` のモードとして指定することができる．具体的には，`DMI_write(...)` を呼び出す際には，WRITE_REMOTE，WRITE_LOCAL の 2 種類のモードを指定可能であり，それぞれ以下のようなプロトコルが動く：

WRITE_REMOTE　ノード i が write を発行した場合，ノード i は write したいデータをオーナーに送信することで，ページの更新作業と invalidate 要求+update 要求+その応答の回収作業をオーナーに行わせる（**図 8** (A)）．

WRITE_LOCAL　ノード i が write を発行した場合，まずノード i がオーナーからオーナー権を奪ってきてオーナーになった後，ページの更新作業と invalidate 要求+update 要求+その応答の回収作業をノード i が行う（図 8 (B)）．

さてここで，WRITE_REMOTE と WRITE_LOCAL のどちらが効率的かを比較する．N 個のノードが合計 K 回の write を行うようなプログラムを考え，簡単のため read はいっさい行われないものとして，K 回の write すべてを WRITE_REMOTE で行う場合と，K 回の write すべてを WRITE_LOCAL で行う場合を比較する．また，プログラム開始時にオーナー権はノード v にあるとし，i（$1 \leq i \leq K$）回目の write を行うノードを $f(i)$ で表す．

まず，K 回すべての write が WRITE_REMOTE で行われる場合を考える．この場合，プログラムの開始から終了までオーナーはノード v に固定されるため，プログラム開始時に全ノードが正しいオーナー 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，オーナー以外の各ノードで生じる 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届けられる．したがって，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 回とすれば，$O(K_1)$ になる．次に，K 回すべての write が WRITE_LOCAL で行われる場合を考える．この場合には，プログラムの開始から終了までオーナーは動的に遷移し続けるため，各ノードで生じた write は，そのノードがオーナーでない限り write フォルトを引き起こし，オーナー権を奪うための要求をオーナー追跡グラフに沿ってオーナーまで届ける必要がある．一般に，オーナー追跡グラフを最適に管理した場合には，オーナー追跡グラフに沿ってフォワーディングする際のホップ数は平均 $O(\log N)$ になることが知られている[23),35)]．したがって，飛び交うメッセージ数は，write フォルトが発生するのは現在 write を行っているノードと直前に write を行ったノード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，$O(K_2 \log N)$ となる．

以上より，単純にメッセージ数で比較すれば，$K_1 \leq K_2 \log N$ であれば WRITE_REMOTE の方が有利であり，$K_1 > K_2 \log N$ であれば WRITE_LOCAL の方が有利といえるが，K_1 と $K_2 \log N$ の大小関係は完全にアプリケーション依存であると考えられる．しかも，WRITE_REMOTE の場合には write フォルト発生時に write すべきデータそのものをオーナーに送信しなければならないことをふまえると，write すべきデータが巨大な場合には $K_1 \leq K_2 \log N$ であっても WRITE_LOCAL の方が有利になる可能性も十分ある．要するに，どちらのアプローチが効率的かは，アプリケーションの各局面や各データによってさまざまであるというのがもっともな考え方であり，DMI のように，各 `DMI_write(...)` の粒度で選択可能にすることは最適化手段として有効であるといえる．

5.3　コンシステンシプロトコルの詳細

固定的なノードを設置することなく，read 操作（READ_ONCE，READ_INVALIDATE，READ_UPDATE），write 操作（WRITE_REMOTE，WRITE_LOCAL），追い出し操作を実現するプロトコルを説明する．アルゴリズムの疑似コードは付録 A.1 に付す．なお，本節で述べるアルゴリズムに基づき，これら 6 種類の操作をランダムに合計 10000 回行うプ

ロセスを同時に 20 個走らせた結果，デッドロックを引き起こしたりコンシステンシを崩したりすることなく，20 個すべてのプロセスが正常に終了することを確認している．

5.3.1　データ構造

各ノードは，DMI システムに割り当てられている全ページに対して，*owner*，*probable*，*buffer*，*state*，*state_array*，*valid*，*msg_set*，*seq_array* の 8 つのデータを管理する．コンシステンシ管理はページ単位で独立であるため，以降では，ある特定のページ p に関するプロトコルを考えることとし，ノード i のページ p に関する *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　ノード i がオーナーであれば $i.owner$ = TRUE，そうでなければ $i.owner$ = FALSE である．言い換えると，$i.owner$ = TRUE であることが，ノード 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たかだか 1 個しか存在しない．プロトコル上，オーナーが遷移中の状態では系内にオーナーが存在しない時刻が存在するが，有限時間内には必ずオーナーが確定する．

probable　ノード i がオーナーをノード j だと思っているとき $i.probable = j$ である．全ノードを通じた *probable* の参照関係がオーナー追跡グラフであり，任意のノードで発生したオーナー宛のメッセージは，各ノード 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オーナーにたどり着く．なお，$i.probable = i$ であってもノード i がオーナーであるとは限らない．

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　$i.state$ はノード 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，オーナーにおいてノード i は invalidate 型のノードとして登録されている．$i.state$ = UP_VALID ならば，$i.buffer$ には最新ページが格納されており，オーナーにおいてノード i は update 型のノードとして登録されている．$i.state$ = INVALID ならば，ノード 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．

msg_set　系内に無駄なメッセージが流れるのを防ぐため，一時的に $i.probable$ = NIL としてメッセージのフォワーディングを抑止することがあり，その期間中はメッセージを $i.msg\_set$ に保留する．

state_array　オーナー v のみが管理する配列データで，全ノードにおけるページの状態が記録される．$v.state\_array[i]$ には $i.state$ の値が格納される．

valid　オーナー v のみが管理するデータで，DOWN_VALID または UP_VALID な状態にあるノード数を管理する．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，プログラム開始から現在に至るまでにオーナーから各ノード i に対して送信したメッセージ数が記録される．これはノード v がオーナーになった時点以降にノード v がノード i へ送信したメッセージ数ではなく，プログラムの開始以降，オーナーがどう遷移したかにかかわらず，オーナーであるようなノードがノード i へ送信したメッセージ数の総和である．

5.3.2　プロトコルの設計法

DMI では，非同期 read/write，マルチモード read/write，ページの追い出しなどの多様な操作を，固定的なオーナーを設置することなく実現する必要があり，各操作による多様な影響を同時に考慮しつつコンシステンシプロトコルを設計するのは難解である．そこで，まず問題を単純化するため，オーナーが固定されており，かつオーナーと各ノードが FIFO な通信路で接続されている状況を考える．これはいわゆる単純なクライアント・サーバ方式に相当し，ノード i で各種操作が発行されると，ノード i からオーナーに対して要求メッセージが送信され，それがオーナーによってシリアライズされて処理され，やがてオーナーからノード i へ必要な情報を含んだ応答メッセージが送信されるという非常に単純なモデルになる．そして，このような単純なモデルの上では，複雑な操作に対してもプロトコルの正しさはほぼ自明である．このように，オーナーが固定されオーナーと各ノードが FIFO に接続されるモデルを仮定すると，正しいプロトコルを設計するのが容易になるが，その理由は，

- 各ノードが，要求メッセージを送信すべき対象であるオーナーの位置をつねに知っていること
- 単独のオーナーが，すべての要求をシリアライズして処理すること
- オーナーがノード 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ノード i によって受信されること

が保証できるためである．

以上の観察に基づくと，動的にオーナーが遷移する場合でも，上記の 3 つの性質が満足されるようなプロトコルを設計しさえすれば，マルチモード read/write や追い出し操作などの複雑な操作に対しても正しいプロトコルを容易に設計できる．したがって，DMI のプロトコル設計の基本アイディアは，まず第 1 ステップとして，

条件 1　各ノード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

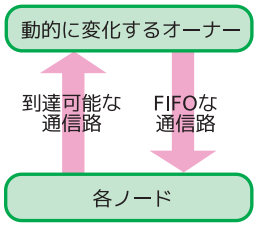

図 9　DMI のコンシステンシプロトコルが実現する各ノードとオーナーとの通信形態
Fig. 9　Relationship between an owner and each node in the consistency protocol of DMI.

条件 2　たかだか 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，オーナーから各ノードへの FIFO な通信路が存在すること

が保証できるようなプロトコルを設計し（**図 9**），第 2 ステップとして，その上にマルチモード read/write などの高度な操作に対するプロトコルを定義することである．

まず，条件 1，条件 2，条件 3 を満たすプロトコルの設計法について説明する．

条件 2 に関しては，オーナーが遷移中であるような一時的な状態ではオーナーは存在せず，それ以外の状態ではオーナーは 1 個しか存在しないようにし，同一ページに関する要求メッセージは，その単独のオーナーがシリアライズして処理することにすればよい．

条件 3 に関しては，オーナーから各ノード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ノード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 からノード i へメッセージを送信する際には，$v.seq\_array[i]$ の値を 1 増やすとともに，その値を順序番号として付与する．そして，各ノード i は，オーナーからのメッセージに関しては，順序番号の順にメッセージを受信するような順序制御を行う．また，オーナーをノード v からノード v' に遷移させる際には，$v.seq\_array$ をノード v からノード v' へ丸ごとコピーすることで，オーナーの遷移を越えた順序番号の連続性を保証する．たとえば，**図 10** のような状況では，物理的にはメッセージ m_2 が m_1 よりも先に到着したとしても，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うことで，動的なオーナーの遷移を越えてオーナーから各ノードへの FIFO な通信路を保証できる．

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，任意のノードから *probable*をたどることでやがてオーナーに到達できるように，各ノードの *probable* を適切

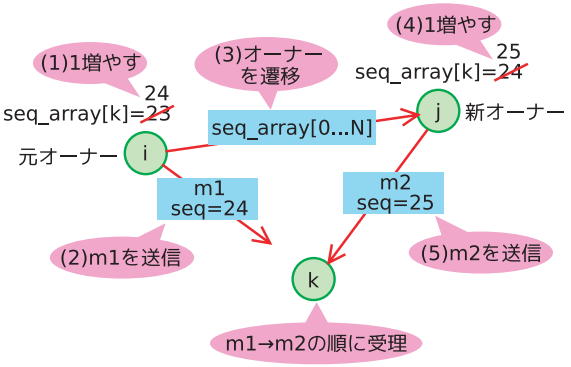

図 10　オーナーからのメッセージの順序制御
Fig. 10　Sequencing of messages from an owner.

に管理すればよい．これを実現するため，各ノードの *probable* の値に関して，ノード i が $i.probable$ を参照することは任意の時点で可能だが，ノード 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，順序制御されたオーナーからのメッセージをノード i が受信した時点でのみ可能という制約を課す．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全ノードに分散しているものの，データの値としてはオーナーによって一括管理されることとなるため，正しいオーナー追跡グラフを容易に管理できる．このように，DMI では各ノード上のコンシステンシ管理のためのデータに関して以下の制約を課す：

条件 4　各ノード上に分散されているコンシステンシ管理のためのデータは，(I) 各ノードは任意の時点で参照することができるが，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ようなデータであるか，(II) オーナーのみが参照し更新することができるデータであるかのいずれかである．

具体的には，*owner*，*buffer*，*state*，*probable* は (I) の性質を満たすデータとして管理し，*state_array*，*valid*，*seq_array* は (II) の性質を満たすデータとして管理する．しかし，この強い制約の下では，*probable* が更新される機会があまりにも限定されすぎ，オーナー追跡のパスが長くなる傾向があるため，オーナー追跡グラフの正しさを損なわない範囲で 5.3.6 項で述べる最適化を施す．

さらに，プロトコルを容易化するために，ノードの参加/脱退と DMI 仮想メモリの確保/

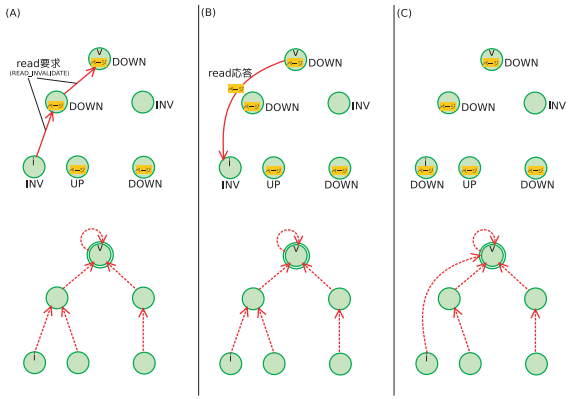

図 11　read 操作のコンシステンシプロトコル（各コマにおいて，下段がオーナー追跡グラフの形状，上段が送受信メッセージやページの状態の様子を表す．下段において二重丸で囲まれたノードはオーナーを表す．INV は INVALID，DOWN は DOWN_VALID，UP は UP_VALID を意味する）
Fig. 11　A consistency protocol of a read operation.

解放に関して以下の制約を課す：

条件 5　ノードの参加/脱退および DMI 仮想メモリの確保/解放は，1 個のグローバルロックを利用してシリアライズして実行する．

5.3.3　read 操作

ノード i で read 操作が発行された場合，$i.state$ = DOWN_VALID または $i.state$ = UP_VALID であり，かつ $i.state$ と read 操作によって指定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 read 操作が完了する．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 11** (A)）．なお，図 7 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 READ_UPDATE の場合，または $i.state$ = UP_VALID かつモードが READ_INVALIDATE または READ_ONCE の場合を指す．

ノード i の read 要求がオーナー v に受信された場合，以下の 2 通りの場合が考えられる：

(I) $v.state\_array[i]$ = INVALID の場合

(1)　ノード i が要求するモードが READ_INVALIDATE の場合には $v.state\_array[i]$ = DOWN_VALID とし，要求するモードが READ_UPDATE の場合には $v.state\_array[i]$ = UP_VALID としたうえで，$v.valid$ を 1 増やす．ノード 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 11 (B)）．

(2)　read 応答を受信したノード 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 11 (C)）．$i.buffer$ を読み込み read 操作を完了させる．

(II) それ以外の場合

この状況は，ノード i の read 要求が $i.state$ と read 操作が指定するモードの矛盾に起因したものである場合や，過去にノード 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，すでにノード 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．なお，後者の状況は，各ノード上の複数の DMI スレッドが同時に read 操作を発行する場合や，非同期 read が発行される場合に発生しうる．

(1)　オーナー v は，ノード i が要求するモードが READ_ONCE または READ_INVALIDATE であり，かつ $v.state\_array[i]$ = UP_VALID ならば $v.state\_array[i]$ を DOWN_VALIDに更新し，要求するモードが READ_UPDATE かつ $v.state\_array[i]$ = DOWN_VALID ならば $v.state\_array[i]$ を UP_VALID に更新する．ノード 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．

(2)　read 応答を受信したノード i は，$i.state$ を $v.state\_array[i]$ に更新し，$i.probable$ を v に更新する．$i.buffer$ を読み込み read 操作を完了させる．

5.3.4　write 操作

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

(I) モードが WRITE_REMOTE の場合

(1)　ノード i は，書き込むべきデータを載せた write 要求をオーナー v に対して送信する（**図 12** (A)）．

(2)　write 要求を受信したオーナー v は，受信したデータを $v.buffer$ に格納する．オーナー v は，$v.owner$ を FALSE に更新し，オーナー権を放棄する．ノード v は，$v.state\_array[j]$ = DOWN_VALID を満たすすべてのノード j（$\neq v$）に対して，invalidate 要求を順序番号を付与して送信し，そのつど $v.state\_array[j]$ を INVALID に更新し，$v.valid$ を 1 減らす．$v.state\_array[j]$ = UP_VALID を満たすすべての

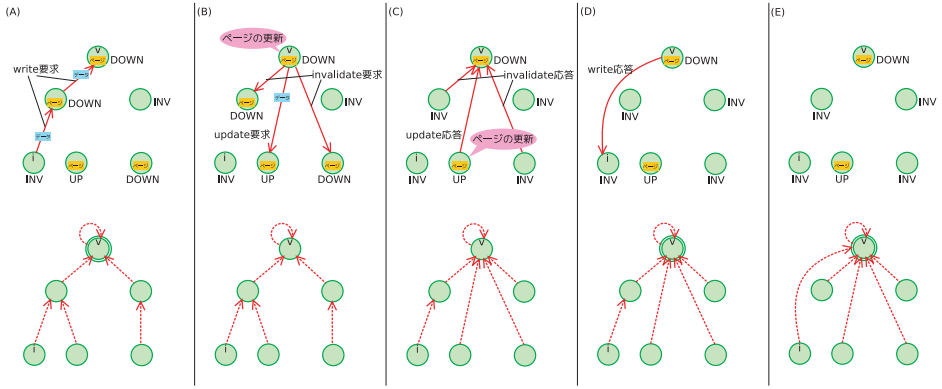

図 12　write 操作（WRITE_REMOTE）のコンシステンシプロトコル
Fig. 12　A consistency protocol of a write operation (WRITE_REMOTE).

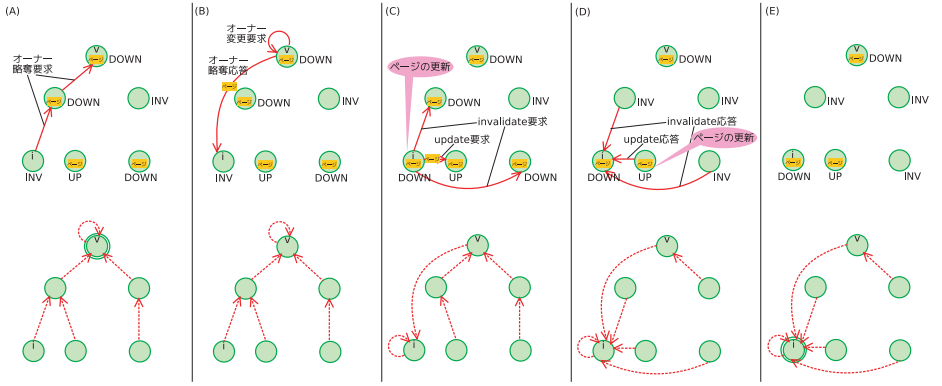

図 13　write 操作（WRITE_LOCAL）のコンシステンシプロトコル
Fig. 13　A consistency protocol of a write operation (WRITE_LOCAL).

ノード j（$\neq v$）に対して，最新ページを載せた update 要求を，順序番号を付与して送信する（図 12 (B)）．

(3)　invalidate 要求を受信した各ノード j は，$j.state$ を INVALID に更新し，$j.probable$ を v に更新したうえで，ノード v に対して invalidate 応答を送信する（図 12 (C)）．

(4)　update 要求を受信した各ノード j は，$j.buffer$ を最新ページに更新し，$j.probable$ を v に更新したうえで，ノード v に対して update 応答を送信する（図 12 (C)）．

(5)　ノード 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 応答を回収したら，$v.owner$ を TRUE に更新し，再度オーナー権を得る．そして，ノード 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 12 (D)）．なお，先ほど $v.owner$ を FALSE に変えてから，今ここで再度 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が，この期間にノード v に届いたメッセージは，ノード 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（ただしこの部分は 5.3.6 項で改善する）．

(6)　write 応答を受信したノード i は，$i.probable$ を v に更新する（図 12 (E)）．write 操作を完了させる．

（Ⅱ）モードが WRITE_LOCAL の場合

ノード i がオーナーでない場合には以下の (1)〜(6) の手順を行う．ノード i がオーナーである場合には以下の (5)〜(6) の手順のみを行う：

(1)　ノード 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 13** (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 13 (B)）．ここで，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．オーナー v は，$v.owner$ を FALSE に更新し，オーナー権を放棄する．ノード 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，ノード i は最新ページを持っていないため，$v.state\_array[i]$ = DOWN_VALID と更新し，$v.valid$ 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 13 (B)）．

(3)　オーナー変更要求を受信したノード v は，$v.probable$ を i に更新する（図 13 (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)　オーナー略奪応答を受信したノード 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．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し，$i.state$ を DOWN_VALID に更新する．$i.probable$ を i に更新する（図 13 (C)）．

(5)　ノード i は，書き込むべきデータを $i.buffer$ に格納する．ノード i は，$i.owner$ を

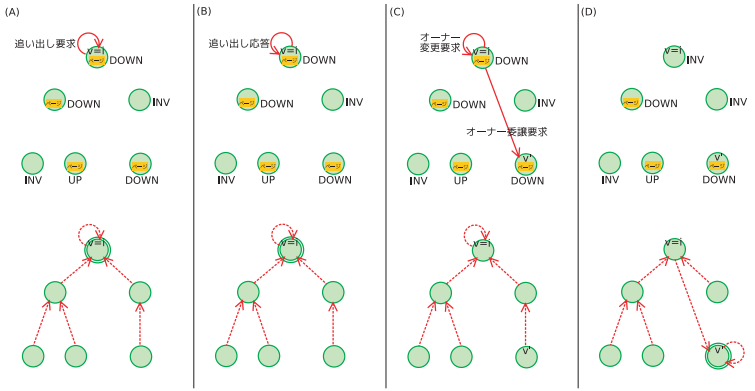

図 14　追い出し操作のコンシステンシプロトコル
Fig. 14　A consistency protocol of an eject operation.

FALSE に更新し，オーナー権を放棄する．以降，REMOTE_WRITE モードと同様にして，invalidate 要求と update 要求を送信し，それらに対する応答を回収する（図 13 (D)）．

(6)　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 応答を回収した後，$i.owner$ を TRUE に更新してオーナー権を得る（図 13 (E)）．

なお，DMI では fetch-and-store 操作，compare-and-swap 操作も実装しているが，それらのプロトコルは write 操作のプロトコルと同様である．fetch-and-store 操作，compare-and-swap 操作では，write 操作においてオーナーが $i.buffer$ にデータを書き込む部分が，該当のアトミック操作に切り替わるだけである．

5.3.5　追い出し操作

ノード i で追い出し操作が発行された場合，$i.state$ = INVALID ならば，何の処理も発生せず追い出し操作が完了する．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 14** (A)）．

(1)　追い出し要求を受信したオーナー v は，$v.state\_array[i]$ = DOWN_VALID または $v.state\_array[i]$ = 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 はノード 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 14 (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，オーナー権を他ノードに追い出す必要があるため，(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ノード i は，$i.state$ を INVALID に更新し，$i.probable$ を v に更新する（図 14 (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 14 (C)）．オーナー v は，$v.owner$ を FALSE に更新し，オーナー権を放棄する．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 $v.state\_array[v']$ = 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，$v.state\_array[v']$ = DOWN_VALID と更新し，$v.valid$ 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 14 (C)）．よって，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ノードを新オーナー v' として選ぶのが望ましい．

(4)　オーナー変更要求を受信したノード v は，$v.probable$ を v' に更新する（図 14 (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．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し，$v'.state$ を DOWN_VALID に更新する．$v'.owner$ を TRUE に更新してオーナーになり，$v'.probable$ を v' に更新する（図 14 (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，適切な排他によって，メモリ解放作業中の DMI 仮想メモリに属するページの追い出し操作を防いでいる．

5.3.6　オーナー追跡の最適化

オーナー追跡のパスを短縮し，系内に無駄なメッセージを流さないようにするため，「有限時間だけ待てば，今知っている *probable* よりも確からしい *probable* を知ることができる」ような場合には，その期間に受信した各種要求をすぐにフォワーディングするのではなく，一時的に保留しておき，より確からしい *probable* が確定した時点で，保留中の全要求を *probable* へとフォワーディングするという最適化を施す．具体的には，ノード i が，read 要求，write 要求，オーナー略奪要求，追い出し要求を送信する際，これらの要求 m を

$i.probable$ へと送信した後で，$i.probable$ を NIL に書き換え，以降にノード i に届く要求は $i.msg\_set$ に保留する．そして，やがてノード i がオーナー v からの順序制御された何らかのメッセージを受信し，$i.probable$ が v に更新された時点で，$i.msg\_set$ 内の全要求をノード v へとフォワーディングする．証明は略すが，要求 m に対する応答は有限時間内に返ってくることが保証されるため，$i.probable$ が NIL であるような時間は有限であることがいえ，よって，メッセージが永遠に $i.msg\_set$ に保留され続けることはない．なお，実際には，要求 m に対するオーナーからの応答が返る前に，オーナーからの invalidate 要求などを通じて $i.probable$ が確定し，保留されたメッセージがフォワーディングされる場合もある．

なお，オーナー追跡の最適化に関しては，ノード i がノード j からの要求をフォワーディングした時点で，「いずれはノード j が，ノード i が知っている $probable$ よりも確からしい $probable$ を知るだろう」と判断し，$i.probable$ を j に書き換えることで，オーナー追跡におけるホップ数の計算量を削減できることが知られている[23),35)]．しかし，この最適化手法は，オーナーからの順序制御されたメッセージを受信したタイミング以外で $probable$ の値を更新する操作を含んでいる．そのため，オーナーからの順序制御されたメッセージを受信したタイミングでしか $probable$ の値を更新しないという制約を課すことによってプロトコルの正しさの推論を容易化している DMI にとっては，この最適化手法を単純に導入できるかどうかは不明である．よって，現時点ではこの最適化手法は導入できていない．

5.4　ノードの動的な参加/脱退

ノードの参加は，実行中の任意のノード 1 個をブートストラップとして実現される．ノード i がノード 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)　ノード i は，ノード j に参加要求を送信する．
(2)　参加要求を受理したノード j はグローバルロックを取得する．
(3)　ノード j は，ノード i に対して，全ノード情報と全ページに対する $j.probable$ を送信する．
(4)　ノード i は，全ページを割り当て，全ページに関して $i.owner$ を FALSE に，$i.probable$ を $j.probable$ に，$i.state$ を INVALID に初期化する．なお，ここでは $i.probable$ を $j.probable$ に初期化しているが，プロトコル上は，$i.probable$ に任意のノードを代入してもオーナー追跡グラフの正しさは維持される（**図 15** (A)）．
(5)　ノード i は，全ノードに対してコネクション接続を確立し，全ノードとネゴシエーションを行って必要な初期化処理を行う．
(6)　ノード i は，グローバルロックを解放する．

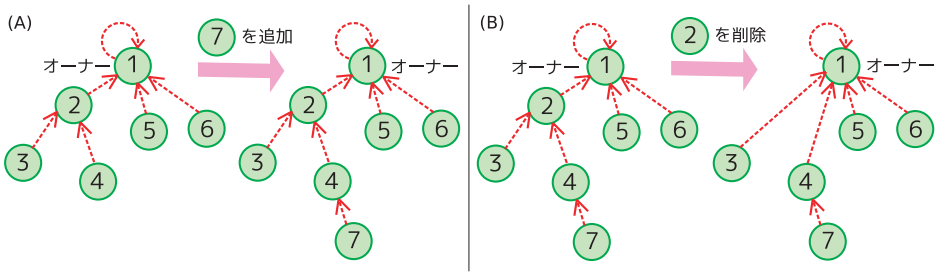

図 15　ノードの参加/脱退時におけるオーナー追跡グラフの再形成（(A) ノードの参加，(B) ノードの脱退）
Fig. 15　Restructuring of an owner tracking graph at joining/leaving of nodes ((A) Joining of nodes, (B) Leaving of nodes).

一方，ノード i が脱退する手順は以下のとおりである：

(1)　ノード i は，グローバルロックを取得する．
(2)　ノード i は，全ページの追い出し操作を行う．
(3)　ノード i は，全ノードに対して，全ページに関する $i.probable$ を送信する．これを受信した各ノード 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．この作業により，全ページのオーナー追跡グラフが，ノード i を含まないオーナー追跡グラフへと再形成される（図 15 (B)）．
(4)　ノード i は，全ノードとのコネクション接続を切断したうえで，終了処理を行う．
(5)　ノード i は，実行中の適当なノード j に対して脱退要求を送信する．
(6)　脱退要求を受信したノード j は，グローバルロックを解放する．

なお，適切な排他によって，脱退作業中のノードに対するページの追い出しや DMI スレッド生成を防いでいる．

5.5　ページ置換

DMI では，遠隔スワップシステムとしての機能を備えるにあたってページ置換アルゴリズムを実装する必要がある．具体的には，sweeper スレッドに適宜ページアウトを実装することになるが，効率的なページ置換を実現するためには，どのページをどのノードに追い出すかを検討する必要がある．

第 1 に，どのページを追い出すかを考える．まず，あるノード i からページを追い出すとは，$i.buffer$ に消費されているメモリ領域を解放することであるが，そのためには，追い出し操作によって $i.state$ を INVALID に変化させればよい．ここで，5.3.5 項で述べた追い

出し操作のプロトコルを観察すると，ページを INVALID に変化させるための負荷の大きさは，ページの状態に応じて，

（I）INVALID なページ
（II）DOWN_VALID または UP_VALID であるが，自分はオーナーではないようなページ
（III）自分がオーナーであり，自分以外に DOWN_VALID または UP_VALID な状態にあるノードが存在するようなページ
（IV）自分がオーナーであり，自分以外には DOWN_VALID または UP_VALID な状態にあるノードが存在しないようなページ

の順に大きいことが分かる．また，DMI 物理メモリ使用量を減らす目的では，できるだけページサイズの大きいページを追い出す方が効果的である．さらに，アプリケーションのアクセス特性も考慮に入れるため，ページを追い出し対象から外すための `DMI_save(...)` 関数と，追い出し対象に含めるための `DMI_unsave(...)` 関数を提供する．なお，メモリ確保時にはすべてのページが追い出し対象に含まれている．以上の考察に基づき，DMI では，追い出し対象のページに関して，ページサイズの大きい順に，（I）→（II）→（III）→（IV）の優先度順にページの追い出しを行う．

第 2 に，どのノードに追い出すかを考える．これが問題となるのは，追い出し操作において最新ページの転送をともなう（IV）の場合である．もし，すでに DMI 物理メモリ使用量が飽和状態に近いノードに対してページを追い出せば，追い出し先のノードでも再度追い出し操作が発生し，結果的にノード間で追い出し操作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，できるだけ DMI 物理メモリ使用量に空きが多いノードを選択することが重要である．そこで，コンシステンシ維持のためにノード間を常時飛び交っているメッセージに対して，メッセージの送信元ノードのメモリ使用状況を付与することで，全ノードが全ノードのメモリ使用状況を 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ノードを選択するのが望ましい．ただし，現状の実装では単純に乱数で追い出し先ノードを決定している．

なお，各ノードが提供する DMI 物理メモリ量 L は各ノードの起動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．つまり，sweeper スレッドは DMI 物理メモリ使用量が可能な限り L 以下になるように努力するだけであって，つねに L 以下になることが保証されるわけではない．これは主に性能上の理由による．したがって，原理的には，全ノードを通じた DMI 物理メモリ量の総量を超える DMI 仮想メモリを確保して利用することも可能ではあるが，その場合にはノード間で頻繁なスラッシングが発生して著しい性能低下が引き起こされる．これは，共有メモリ環境におけるディスクスワップ処理に相当する現象であるととらえることができる．

5.6　共有メモリベースの排他制御

DMI では，複数の共有メモリベースの排他制御アルゴリズムを分析した結果，Permission Word アルゴリズム[15] が最適であると判断し，Permission Word アルゴリズムに基づいて，pthread 型のインタフェースによる排他制御の機構を提供する．Permission Word アルゴリズムは以下のような特徴を持つ：

- read/write/fetch-and-store/compare-and-swap を用いたアルゴリズムである．
- Weak Fairness を満たす．
- 各クリティカルセクションあたり，プロセスローカルでない変数への参照回数が $O(1)$ である．
- 本来の Permission Word アルゴリズムは図 3 のモデルに従って記述されているが，Entry Section と Exit Section を，効率的に pthread 型のロック関数とアンロック関数に分離できる．

Permission Word アルゴリズムを，pthread 型のインタフェースで記述したコードを**図 16** に示す．このコードの利点は，図 5 とは異なり，クリティカルセクションのたびに malloc が発生しない点である．`mutex` で示される排他制御変数と，`lock(...)` のスタック領域上の変数のみを使って，排他制御を巧みに実現している．

以下，図 16 のアルゴリズムの概略を説明する．pthread 型のインタフェースに従わせるため，本来の Permission Word アルゴリズムとはデータ構造を変更しているが，アルゴリズムは本質的に同等である：

(1)　初期状態では，`head`，`next`，`p1`，`p2` はすべて `NULL` である（**図 17** (A)）．`head` は待ちプロセスリストの先頭を示す変数で，`next` は，あるプロセスがクリティカルセクションを実行しているとき，そのプロセスがクリティカルセクションを抜けた時点でどのプロセスを起こせばよいかを示す変数である．`p1` と `p2` の意味は後述する．また，20 行目の `convert(...)` と 39 行目と 43 行目の `revert(...)` についても後述する．

(2)　プロセス 1 が `lock(...)` を呼び出し，21 行目で `head` に対して fetch-and-store を実行すると，図 17 (B) の状態になる．このとき，自分の後ろが `NULL` であるためクリティカルセクションに突入することができ，`lock(...)` はすぐに返る．

```
01: struct mutex_t {
02:   int *head;
03:   int *next;
04:   int *p1;
05:   int *p2;
06: };
07:
08: void init(struct mutex_t *mutex) {
09:   mutex->head = NULL;
10:   mutex->next = NULL;
11:   mutex->p1 = NULL;
12:   mutex->p2 = NULL;
13: }
14:
15: void lock(struct mutex_t *mutex) {
16:   int flag;
17:   int *prev, *curr;
18:
19:   flag = 0;
20:   curr = convert(&flag, mutex->p1, mutex->p2);
                        /* address conversion */
21:   prev = fetch_and_store(&mutex->head, curr);
22:   if(prev == NULL) {
23:     mutex->p1 = curr;
24:   } else {
25:     while(flag == 0); /* spin */
26:   }
27:   mutex->next = prev;
28: }
29:
30: void unlock(struct mutex_t *mutex) {
31:   int *curr;
32:
33:   if(mutex->next == NULL
        || mutex->next == mutex->p1) {
34:     if(mutex->next == mutex->p1) {
35:       mutex->p1 = mutex->p2;
36:     }
37:     if(!compare_and_swap(&mutex->head, mutex->p1, NULL)) {
38:       mutex->p2 = mutex->head;
39:       curr = revert(mutex->p2); /* address reversion */
40:       *curr = 1;
41:     }
42:   } else {
43:     curr = revert(mutex->next); /* address reversion */
44:     *curr = 1;
45:   }
46: }
47:
48: void destroy(struct mutex_t *mutex) {
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 16　pthread 型のインタフェースに従った Permission Word アルゴリズム
Fig. 16　The Permission Word algorithm expressed in pthread-like programming interfaces.

(3)　プロセス 2，プロセス 3，プロセス 4 が `lock(...)` を呼び出し，この順に 21 行目の fetch-and-store を実行すると，図 17 (C) の状態になる．

(4)　プロセス 1 が `unlock(...)` を呼び出すと，プロセス 1 は `next` で示されるプロセスを起こそうとするが，今は `next` が `NULL` なので，自分より後ろに起こすプロセスは存在しないと判断し，`head` 側から起こそうと試みる．今の場合，`head` の後ろに待ちプロセスが存在するため，37 行目の compare-and-swap は失敗し，38 行目で `p2` にプロセス 4 を入れたうえで（正確には，`p2` にプロセス 4 の `flag` のアドレスを入れたうえで），40 行目でプロセス 4 を起こす．起こされたプロセス 4 は，`next` にプロセス 3 を入れたうえで，`lock(...)` を抜ける（図 17 (D)）．

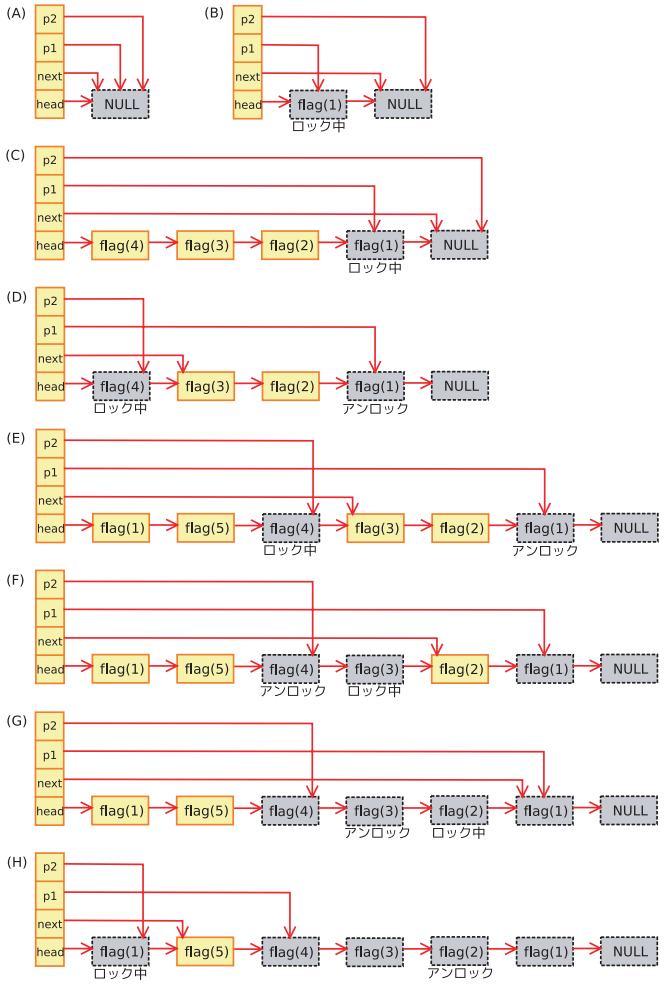

図 17　図 16 のコードによる Permission Word アルゴリズムの動き（flag(i) は，プロセス i における `lock(...)` 関数内の `flag` 変数を意味する．実線枠で囲まれた flag(i) は，まだ `lock(...)` 関数が実行中であるため実体が存在している変数，点線枠で囲まれた flag(i) は，すでに `lock(...)` 関数が終了しているため実体が消滅している変数である）
Fig. 17　Behavior of the Permission Word algorithm programmed in Fig. 16.

(5)　プロセス 5，プロセス 1 が `lock(...)` を呼び出すと，21 行目の fetch-and-store により，プロセス 5 とプロセス 1 が `head` に連結される（図 17 (E)）．

(6)　プロセス 4 が `unlock(...)` を呼び出すと，プロセス 4 は，`next` が指しているプロセス 3 を 44 行目で起こし，プロセス 3 は，`next` にプロセス 2 を入れたうえで，`lock(...)` を抜ける（図 17 (F)）．

(7)　同様に，プロセス 3 が `unlock(...)` を呼び出すと，プロセス 3 は，`next` が指しているプロセス 2 を 44 行目で起こし，プロセス 2 は，`next` にプロセス 1 を入れたうえで，`lock(...)` を抜ける（図 17 (G)）．このように，プロセス 4 → プロセス 3 → プロセス 2 → · · · の順に起こされていくが，この起床処理の連鎖をどこで止めればよいかを示すのが `p1` である．プロセス 2 が `unlock(...)` を呼び出すと，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プロセス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` に代入することで，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．それと同時に，次に行われる起床処理の連鎖がどこから始まるのかを 38 行目で `p2` に記憶することで，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教えられるようにしておく（図 17 (H)）．

最後に，`convert(...)` と `revert(...)` について説明する．図 17 (B) において，プロセス 1 がクリティカルセクションに突入した時点では，プロセス 1 の `lock(...)` はすでに返っているため，`lock(...)` 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，起床処理の連鎖が中止されて再度 `head` から起床処理を開始しようとする際に，`head` の先に起こすべきプロセス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．したがって，図 17 (E) のように，プロセス 1 が再度 `lock(...)` を呼び出したとき，この `lock(...)` 内の `flag` が，前回 `lock(...)` 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と，図 17 (H) において，プロセス 2 が `unlock(...)` 内で 37 行目の compare-and-swap を呼び出す際に，本当は `head` の先に起こすべきプロセスが存在するにもかかわらず，`head` と `p1` の値が一致しているために，`head` の先には起こすべきプロセスが存在しないと判断してしまう．以上の問題を回避するためには，つねに `p1`，`p2`，`flag` のアドレスが一致しないように管理すればよい．そこで，`convert(...)` によってアドレスの下位 2 ビットを適宜ずらす処理を入れ，実際にそのアドレスの値を読む際には `revert(...)` によって正規のアドレスに還元する処理を行っている．

6. 性能評価

6.2.1 項では DMI における read 操作のオーバヘッドを，6.2.2 項ではマルチモード read/write の有効性を，6.2.3 項では遠隔スワップシステムとしての性能と非同期 read/write の有効性を，6.3.1，6.3.2，6.3.3 項では各種アプリケーションに対する DMI と MPI との性能比較およびノードの動的な参加/脱退を行うことの有効性を，6.3.4 項では任意のページサイズを設定できることの有効性を検証する．なお，DMI が採用する Sequential Consistency や Single Writer 型のコンシステンシプロトコルは，既存の緩和型コンシステンシモデルや Multiple Writer 型のコンシステンシプロトコルよりも並列性が絞られるため，データアクセスの競合が頻発する処理に対しては性能上不利になると考えられるが，これに関する評価は行えていない．ただし，3.1 節で述べたように，そのような処理の多くは台数効果が出にくくノードの参加/脱退による効果が期待できないため，DMI が主に対象とする処理ではない．

6.1 実験環境

実験環境としては，Intel Xeon E5410 2.33 GHz（4 コア）× 2 の CPU，32 GB のメモリ，カーネル 2.6.18-6-amd64 の Linux で構成されるマシン 16 ノードを 1 Gbit イーサネット × 2 でネットワーク接続した，合計 128 プロセッサのクラスタ環境を用いた．以降の実験では，DMI/MPI を n プロセッサで実行する際には，8 本の DMI スレッド/MPI プロセスを $\lfloor n/8 \rfloor$ 台のノードに立て，残りの $n - 8 \times \lfloor n/8 \rfloor$ 本の DMI スレッド/MPI プロセスを別の 1 つのノードに立てるプロセス構成とした．また，コンパイラには gcc 4.1.2，MPI には OpenMPI 1.3.3，最適化オプションには -O3 を使用した．

6.2 マイクロベンチマーク

6.2.1 read 操作の性能

DMI における read のオーバヘッドを評価した．DMI における read には，そのノードがすでにキャッシュを保有していてローカルに完了する場合（ローカル read）と，オーナーに対して read フォルトを送信して最新ページの転送を要求する場合（リモート read）の 2 種類がある．ローカル read の処理の内訳は，DMI 物理メモリのメモリ空間からユーザプログラムのメモリ空間への memcpy と，ユーザレベルによるコンシステンシ管理などの処理系のオーバヘッドである．一方，リモート read の処理の内訳は，オーナーからのページ転送と，memcpy と，処理系のオーバヘッドである．

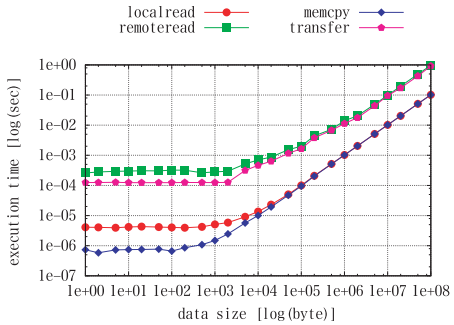

図 18　データサイズを変化させたときのローカル read（localread），リモート read（remoteread），memcpy（memcpy），データ転送（transfer）の実行時間
Fig. 18　The execution time of local read (localread), remote read (remoteread), memcpy (memcpy) and data transfers (transfer) with various data size.

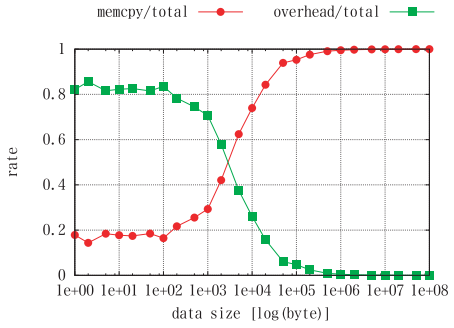

図 19　データサイズを変化させたときのローカル read の内訳
Fig. 19　The breakdown of local read with various data size.

図 18 には，さまざまなデータサイズ x に関して，ページサイズ x のページ 1 個をローカル read するのに要する時間，およびリモート read するのに要する時間，サイズ x のデータの memcpy に要する時間，サイズ x のデータの転送に要する時間を比較した結果を示す．また，ページサイズを変化させたときに，ローカル read 全体の実行時間に占める memcpy の比率および処理系のオーバヘッドの比率を**図 19** に，リモート read 全体の実行時間に占めるページ転送の比率，memcpy の比率，および処理系のオーバヘッドの比率を**図 20** に示す．

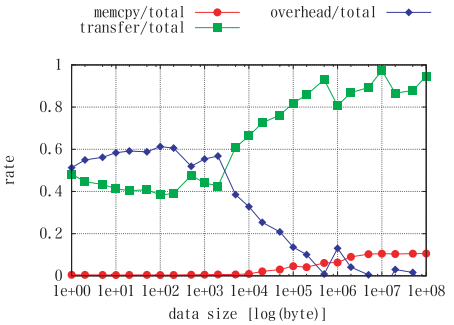

図 20　データサイズを変化させたときのリモート read の内訳
Fig. 20　The breakdown of remote read with various data size.

図 18 より，リモート read はローカル read よりも，2 MB 以上では 9.52 倍〜9.96 倍遅く，10 KB 以下では 50.8 倍〜81.4 倍遅いことが分かる．ローカル read に関して図 19 の内訳を見ると，500 KB 以上では処理系のオーバヘッドが占める比率が 1%以下となるものの，100 バイト以下では 80%以上を占めることが分かる．しかし，ローカル read では，（1）ページテーブルから該当ページの管理情報を取り出し，（2）そのページに対する他の操作を排他し，（3）アクセスチェックを行い，（4）memcpy し，（5）排他を解除するという操作を行うのみであり，これらはユーザレベルでコンシステンシ管理を行う以上は最低限必要になる操作であるため，やむをえない．また，リモート read に関して図 20 の内訳を見ると，100 KB 以上ではオーバヘッドの比率が 15%以下に抑えられるものの，2 KB 以下では 50%以上を占めており，今後ネットワーク性能が向上すれば，さらにこの比率は増加すると予測される．以上より，現状の DMI は細粒度なアクセスが頻発する処理には弱く，オーバヘッドの低減が重要な課題といえる．

6.2.2　マルチモード read/write の性能

図 16 の排他制御アルゴリズムを題材にして，マルチモード read/write の有効性を評価した．図 16 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 を行うが，DMI ではアクセス競合時の性能低下を防ぐために，すべての write と fetch-and-store と compare-and-swap を WRITE_REMOTE モードで，すべての read を READ_ONCE モードで行っている．この理由は，write などを WRITE_LOCAL モードで行うと，排他制御のたびに

オーナーが変化するため，アクセス競合時にオーナーの頻繁な移動が発生してオーナー追跡のための通信が多量に発生するためである．また，すべての read を READ_ONCE モードで行う理由は，アクセス競合時にはデータを read してから次に read するまでの間に他ノードによってそのデータが更新される可能性が高いため，READ_INVALIDATE モードや READ_UPDATE モードによってデータをキャッシュすることに意味がないうえに，多量の invalidate 要求や update 要求が発生してしまうためである．

以上のようなマルチモード read/write の使い分けの効果を検証するため，すべての write と fetch-and-store と compare-and-swap を X モードで，すべての read を Y モードで行う場合の性能を，(I) X = WRITE_REMOTE，Y = READ_ONCE，(II) X = WRITE_LOCAL，Y = READ_ONCE，(III) X = WRITE_REMOTE，Y = READ_INVALIDATE，(IV) X = WRITE_REMOTE，Y = READ_UPDATE，の 4 通りに関して，排他制御された 1 個のカウンタ変数を各 DMI スレッドが 300 回インクリメントする処理を，128 DMI スレッドで行う処理の時間を測定した．その結果，(I) が 44.97 sec，(II) が 379.87 sec，(III) が 53.11 sec，(IV) が 70.74 sec となった．この結果より，オーナーの位置やキャッシュの状況などを意識してマルチモード read/write を適切に使い分けることによって，大きな性能向上を実現できる可能性があり，マルチモード read/write は分散共有メモリにおける有効な最適化手段であるといえる．

6.2.3　非同期 read/write を利用する遠隔スワップシステムの性能

STREAM ベンチマーク[30] で採用されている copy，scale，add，triadd の各処理に関して，非同期 read/write を利用する遠隔スワップシステムの性能と HDD アクセスの性能を比較した．DMI の遠隔スワップシステムにおける評価では，16 ノードを使用し，各ノードが提供する DMI 物理メモリ量を 2 GB に設定し，3 つの 8 GB 配列 A，B，C をページサイズ 1 MB で DMI 仮想共有メモリ上に確保した．そして，各配列 A，B，C に関して，先頭から $i \times 512$ MB 以上 $(i+1) \times 512$ MB 未満 ($0 \leq i < 16$) の領域が，ノード i にはオーナー権をともなった DOWN_VALID 状態で存在し，他のノードには INVALID 状態で存在するように配列領域を分散配置したうえで，配列 A から配列 C への 8 GB の copy をノード 0 が逐次で行う実行時間を測定した．同様にして，測定のたびに前述の分散配置をやり直し，ノード 0 が逐次で scale，add，triadd を行う実行時間を測定した．この実験では，各配列の read 操作には READ_ONCE モードを，各配列の write 操作には WRITE_LOCAL モードを使用した．なお，各ノードが提供する DMI 物理メモリ量を 2 GB に設定しているため，この処理ではたえずページの追い出し処理が発生する．

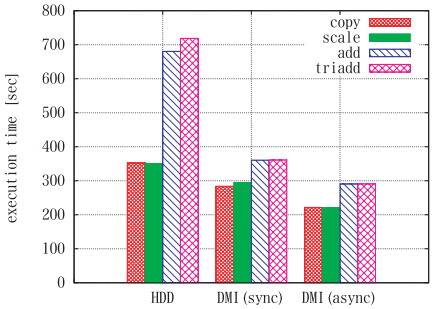

図 21　STREAM ベンチマークにおける，HDD アクセス（HDD），`DMI_read(...)/DMI_write(...)` の実行時間（DMI(sync)），非同期 `DMI_read(...)/DMI_write(...)` の実行時間（DMI(async)）

Fig. 21　The execution time of HDD access (HDD), normal `DMI_read(...)/DMI_write(...)` (DMI(sync)) and asynchronous `DMI_read(...)/DMI_write(...)` (DMI(async)) in the STREAM benchmark.

図 21 に，copy，scale，add，triadd の各処理を，(I) 通常の `DMI_read(...)/DMI_write(...)` を利用して同期的に行った場合，(II) 非同期 read/write を利用して，4 MB 先のプリフェッチおよび 4 MB 後のポストストアを行った場合，(III) SATA 7200 rpm の HDD への初回アクセスで行った場合の性能を比較した結果を示す．図 21 より，4 つの処理を平均すると，(I) は (III) より 1.58 倍高速で，(II) は (III) より 2.00 倍高速である．他の遠隔スワップシステムとの比較は行えていないが，この結果より，DMI が実現する大規模メモリが HDD よりも高速なストレージとして利用可能なこと，および非同期 read/write が有効な最適化手段であることが分かる．

6.3　アプリケーションベンチマーク

以降の実験で使用する DMI プログラムはすべて，pthread で記述したプログラムに対して，ほぼ機械的な変換作業を手動で施す形で作成した．

6.3.1　ヤコビ反復法による 3 次元熱伝導方程式の求解

3 次元熱伝導方程式をヤコビ反復法で解く処理を題材にして，ノードの動的な参加/脱退にともなって並列度を動的に変化させる効果を評価した．$n = 512$ として，$1 \leq x \leq n$，$1 \leq y \leq n$，$1 \leq z \leq n$ の各格子点を要素とする立方体の物体を考え，初期状態として，$y = 1$ の面の要素に $T_0(x, 1, z) = 1$ の温度を与えて，その他の要素の温度は $T_0(x, y, z) = 0$ とした．ヤコビ反復法の第 i イテレーションではすべての要素 (x, y, z) に関して，

$$T_i(x, y, z) = (T_{i-1}(x-1, y, z) + T_{i-1}(x, y-1, z) + T_{i-1}(x, y, z-1)$$

$$+T_{i-1}(x+1,y,z)+T_{i-1}(x,y+1,z)+T_{i-1}(x,y,z+1))/6$$

の更新を行い，$\sum_{x=1}^{n}\sum_{y=1}^{n}\sum_{z=1}^{n}|T_i(x,y,z)-T_{i-1}(x,y,z)|/n^3 < 10^{-5}$ を満たすまでイテレーションを繰り返した．DMI では，各イテレーションの先頭でノードの参加/脱退を処理し，その時点で参加しているノード間で z 軸方向に領域を均等に分割して，そのイテレーションを実行するようなプログラムを記述した．各イテレーションでは，各プロセッサが自分の担当領域の境界面の温度を計算する際に，直前のイテレーションで隣のプロセッサが更新した領域を参照する必要が生じるため，この境界面が 1 ページになるようにページサイズを設定した．また，自分の担当領域の更新には WRITE_LOCAL モードを利用し，境界面の計算時に直前のイテレーションで隣のプロセッサが更新した領域を参照する際には READ_INVALIDATE モードを使用した．この理由は，仮に担当領域の更新に WRITE_REMOTE モードを使用してしまうと，第 i イテレーションの先頭でノードの参加/脱退が発生して領域に対するプロセッサ割当てが変化した場合に，第 i イテレーション以降では他のプロセッサがオーナー権を持つページに対する更新作業につねに必要になり，多量の通信が生じてしまう（**図 22** (A)）．これに対して，WRITE_LOCAL モードを使用すれば，第 i イテレーションの先頭でノードの参加/脱退が発生して領域に対するプロセッサ割当てが変化したとしても，第 i イテレーション終了時には，その時点でのプロセッサ割当てに一致したページのオーナーの割当てが実現されるため，第 i イテレーション以降における更新作業の対象は自分がオーナー権を持つページになる（図 22 (B)）．このように，マルチモード read/write をうまく利用することで，データへのプロセッサ割当てが重要となるデータパラレルなアプリケーションに関して，動的な参加/脱退にともなうプロセッサ割当ての変化にロバストなプログラムを記述することができる．

まず，**図 23** には，さまざまなプロセッサ数に関する DMI の実行時間（DMI(time)）と速度向上度（DMI(speedup)）を示す．図 23 より，32 プロセッサまでスケールしており，32 プロセッサで 8.90 の速度向上度を達成していることが分かる．

次に，ノードの動的な参加/脱退の効果を調べるため，最初は 8 プロセッサで実行し，第 50 イテレーション終了時に 56 プロセッサを追加し，第 85 イテレーション終了時に 48 プロセッサを脱退させた場合の，各イテレーションの実行時間を**図 24** に示す．図 24 より，ノードの動的な参加/脱退にともなって動的に並列度を変化させられていることが分かる．また，第 51 イテレーションと第 86 イテレーションの実行時間が長いのは，更新領域に対するプロセッサ割当ての変化にともなって，WRITE_LOCAL モードによる更新時に，最新のページ転送をともなったオーナー権の移動が多量に発生するためである．

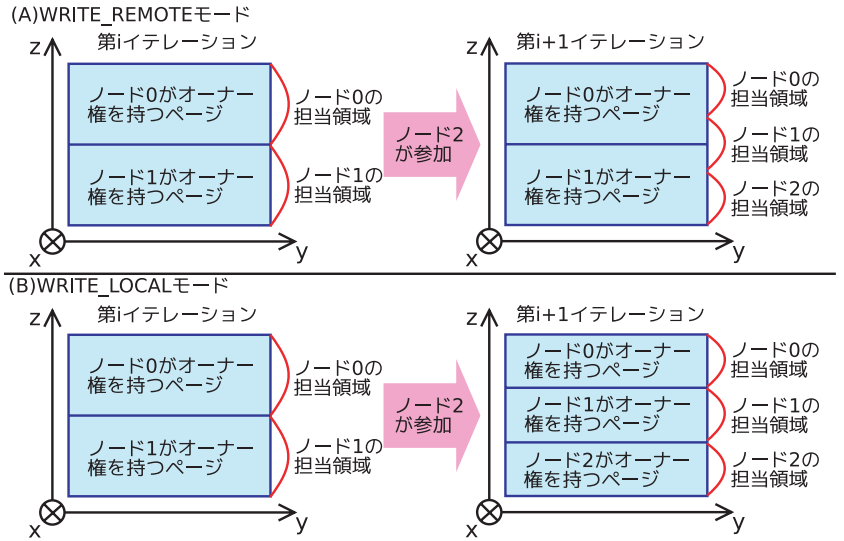

図 22　ヤコビ反復法においてプロセッサ数を増減させた場合における，領域に対するプロセッサ割当ての変化（(A) 領域更新に WRITE_REMOTE を使用した場合，(B) 領域更新に WRITE_LOCAL を使用した場合）

Fig. 22　Change of processor assignment to domains in the Jacobi iterative method when the number of processors changes ((A) WRITE_REMOTE is used to update each domain, (B) WRITE_LOCAL is used to update each domain).

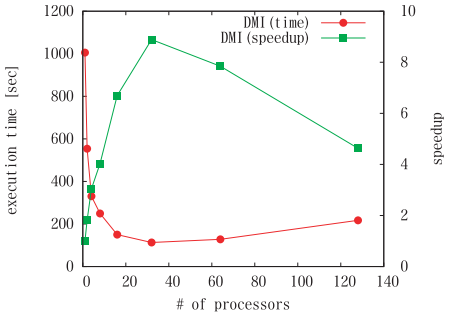

図 23　ヤコビ反復法に関する，DMI の実行時間とスケーラビリティ

Fig. 23　The execution time and scalability of DMI in the Jacobi iterative method.

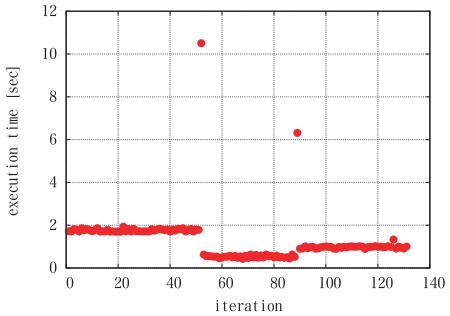

図 24　ノードを動的に参加/脱退させる場合の，ヤコビ反復法の各イテレーションの実行時間
Fig. 24　The execution time of each iteration of the Jacobi iterative method when nodes join and leave dynamically.

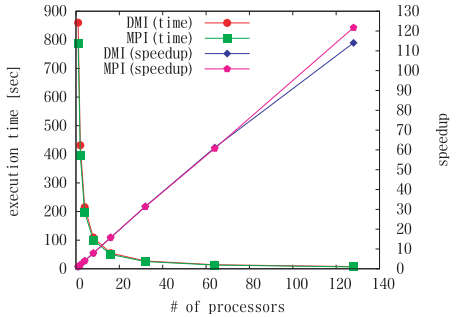

図 25　EP に関する，DMI と MPI の実行時間とスケーラビリティ
Fig. 25　The execution time and scalability of DMI and MPI in the EP benchmark.

以上のように，単純なクライアント・サーバ方式では記述できないような，プロセッサが密に協調しながら動作するアプリケーションに対しても計算資源の動的な参加/脱退をサポートし，参加/脱退に相応して動的に並列度を変えられる処理系は，我々の知る限りでは新規性の高いものである．この結果は，従来の処理系では計算資源の動的な参加/脱退をサポートできなかったアプリケーション領域に対しても，DMI によるアプローチが応用できる可能性を示唆している．

6.3.2　NAS Parallel Benchmark（EP）

NAS Parallel Benchmark [6),9)] における EP（Class C）を題材にして，DMI と MPI のスケーラビリティを比較するとともに，DMI においてノードの動的な参加/脱退にともなって並列度を動的に変化させる効果を評価した．EP（Class C）は，$n = 2^{32}$ 個の乱数の組 (x_i, y_i) を生成し，$t_i = x_i^2 + y_i^2 \leq 1$ なる (x_i, y_i) に関して $\max(|x_i\sqrt{(-2\log t_i)/t_i}|, |y_i\sqrt{(-2\log t_i)/t_i}|)$ の分布を求める処理である．DMI では，ノードの動的な参加/脱退に対応させるため，$n = 2^{32}$ を 128 個の均等なタスクに分割し，単純なマスタ・ワーカ方式によってプログラムを記述した．一方，MPI では，NPB3.3-MPI のプログラムをそのまま評価に用いた．

まず，**図 25** には，さまざまなプロセッサ数に関する DMI の実行時間（DMI(time)）と速度向上度（DMI(speedup)），MPI の実行時間（MPI(time)）と速度向上度（MPI(speedup)）を示す．この処理はほとんど通信をともなわないため，DMI は MPI とほぼ同等のスケーラビリティを達成している．DMI の実行時間が MPI よりも長いのは，演算部分に関するコンパイラの最適化などが原因で，MPI のプログラム（Fortran）では，DMI のプログラム（C 言語）よりも効率的な実行バイナリが生成されたためと思われる．次に，ノードの動的な参加/脱退の効果を調べるため，(I) 最初から最後まで 32 プロセッサで実行する場合，(II) 最初は 32 プロセッサで実行するが，約 10 sec 後に 96 プロセッサの参加を開始させる場合，(III) 最初は 32 プロセッサで実行するが，約 10 sec 後に 24 プロセッサの脱退を開始させる場合について実行時間を計測した．なお，この実験では 10 sec 後に参加/脱退を開始させただけであって，実際に参加/脱退が完了するまでには数 sec を要しており，10 sec を境目として一気にプロセス数が増減したわけではない．その結果，(I) が 27.3 sec，(II) が 17.5 sec，(III) が 47.8 sec となり，このような embarrassingly parallel な処理では，ノードの参加/脱退にともなって，効果的に並列度を変化させられることが分かった．

6.3.3　NAS Parallel Benchmark（CG）

NAS Parallel Benchmark における CG（Class B）を題材にして，DMI と MPI のスケーラビリティを比較するとともに，DMI においてノードの動的な参加/脱退にともなって並列度を動的に変化させる効果を評価した．CG（Class B）は，75000×75000 のサイズの疎行列 A に関して，以下の手順によって A の最小固有値を求める：

(1)　$\boldsymbol{x} = {}^t(1, 1, \cdots, 1)$.
(2)　$A\boldsymbol{z} = \boldsymbol{x}$ を 25 イテレーションの CG 法で解く．
(3)　$\boldsymbol{x} = \boldsymbol{z}/\|\boldsymbol{z}\|$.
(4)　(2) と (3) を 75 イテレーション繰り返す．

DMI では，(2) と (3) を構成する各イテレーションの先頭でノードの参加/脱退を処理し，その時点で参加中のノード間で行列やベクトルを均等に横ブロック分割し，そのイテ

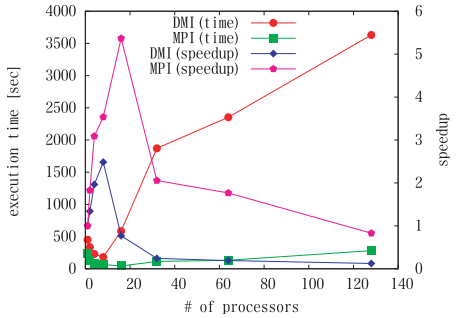

図 26　CG に関する，DMI と MPI の実行時間とスケーラビリティ
Fig. 26　The execution time and scalability of DMI and MPI in the CG benchmark.

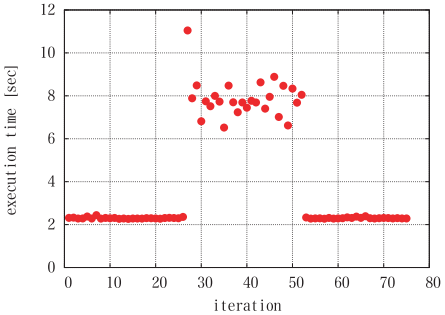

図 27　ノードを動的に参加/脱退させる場合の，CG の各イテレーションの実行時間
Fig. 27　The execution time of each iteration of the CG benchmark when nodes join and leave dynamically.

レーションを行うようなプログラムを記述した．また，この処理で最大のボトルネックになるのは (2) の CG 法内部で行われる行列ベクトル積であるが，この行列ベクトル積を，行列 A の横ブロック分割によって最大 128 プロセッサで並列演算することを想定し，行列 A のページサイズは $75000 \times 75000/128 \times sizeof(double)$ バイトとし，各ベクトルのページサイズは $75000/128 \times sizeof(double)$ バイトに設定した．一方，MPI では，NPB3.3-MPI のプログラムをそのまま評価に用いた．

まず，**図 26** には，さまざまなプロセッサ数に関する DMI の実行時間（DMI(time)）と速度向上度（DMI(speedup)），MPI の実行時間（MPI(time)）と速度向上度（MPI(speedup)）を示す．図 26 を見ると，DMI は 8 プロセッサまでしかスケールしておらず，16 プロセッサ以上では MPI よりも著しく実行時間が長い．この原因は，プロセッサ数を p，行列サイズを n としたとき，行列ベクトル積に関して，MPI では総通信量 $O(\sqrt{p}n)$ の洗練されたアルゴリズムが採用されているのに対して，DMI では簡単化のため，行列の横ブロック分割による総通信量 $O(pn)$ のアルゴリズムを採用したためと考えられる．

次に，ノードの動的な参加/脱退の効果を調べるため，上記手順の (2) と (3) の 75 イテレーションのうち，最初は 8 プロセッサで実行し，第 26 イテレーション終了時に 8 プロセッサを参加させ，第 51 イテレーション終了時に 8 プロセッサを脱退させた場合の，各イテレーションの実行時間を**図 27** に示す．図 27 ではノードの参加によって性能が落ちており，当然ながら，このようなスケーラビリティの悪い処理は参加/脱退に適さない．

6.3.4　行列行列積

4096×4096 のサイズの行列を用いた行列行列積 $AB = C$ を題材にして，DMI と MPI のスケーラビリティを比較するとともに，任意のページサイズを指定できることの有効性を評価した．アルゴリズムを MPI 風に記述すると以下のとおりである：

(1)　プロセス 0 が行列 A をプロセス数分だけ横ブロック分割し，それを全プロセスに scatter する．
(2)　プロセス 0 が行列 B を broadcast する．
(3)　各プロセス i はプロセス 0 から送信され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．
(4)　各プロセス i は部分行列 C_i をプロセス 0 に gather する．

DMI においても，read/write ベースで記述することを除けば，MPI と同様のデータ操作が起きるアルゴリズムで記述した．

まず，**図 28** には，さまざまなプロセッサ数に関して以下を測定した結果を示す：

- DMI で，行列 A，C については各横ブロックが 1 ページになるようにページサイズを設定し，行列 B については行列丸ごと 1 個が 1 ページとなるようページサイズを設定し，行列行列積を通じてページフォルトが各ページあたり 1 回しか発生しないようにした場合の実行時間（DMI_nrm(time)），その速度向上度（DMI_nrm(speedup)）
- DMI で，行列 A, B, C のページサイズを 2 KB にした場合の実行時間（DMI_2KB(time)），その速度向上度（DMI_2KB(speedup)）
- MPI で，scatter，broadcast，gather の操作に集合通信を用いた場合の実行時間（MPI_nrm(time)），その速度向上度（MPI_nrm(speedup)）

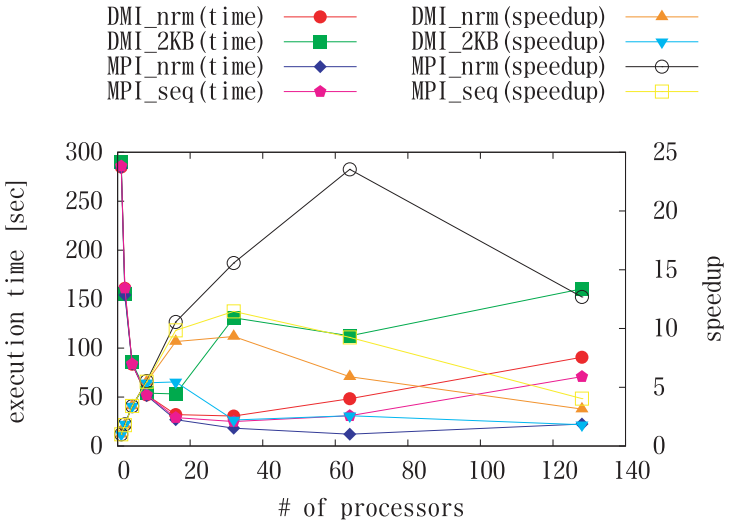

図 28　行列行列積に関する，DMI と MPI の実行時間とスケーラビリティ
Fig. 28　The execution time and scalability of DMI and MPI in a matrix-matrix multiplication.

- MPI で，集合通信を用いずに `MPI_Send(...)` による逐次転送で行った場合の実行時間（MPI_seq(time)），その速度向上度（MPI_seq(speedup)）

図 28 より，DMI_2KB の実行時間およびスケーラビリティが DMI_nrm より大きく劣っており，任意のページサイズ指定によってページフォルト回数を大幅に削減できることの有効性が分かる．また，図 28 より，DMI_nrm のスケーラビリティは MPI_nrm よりも大きく劣るが，DMI_nrm と MPI_nrm と MPI_seq の結果を比較すると，その原因の大部分が MPI の集合通信に起因していると分かる．特に影響が大きいのは行列 B の broadcast であり，現状の DMI では，**図 29** (A) のように，ページをキャッシュするノード（DOWN_VALID または UP_VALID なノード）をつねにオーナーの直下に配置する構造になっているため，read 要求を発行してきたノードへの最新ページの転送がオーナーによって逐次化され性能低下につながっている．これに対する解決策としては，図 29 (B) に示すように，オーナーに read 要求が到着した際には，オーナーが，すでにページ転送を完了したノードに対して実際のページ転送処理を動的に委譲することによって，ページ転送を全体として木構造化させるなどの工夫が考えられる．

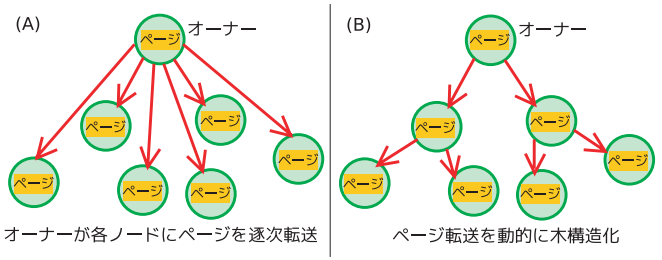

図 29　ページ転送の木構造化（(A) read 要求に対してオーナーがページを逐次転送する場合，(B) ページを木構造転送する場合）
Fig. 29　A topology of page transfers ((A) An owner transfers pages sequentially to all nodes, (B) An owner transfers pages in some balanced tree topology).

7. 関 連 研 究

メッセージパッシングをベースとして，計算資源の動的な参加/脱退をサポートした並列計算プラットフォームに Phoenix [44),49)] がある．Phoenix では，参加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 個の物理ノードに保持されるよう，各物理ノードに対して仮想ノード名集合の割当てを行う．そして，ノードが参加する場合には，すでに参加中のノードが持つ仮想ノード名集合の一部をそのノードに分け与え，脱退する場合には，そのノード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とで，計算を通じて，参加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．この管理の下では，ノードの参加/脱退を局所的な変更操作のみで実現可能であるうえ，仮想ノード名を用いたメッセージ送受信を行えば，ノードの参加/脱退が生じてもメッセージの損失が起こらない．Phoenix は，メッセージパッシングで記述された各種のプログラムを広域分散化できる記述力を持つと同時に，スケーラビリティにも優れるが，分散共有メモリをベースとして参加/脱退をサポートする DMI と比較すると，Phoenix におけるユーザプログラムの記述は相当複雑である．

研究 42) では，計算資源が動的に参加/脱退しうる広域環境上への分散共有メモリの適用可能性が論じられている．しかし，これは固定的なサーバを設けるクライアント・サーバ方式のアプローチであり，固定的な計算資源を設けることなく計算環境の動的なマイグレーションを実現可能とする DMI とは本質的に異なっている．

遠隔スワップシステムの機能を組み込んだ分散共有メモリの実装としてはCashmere-VLM[10]やJIAJIA[14]がある．Cashmere-VLMのページ置換は，5.5節で述べたようにページの状態に応じて追い出し操作の負荷が異なることを受けて，ページの状態と最終更新時刻に基づくアルゴリズムが採用されている．しかし，Cashmere-VLMの追い出し操作のプロトコルは，各ページに対して固定的な帰属ノードを設けることで実現されているため，DMIのように固定的なノードを仮定できない動的環境には適用できない．JIAJIAも各ページに対して固定的な帰属ノードを設けている．

また，DMIが採用する大規模分散共有メモリのモデルはCOMA（Cache Only Memory Architecture）に似ている．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，各アイテム（コンシステンシ維持の単位）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プロトコルが管理される．COMAの代表的な実装としてはDDM[12]がある．ただし，DDMはハードウェア上の技術であるため計算資源はつねに一定であることが前提とされており，そのプロトコルは動的な参加/脱退に対応しておらず，各アイテムに対して固定的な帰属キャッシュを設けることでアイテムの追い出し操作に対処している．

マルチコア上の並列プログラムを分散化させる際の敷居を下げることを目指し，分散共有メモリベースでpthreadを分散環境に拡張した実装としてはDSM-Threads[32),33),39)]がある．しかし，DSM-Threadsはノードの動的な参加/脱退に対応しておらず，DMIのように動的な参加/脱退を容易化させるためにpthread型のプログラミングスタイルを採用するという視点は含まれていない．また，DSM-Threadsはtoken-basedなメッセージパッシングベースの排他制御を採用しており，DMIの方がより共有メモリ環境の事情に近い排他制御を実現できているといえる．

8. 結　　論

8.1 ま と め

本稿では，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の処理系として，DMI（Distributed Memory Interface）を提案して実装し，評価した．DMIは，多様な並列分散アプリケーションを支援することを目的に多種多様な機能を備えているが，本研究の主な貢献は以下のとおりである：

- 分散共有メモリが計算資源の動的な参加/脱退に適したプログラミングモデルであることに着眼し，サーバのような固定的な計算資源を設けることなく，計算資源の動的な参加/脱退を実現できるコンシステンシプロトコルを新たに提案して実装している．
- 動的なスレッド生成/破棄が可能なpthread型のプログラミングスタイルを採用することで，シンタックスとセマンティクスの両面においてマルチコア並列プログラミングとの対応性に優れ，かつ計算資源の動的な参加/脱退に対応したユーザプログラムを容易に記述できるようなインタフェースを提供している．
- ユーザの指定する任意のサイズを粒度とするコンシステンシ維持，非同期read/write，マルチモードread/writeなど，ユーザプログラムに対して明示的で細粒度な最適化手段を提供している．

評価の結果，DMIは，ヤコビ反復法による熱伝導方程式の求解のように，単純なクライアント・サーバ方式では記述できないような，計算資源が密に協調しながら動作するアプリケーションに対しても，計算資源の動的な参加/脱退を越えた計算の継続実行をサポートし，参加/脱退に相応して動的に並列度を増減できることを確認した．このような処理系は，我々の知る限りでは新規性のあるものであり，従来の処理系では計算資源の動的な参加/脱退をサポートできなかったアプリケーション領域に対しても，DMIによるアプローチが応用できる可能性を示唆している．また，マルチモードread/writeなどを利用して明示的で細粒度なチューニングを施すことで，アプリケーションの性能を改善できることを確認した．さらに，pthreadプログラムに対してほぼ機械的な変換作業を手動で施すことで，さまざまなDMIプログラムが得られることも確認した．

8.2 今後の課題

計算資源の動的な参加/脱退が本当に重要になるのは，マルチクラスタ環境や地理的に広域分散したWAN環境などの大規模な計算環境であるが，現状のDMIは主に単一クラスタ環境上での実行しか想定できていない．その主な理由としては，計算に参加しているノード間に全対全のコネクションが張られる点，ノードの参加/脱退とDMI仮想共有メモリへのメモリ確保/解放がグローバルロックを握って行われる点などがあげられる．すなわち，現状のDMIは，各ノードが何らか“大域的な”知識を有していることを前提にデザインされており，ノードの参加/脱退に対するプロトコルは備えているものの，大規模な並列環境に適応できるデザインにはなっていない．したがって，DMIを大規模環境に適応させるためには，少なくとも，各ノードがもっと“局所的な”知識のみに基づいて動作するような設計を施す必要があり，現在処理系のデザインの再検討を行っている．また，実際に大規模環境で運用する局面を考えると，NATやファイアウォールなどの複雑なネットワーク構成への対応や耐故障性などについてもアプローチを検討していく必要がある．

謝辞　本研究は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する高度にスケーラブルなソフトウェア構成基盤」の助成を得て行われました．また，本研究を進めるにあたって，処理系のデザインから実装に至るまで，幅広くアドバイスしてくださった弘中健さんと藤澤徹さんに感謝いたします．

参 考 文 献

1) Java RMI [Online]. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
2) MPI [Online]. http://www-unix.mcs.anl.gov/mpi/
3) OpenMP [Online]. http://openmp.org/wp/
4) Amza, C., Cox, A.L., Dwarkadas, H., Keleher, P., Lu, H., Yu, W., Rajamony, R. and Zwaenepoel, W.: TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, Vol.29, No.2, pp.18–28 (1996).
5) Baiardi, F., Dobloni, G., Mori, P. and Ricci, L.: Hive: Implementing a Virtual distributed Shared Memory in Java, *Proc. Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pp.169–172 (2000).
6) Bailey, D., Harris, T., Saphir, W., vander Wijngaart, R., Woo, A. and Yarrow, M.: The NAS Parallel Benchmarks 2.0, Technical report, NAS-95-020 (Dec. 1995).
7) Basumallik, A. and Eigenmann, R.: Towards Automatic Translation of OpenMP to MPI, *Proc. 19th annual International Conference on Supercomputing*, pp.189–198 (2005).
8) Carlson, W., Sterling, T., Yelick, K. and El-Ghazawi, T.: *UPC Distributed Shared Memory Programming*, WILEY INTER-SCIENCE (June 2005).
9) Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V. and Weeratunga, S.: THE NAS PARALLEL BENCHMARKS, Technical report, RNR-94-007 (Mar. 1994).
10) Dwarkadas, S., Hardavellas, N., Kontothanassis, L., Nikhil, R. and Stets, R.: Cashmere-VLM: Remote Memory Paging for Software Distributed Shared Memory, *The 13th International Parallel Processing Symposium* (Apr. 1999).
11) El-Ghazawi, T. and Cantonnet, F.: UPC Performance and Potential: A NPB Experimental Study, *Proc. 2002 ACM/IEEE conference on Supercomputing*, pp.1–26 (Nov. 2002).
12) Hagersten, E., Landin, A. and Haridi, S.: DDM — a Cache-Only Memory Architecture, *Multiprocessor performance measurement and evaluation*, pp.304–314 (1995).
13) Anderson, J.H. and Kim, Y-J.: Shared-memory Mutual Exclusion: Major Research Trends Since 1986, *Distributed Computing*, Vol.16, No.2-3, pp.75–110 (2003).
14) Hu, W., Shi, W., Tang, Z. and Zhou, Z.: 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).
15) Huang, T-L.: Fast and Fair Mutual Exclusion for Shared Memory Systems, *Proc. 19th IEEE International Conference on Distributed Computing Systems*, pp.224–231 (1999).
16) Holzmann, G.J.: A Stack-Slicing Algorithm for Multi-Core Model Checking, *Electronic Notes in Theoretical Computer Science*, Vol.198, No.1, pp.3–16 (2008).
17) Holzmann, G.J. and Bosnacki, D.: The Design of a Multicore Extension of the SPIN Model Checker, *IEEE Trans. Softw. Eng.*, Vol.33, No.10, pp.659–674 (2007).
18) Johnson, T.: A Performance Comparison of Fast Distributed Synchronization Algorithms, *International Conference on Parallel Processing*, pp.258–264 (1994).
19) Bennett, J.K., Carter, J.B. and Zwaenepoel, W.: Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence, *Proc. 2nd ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, Vol.25, No.3, pp.168–176 (1990).
20) Kermarrec, Y. and Pautet, L.: Integrating Page Replacement in a Distributed Shared Virtual Memory, *Proc. 14th International Conference on Distributed Computing Systems*, pp.355–362 (June 1994).
21) Sinha, P.K.: *Distributed Operating Systems: Concepts and Design*, IEEE COMPUTER SOCIETY PRESS, IEEE PRESS (Dec. 1996).
22) Lamport, L.: Time, Clocks and the Ordering of Events in a Distributed System, *Comm. ACM*, Vol.21, No.7, pp.558–565 (1978).
23) Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. Computer Systems*, Vol.7, No.4, pp.321–359 (1989).
24) Li, S., Lin, Y. and Walker, M.: Region-based Software Distributed Shared Memory (May 2000).
25) Johnson, K.L., Kaashoek, M.F. and Wallach, D.A.: CRL: High-Performance All-Software Distributed Shared Memory, *Proc. 15th Symposium on Operating Systems Principles*, Vol.29, No.5, pp.213–228 (1995).
26) Neilsen, M.L. and Mizuno, M.: A Dag-Based Algorithm for Distributed Mutual Exclusion, *Proc. 11th International Conference on Distributed Computing Systems*, pp.354–360 (May 1991).
27) Lodi, G., Ghini, V., Panzieri, F. and Carloni, F.: An Object-based Fault-Tolerant Distributed Shared Memory Middleware, Technical report, Department of Computer Science University of Bologna (July 2007).

28) Maekawa, M.: A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems, *ACM Trans. Computer Systems*, Vol.3, No.2, pp.145–159 (1985).
29) Mazzucco, M., Morgan, G. and Panzieri, F.: Design and Evaluation of a Wide Area Distributed Shared Memory Middleware, Technical report, Department of Computer Science University of Bologna (July 2007).
30) McCalpin, J.D.: 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).
31) Mellor-Crummey, J.M. and Scott, M.L.: Algorithms for Scalable Synchronization on Shared-memory Multiprocessors, *ACM Trans. Computer Systems*, Vol.9, No.1, pp.21–65 (1991).
32) Mueller, F.: Distributed Shared-Memory Threads: DSM-Threads, *Workshop on Run-Time Systems for Parallel Programming*, pp.31–40 (Apr. 1997).
33) Mueller, F.: On the Design and Implementation of DSM-Threads, *Conference on Parallel and Distributed Processing Techniques and Applications*, pp.315–324 (June 1997).
34) Mueller, F.: Priority Inheritance and Ceilings for Distributed Mutual Exclusion, *IEEE Real-Time Systems Symposium*, pp.340–349 (Dec. 1999).
35) Naimi, M., Trehel, M. and Arnold, A.: A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal, *Journal of Parallel and Distributed Computing*, Vol.34, No.1, pp.1–13 (1996).
36) Bershad, B.N., Zekauskas, M.J. and Sawdon, W.A.: The Midway Distributed Shared Memory System, *Proc. IEEE Compcon Spring 1993*, pp.528–537 (1993).
37) Raymond, K.: A Tree-Based Algorithm for Distributed Mutual Exclusion, *ACM Trans. Computer Systems*, Vol.7, No.1, pp.61–77 (1989).
38) Ricart, G. and Agrawala, A.K.: An Optimal Algorithm for Mutual Exclusion in Computer Networks, *Comm. ACM*, Vol.24, No.1, pp.9–17 (1981).
39) Roblitz, T. and Mueller, F.: Combining Multi-Threading with Asynchronous Communication: A Case Study with DSM-Threads using Myrinet via BIP and Madeleine, *Myrinet User Group Conference*, pp.131–138 (Sep. 2000).
40) Saito, H. and Taura, K.: 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).
41) Fu, S.S., Tzeng, N.F. and Li, Z.: Empirical Evaluation of Distributed Mutual Exclusion Algorithms, *International Parallel Processing Symposium*, pp.255–259 (Apr. 1997).
42) Shi, W.: Heterogeneous Distributed Shared Memory on Wide Area Network, *IEEE TCCA Newsletter*, pp.71–80 (Jan. 2001).
43) Taura, K.: GXP: An Interactive Shell for the Grid Environment, *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp.59–67 (Apr. 2004).
44) Taura, K., Endo, T., Kaneda, K. and Yonezawa, A.: 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).
45) Yang, J-H. and Anderson, J.H.: A Fast, Scalable Mutual Exclusion Algorithm, *Distributed Computing*, Vol.9, No.1, pp.51–60 (1994).
46) 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).
47) 高橋　慧：Distributed Aggregate with Migration，東京大学卒業論文 (Feb. 2005).
48) 弘中　健，斎藤秀雄，高橋　慧，田浦健次朗：複雑なグリッド環境で柔軟なプログラミングを実現するフレームワーク，*Symposium on Advanced Computing Systems and Infrastructures 2008*, pp.349–358 (June 2008).
49) 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).
50) 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 10 GbEthernet 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 (2008).
51) 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 (2001).
52) 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).

付　　録

A.1　コンシステンシプロトコルのアルゴリズム

ページ *page* に関するコンシステ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* などは，*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 はそのコードを実行する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 $seq \geq 0$ 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できるものとする．

structure for page {*state, owner, probable, valid, buffer, msg_set, 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:     page.msg_set := ∅
017:
018: read operation for page with buffer and mode:
019:     lock page
020:     if page.state == INVALID
021:         || (page.state == DOWN_VALID && mode == READ_UPDATE)
022:         || (page.state == UP_VALID && (mode == READ_INVALIDATE
023:         || mode == READ_ONCE)) then
024:         REQ_READ.seq := −1
025:         REQ_READ.src := my_rank
026:         REQ_READ.mode := mode
027:         try_send(REQ_READ, page)
028:         page.probable := NIL
029:         unlock page
030:         wait for ACK_READ to be received
031:         lock page
032:         page.probable := ACK_READ.src
033:         flush_msg(page)
034:         if ACK_READ.buffer != NIL then
035:             page.buffer := ACK_READ.buffer
036:         endif
037:         page.state := ACK_READ.state
038:     endif
039:     buffer := page.buffer
040:     unlock page
041:
042: when REQ_READ for page is received:
043:     lock page
044:     if page.owner == TRUE then
045:         if page.state_array[REQ_READ.src] == INVALID then
046:             ACK_READ.buffer := page.buffer
047:             if REQ_READ.mode == READ_INVALIDATE then
048:                 page.state_array[REQ_READ.src] := DOWN_VALID
049:                 page.valid := page.valid + 1
050:             elseif REQ_READ.mode == READ_UPDATE then
051:                 page.state_array[REQ_READ.src] := UP_VALID
052:                 page.valid := page.valid + 1
053:             endif
054:         else
055:             ACK_READ.buffer := NIL
056:             if page.state_array[REQ_READ.src] == UP_VALID
057:                 && (REQ_READ.mode == READ_INVALIDATE
```

```
058:            || REQ_READ.mode == READ_ONCE) then
059:            page.state_array[REQ_READ.src] := DOWN_VALID
060:          elseif page.state_array[REQ_READ.src] == DOWN_VALID
061:                && REQ_READ.mode == READ_UPDATE then
062:            page.state_array[REQ_READ.src] := UP_VALID
063:          endif
064:        endif
065:        ACK_READ.state := page.state_array[REQ_READ.src]
066:        ACK_READ.seq := page.seq_array[REQ_READ.src]
067:        page.seq_array[REQ_READ.src] := page.seq_array[REQ_READ.src] + 1
068:        ACK_READ.src := my_rank
069:        send ACK_READ to REQ_READ.src
070:      else
071:        try_send(REQ_READ, page)
072:      endif
073:      unlock page
074:
075: write operation for page with buffer and mode:
076:      lock page
077:      if page.owner == TRUE && page.valid == 1 then
078:        page.buffer := buffer
079:      elseif mode == WRITE_REMOTE then
080:        REQ_WRITE.seq := −1
081:        REQ_WRITE.src := my_rank
082:        REQ_WRITE.buffer := buffer
083:        try_send(REQ_WRITE, page)
084:        page.probable := NIL
085:        unlock page
086:        wait for ACK_WRITE to be received
087:        lock page
088:        page.probable := ACK_WRITE.src
089:        flush_msg(page)
090:      elseif mode == WRITE_LOCAL then
091:        if page.owner == FALSE then
092:          REQ_STEAL.seq := −1
093:          REQ_STEAL.src := my_rank
094:          try_send(REQ_STEAL, page)
095:          page.probable := NIL
096:          unlock page
097:          wait for ACK_STEAL to be received
098:          lock page
099:          page.probable := my_rank
100:          flush_msg(page)
101:          if ACK_STEAL.buffer != NIL then
102:            page.buffer := ACK_STEAL.buffer
103:            page.state := DOWN_VALID
104:          endif
105:          page.state_array := ACK_STEAL.state_array
106:          page.seq_array := ACK_STEAL.seq_array
107:          page.valid := ACK_STEAL.valid
108:          page.owner := TRUE
109:        endif
110:        page.buffer := buffer
111:        update_and_invalidate(page)
112:      endif
113:      unlock page
114:
115: when REQ_WRITE for page is received:
116:      lock page
117:      if page.owner == TRUE then
118:        page.buffer := REQ_WRITE.buffer
119:        update_and_invalidate(page)
```

```
120:        ACK_WRITE.seq := page.seq_array[REQ_WRITE.src]
121:        page.seq_array[REQ_READ.src] := page.seq_array[REQ_READ.src] + 1
122:        ACK_WRITE.src := my_rank
123:        send ACK_WRITE to REQ_WRITE.src
124:      else
125:        try_send(REQ_WRITE, page)
126:      endif
127:      unlock page
128:
129: update_and_invalidate(page):
130:      if page.valid != 1 then
131:        for each rank s.t. rank != my_rank
132:                  && page.state_array[rank] == DOWN_VALID do
133:          page.state_array[rank] := INVALID
134:          page.valid := page.valid − 1
135:          REQ_INVALIDATE.seq := page.seq_array[rank]
136:          page.seq_array[rank] := page.seq_array[rank] + 1
137:          REQ_INVALIDATE.src := my_rank
138:          send REQ_INVALIDATE to rank
139:        endfor
140:        for each rank s.t. rank != my_rank
141:                  && page.state_array[rank] == UP_VALID do
142:          REQ_VALIDATE.seq := page.seq_array[rank]
143:          page.seq_array[rank] := page.seq_array[rank] + 1
144:          REQ_VALIDATE.src := my_rank
145:          REQ_VALIDATE.buffer := page.buffer
146:          send REQ_VALIDATE to rank
147:        endfor
148:        page.owner := FALSE
149:        page.probable := NIL
150:        unlock page
151:        wait for all ACK_VALIDATE to be received
152:        wait for all ACK_INVALIDATE to be received
153:        lock page
154:        page.probable := my_rank
155:        flush_msg(page)
156:        page.owner := TRUE
157:      endif
158:
159: when REQ_VALIDATE for page is received:
160:      lock page
161:      page.probable := REQ_VALIDATE.src
162:      flush_msg(page)
163:      page.buffer := REQ_VALIDATE.buffer
164:      ACK_VALIDATE.seq := −1
165:      ACK_VALIDATE.src := my_rank
166:      send ACK_VALIDATE to REQ_VALIDATE.src
167:      unlock page
168:
169: when REQ_INVALIDATE for page is received:
170:      lock page
171:      page.probable := REQ_INVALIDATE.src
172:      flush_msg(page)
173:      page.state := INVALID
174:      ACK_INVALIDATE.seq := −1
175:      ACK_INVALIDATE.src := my_rank
176:      send ACK_INVALIDATE to REQ_INVALIDATE.src
177:      unlock page
178:
179: when REQ_STEAL for page is received:
180:      lock page
181:      if page.owner == TRUE then
```

```
182:        page.owner := FALSE
183:        REQ_CHANGE.seq := page.seq_array[my_rank]
184:        page.seq_array[my_rank] := page.seq_array[my_rank] + 1
185:        REQ_CHANGE.src := REQ_STEAL.src
186:        send REQ_CHANGE to my_rank
187:        if page.state_array[REQ_STEAL.src] == INVALID then
188:          ACK_STEAL.buffer := page.buffer
189:          page.state_array[REQ_STEAL.src] := DOWN_VALID
190:          page.valid := page.valid + 1
191:        else
192:          ACK_STEAL.buffer := NIL
193:        endif
194:        ACK_STEAL.seq := page.seq_array[REQ_STEAL.src]
195:        page.seq_array[REQ_STEAL.src] := page.seq_array[REQ_STEAL.src] + 1
196:        ACK_STEAL.src := my_rank
197:        ACK_STEAL.state_array := page.state_array
198:        ACK_STEAL.seq_array := page.seq_array
199:        ACK_STEAL.valid := page.valid
200:        send ACK_STEAL to REQ_STEAL.src
201:      else
202:        try_send(REQ_STEAL, page)
203:      endif
204:      unlock page
205:
206: when REQ_CHANGE for page is received:
207:      lock page
208:      page.probable := REQ_CHANGE.src
209:      flush_msg(page)
210:      unlock page
211:
212: sweep operation for page:
213:      lock page
214:      if page.state == DOWN_VALID || page.state == UP_VALID then
215:        REQ_SWEEP.seq := −1
216:        REQ_SWEEP.src := my_rank
217:        try_send(REQ_SWEEP, page)
218:        page.probable := NIL
219:        unlock page
220:        wait for ACK_SWEEP to be received
221:        lock page
222:        page.probable := ACK_SWEEP.src
223:        flush_msg(page)
224:        page.state := INVALID
225:      endif
226:      unlock page
227:
228: when REQ_SWEEP for page is received:
229:      lock page
230:      if page.owner == TRUE then
231:        if page.state_array[REQ_SWEEP.src] == UP_VALID
232:          || page.state_array[REQ_SWEEP.src] == DOWN_VALID then
233:          page.state_array[REQ_SWEEP.src] := INVALID
234:          page.valid := page.valid − 1
235:        endif
236:        ACK_SWEEP.seq := page.seq_array[REQ_SWEEP.src]
237:        page.seq_array[REQ_SWEEP.src] := page.seq_array[REQ_SWEEP.src] + 1
238:        ACK_SWEEP.src := my_rank
239:        send ACK_SWEEP to REQ_SWEEP.src
240:        if REQ_SWEEP.src == my_rank then
241:          rank := select new owner except my_rank
242:          page.owner := FALSE
243:          REQ_CHANGE.seq := page.seq_array[my_rank]
```

```
244:          page.seq_array[my_rank] := page.seq_array[my_rank] + 1
245:          REQ_CHANGE.src := rank
246:          send REQ_CHANGE to my_rank
247:          if page.state_array[rank] == INVALID then
248:            REQ_DELEGATE.buffer := page.buffer
249:            page.state_array[rank] := DOWN_VALID
250:            page.valid := page.valid + 1
251:          else
252:            REQ_DELEGATE.buffer := NIL
253:          endif
254:          REQ_DELEGATE.seq := page.seq_array[rank]
255:          page.seq_array[rank] := page.seq_array[rank] + 1
256:          REQ_DELEGATE.src := my_rank
257:          REQ_DELEGATE.state_array := page.state_array
258:          REQ_DELEGATE.seq_array := page.seq_array
259:          REQ_DELEGATE.valid := page.valid
260:          send REQ_DELEGATE to rank
261:        endif
262:      else
263:          try_send(REQ_SWEEP, page)
264:      endif
265:      unlock page
266:
267: when REQ_DELEGATE for page is received:
268:      lock page
269:      if REQ_DELEGATE.buffer != NIL then
270:        page.buffer := REQ_DELEGATE.buffer
271:        page.state := DOWN_VALID
272:      endif
273:      page.state_array := REQ_DELEGATE.state_array
274:      page.seq_array := REQ_DELEGATE.seq_array
275:      page.valid := REQ_DELEGATE.valid
276:      page.owner := TRUE
277:      unlock page
278:
279: try_send(msg, page):
280:      if page.probable == NIL then
281:        page.msg_set := page.msg_set ∪ msg
282:      else
283:        send msg to page.probable
284:      endif
285:
286: flush_msg(page):
287:      for each msg in page.msg_set do
288:        send msg to page.probable
289:      endfor
290:      page.msg_set := ∅
```

A.2　API

API の一覧を示す．`xxx_o` の形式の引数は出力変数である．

- `int32_t DMI_rank(int32_t *rank_o)`

  自ノードのランクを `rank_o` に格納する．ランクはその時点で実行されている全ノードを通じて一意であるが，全参加ノードを通じたランクの連続性は保証されない．
- `int32_t DMI_welcome(int32_t rank)`

  `rank` で指定されるランクのノードを参加させる．
- `int32_t DMI_goodbye(int32_t rank)`

  `rank` で指定されるランクのノードを脱退させる．
- `int32_t DMI_poll(DMI_node_t *node_o)`

  参加/脱退宣言をポーリングし，参加/脱退が生じた場合にノード情報を `node_o` に格納して返る．参加/脱退宣言が存在しない場合には待機する．`DMI_node_t` のメンバ変数には，`state`（参加宣言/脱退宣言の区別），`core`（コア数），`memory`（提供する DMI 物理メモリ量），`rank`（ランク）が存在する．

- `int32_t DMI_peek(DMI_node_t *node_o)`
  `DMI_poll(...)` と同様だが，参加/脱退宣言が存在しない場合にはすぐに返る．
- `int32_t DMI_interrupt(void)`
  `DMI_poll(...)` を強制的に起こす．
- `int32_t DMI_nodes(DMI_node_t *nodes_o, int32_t *num_o, int32_t capacity)`
  参加中のノードの情報を，最大で `capacity` ノード分だけ配列 `nodes_o` に格納する．
- `int32_t DMI_mmap(int64_t *addr_o, int64_t page_size, int64_t page_num, DMI_status_t *status_o)`
  ページサイズが `page_size` でページ数が `page_num` 個の DMI 仮想メモリを確保し，そのアドレスを `addr_o` に格納する．`DMI_mmap(...)` はページテーブルを割り当てるだけで，ページの実体が割り当てられるのは各ページへの初回アクセス時である．また，`status_o` に `NULL` 以外を渡すと，この `status_o` をハンドルとした非同期操作として実行される．
- `int32_t DMI_munmap(int64_t addr, DMI_status_t *status_o)`
  DMI 仮想メモリを解放する．
- `int32_t DMI_read(int64_t addr, int64_t size, void *buf_o, int32_t mode, DMI_status_t *status_o)`
  アドレス `addr` から `size` バイトを `buf_o` に読み込む．`mode` には READ_ONCE, READ_INVALIDATE, READ_UPDATE のいずれかを指定する．読み込むアドレスは複数ページにまたがってもよい．
- `int32_t DMI_oneread(int64_t addr, int64_t size, void *buf_o, int32_t mode, DMI_status_t *status_o)`
  読み込むアドレスが 1 ページ以内に収まっていることを前提にして，`DMI_read(...)` よりも高速に動作する．
- `int32_t DMI_write(int64_t addr, int64_t size, void *buf, int32_t mode, DMI_status_t *status_o)`
  アドレス `addr` に `buf` から `size` バイトを書き込む．`mode` には WRITE_REMOTE, WRITE_LOCAL のいずれかを指定する．読み込むアドレスは複数ページにまたがってもよい．
- `int32_t DMI_onewrite(int64_t addr, int64_t size, void *buf, int32_t mode, DMI_status_t *status_o)`
  読み込むアドレスが 1 ページ以内に収まっていることを前提にして，`DMI_write(...)` よりも高速に動作する．
- `int32_t DMI_fas(int64_t addr, int64_t size, void *fetch_buf_o, void *store_buf, int32_t mode, DMI_status_t *status_o)`
  アドレス `addr` に対して fetch-and-store を発行する．`mode` には WRITE_REMOTE, WRITE_LOCAL のいずれかを指定する．
- `int32_t DMI_cas(int64_t addr, int64_t size, void *cmp_buf, void *swap_buf, int32_t *flag_o, int32_t mode, DMI_status_t *status_o)`
  アドレス `addr` に対して compare-and-swap を発行する．`mode` には WRITE_REMOTE, WRITE_LOCAL のいずれかを指定する．compare-and-swap の成否が `flag_o` に格納される．
- `int32_t DMI_save(int64_t addr, int64_t size)`
  アドレス `addr` から `size` バイトを，ページの追い出し処理の対象から外す．デフォルトではすべてのページが追い出し対象に含まれている．
- `int32_t DMI_unsave(int64_t addr, int64_t size)`
  アドレス `addr` から `size` バイトを，ページの追い出し処理の対象に含める．
- `int32_t DMI_create(DMI_thread_t *handle_o, int32_t rank, int64_t addr, DMI_status_t *status_o)`
  ランクが `rank` のノード上に DMI スレッドを生成し，そのハンドルを `handle_o` に格納する．`addr` は，生成される DMI スレッドが実行する `DMI_thread(int64_t addr)` 関数の引数に渡される．
- `int32_t DMI_join(DMI_thread_t handle, int64_t *addr_o, DMI_status_t *status_o)`
  DMI スレッドを回収する．`addr_o` が `NULL` でなければ，該当の DMI スレッドの返り値が `addr_o` に格納される．
- `int32_t DMI_detach(DMI_thread_t handle)`
  DMI スレッドを detach する．
- `int32_t DMI_self(DMI_thread_t *handle_o)`
  この DMI スレッドのハンドラを `handle_o` に格納する．
- `int32_t DMI_wake(DMI_thread_t handle)`
  `handle` で指定される DMI スレッドに対して起床通知を送る．

- `int32_t DMI_suspend(void)`
  この DMI スレッドに対して起床通知が届いているならばすぐに返る．起床通知が届いていない場合，起床通知が届くまでこの DMI スレッドをスリープさせる．
- `int32_t DMI_mutex_init(int64_t mutex_addr)`
  排他制御変数を初期化する．
- `int32_t DMI_mutex_destroy(int64_t mutex_addr)`
  排他制御変数を破棄する．
- `int32_t DMI_mutex_lock(int64_t mutex_addr)`
  排他制御変数を lock する．
- `int32_t DMI_mutex_unlock(int64_t mutex_addr)`
  排他制御変数を unlock する．
- `int32_t DMI_mutex_trylock(int64_t mutex_addr, int32_t *flag_o)`
  排他制御変数を trylock し，その成否を `flag_o` に格納する．
- `int32_t DMI_cond_init(int64_t cond_addr)`
  条件変数を初期化する．
- `int32_t DMI_cond_destroy(int64_t cond_addr)`
  条件変数を破棄する．
- `int32_t DMI_cond_signal(int64_t cond_addr)`
  条件変数の signal 操作を行う．
- `int32_t DMI_cond_broadcast(int64_t cond_addr)`
  条件変数の broadcast 操作を行う．
- `int32_t DMI_cond_wait(int64_t cond_addr, int64_t mutex_addr)`
  排他制御変数と条件変数による wait 操作を行う．
- `int32_t DMI_check(DMI_status *status, int32_t *ret_o)`
  `status` をハンドルとする非同期操作が終了したかどうか検査する．非同期操作が完了している場合，成否を `ret_o` に格納する．
- `void DMI_wait(DMI_status *status, int32_t *ret_o)`
  `status` をハンドルとする非同期操作の終了を待機する．非同期操作の成否を `ret_o` に格納する．

(平成 21 年 7 月 6 日受付)
(平成 21 年 10 月 14 日採録)

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

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

近山　　隆（正会員）
1953 年生．1982 年 3 月東京大学大学院工学系研究科情報工学専門課程博士課程修了．同年 4 月富士通株式会社入社．同年 6 月（財）新世代コンピュータ技術開発機構に出向．1995 年 4 月東京大学工学系研究科助教授，その後学内の異動を経て，2008 年 4 月より東京大学工学系研究科教授．日本ソフトウェア科学会，人工知能学会，ACM 各会員．プログラム言語と処理系，並列分散処理，機械学習に興味を持つ．