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上では，計算資源の動的な参加/脱退を越えてひと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同士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．以上を踏まえて本研究では，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，ユーザの指定する任意のサイズを単位としたコンシステンシ維持，非同期 read/write，マルチモード read/write など，分散共有メモリが抱える潜在的な台数効果の鈍さを補償するための最適化手段も提供する．評価の結果，DMI は，二分探索木への並列なデータの挿入/削除のような，多数の計算資源が密に協調しながら動作する共有メモリベースの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できることを確認した．また，マンデルブロ集合の並列描画のような Embarrassingly Parallel なアプリケーションに対しては，32 プロセッサ程度までは DMI が MPI とほぼ同等のスケーラビリティを示すことも確認できた．

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 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 several optimization methods, for example, consistency maintenance based on the user-specified arbitrary unit size, asynchronous read/write and multi-mode read/write, to compensate for the potential weak scalability of the distributed shared memory model. Our performance evaluation confirmed that beyond dynamic joining/leaving DMI can support the continuous computation of tightly coupled applications for the shared memory, such as parallel data insertion/deletion into/from a binary search tree. We also confirmed that DMI achieves scalability equivalent to MPI up to about 32 processors for embarrassingly parallel applications, such as parallel drawing of the Mandelbrot set.

†1 東京大学情報理工学系研究科
School of Information Science and Technology, The University of Tokyo
†2 東京大学工学系研究科
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伴い，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．

中でも，計算資源の動的な参加/脱退のサポートは重要な要請の一つである．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ー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できるような枠組みが求められている[42]．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同士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上では，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上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）を提案して評価する．DMIでは，特に，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） マルチコアレベルの並列性と分散レベルの並列性の効率的な活用
（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
（V） 分散共有メモリにおける潜在的な性能の鈍さを補償するための最適化手段の提供

という 5 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 プログラミングは一層と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，論理的な思考を伴う変換作業が要求されてしまう．以上を踏まえて，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，マルチコア化の加速に伴ってクラスタ環境の構成ノードのコア数も一段と増加する傾向にあるため，今後の並列分散処理系には，分散レベルの並列性と同時にマルチコアレベルの並列性を考慮した設計が必須となる．たとえば MPI では，ノード間通信にはソケット通信を利用するものの，同一ノード内では物理的な共有メモリ経由のプロセス間

通信を行うことで，プログラムからは透過的にマルチコアレベルの並列性を活用する実装が施されている[11]．DMI では，同一ノード内では複数のスレッドが物理的な共有メモリ上のリソースを “共有キャッシュ” として利用できる設計とし，マルチコアレベルの並列性を活用する．

（Ⅲ）に関して，大規模メモリは，モデル検査[15),16)] 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[9),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，エンドユーザエンド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．

（Ⅴ）に関して，分散共有メモリは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，メッセージパッシングと比較すると台数効果は得られにくい．そこで DMI では，ユーザの指定したサイズを単位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，明示的で細粒度なアプリケーションの最適化手段をユーザプログラムに対して提供する．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，段階的にチューニングを施すことで高性能なアプリケーションを開発できる．

分散共有メモリは過去 20 年以上に渡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．さらに，DMI では，pthread 型のプログラミングスタイルによる動的な参加/脱退の容易な表現，非同期モードの read/write，マルチモード read/write など，独自的なアプローチを多数採用している．

1.2 本稿の構成

2 節では，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 節では，DMI の設計コンセプトや機能について述べる．4 節では，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 節では，DMI の設計コンセプトを実現するための実装について説明する．6 節では，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 節では，関連する既存技術を紹介し DMI との比較を行う．8 節では，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」，その上に構築されるアドレス空間を「共有メモリアドレス空間」，共有メモリアドレス空間上に確保されるメモリを「仮想メモリ」，分散環境上に仮想的な共有メモリアドレス空間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」，そのアドレス空間を「仮想共有メモリアドレス空間」，特に DMI が構築する仮想共有メモリを「DMI 仮想共有メモリ」，そのアドレス空間を「DMI 仮想共有メモリアドレス空間」，DMI 仮想共有メモリアドレス空間上に確保される仮想メモリを「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[3)] のようなメッセージパッシング，TreadMarks[5)] や UPC[8)] のような分散共有メモリ，Java RMI[2)] や gluepy[48)] のような分散オブジェクト/RPC，OpenMP[4)] 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一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[42)]．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[10),39),45)]．

一方で，メッセージパッシングの第一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づらい．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．中には，Phoenix[42),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．

第二の欠点として，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り当てをどう行うべきかが問題である．たとえば，$0...i...N-1$ 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[42)]．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[3)]．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アドレス空間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，まず第一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[5)]．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．第二の利点として，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関わらず，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[46]，ノードの動的な参加/脱退が容易に記述できる．第三の利点として，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[20]．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メッセージパッシング的な通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，その抽象度の高さ故に，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[20]．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づら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[10] 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パフォーマンス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上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．

第一に，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パフォーマンスが向上するため，Sequential Consistency の IVY[22]，Eager Release Consistency の Munin[18]，Lazy Release Consistency の TreadMarks，Entry Consistency の Midway[35] 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感的な動作が掴みにくくなる上，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[20),41]．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[6),32]．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．

第二に，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 の独立性を高める上では 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ー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常に最新状態に保つ 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様々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

問い合わせ型，オーナー追跡型のアプローチが存在する[20]．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．ホーム問い合わせ型では，オーナーは動的に変化させるが，オーナーの位置を常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よい．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い合わせ型やオーナー追跡型を実装するシステムもあれば，UPC のようにデータの通信パターン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．

第三に，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ー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る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),41)]．処理系としては，IVY，TreadMarks，DSM-Threads[31),32),38)]，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),41)]．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ー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言える[5)]．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言える．処理系としては，CRL[25)]，HIVE[6)]，研究[24)] などで採用されている．

3. コンセプト

3.1 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，各ノードによって提供される DMI 物理メモリを集め，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 仮想共有メモリアドレス空間を構築する．

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ユーザスレッドが生成されていくという形態を採る．ユーザ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ユーザスレッドが 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ユーザスレッドの “共有キャッシュ” の

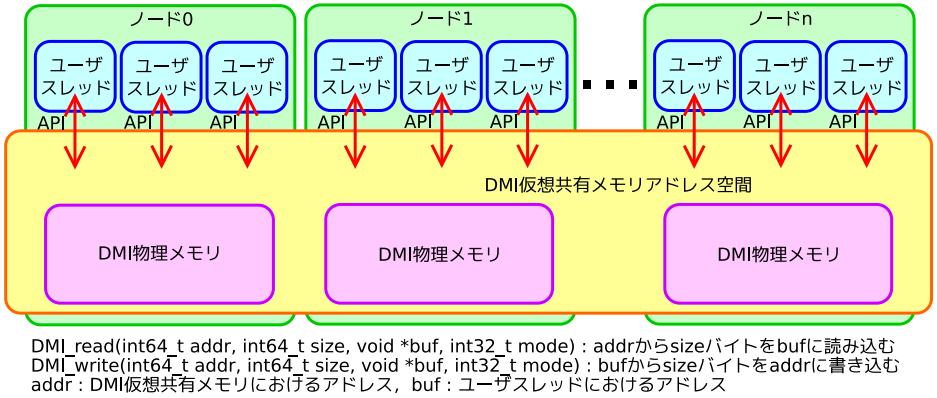

図 1 DMI における大規模分散共有メモリの構成.

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，任意のユーザスレッドは，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 で述べる．

DMI の処理系は全てユーザレベルで実装されており，OS やコンパイラには一切手を加えていないため移植性が高い．

3.2 メモリモデル

DMI では，図 1 のように，ユーザ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，ユーザ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採る．

read に関しては，`DMI_read(addr, size, buf, mode)` を呼び出すことで，DMI 仮想共有メモリアドレス空間におけるアドレス `addr` から `size` バイトを，ユーザ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，ユーザ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 仮想共有メモリアドレス空間におけるアドレス `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, chunk_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上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る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ーヘッドが小さい．

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．

3.3 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，論理的な直感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踏まえると，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 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[9),12),19),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．

3.4 関数呼び出し型の read/write

3.4.1 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．

第一の利点は，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．第二の利点は，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挙げられる．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他，OS のアドレッシング範囲に囚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．第三の利点は，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挙げられる．

分散共有メモリというプログラミングモデルは，その抽象度の高さゆえに性能が鈍い面があるが，DMI の採用するアプローチは，分散共有メモリにおける潜在的な性能の鈍さを補償する上での一つの解決策を提示していると言える．

一方で，欠点としては，第一に，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挙げられる．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．第二の欠点としては，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度にメモリコピーが発生してしまう点がある．第三の欠点としては，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ー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，オーバーヘッドが大きいという問題がある．

3.4.2 マルチモード read/write

並列分散プログラムを最適化する上では，データの所在を把握して適切に管理することが極めて重要である．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 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踏まえると，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（非キャッシュ型）

- 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 する頻度はそれほど高くないため，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（invalidate 型）
- ページが write される頻度よりも頻繁に read を行うため，自ノードのキャッシュを常に最新ページに保っておきたい状況（update 型）

などが考えられる．一方，データを write する際の状況としては，

- UPC のように，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
- 汎用マルチコアプロセッサのキャッシュのように，まず自ノードに最新ページとオーナー権を移動した上で，自ノードで write したい状況

などが考えられる．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．以上を踏まえ，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手段を提供する．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 で述べる．

3.4.3 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，ネットワーク通信を伴いうる `DMI_read(...)`/`DMI_write(...)` とローカルな計算をオーバーラップ実行したり，いわゆるプリフェッチを実現することができ，プログラムの並行実行性を高めてパフォーマンスを引き出すことができる．

3.5 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．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が実装しているが，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．

3.5.1 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),40)]．

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[21)]，$O(2(N-1))$ のメッセージ数を要する Ricart Agrawala のアルゴリズム[37)]，$O(\sqrt{N})$ 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[34)]，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[36)]，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 のアルゴリズムなどがある[26)]，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[17),40)]．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言えないが，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．

以上を踏まえ，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．

第一の理由は，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．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．その他，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上では，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言える．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わかる．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．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．

第二の理由は，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),33),34),36)]．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．このような設計は明らかに冗長であり，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，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言える．

3.5.2 共有メモリベースの排他制御

共有メモリベースの排他制御は，read/write の他に，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-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),14),43)]．たとえば，read/write/fetch-and-store/compare-and-swap を用いる MCS アルゴリズム[30)] 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[43)] 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 2** 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),14),43)]．具体例として，この構造に沿った MCS アルゴリズムを**図 3** 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 3 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 $O(1)$ であることがわかる．

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が構築できるため，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．

3.6 pthread 型のプログラミングスタイル

3.6.1 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

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

図 2 共有メモリベースの排他制御におけるプロセスモデル.

```
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();  /* Noncritical Section */
    p->next = NULL;
    pred = fetch_and_store(&tail, p);
    if(pred != NULL) {
      p->locked = 1;
      pred->next = p;
      while(p->locked == 1); /* spin */
    }
    CriticalSection();  /* Critical Section */
    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 3 MCS アルゴリズム.

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づらい．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上では 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ることで，ノードの動的な参加/脱退を容易に記述させることができる．

3.6.2 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[5),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上では重要であり，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上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 2 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 個を引数に取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構わないのに対して，図 2 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 2 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 3 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上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 4** 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度に malloc が必要となるため，重い．5.6 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．

3.7 補　　足

現状の DMI では，順序制御と信頼性が保証された通信レイヤー（TCP）を仮定している．また，ネットワーク的には均質な単一クラスタ環境を想定しており，参加ノード間で全対全のコネクション接続を張っている．ヘテロジニアスな環境や耐故障性に関しても現段階では考慮できていない．

4. プログラミングインタフェース

並列分散プログラミングフレームワークにおける DMI の位置付けを**図 5** に示す．DMI の処理系は，dmisystem と dmithread の 2 つに分類される．

dmisystem は，大規模分散共有メモリの構築，ノードの参加/脱退，コンシステンシ管理，スレッド管理など，DMI のシステムを動作させる上で本質的に必要な関数を SPI（System Programming Interface）として提供する．SPI は，基盤レイヤーとしての汎用性を最優先に設計されており，SPI の上に直接アプリケーションを記述することは想定していない．

```
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();  /* Entry Section */
  mutex->vars = vars;
}

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

  vars = mutex->vars;
  ExitSection();  /* Exit Section */
  free(vars);
}
```

図 4　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.

一方，dmithread は SPI の上に実装されている処理系であり，pthread 型のプログラミングスタイルでノードの動的な参加/脱退を容易に記述できるようにするための各種関数を API（Application Programming Interface）として提供する．たとえば，現在参加中の全ノードの情報を取得する関数，ノードの参加/脱退イベントをポーリングする関数，参加ノードの合計コア数が指定した数値に達するまで待機する関数など，各種の便利な API が整備されている．また，dmithread におけるプログラムの実行形態としては，コマンドラインで実行する際にすでに参加中の任意のノードを 1 個指定することで参加が完了し，コマンドラインからの Ctrl+C 割り込みを引き金にして脱退処理が始まる形態を取っている．したがって，GXP[1] 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/脱退を簡単に実現できる．

以上のような設計により，アプリケーションの開発者は，API を利用して並列分散プログラミング処理系や高性能なユーザエンドアプリケーションを開発できる他，必要であれば dmithread とは設計ポリシーの異なる並列分散ミドルウェアを SPI の上に直接構築することもできる．

SPI と API の全容を A.2 節に示す．また，ノードの参加/脱退に対応したプログラムの

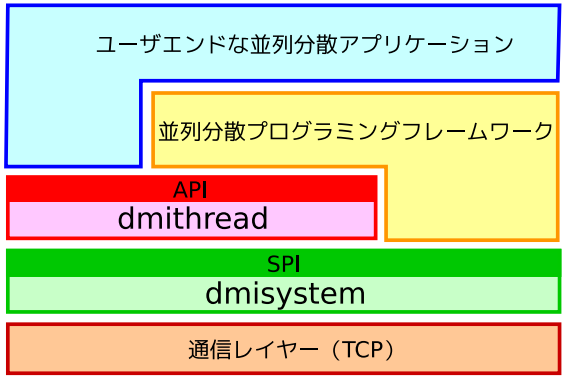

図 5　並列分散プログラミングフレームワークにおける DMI の位置付け.

記述例を**図 6** に示す.

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 スレッドと複数のユーザスレッドであり，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，全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常

```
#include "dmi.h"
#define NODE_MAX 1024
#define THR_MAX 32

int DMI_main(int argc, char **argv) { /* メイン関数 */
  int32_t i, thr_id, node_id, node_num;
  int64_t addr;
  int64_t handle[NODE_MAX][THR_MAX];
  DMI_member member;
  DMI_node nodes[NODE_MAX];

  ...;
  DMI_alloc(&addr, sizeof(int32_t), NODE_MAX * THR_MAX, 1);  /* DMI 仮想メモリを確保 */
  DMI_member_init(&member);

  thr_id = 0;
  while(1) {
    DMI_member_poll(member, nodes, &node_num, NODE_MAX); /* ノードの参加/脱退イベントをポーリング */

    for(node_id = 0; node_id < node_num; node_id++) { /* イベントが発生した各ノードに関して */
      if(nodes[node_id].state == DMI_OPEN) { /* 参加中のノードならば */
        for(i = 0; i < nodes[node_id].core; i++) { /* コア数分だけスレッドを生成 */
          DMI_write(addr + thr_id * sizeof(int32_t), sizeof(int32_t), &thr_id); /* スレッド id を書き込む */
          DMI_create(&handle[node_id][i], nodes[node_id].id, addr + thr_id * sizeof(int32_t)); /* スレッド生成 */
          thr_id++;
        }
      }
      else if(nodes[node_id].state == DMI_CLOSING) { /* 脱退処理中のノードならば */
        for(i = 0; i < nodes[node_id].core; i++) { /* 生成したスレッドを全て回収する */
          DMI_join(handle[node_id][i], NULL); /* スレッド回収 */
        }
      }
      else if(nodes[node_id].state == DMI_CLOSE) { /* 未参加のノードならば */
      }
    }
  }

  DMI_member_destroy(&member);
  DMI_free(addr);               /* DMI 仮想メモリ解放 */
  ...;
  return 0;
}

int64_t DMI_thread(int64_t addr) { /* DMI スレッド */
  int32_t thr_id;

  DMI_read(addr, sizeof(int32_t), &thr_id); /* 自分のスレッド id を読む */
  printf("my id = %d\n", thr_id);
  ...;
  return DMI_NULL;
}
```

図 6　ノードの参加/脱退に対応したプログラムの記述例．API の詳細は A.2 節を参照.

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ー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．

ユーザスレッド　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全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．

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挙げる：

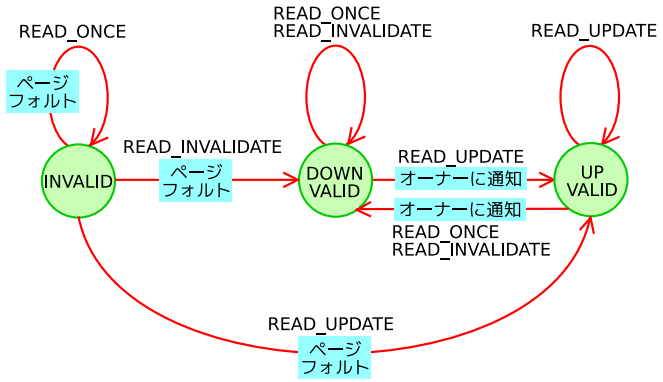

図 7　マルチモード read における状態遷移図.

- 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 なノードたちに対しては invalidate 要求を送信し，UP_VALID なノードたち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

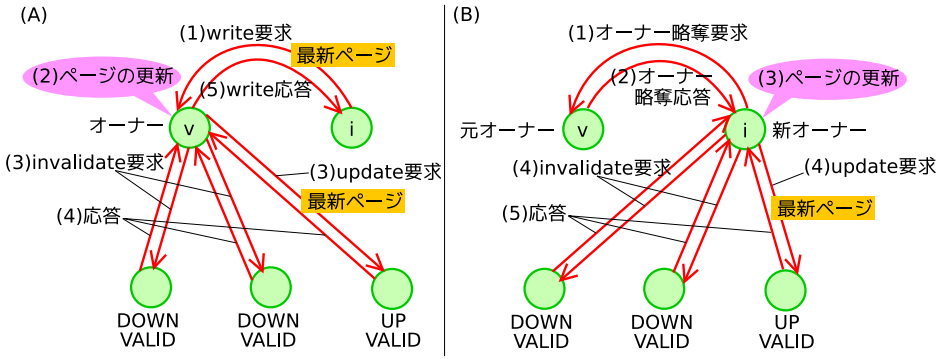

図 8 マルチモード write の挙動（(A)WRITE_REMOTE，(B)WRITE_LOCAL）．

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 v 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 v 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 は一切行われないものとして，K 回の write 全てを WRITE_REMOTE で行う場合と，K 回の write 全てを 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 回全ての 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常にオーナー 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 は常に write フォルトを引き起こすが，それらの write フォルトは常に 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 回全ての 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),34)]．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 の方が有利と言えるが，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踏まえると，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様々であるというのが最もな考え方であり，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言える．

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 A.1 節に付す．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 個全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

各ノードは各ページに対して，*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

の通りである：

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高々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辿り着く．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 のいずれかの値を取る．$i.state$ = 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関わらず，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常に知っていること
- 単独のオーナーが，全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通りにノード 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，まず第一ステップとして，

（I） 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

（II） 高々1 個だけ存在するオーナーが，全ての要求をシリアライズして処理すること

（III） 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**），第二ステップとして，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，(I)（II）（III）を満たすプロトコルの設計法について説明する．

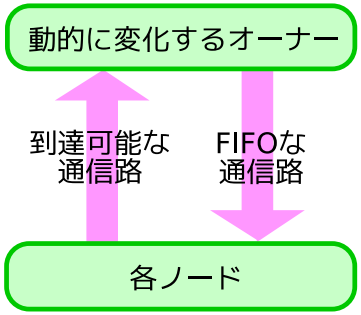

図 9 DMI のコンシステンシプロトコルが実現する各ノードとオーナーとの通信形態.

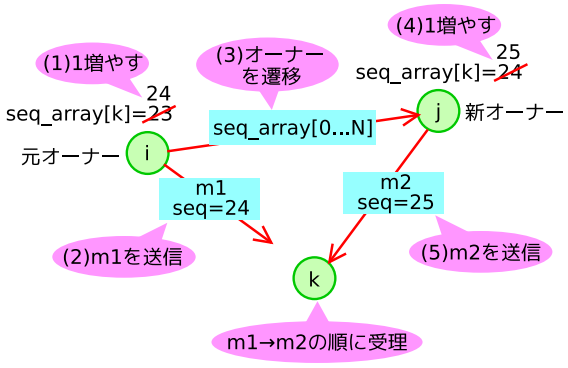

図 10 オーナーからのメッセージの順序制御.

（Ⅱ）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．

（Ⅲ）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．

（Ⅰ）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* を辿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．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過ぎ，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 で述べる最適化を施す．

以降では，上記のようにして（Ⅰ）（Ⅱ）（Ⅲ）の性質を満足したプロトコルの上に，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．これらのプロトコルでは，各ノードの *buffer*，*state*，*owner*，*state_array*，*valid*，*seq_array* の値を適宜参照/更新することでコンシステンシ維持を図るが，プロトコルの正しさの推論を容易化するため，この参照/更新に関して以下の制約を課している：

- *buffer*，*state* に関しては，（*probable* と同じく，）各ノードが参照することは任意の時点で可能だが，ノード i がそれらのデータ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である．つまり，データの所在は全ノードに分散させるが，データの値はオーナーが一括管理する．
- *owner*，*state_array*，*valid*，*seq_array* に関しては，オーナーのみが参照し書き換えることが可能である．つまり，データの所在も値もオーナーが一括管理する．

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 とした上で，$v.valid$ 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上で送信する（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ユーザスレッドが同時に 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上で送信する．

（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 を満たす全てのノード $j(\neq v)$ に対して，invalidate 要求を順序番号を付与して送信し，その都度 $v.state\_array[j]$ を INVALID に更新し，$v.valid$ を 1 減らす．$v.state\_array[j]$ = UP_VALID を満たす全てのノード $j(\neq v)$ 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)）．

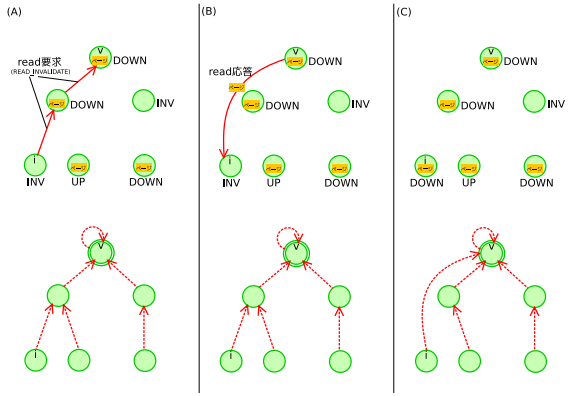

図 11 read 操作のコンシステンシプロトコル．各コマにおいて，下段がオーナー追跡グラフの形状，上段が送受信メッセージやページの状態の様子を表す．下段において二重丸で囲まれたノードはオーナーを表す．INV は INVALID，DOWN は DOWN_VALID，UP は UP_VALID を意味する．

（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 に更新した上で，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 に更新した上で，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，先ほど発行した全ての 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上で送信する（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 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

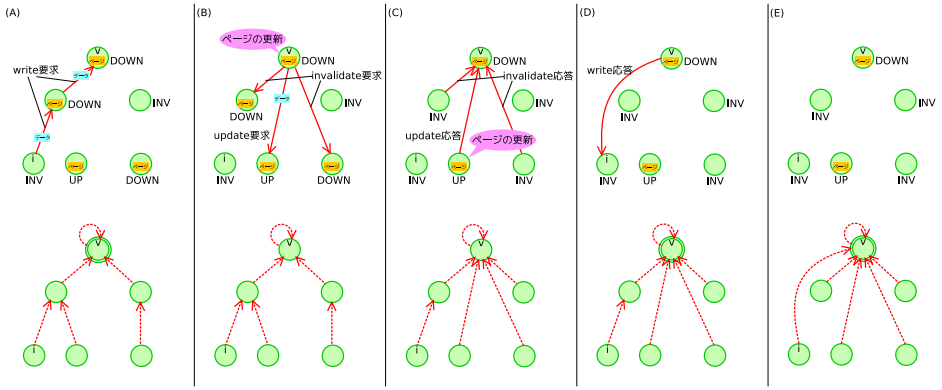

図 12 write 操作（WRITE_REMOTE）のコンシステンシプロトコル.

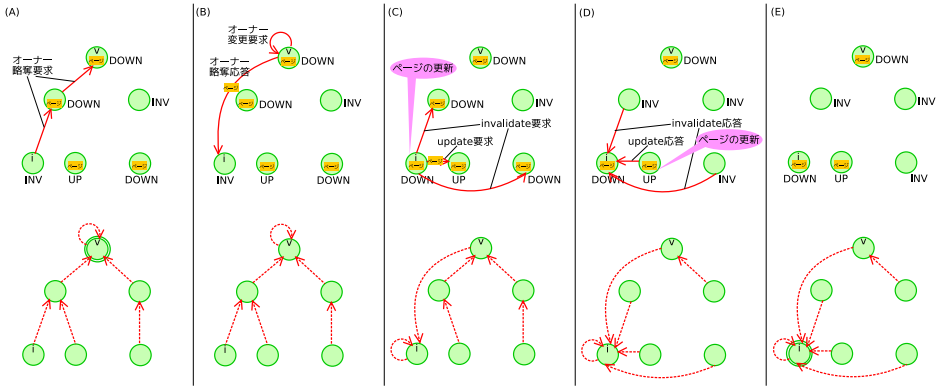

図 13 write 操作（WRITE_LOCAL）のコンシステンシプロトコル.

ノード i がオーナーでない場合には以下の（1）〜（8）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）〜（8）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上で送信する（図 13(B)）. オーナー v は，$v.owner$ を FALSE に更新し，オーナー権を放棄する．このとき $v.state\_array[i]$ = INVALID ならば，$v.state\_array[i]$ = DOWN_VALID と更新し，$v.valid$ を 1 増やした上で，ノード i に対して，最新ページと配列 $v.state\_array$ 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上で送信する（図 13(B)）．一方，$v.state\_array[i]$ = DOWN_VALID または $v.state\_array[i]$ = UP_VALID の場合には，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上で送信する.

（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 FALSE に更新し，オーナー権を放棄する．$i.state\_array[j]$ = DOWN_VALID を満たす全てのノード $j(\neq i)$ に対して，invalidate 要求を順序番号を付与して送信し，その都度 $i.state\_array[j]$ を INVALID に更新し，$i.valid$ を 1 減らす．$i.state\_array[j]$ = UP_VALID を満たす全てのノード $j(\neq i)$ 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 13(C)）.

（6） 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 i に更新した上で，ノード i に対して invalidate 応答を送信する（図 13(D)）.

（7） 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 i に更新した上で，ノード i に対して update 応答を送信する（図 13(D)）.

（8） これらの invalidate 要求または update 要求に対する全ての 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 操作において書き込むべきデータをオーナーが $i.buffer$ に格納する部分が，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)）．追い出し要求を受信したオーナー v は，手順 2 が終了した時点で $i = v$ ならば（3）～（5）も行う．$i \neq v$ であれば（3）以降は行わない．

（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上で送信する（図 14(B)）．

（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上で送信する（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．このとき $v.state\_array[v']$ = INVALID ならば，$v.state\_array[v']$ = DOWN_VALID と更新し，$v.valid$ を 1 増やした上で，新オーナー v' に対して，最新ページと配列 $v.state\_array$ 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上で送信する．一方，$v.state\_array[v']$ = DOWN_VALID または $v.state\_array[v']$ = UP_VALID の場合には，新オーナー v' に対して，配列 $v.state\_array$ と配列 $v.seq\_array$ と $v.valid$ を載せたオーナー委譲要求を，順序番号を付与した上で送信する（図 14(C)）．したがって，最新ページの転送を省略するためには，新オーナー v' としては，できるだけ $v.state\_array[v']$ = DOWN_VALID または $v.state\_array[v']$ = UP_VALID であるようなノードを選択するのが望ましい．

（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．

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言え，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．

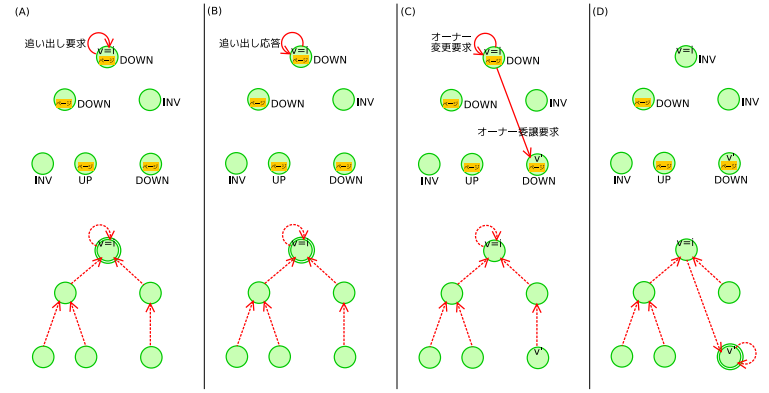

図 14 追い出し操作のコンシステンシプロトコル．

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),34)]．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/脱退処理はグローバルロックを握って行われる．グローバルロックは系内に 1 個だけ存在する排他制御変数によって実現されるもので，ノードの参加/脱退と DMI 仮想メモリの確保/解放をシリアライズするために使用される．通常のページアクセス，同期操作，スレッド管理，追い出し処理などはグローバルロックを必要としないため，グローバルロックの状態に関わらず並行して実行可能である．

第一に，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通りである：

(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．

第二に，ノード i が脱退する手順は以下の通りである：

(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全てに関して，$k.probable$ の値を $i.probable$ 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上で，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．

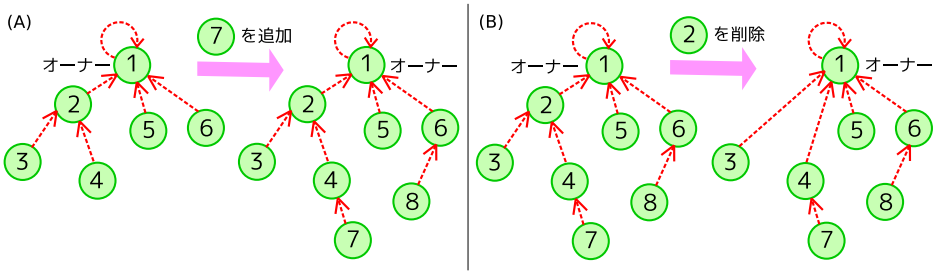

図 15 ノードの参加/脱退時におけるオーナー追跡グラフの再形成（(A) ノードの参加，(B) ノードの脱退）．

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．

第一に，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 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 な状態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わかる．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と言える．以上の考察に基づき，DMI では，ページサイズの大きい順に，(II) → (III) → (IV) の優先度順にページの追い出しを行う．

第二に，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伴う（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-base 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．ここでメッセージに付与するデータは高々20 バイト程度であり，通信量としては無視できる．

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過ぎない．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，常に L 以下になることが保証されるわけではない．これは主にパフォーマンス上の理由による．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捉えることができる．

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[14] 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 2 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

```
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

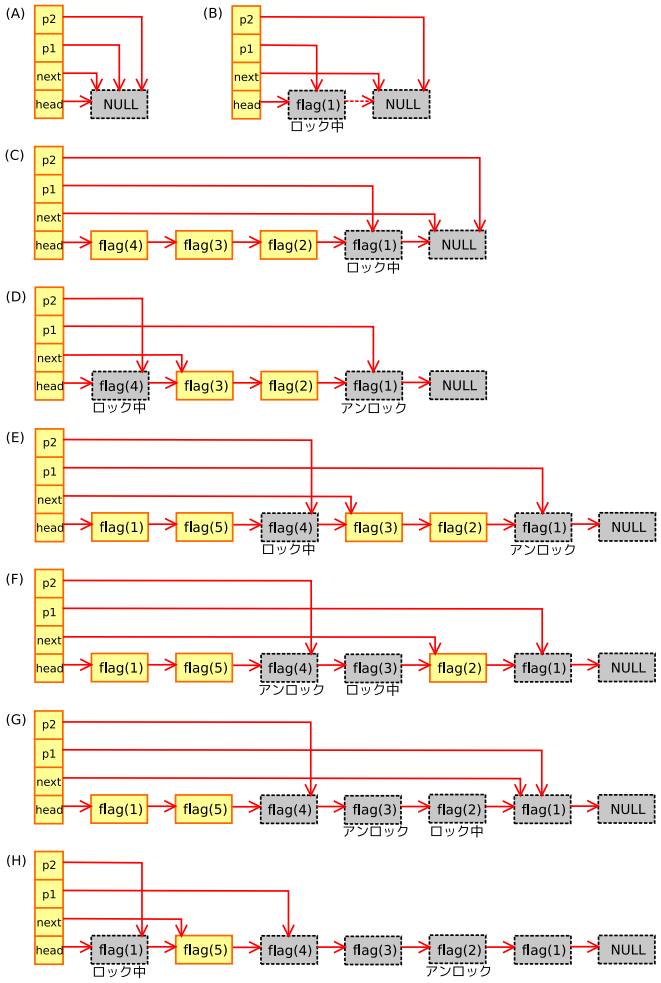

図 17 図 16 のコードによる Permission Word アルゴリズムの動き．flag(i) は，プロセス i における `lock(...)` 関数内の `flag` 変数を意味する．実線枠で囲まれた flag(i) は，まだ `lock(...)` 関数が実行中であるため実体が存在している変数，点線枠で囲まれた flag(i) は，すでに `lock(...)` 関数が終了しているため実体が消滅している変数である．

に示す．このコードの利点は，図 4 とは異なり，クリティカルセクションの度に 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` は全て `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．

(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 を入れた上で（正確には，`p2` にプロセス 4 の `flag` のアドレスを入れた上で），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 を入れた上で，`lock(...)` を抜ける（図 17(D)）．

(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 を入れた上で，`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 を入れた上で，`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関わらず，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，常に 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

現状の DMI は開発段階にあり，マルチモード read/write，共有メモリベースの分散排他制御などの実装が完了しておらず，A.2 節に示す全てのインタフェースを実装できているわけではない．本節で述べる性能評価の結果は，文献[47] に示す時点の DMI の実装に基づくものである．

使用した実験環境を**表 1** に示す．istbs 環境では 1 ノードに 1 スレッド（MPI の場合には 1 プロセス）を生成，tsukuba 環境では 1 ノードに 8 スレッド（MPI の場合には 8 プロセス）を生成して実験を行う．また，コンパイラには gcc 4.1.2 を，MPI には mpich 1.2.7p1 を使用した．

表 1 実験環境.

環境名	CPU	メモリ	OS	NIC
istbs 環境	Intel Xeon(TM) 2.40GHz×4	20GB	2.6.18-6-686	GigE
tsukuba 環境	Intel Xeon E5410 2.33GHz×8	32GB	2.6.18-6-amd64	GigE×2

6.1 マイクロベンチマーク

以下に示すマイクロベンチマークは全て istbs 環境で測定した.

第一に，DMI における read の性能を評価する．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ーヘッドである．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，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，処理系のオーバーヘッドである．

まず，**図 18** には，データサイズを変化させたときの，ローカル read，リモート read，memcpy の処理時間を示す．図 18 より，リモート read はローカル read と比較して 7〜27 倍程度遅いことがわかる．また，データサイズが 1000 バイト以下では処理系のオーバーヘッドが原因でローカル read は memcpy と比較して 30〜40 倍程度遅いが，500KB 以上になると処理系のオーバーヘッドはほぼ無視できるようになる．より詳しく観察するため，**図 19** にローカル read の全体の処理時間に占める memcpy の比率および処理系のオーバーヘッドの比率を，**図 20** にリモート read の全体の処理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ーヘッドの比率を示す．図 19 の結果を見ると，ローカル read では 1000 バイト以下のデータサイズでは処理系のオーバーヘッドが 95%以上を占めている．また，図 20 の結果を見ると，リモート read では 200 バイト以下では処理系のオーバーヘッドが 50%強を占めており，500KB 以上ではページ転送時間が全体の 70%程度を支配するものの，依然として処理系のオーバーヘッドが 20%程度を占めている．以上の結果より，DMI にとっては処理系のオーバーヘッドの削減が重要な課題であると言える．

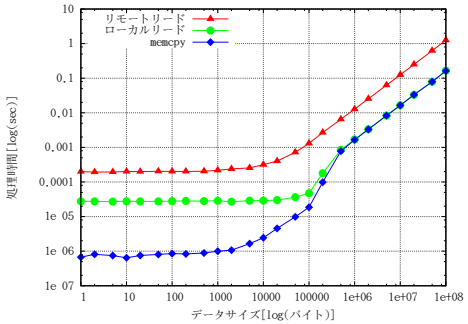

図 18 データサイズを変化させたときのローカル read，リモート read，memcpy の処理時間.

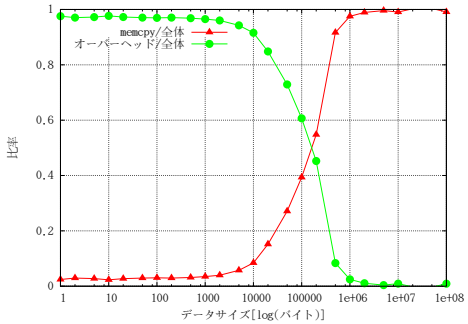

図 19 データサイズを変化させたときのローカル read の内訳.

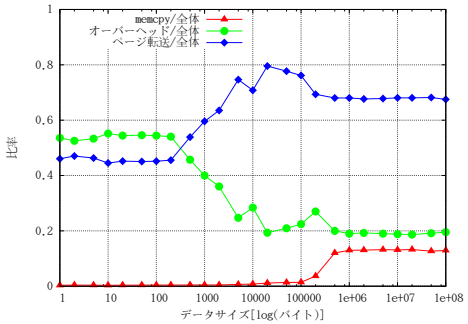

図 20 データサイズを変化させたときのリモート read の内訳.

第二に，動的な参加/脱退の性能を評価する．DMI では全対全でコネクション接続を張るため，N 個のノードがほぼ同時に参加/脱退する場合のコネクションに関する計算量は $O(N^2)$ である．しかし，実際に要する処理時間は，DMI 仮想共有メモリにどの程度のメモリが割り当てられているかや，脱退するノードがノード内の DMI 物理メモリにどの状態のページをどれくらい保有しているかによって相当に変化する．そこで，実験として，

（I）ページを何も割り当てない場合

（II）ページサイズが 4 バイトのページを 8192 個割り当て，全ノードが自ノード内にそのキャッシュを保有している場合

の 2 通りを考え，全ノードがほぼ同時に参加/脱退を試みたとき，参加/脱退が完了するのに要する時間がノード数に応じてどう変化するかを調べた．(II) の場合は（I）の場合と比較すると，参加の際には 8192 個のページに関するページ情報の転送が必要になり，脱退の際には 8192 個のページの追い出し操作が必要になる．結果を**図 21** および**図 22** に示す．まず，参加に関して図 21 を見ると，(I)（II）の場合ともに 96 ノード以上で処理時間が急激に増加しているが，これは，実装の都合上，ノード数が大規模になると初回のコネクション接続がタイムアウトする確率が高く，コネクション接続の再試行を行っていることが影響している．次に，脱退に関して図 22 を見ると，(I）は脱退時に何もページを追い出す必要がないため 128 ノードの脱退が 1.76 秒で完了しているが，(II）では 38.2 秒を要しており，脱退に要する時間はメモリの使用状況に大幅に依存することが読み取れる．

第三に，ページ置換の性能を評価する．実験として，36 ノードが 1 ノードあたり 1MB の DMI 物理メモリを提供する状況で，まず 32MB の DMI 仮想メモリに確保し，ノード 0 がその 32MB をシーケンシャルに繰り返し read し続けるプログラムを実行する．当然，ページ置換が一切行われなければノード 0 の DMI 物理メモリには 32MB が格納される状態になる．**図 23** に，ノード 0 における DMI 物理メモリの使用量を 1 秒ごとに 15000 秒間観測した結果を示す．図 23 において，点線は，指定した DMI 物理メモリ量である 1MB のラインを表す．この結果より，ノード 0 の DMI 物理メモリの使用量は，常にほぼ 400KB から 1.8MB の間の値を取っていることがわかり，ページ置換アルゴリズムが妥当に動作していると判断できる．なお，5.5 で述べたように，DMI では指定された DMI 物理メモリ量を厳格に守ってページ置換を行うわけではない．

6.2 アプリケーションベンチマーク

6.2.1 二分探索木への並列なデータの挿入/削除

動的に参加/脱退する多数のノードが，1 個の二分探索木に対して，適切な排他制御を行

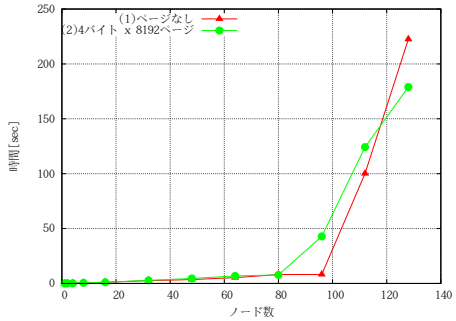

図 21 ノード数と，全ノードを参加させるのに要する時間の関係.

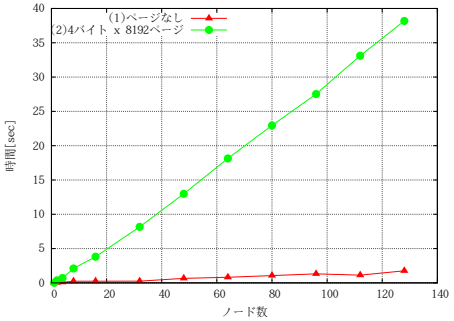

図 22 ノード数と，全ノードを脱退させるのに要する時間の関係.

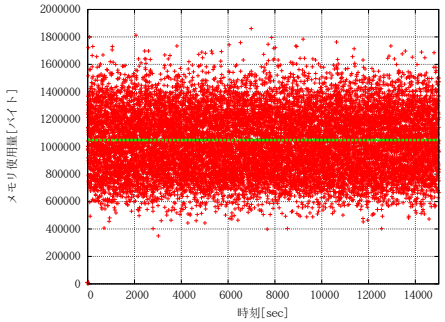

図 23 ページ置換アルゴリズムのもとでの DMI 物理メモリ使用量の時間的変化.

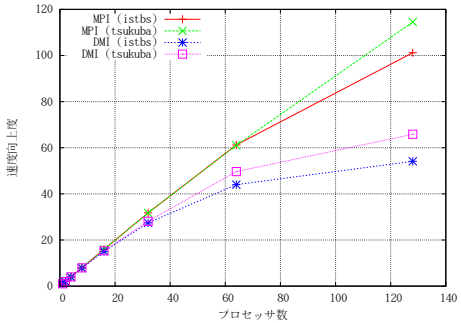

図 24 マンデルブロ集合の描画に関する DMI と MPI のスケーラビリティ.

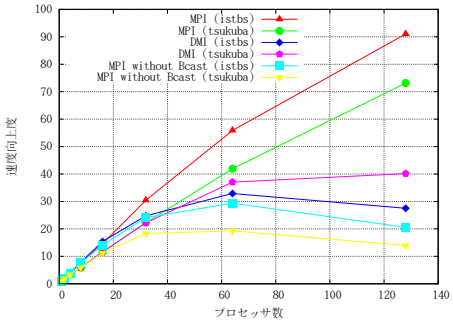

図 25 行列行列積に関する DMI と MPI のスケーラビリティ.

いながら，データをランダムに挿入/削除する DMI プログラムを作成した．この処理は，動的に変化する複雑なグラフ構造を取り扱う処理であり，木の節が動的に生成/消滅するとともにポインタが随所で書き換わるため，メッセージパッシングで記述するのは事実上困難であり，共有メモリベースであるからこそ記述できるアプリケーションと言える．

実験の結果，単一クラスタ内の 22 ノード 88 スレッドを動的に参加/脱退させて計算環境をマイグレーションさせても，依然として二分探索木の中身は正しくソートされた状態で計算が継続実行されることが確認できた．このように，多数の計算資源が密に協調しながら動作する共有メモリベースのアプリケーションに対して，ノードの参加/脱退を越えて計算の継続実行をサポートできる処理系は，我々の知る限り，新規性のあるものである．この結果は，従来の処理系では計算資源の参加/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，ここで作成した DMI プログラムは，pthread プログラムを手動で翻訳することによって得たが，この翻訳がほぼ機械的な思考に基づく変換作業で可能であることも確認した．動的な参加/脱退を記述している部分を除くと，行数としては，pthread プログラムが 663 行，DMI プログラムが 759 行であり，DMI プログラムの方が極端に分量が増えるわけではない．

6.2.2 マンデルブロ集合の並列描画

マンデルブロ集合とは，$z_0 = 0, z_{n+1} = z_n^2 + c$ で定義される複素数列 $\{z_n\}$ が $n \to \infty$ で発散しないような c の範囲を描画する問題である．マンデルブロ集合の並列描画は Embar-

rassingly Parallel なアプリケーションであるが，描画範囲によって計算量が大きく異なるため，この実験では以下に示すようなマスタワーカモデル型のアルゴリズムを用いた：

(1) マスタが，描画領域を横に K 分割し，この K 個のタスクをタスクキューに挿入する．

(2) ワーカは，タスクキューが空になるまで，タスクを取り出しては担当領域中の各点に関して発散判定を行い，その結果を描画し，描画結果をマスタに送信する．

この実験では描画領域のサイズを 480×480，$K = 480$，発散を判定するまでのイテレーション数の上限値を 1000000 とした．

図 24 に，tsukuba 環境と istbs 環境において DMI と MPI のスケーラビリティを比較した結果を示す．図 24 の結果より，32 プロセッサ程度までは DMI は MPI と同程度のスケーラビリティを達成しているが，それ以上では MPI に劣ることがわかる．tsukuba 環境で 128 プロセッサ使用時の速度向上度は，MPI が 114.5，DMI が 65.85 である．この実験においても，pthread プログラムに対してほぼ機械的な翻訳作業を手動で施すことによって DMI のプログラムが得られることを確認した．行数は pthread プログラムが 302 行，DMI プログラムが 365 行である．

6.2.3 行列行列積の並列演算

2048×2048 のサイズの行列を用いた行列行列積 $AB = C$ の並列演算を行い，DMI と MPI を性能比較した．アルゴリズムを MPI 風に記述すると以下の通りである：

(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 でも，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ようなアルゴリズムを組んだ．また，DMI では，行列 A と行列 C に関しては，各横ブロックが 1 ページになるようページサイズを設定し，行列 B に関しては行列丸ごと 1 個が 1 ページとなるようページサイズを設定したため，ページフォルトは処理を通じて各ブロックあたり 1 回しか発生しない．

図 25 に DMI と MPI のスケーラビリティを比較した結果を示す．図 25 には，istbs 環境と tsukuba 環境における DMI と MPI のスケーラビリティを示す．また，DMI においては，全ノードが行列 B を read する際にオーナーから各ノードへのページ転送が逐次的に発生することを踏まえて，それに相当する MPI の処理として，MPI において行列 B を全プロセスに `MPI_Bcast(...)` する部分を，`MPI_Send(...)` による逐次転送に置き換えたプログラムによる結果も図 25 に示す．図 25 の結果より，DMI は 64 プロセッサ付近でスケーラビリティが飽和するのに対して，MPI は 128 プロセッサでも依然として高いスケーラビリティを示すことがわかる．しかし，行列 B を `MPI_Send(...)` で逐次転送する MPI が DMI より性能が劣る点に着目すると，DMI と MPI の性能差のほぼ全てが，行列 B の転送が `MPI_Bcast(...)` によって木構造化されるのか，それとも逐次転送されるのかの違いに起因していることがわかる．すなわち，現状の DMI では，**図 26**(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）を常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たちへの最新ページの転送が逐次化され，オーナーが通信上のボトルネックになることが問題である．これに対する解決策としては，図 26(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．

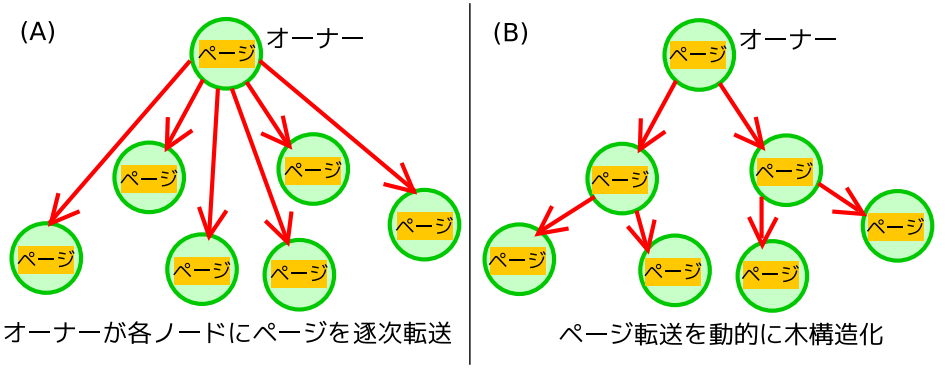

図 26 ページ転送の木構造化（(A)read フォルトを送信してきた各ノードに対してオーナーがページを逐次転送する場合，(B) ページを木構造転送する場合）．

7. 関 連 研 究

7.1 ノードの動的な参加/脱退のサポート

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[42),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り当てを行う．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常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上，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[41)] 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サーバ・クライアント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．

WAN 上での並列分散ミドルウェア基盤を狙った耐故障な分散共有メモリとして，ObFT-DSM[27),29)] がある．ObFT-DSM は，基盤言語として Java を用いることでヘテロジニアスな環境への適応性を高めるとともに，通信レイヤーとして JMS（Java Message Service）を利用することで，下層の通信事情に依存しない，スケーラブルで信頼性のある通信を実現している．また，耐故障性に対するプロトコルが提示されている．しかし，一般的に，耐故障であることとノードの脱退をサポートすることは本質的に異なる機能である．耐故障では，ある想定する故障モデルに従う故障が発生した場合におけるプログラムの継続実行は保証されるが，任意のタイミングにおける任意のノードの脱退宣言が正しく成功するとは限らず，想定する故障モデルから外れるような “運の悪い” タイミングにおけるノードの脱退に対する安全性は保証されない．これに対して，ノードの脱退のサポートでは，突然の故障（＝脱退宣言を行うことなく突然終了してしまうこと）は考慮されていないが，いかなるタイミングでいかなるノードが脱退を宣言したとしても，その脱退処理が正しく成功されてプログラムが継続実行されることは保証される．ObFT-DSM においても，ユーザプログラムによる能動的な脱退は考慮されておらず，DMI のようにノードの動的な参加/脱退に対応したユーザプログラムを記述できるわけではない．

7.2 大規模分散共有メモリ

大規模メモリを実現する遠隔スワップシステムの研究事例としては，DLM[50)] や Teramem[52)] がある．DLM では，1 台のノード上で動作する逐次プログラムが，クラスタ上の多数のノードの遠隔メモリを容易に利用可能にした遠隔スワップシステムであり，ページ置換アルゴリズムとしては単純なラウンドロビン方式が採用されている．10Gbit Ethernet 結合クラスタを用いた評価の結果，77GB の大規模メモリを実現し，ディスクスワップを用いる場合と比較して約 10 倍の性能を発揮した．また，Teramem では，MMU の参照ビットや dirty ビットの情報を活用した疑似 LRU に基づくページ置換アルゴリズムや，スワップ時に複数ページをまとめてブロック転送することによるバンド幅の有効活用などが提案されている．その結果，GNU sort をベンチマークに用いた評価においてディスクスワップの 40 倍以上の性能を発揮した．しかし，これらの遠隔スワップシステムは，DMI とは異なり，逐次プログラムに対する大規模メモリの提供のみを目的としており，分散共有メモリのような並列実行環境の提供は考慮されていない．

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[9)] がある．Cashmere-VLM のページ置換は，5.5 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．

また，DMI が採用する大規模分散共有メモリのモデルは，ハードウェアにおける COMA（Cache Only Memory Architecture）の技術に似ている．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）が常に少なくとも 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常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．

7.3 マルチコア並列プログラムとの類似性

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[31),32),38)] がある．DSM-Threads では，効率的な同期操作の実装，データの表現形式やアラインメントなどに関してヘテロジニアスな環境への対応，可搬性を確保するための標準規格への準拠などが追求されている．しかし，DMI とは異なり，OS のメモリ保護違反機構を利用しているために任意のページサイズが許されているわけではなく，ノードの動的な参加/脱退にも対応していない．また，DSM-Threads は token-based なメッセージパ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言える．

7.4 分散共有メモリの性能改善

分散共有メモリの性能改善に関して，非同期化のアイディアを導入した研究としては，非同期な Release Consistency に関する研究[44)] がある．この研究では，コンシステンシ維持のために必要な同期のためのメッセージ通信とデータ本体の通信を分離し，データ本体の通信を同期のためのメッセージ通信と非同期に並行実行できるようにした結果，invalidate 型の Lazy Release Consistency と比較して 29%高い性能を発揮したとしている．しかし，この研究は処理系内部のプロトコルの並行実行性を高めることを目的としており，非同期 read/write のインタフェースによってユーザプログラムにおける並行実行性を高めようとする DMI とは意図が異なる．

研究[7)] では，分散共有メモリにおける高い記述力とメッセージパッシングにおける高いスケーラビリティを両立させることを目的として，OpenMP で記述された分散共有メモリベースのプログラムを MPI に自動翻訳する試みが成されている．そして，翻訳時に集合通信の導入などの各種最適化を適用した結果，翻訳後の MPI コードは，翻訳前の OpenMP コードを TreadMarks ベースの OpenMP 上で実行する場合と比較して，約 30%高いスケーラビリティを達成したとしている．

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）を提案して評価した．本研究の主な貢献は以下の通りである：

- 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整備している．
- 遠隔スワップシステムとしての機能を兼ね備えた分散共有メモリであり，CPU の意味でのスケーラビリティだけでなく，メモリの意味でのスケーラビリティを達成できる設計を施している．
- ユーザの指定する任意のサイズを単位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アプリケーションの最適化を可能とするような柔軟なインタフェースを提供しており，分散共有メモリにおける潜在的な性能の鈍さを補償するためのひとつの解決策を提示している．

評価の結果，DMI は，二分探索木への並列なデータの挿入/削除のような，多数の計算資源が密に協調しながら動作する共有メモリベースの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，マンデルブロ集合の並列描画のような Embarassingly Parallel なアプリケーションに対しては，DMI は，32 プロセッサ程度までは MPI とほぼ同等のスケーラビリティを示すことも確認できた．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，これらの 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

第一に，現状の DMI は開発段階にあり，提案手法のうち一部の機能や API がまだ実装できていないため，それらの機能の実装を完了させ，より緻密な性能評価を行う必要がある．特に，任意のページ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 などの柔軟なインタフェースを駆使した場合に，DMI がメモリの意味でのスケーラビリティをどの程度まで発揮できるかは興味深く，実用的なアプリケーションベンチマークを用いて，既存の遠隔スワップシステムとの性能比較を行う必要がある．その際には，ページ置換アルゴリズムの効率化も重要な課題になると思われる．

第二に，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 仮想共有メモリ空間へのメモリ確保/解放がグローバルロックを握って行われる点などが挙げられる．すなわち，現状の DMI は，各ノードが何らか “全体” の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) GXP. http://www.logos.ic.i.u-tokyo.ac.jp/gxp/index.php.
2) Java RMI. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.
3) MPI. http://www-unix.mcs.anl.gov/mpi/.
4) OpenMP. http://openmp.org/wp/.
5) Christiana Amza, Alan L.Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, WeiminYu RamakrishnanRajamony, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 1996.
6) Fabrizio Baiardi, Gianmarco Dobloni, Paolo Mori, and Laura Ricci. Hive: Implementing a Virtual distributed Shared Memory in Java. *Proceedings of Austrian-Hungarian Workshop on Distributed and Parallel Systems*, 2000.
7) Ayon Basumallik and Rudolf Eigenmann. Towards Automatic Translation of OpenMP to MPI. *Proceedings of the 19th annual International Conference on Supercomputing*, 2005.
8) William Carlson, Thomas Sterling, Katherine Yelick, and Tarek El-Ghazawi. *UPC Distributed Shared Memory Programming*. WILEY INTER-SCIENCE, 2005.
9) Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Kontothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-VLM:Remote Memory Paging for Software Distributed Shared Memory. *the 10th Symposium on Parallel and Distributed Processing*, 1999.
10) Tarek El-Ghazawi and Francois Cantonnet. UPC Performance and Potential: A NPB Experimental Study. *Supercomputing 2002*, 2002.
11) Brice Goglin. High Throughput Intra-Node MPI Communication with Open-MX. *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing 2009*, 2009.
12) Erik Hagersten, Anders Landin, and Seif Haridi. DDM — a Cache-Only Memory Architecture. *IEEE Computer*, 1992.
13) James H.Anderson and Yong-Jik Kim. Shared-memory Mutual Exclusion: Major Research Trends Since 1986. *Distributed Computing*, 2001.
14) Ting-Lu Huang. Fast and Fair Mutual Exclusion for Shared Memory Systems. *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, 1999.
15) Gerard J.Holzmann. A Stack-Slicing Algorithm for Multi-Core Model Checking. *Electronic Notes in Theoretical Computer Science*, 2008.
16) Gerard J.Holzmann and Dragan Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, 2007.
17) Theodore Johnson. A Performance Comparison of Fast Distributed Synchronization Algorithms. *International Conference on Parallel Processing*, 1994.
18) John K.Bennett, John B.Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. *Proceedings of the Second ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming*, 1990.
19) Yvon Kermarrec and Laurent Pautet. Integrating Page Replacement in a Distributed Shared Virtual Memory. *Proceedings of the 14th International Conference on Distributed Computing Systems*, 1994.
20) Pradeep K.Sinha. *Distributed Operating Systems*. IEEE COMPUTER SOCIETY PRESS,IEEE PRESS, 1996.
21) Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 1978.
22) Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. *In Proceedings of the 1988 International Conference on Parallel Processing*, 1988.
23) Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 1989.
24) Song Li, YuLin, and Michael Walker. Region-based Software Distributed Shared Memory. 2000.
25) Kirk L.Johnson, M.Frans Kaashoek, and Deborah A.Wallach. CRL: High-Performance All-Software Distributed Shared Memory. *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.
26) Mitchell L.Neilsen and Masaaki Mizuno. A Dag-Based Algorithm for Distributed

Mutual Exclusion. *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1991.
27) Giorgia Lodi, Vittorio Ghini, Fabio Panzieri, and Filippo Carloni. An Object-based Fault-Tolerant Distributed Shared Memory Middleware. Technical report, Department of Computer Science University of Bologna, 2007.
28) Mamoru Maekawa. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, 1985.
29) Michele Mazzucco, Graham Morgan, and Fabio Panzieri. Design and Evaluation of a Wide Area Distributed Shared Memory Middleware. Technical report, Department of Computer Science University of Bologna, 2007.
30) John M.Mellor-Crummey and Michael L.Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 1991.
31) Frank Mueller. Distributed Shared-Memory Threads: DSM-Threads. *Workshop on Run-Time Systems for Parallel Programming*, 1997.
32) Frank Mueller. On the Design and Implementation of DSM-Threads. *Conference on Parallel and Distributed Processing Techniques and Applications*, 1997.
33) Frank Mueller. Priority Inheritance and Ceilings for Distributed Mutual Exclusion. *IEEE Real-Time Systems Symposium*, 1999.
34) Mohamed Naimi, Michel Trehel, and Andr Arnold. A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal. *Journal of Parallel and Distributed Computing*, 1996.
35) Brian N.Bershad, Matthew J.Zekauskas, and Wayne A.Sawdon. The Midway distributed shared memory system. *Compcon Spring '93, Digest of Papers*, 1993.
36) Kerry Raymond. A Tree-Based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems*, 1989.
37) Glenn Ricart and Ashok K.Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 1981.
38) Thomas Roblitz and Frank Mueller. Combining Multi-Threading with Asynchronous Communication: A Case Study with DSM-Threads using Myrinet via BIP and Madeleine. *Myrinet User Group Conference*, 2000.
39) Hideo Saito and Kenjiro Taura. Locality-aware Connection Management and Rank Assignment for Wide-area MPI. *IEEE International Symposium on Cluster Computing and the Grid 2007*, 2007.
40) Shiwa S.Fu, Nian feng Tzeng, and Zhiyuan Li. Empirical Evaluation of Distributed Mutual Exclusion Algorithms. *International Parallel Processing Symposium*, 1997.
41) Weisong Shi. Heterogeneous Distributed Shared Memory on Wide Area Network. *IEEE TCCA Newsletter*, 2001.
42) Kenjiro Taura, Toshio Endo, Kenji Kaneda, and Akinori Yonezawa. Phoenix:a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.
43) Jae-Heon Yang and James H.Anderson. A Fast, Scalable Mutual Exclusion Algorithm. *Distributed Computing*, 1994.
44) Jaeheung Yeo, Heon Y.Yeom, Taesoon Park, and Heon Y.YeomTaesoon Park. An Asynchronous Protocol for Release Consistent Distributed Shared Memory Systems. *Proceedings of the 2000 ACM symposium on Applied computing*, 2000.
45) 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. *Summer United Workshops on Parallel Distributed and Cooperative Processing 2008*, 2008.
46) 高橋慧. Distributed Aggregate with Migration. 東京大学 卒業論文, 2005.
47) 原健太朗. DMI：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース. 東京大学 卒業論文, 2009.
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. *SACSIS 2008*, 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. *Summer United Workshops on Parallel Distributed and Cooperative Processing 2001*, 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 10GbEthernet における初期性能評価. 情報処理学会論文誌コンピューティングシステム, 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], 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. *SACSIS 2009*, 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，各プロセスは *seq*≥ 0 なる *seq* をデータに持つメッセージに関しては，*seq* の昇順に順序制御してメッセージを受信するものとする．*seq*$= -1$ のメッセージは任意の時点ですぐ

に受信可能である．

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 := 0
```

A.2 プログラミングインタフェース
Application Programming Interface

DMI の API では，ノードの動的な参加/脱退に対応したプログラムを容易に記述可能とする関数や，マルチコア上の pthread プログラムに対応する各種関数を整備している．API を用いたプログラムでは，図 6 のように，`DMI_main(int argc, char **argv)` から実行が開始され，ユーザプログラム側で `DMI_create(...)` を呼び出す度に，指定したノード上に DMI スレッドを任意個生成できる．DMI スレッドとして実行される関数は `DMI_thread(int64_t addr)` に記述する．

API として利用可能な関数の一覧を以下に示す．なお，返り値の型が `int32_t` である API の返り値は，`DMI_SUCCESS` もしくは `DMI_FAILURE` である．また，引数における `xxx_i`，`xxx_o`，`xxx_io` の形式の変数名は，その変数がそれぞれ入力変数，出力変数，入出力変数であることを示す．

- `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_isalive(void)
  自ノードが参加状態にあるのか脱退処理中の状態にあるのかを調べる．
- int32_t DMI_nodes(DMI_node *nodes_o, int32_t *num_o, int32_t capacity)
  参加ノードにおけるランクの最大値と capacity のうち大きい方を *max_rank* とすると，ランクが 0 以上 *max_rank* 以下のノードの一覧を nodes_o に格納する．実際に格納されたノードの個数が num_o に格納される．DMI_node はノードを表す構造体であり，core（コア数），memory（メモリ量），state（状態），rank（ランク）を持つ．状態としては，DMI_OPEN（参加中），DMI_CLOSING（脱退処理中），DMI_CLOSE（未参加）の 3 状態がある．
- int32_t DMI_gather_cores(int32_t target_value)
  参加ノードの総コア数が target_value 個になるまで待機する．
- int32_t DMI_gather_nodes(int32_t target_value)
  参加ノード数が target_value 個になるまで待機する．
- int32_t DMI_gather_memory(int64_t target_value)
  参加ノードが提供する総メモリ量が target_value バイトになるまで待機する．
- void DMI_member_init(DMI_member *member_o)
  メンバを初期化する．
- void DMI_member_destroy(DMI_member *member_i)
  メンバを破棄する．
- int32_t DMI_member_poll(DMI_member *member_i, DMI_node *nodes_o, int32_t num_o, int32_t capacity)
  ノードの参加/脱退イベントをポーリングする．前回この関数を呼び出してから参加ノードの状態に変化がない限り待機し，変化が生じた時点で，参加ノードのリストを nodes_o に格納して返る．
- int32_t DMI_member_notify(DMI_member *member_i)
  DMI_member_poll(...) を強制的に起こす．
- int32_t DMI_alloc(int64_t *addr_o, int64_t page_size, int64_t page_num, int64_t chunk_num)
  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 DMI ヒープ領域に確保し，そのアドレスを addr_o に格納する．page_num 個のページのオーナーは，全ノードを通じて chunk_num 個ずつ，ラウンドロビン方式で割り当てられる．
- int32_t DMI_free(int64_t addr)
  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 仮想共有メモリにおけるアドレス 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．
- int32_t DMI_write(int64_t addr, int64_t size, void *buf_i, int32_t mode)
  DMI 仮想共有メモリにおけるアドレス addr に buf_i 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．
- int32_t DMI_fas(int64_t addr, int64_t size, void *buf_io, int32_t mode)
  DMI 仮想共有メモリにおけるアドレス addr に対して fetch-and-store を発行する．つまり，「buf_io から size バイトを addr に書き込み，書き込む直前の addr の値を buf_io に格納する」という操作をアトミックに行う．
- int32_t DMI_cas(int64_t addr, int64_t size, void *buf_i, void *swap_i, int32_t mode, int32_t *flag_o)
  DMI 仮想共有メモリにおけるアドレス addr に対して compare-and-swap を発行する．つまり，「addr から size バイトの領域が，buf_i から size バイトの領域と完全に一致しているならば，swap_i から size バイトを addr に書き込んだ上で，flag_o に TRUE を格納する．そうでなければ flag_o に FALSE の値を格納する」という操作をアトミックに行う．
- int32_t DMI_create(int64_t *handle_o, int32_t rank, int64_t addr)
  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．引数の addr は，生成される DMI スレッド DMI_thread(int64_t addr) の引数に渡される．
- int32_t DMI_join(int64_t handle, int32_t *exit_code_o)
  スレッドを回収する．該当の DMI スレッド DMI_thread(int64_t addr) の返り値が exit_code_o に格納される．
- int32_t DMI_detach(int64_t handle)
  スレッドを detach する．
- int32_t DMI_self(int64_t *handle_o)
  自スレッドのハンドラを handle_o に格納する．
- int32_t DMI_wake(int64_t handle)
  handle のハンドラで指定されるスレッドに対して起床通知を送る．

- `int32_t DMI_suspend(void)`
  自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自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．
- `void DMI_iread(int64_t addr, int64_t size, void *buf_o, int32_t mode, DMI_status *status_o)`
  非同期モードの `DMI_read(...)` であり，`status_o` にこの非同期操作のハンドルが格納される．非同期操作のため，当然この関数はすぐに返る．以下省略するが，`DMI_read(...)`, `DMI_write(...)`, `DMI_fas(...)`, `DMI_cas(...)`, `DMI_create(...)`, `DMI_join(...)`, `DMI_detach(...)`, `DMI_wake(...)` の各関数には，それに対応する非同期モードの関数が存在する．
- `int32_t DMI_check(DMI_status *status_i)`
  `status_i` 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．
- `int32_t DMI_wait(DMI_status *status_i)`
  `status_i` をハンドルとする非同期操作の終了を待機する．

System Programming Interface

DMI の SPI では，ノードの動的な参加/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，システムにとって本質的に必要な関数のみを提供する．ここで，システムにとって本質的に必要であるとは，その SPI が他の SPI の組み合わせによっては実現できないことを意味する．例外として，排他制御変数と条件変数に関する SPI については，他の SPI を組み合わせることで実現可能ではあるが，その実装の複雑さと利用度の高さを考慮して SPI の一部に入れている．SPI は一部を除けば全て非同期モードの関数である．以下に一覧を示す．

- `dmi_t* dmi_spi_init_dmi(uint16_t listen_port, int64_t mem_size)`
  DMI 物理メモリとして `mem_size` バイトのメモリを提供し，待ち受けポートとして `listen_port` を使用するノードを生成する．
- `void dmi_spi_final_dmi(dmi_t *dmi)`
  ノードを破棄する．
- `void dmi_spi_join_dmi(dmi_t *dmi, char *ip, uint16_t port, int64_t page_size, int64_t page_num, status_t *status)`
  IP アドレスが `ip`，ポートが `port` のノードをブートストラップとして参加処理を行う．このノード上に DMI スレッドを生成すると，ページサイズが `page_size` でページ数が `page_num` 個の DMI スタック領域が割り当てられる．
- `void dmi_spi_leave_dmi(dmi_t *dmi, status_t *status)`
  この関数を呼び出すと，これ以降にこのノード上で実行される全ての DMI 操作が失敗するように設定された上で，その時点で実行されている全 DMI 操作の完了が待機され，その後脱退処理を行う．ここで，DMI 操作とは，`dmi` を引数に渡す必要がある全ての SPI 呼び出しと DMI スレッドを意味する．したがって，`void dmi_spi_leave_dmi(...)` を呼び出したとしても，このノード上の DMI スレッドが全て回収されない限り，脱退処理は始まらない．よって，ユーザプログラム側では，脱退を宣言しているノードを検知して，そのノード上の DMI スレッドを回収する処理を明示的に記述する必要がある．このような参加/脱退の検知は，DMI 仮想共有メモリ上に適当なデータ構造を用意した上で参加/脱退する旨を適宜 read/write することで実装可能だが，この実装はやや面倒なため，DMI の API では

参加/脱退に関わる一連の作業を，DMI_node と DMI_member を用いる容易なインタフェースに抽象化している．

- void dmi_spi_open_threads(dmi_t *dmi, status_t *status)
  このノード上への DMI スレッドの生成を許可する．
- void dmi_spi_close_threads(dmi_t *dmi, status_t *status)
  この関数が呼び出された時点以降におけるこのノード上への DMI スレッドの生成を禁止した上で，この時点で実行されている全 DMI スレッドの完了を待機する．

その他，前述の API にほぼそのまま対応する SPI として，dmi_spi_init_mutex(...), dmi_spi_destroy_mutex(...), dmi_spi_lock_mutex(...), dmi_spi_unlock_mutex(...), dmi_spi_trylock_mutex(...), dmi_spi_init_cond(...), dmi_spi_destroy_cond(...), dmi_spi_signal_cond(...), dmi_spi_broadcast_cond(...), dmi_spi_wait_cond(...), dmi_spi_rank_dmi(...), dmi_spi_map_memory(...), dmi_spi_unmap_memory(...). dmi_spi_read_memory(...), dmi_spi_write_memory(...), dmi_spi_cas_memory(...), dmi_spi_fas_memory(...), dmi_spi_create_thread(...), dmi_spi_join_thread(...), dmi_spi_detach_thread(...), dmi_spi_wake_thread(...), dmi_spi_suspend_thread(...), dmi_spi_self_thread(...), dmi_spi_check_status(...), dmi_spi_wait_status(...) が存在する．