卒業論文

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

平成 21 年 2 月 16 日提出

指導教員　近山　隆　教授
田浦 健次朗 准教授

電子情報工学科
70408 原 健太朗

概要

近年では，情報産業に限らず，気象予測や金融計算などの産業界の各種応用分野において，ユーザエンドな並列分散アプリケーションが積極的に開発され利用されている．一般に，これら並列分散アプリケーションの性能や機能は，基盤として用いられる並列分散プログラミング処理系が提供する性能や機能に支えられているため，近年のアプリケーションの多様化や高度化，およびそれを実行する計算環境の大規模化に伴って，並列分散プログラミング処理系に求められる要請も一段と多様化している．

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．計算環境とアプリケーションの大規模化が加速している現在，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．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ではなく，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非常に限定されている．したがって，多様な並列分散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計算資源の動的な参加/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．さらに，計算資源の動的な参加/脱退を越えた並列計算の継続実行を実現するためには，ユーザプログラム側でそれに対応したプログラム記述が成される必要があることを踏まえれば，処理系としては，ユーザプログラムが計算資源の動的な参加/脱退に関わる処理を容易に表現できるようなインタフェースを整備していなければならない．

本研究では，以上の要件をもとに各種の並列分散プログラミングモデルを分析した結果，計算資源の動的な参加/脱退をサポートする上では，分散共有メモリが優れたプログラミングモデルであると考えた．その上で，従来の多くの分散共有メモリが採用してきた SPMD 型のプログラミングスタイルではなく，動的なスレッド生成/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によって，計算資源の動的な参加/脱退に対応したプログラムを容易に記述できるインタフェースを提供できると考えた．

以上のような動機から，本研究では，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）を提案して評価する．また，分散共有メモリベースの処理系を開発するという観点から，さらなる要請として，(1) マルチコア上の並列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，(2) 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，(3) 多数のノードのメモリを集めた大規模メモリの実現，(4) 並列分散ミドルウェア基盤としての柔軟で汎用的なインタフェースの整備という 4 つの要請に焦点を当てた設計を施し，より多様な並列分散アプリケーションを支援できるような処理系を目標とする．

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．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 が，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．

目次

1 序論

1.1 本研究の背景と目的

近年，情報産業に限らず産業界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．たとえば，気象予測，株価計算，衝突解析，創薬研究などの大規模なシミュレーション計算を要求する各種の応用分野において，特定の問題領域に特化したユーザエンドな並列分散アプリケーションが積極的に開発され利用されている．また，これらの並列分散アプリケーションの実行を支援する計算環境の発展もめざましい．最近では，汎用アーキテクチャで構成されたワークステーションによるクラスタ環境が普及するなど，並列分散環境の汎用化が進んでおり，大規模な計算資源が手に入りやすい時代になっている．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技術革新も顕著であり，適用可能な並列分散アプリケーションの領域や規模も飛躍的に拡大している．

一般に，これらのユーザエンドな並列分散アプリケーションは何らかの並列分散プログラミング処理系を基盤として開発されており，並列分散アプリケーションの性能や機能は，基盤として用いられる並列分散プログラミング処理系が提供する性能や機能に支えられている．そのため，並列分散プログラミング処理系に対しては高い記述力とスケーラビリティが共通の要請として課せられる他，近年の並列分散アプリケーションの多様化や高度化，およびそれを実行する計算環境の大規模化に伴って，機能面での要請も一段と多様化している．

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，計算環境とアプリケーションの大規模化が加速している現在，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 [35]．このような枠組みの代表例としては，クライアント・サーバ方式に基づくものがある．クライアント・サーバ方式では，計算に関係する重要な状態がサーバによって一括管理され，通信がクライアントとサーバ間のみに限定されるため，クライアントは任意のタイミングで計算に参加/脱退することができる．しかし，クライアント・サーバ方式は特定の計算資源に負荷が集中するためスケーラブルではなく，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非常に限定される．したがって，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，計算資源の動的な参加/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 [41]．さらに，計算資源の動的な参加/脱退をサポートする上では，高い記述力を兼ね備えたインタフェースの整備も不可欠の課題である．当然，計算資源の動的な参加/脱退を越えた並列計算の継続実行を実現するには，ユーザプログラム側でそれに対応したプログラム記述が成される必要がある．そのため，処理系としては，ユーザプログラムが計算資源の動的な参加/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を整備していなければならない．

詳細は **2** 節で分析するが，本研究では，以上の要件をもとに各種の並列分散プログラミングモデルを分析した結果，計算資源の動的な参加/脱退をサポートする上では，分散共有メモリが優れたプログラミングモデルであると考えている．その上で，従来の多くの分散共有メモリが採用してきた SPMD 型のプログラミングスタイルではなく，動的なスレッド生成/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によって，計算資源の動的な参加/脱退に対応したプログラムを容易に記述できるインタフェースを提供できると考えている．

以上のような動機から，本研究では，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計算資源の動的な参加/脱退をサポートできることを目的として，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を提案して評価する．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，(1) マルチコア並列プログラミングとの類似性，(2) 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，(3) 多数のノードのメモリを集めた大規模メモリの実現，(4) 並列分散ミドルウェア基盤としての柔軟で汎用的なインタフェースの整備という 4 つの要請に焦点を当てた設計を施し，より多様な並列分散アプリケーションを支援できる処理系を目指す．

まず第一の要請として，マルチコア並列プログラミングとの類似性，特にマルチコア上の並列プログラミング手法として最も汎用的で一般的な pthread プログラミングとの類似性について考える．Pollack の法則によれば，CPU の発熱量はトランジスタ数に比例するが，性能はトランジスタ数の平方根にしか比例しない．この経験則はシングルコアによるアプローチの限界を示している．そのため近年の 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 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環境を求めている．ところが，そ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，pthread による並列化は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のが現状である．この原因は，pthread による並列プログラミングと分散プログラミングの間の飛躍の大きさにあると言える．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，pthread プログラムをこれらの分散共有メモリシステム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，本研究では，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．

第二の要請として，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について考える．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り，最近では 8 コアや 16 コアのプロセッサが用いられる場合も多い．したがって，これからの並列分散処理系には，マルチコアレベルの並列性と分散レベルの並列性を同時に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 [14]．本研究では，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．

第三の要請として，多数のノードのメモリを集めた大規模メモリの実現について考える．大規模メモリは，モデル検査 [17, 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リモートメモリへのアクセス時間の方が高速になっているため，ネットワークページングによって大規模メモリを実現する大規模分散共有メモリが出現している [42, 12]．しかし，既存の大規模分散共有メモリの多くは，OS のメモリ保護違反機構を利用しているために 64bit OS が前提とされている．これに対して本研究では，ユーザレベルで OS のメモリ管理機構をシミュレートすることで，OS のアドレッシング範囲を越えた大規模分散共有メモリを実現する．

第四の要請として，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について考える．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作業的な記述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．また，より高性能なアプリケーション開発を支援するためには，ユーザプログラムに対してチューニングの機会を数多く提供できるような柔軟なインタフェース設計が望ましい．以上を踏まえ，本研究では，ユーザエンドな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，ユーザレベルで OS のメモリ管理機構をシミュレートすることによって，柔軟性，汎用性，機能拡張性に富んだインタフェースを提供できるようにする．将来的には，本研究で構築する処理系の上位レイヤーとして，より抽象度の高い並列分散プログラミングフレームワークを開発するなどの応用も視野に入れている．

1.2 本研究の貢献

本研究では，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）を提案して評価する．DMI 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提案する．
- 従来の多くの分散共有メモリが採用している SPMD 型のプログラミングスタイルではなく，動的なスレッド生成/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，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する．
- pthread によるマルチコア並列プログラムとの類似性を重視し，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によって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．
- 並列分散ミドルウェア基盤としての処理系を意識し，ユーザレベルで OS のメモリ管理機構をシミュレートすることによって，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，高い柔軟性と汎用性を兼ね備えたインタフェースを提供する．

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．また，本研究では，**3** 節で述べる設計コンセプトに基づき，非同期モードの read/write，ノードの参加/脱退を容易に表現するための API，データ転送の動的負荷分散など，細部において独自的なアプローチを多数採用している．

1.3 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 処理系の具体的な実装を説明する．**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」，メモリ確保によって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」，メモリ確保によって DMI 仮想共有メモリアドレス空間に確保されるメモリを「DMI 仮想メモリ」と呼ぶ．

2 並列分散プログラミングモデル

本節では，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，計算資源の動的な参加/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．また，既存の分散共有メモリの処理系が採用しているさまざまなアプローチについて説明する．

2.1 概観

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 [4] のようなメッセージパッシング，TreadMarks [7] や UPC [11] のような分散共有メモリ，Java RMI [3] や gluepy [38] のような分散オブジェクト/RPC，OpenMP [5] のようなコンパイラによる並列化技法などがある．メッセージパッシングでは，コネクション接続やトポロジなどの複雑な通信事情やソケット処理が抽象化され，処理系によって提供される名前空間上での任意のノード間通信や集合通信を記述可能である．分散共有メモリでは，内部で実際に発生する通信が隠蔽され，あたかも共有メモリ上の read/write ベースのプログラミングインタフェースが提供される．分散オブジェクト/RPC では，オブジェクトに関連付けられたメソッド呼び出しの形式でリモートノードでの処理が実行されるため，オブジェクト指向ベースでの分散処理の記述が可能である [37]．OpenMP のようなコンパイラによる並列化技法では，逐次プログラムに対してディレクティブを挿入するだけでコンパイラによる並列化が行われるため，並列化作業が非常に容易である．しかし，対象とする問題領域が主としてデータパラレルな処理に限られている [7]．なお，OpenMP は共有メモリマシン上での並列化手法として有名であるが，Omni [40, 39] などにおいて分散系への拡張が成されている．

本研究では，並列分散プログラミングフレームワークの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．そこで次節以降では，メッセージパッシングと分散共有メモリに焦点を絞り，記述力，スケーラビリティ，ノードの動的な増減への適応力などの観点からその特徴を分析する．

2.2 メッセージパッシング

2.2.1 利点と欠点

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り，その意味でメッセージパッシングは，ユーザプログラム側に全ノードとランクの対応関係を把握させ，データの所在を明示的に管理させるモデルである．

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，ユーザプログラム側で記述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．すなわち，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，通信に無駄がない [35]．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 [36, 33, 13]．

一方で，メッセージパッシングの欠点としては，まず第一にプログラミング上の負担の大きさがある．データの所在や通信形態をユーザプログラム側で管理しなければならないため，データフローが比較的単純な並列計算などは容易に記述できても，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のは事実上困難である．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．さらに，メッセージパッシングではデータを扱う際に送信側と受信側の両者を記述しなければならないため，メッセージパッシングベースのアルゴリズムは共有メモリベースのアルゴリズムと比較して論理的に難解であることが多い．

第二の欠点としては，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 [35]．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 [4]．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，データの物理的な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と言える．

2.2.2 処理系のアプローチ

以上の事実を背景として，メッセージパッシングの処理系の研究は，主として，ユーザプログラムで記述されたデータ通信をいかに効率的に実現するかに焦点が当てられている．具体的には，集合通信アルゴリズムの最適化，レイテンシやバンド幅が非均質なマルチクラスタ上でスケーラブルな通信を実現するためのコネクション接続，NAT やファイアウォールなどの複雑なネットワーク構成のために物理的には直接接続できないノード間での通信の実現などが積極的に研究されている [36, 33].

2.3 分散共有メモリ

2.3.1 利点と欠点

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，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技術である．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 操作を発行することで，必要なノード間通信が実現される．すなわち，分散共有メモリは，データの物理的な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等の記述が可能なため，プログラマは，各ノード上のデータ配置や煩雑なメッセージ通信を意識することなく並列アルゴリズムの開発に専念できる [7]．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環境に移植す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，ユーザプログラム側に見えるのは全ノードにとって共通の仮想共有メモリである．そのため，ユーザプログラム側でのデータの所在管理が不要であり，系内に誰が存在しているかに関する情報がユーザプログラムにとって必要ない．そのため，ユーザプログラム側でノードの参加/脱退イベントをハンドリングするコーディングが (アプリケーション的には何らか必要な場合はあるが) 本質的には不要であり，ノードの動的な参加/脱退を容易に記述できる [37]．第三の利点として，応用範囲の広さがある．たとえば，ネットワーク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，分散共有メモリが並列分散プログラミングフレームワークのミドルウェア基盤としての強力さと可能性を兼ね備えていることを示唆している．

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．分散共有メモリで構築される仮想共有メモリの実体は各ノードが提供するローカルメモリの集合であり，あるノードで発行された read/write 操作は，処理系が，操作対象のアドレスを管理するノードとのメッセージ通信を暗黙的に行って，データをコンシステントに更新することで実現されている．すなわち，分散共有メモリは，実際に発生する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る．その意味で，分散共有メモリはメッセージパッシングよりも抽象度が高いプログラミングモデルであると言え，それゆえ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]．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）が一対一に対応するわけではないために，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こと，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こと，集合通信の最適化が行いにくいことなどが，性能劣化の要因として挙げられる．

2.3.2 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，(1) コンシステンシモデル，(2) コンシステンシプロトコルのデザイン，(3) 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 [21], Eager Release Consistency の Munin [18], Lazy Release Consistency の TreadMarks, Entry Consistency の Midway [31]

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 [43]．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, 34]．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 [28, 9]．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イシューが存在し，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設計項目がある：

共有データ更新時の挙動：データ更新時に他ノード上のキャッシュを無効化する invalidate 型プロトコルと，更新データを送りつけることで他ノードのキャッシュを常に最新状態に保つ update 型プロトコルが存在する．一般的なアプリケーションでは write/read の比率がそれほど高くないため，ほとんどのシステムでは invalidate 型プロトコルが採用されている．

共有データへの読み書き：Single Writer/Single Reader 型，Single Writer/Multiple Reader 型，Multiple Writer/Multiple Reader 型の分類がある．複数ノードによる読み書き操作のコンカレンシを確保する上では 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．

オーナー管理：アクセスフォルトが発生した場合，その共有データの管理権限を有するノード（以下，オーナーと呼ぶ）に対してリクエストを送ることになるが，このオーナーの管理技法に関しては，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，オーナーノードを固定する．ホーム問い合わせ型では，オーナーノードを動的に変化させるが，オーナーノードの位置を常に把握するホーム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ノードに問い合わせることでオーナーノードの位置を解決する．オーナー追跡型 [22] では，オーナーノード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 は真のオーナーノード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ノード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する．そして，アクセスフォルト時のリクエストなどは，このオーナー追跡グラフに沿って真のオーナーへとフォワーディングさせる．以上のオーナー管理技法に関しては，各分散共有メモリ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 な手法などが代表的である：

page-based：IVY，TreadMarks，DSM-Threads [27, 28, 32]，SMS [43] などで採用されている手法である．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固定されてい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 [34, 26]．

object-based：ObFT-DSM [26, 25] などで採用されている手法である．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 [34, 26]．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 [7]．

region-based：CRL [24]，HIVE [9]，研究 [23] などで採用されている手法である．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，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．

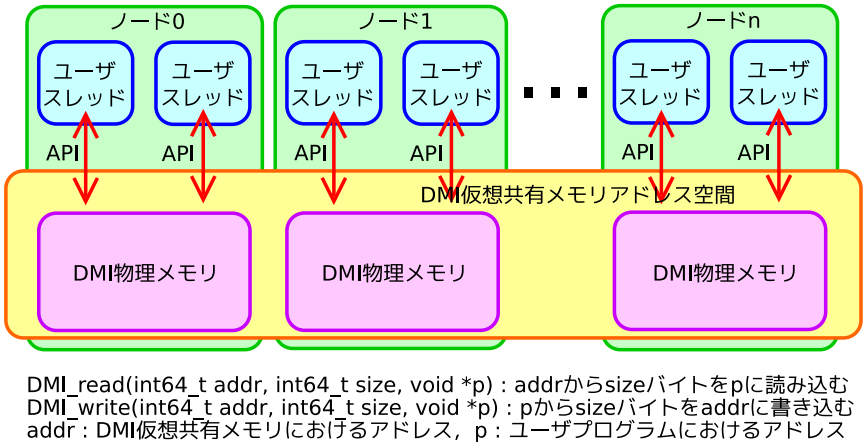

図 1 大規模分散共有メモリの構成.

3 コンセプト

本節では，DMI の設計コンセプトについて述べる．

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造を図 1 に示す．DMI 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 のメモリ管理機構をユーザレベルでシミュレートすることによって，分散環境上に DMI 仮想共有メモリを構築する．DMI 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る．また，これらページテーブルやアドレス空間記述テーブルなどの DMI 仮想共有メモリを管理するためのリソースはスレッドセーフであり，各ノード上には複数のユーザスレッドを生成することができる．

DMI では，ユーザプログラム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いる．そして，ユーザプログラムからは API 呼び出しを通じて，DMI 仮想共有メモリへのメモリ確保/解放や read/write などの各種メモリ操作を発行することができる．たとえば，ユーザプログラムが `DMI_read()` を発行した場合には，処理系はまず，該当アドレスに対応するページがそのノード上の DMI 物理メモリに存在するか否かを検査する．存在しない場合にはコンシステンシプロトコルによってオーナーノードにリードフォルトを通知し，やがてオーナーから最新ページを受け取り，それを DMI 物理メモリに格納する．その後，該当アドレスの領域を 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にコピーすることで，`DMI_read()` の処理が完了する．

DMI では，任意のユーザスレッドは全ノードが提供する DMI 物理メモリを利用可能であり，したがって DMI は大規模分散共有メモリとしての機能も兼ね備えている．また，ネットワークページングを繰り返すうちに DMI 物理メモリが飽和する場合があるが，その場合にはページ置換アルゴリズムが働いて適宜ページアウトが行われる．DMI では同一ノード内の複数のユーザスレッドが DMI 物理メモリを共有する構造を取っているため，各 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．

さらに，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 コンシステンシ管理

DMI のコンシステンシ管理では，region-based に似たアプローチを採用しており，ユーザプログラムは任意のページサイズを持つ DMI 仮想メモリを確保することができる．すなわち，DMI では OS のメモリ保護違反機構に頼ることなくユーザレベルでメモリ管理機構を実装しているため，OS のページサイズとは無関係に，アプリケーションの振る舞いに合致した任意のページサイズの仮想メモリを作成できる．たとえば，ブロック分割による行列の並列計算を行う場合には，各ブロックのサイズをページサイズに指定して行列用の仮想メモリを割り当てることなどが可能である．したがって，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，DMI では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．

DMI では，コンシステンシモデルとして Sequential Consistency を採用している．正確には，複数ページにまたがって `DMI_read()`/`DMI_write()` が呼び出された場合には，その呼び出しがページ単位の “小 `DMI_read()`”/“小 `DMI_write()`” に分割され，それらに関して Sequential Consistency を保証している．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ているのは，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したためである．

3.3 ミドルウェア基盤としてのインタフェース設計

DMI は，ユーザエンドなアプリケーションを直接記述するというよりも，並列分散プログラミングフレームワークを開発するためのミドルウェア基盤としての性格を狙っている．よって，DMI のインタフェース設計では，ユーザプログラム記述の作業的な容易さよりも，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柔軟性や汎用性を重視する方が望ましい．そこで DMI では，従来の多くの分散共有メモリが採用するような，ローカルメモリへのアクセスと同様に配列インデックスによって仮想共有メモリにアクセスできるようなインタフェースではなく，API 呼び出しによって仮想共有メモリにアクセスするインタフェースを採用している．

API 呼び出し型のインタフェースの利点は，まず第一に，同一の操作に対してさまざまなモードを設けることができるため，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を支援できる点である．たとえば，DMI では，`DMI_read()`/`DMI_write()` を含めたほぼ全ての関数に対して同期モードと非同期モードの 2 種類の関数を設けている．これにより，初期的な開発段階では同期モードを利用してプログラムを簡易に記述できる一方で，チューニングを行う段階では非同期モードの `DMI_read()`/`DMI_write()` 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させたり，高度なプリフェッチを実現することができ，プログラムのパフォーマンスをさらに引き出すことができる．また，現状では `DMI_read()` がページフォルトを引き起こした場合には，オーナーから転送されてきた最新ページを自ノード内に read-only な状態でキャッシュする実装になっているが，必要であれば，現状の `DMI_read()` に加えて，キャッシュしないモードの `DMI_read()` も設けることができる．このように，API 呼び出し型のアプローチを採用することにより，処理系としての高い機能拡張性を実現している．第二の利点は，API 呼び出し型とすることにより，OS のメモリ保護違反機構に頼らないページフォルトのハンドリングが可能となるため，OS のアドレッシング範囲に囚われないネットワークページングが実現できる点である．これにより，従来のネットワークページングでは 64 ビット OS が前提とされていたのに対して，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もできる．第三の利点としては，DMI で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，両者が API でしか結ばれていないため，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．

一方で欠点としては，まず第一に，API 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って，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しいわけではない．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挙げられる．

3.4 ノードの動的な参加/脱退のサポート

ノードの動的な参加/脱退をサポートするためには，(1) コンシステンシプロトコルがノードの参加/脱退に対応していることと，(2) 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できるようなプログラミングスタイルや実行形態が整備されていることが必須である．

第一に，ノードの動的な参加/脱退に対応可能なコンシステンシプロトコルについて考える．まず要件として，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することができない．また，ノードの脱退を実現するには，脱退前に，そのノードが DMI 物理メモリ内に保有しているページを他ノードに対して追い出す必要があるため，ページの追い出しプロトコルが定義されている必要がある．したがって，ノードの動的な参加/脱退に対応するためには，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．ページフォルトに関しては，**2.3.2** で述べたように，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 [42, 15, 19, 12]，ノードの動的な参加/脱退には対応できない．そこで DMI では，Sequential Consistency・Single Writer/Multiple Reader 型・オーナー追跡型のコンシステンシプロトコルとして代表的な Dynamic Distributed Manager Algorithm [22]（以下，DDMA）をベースとして拡張し，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コンシステンシプロトコルを作成する．

第二に，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ためのアプローチとして，プログラミングスタイル，API，実行形態について考える．まず，プログラミングスタイルに関しては，従来の多くのメッセージパッシングや分散共有メモリが採用している SPMD 型ではなく，動的なスレッドの生成/破棄を記述できる pthread 型のスタイルを採用する．SPMD 型の利点は，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点などである．しかし，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やや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/脱退を行うには，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には SPMD 型は適していない．これに対して，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るとともに，ユーザプログラムの意志で参加ノード上へのスレッド生成/破棄ができるため，ノードの動的な参加/脱退に応じた動的な並列度変化をプログラムとして記述可能である．よって，DMI

では pthread 型を採用している．次に，API に関しては，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にするため，たとえば，ノードの参加/脱退イベントをポーリングする関数，現在参加中の全ノードの情報を取得する関数，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 を整備している．最後に，プログラムの実行形態に関しては，すでに参加中の任意のノードを 1 個指定してプログラムを実行することで参加が完了し，外部からのシグナル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 [2] 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．

3.5 マルチコア並列プログラムとの類似性

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ことを目指し，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 pthread 型のスタイルを採用しているのは言うまでもないが，同期操作のインタフェースに関しても既存の分散共有メモリとはやや異なるアプローチを採用している．

従来の多くの分散共有メモリにおける同期のインタフェースは，同期用変数（排他制御変数，条件変数など）の初期化関数を呼び出すとその同期用変数の id（整数値）が処理系から与えられ，その id を同期操作用の関数（ロック操作，待機操作など）に渡すことで同期が実現されるようになっている．この id は，仮想共有メモリアドレス空間とは別の “仮想 id アドレス空間” におけるアドレスと捉えることができる．これに対して，pthread における同期のインタフェースは，同期用変数のアドレスを指定して初期化関数を呼び出した後，そのアドレスを同期操作用の関数に渡すことで同期が実現されるようになっている．すなわち，pthread では「ユーザプログラムが与えた共有メモリアドレスの上で」同期が実現されるのに対して，従来の多くの分散共有メモリでは「初期化時に処理系から与えられる，仮想共有メモリアドレス空間とは別の “仮想 id アドレス空間” のアドレスを使って」同期が実現される．この相違は一見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．特に，pthread プログラムから DMI のプログラムへのトランスレータなどを将来的に検討する際には，この相違は核心的な問題になると予想される．以上を踏まえて，DMI では，pthread プログラムとの対応性を重視し，「ユーザプログラムが与えた DMI 仮想共有メモリアドレスの上で」同期を実現するようなインタフェースを整える．

3.6 データ転送の動的負荷分散

分散共有メモリがメッセージパッシングと比較して性能が出にくい主な要因として，分散共有メモリでは集合通信が行いにくい点が指摘されており [13]，DMI の初期性能評価でも同様の考察が得られている．この事情を受け，SPMD 型のスタイルを採用する UPC では，分散共有メモリでありながらも集合通信を導入して最適化する試みが成されている [8, 11]．しかし，分散共有メモリの本質はユーザプログラムに対して通信を隠蔽する部分にあるため，純粋な分散共有メモリを目指す DMI に集合通信を導入するのは好ましくない．また，ノードの参加/脱退をサポートする DMI では，動的なスレッド生成/破棄が可能な pthread 型のスタイルを採用することで，ユーザプログラムに対して “全員” の概念を隠蔽することが鍵となっており，集合通信を導入するというアイディアはこれに矛盾してしまう．

そこで DMI では，データ転送の最適化手法としてページ転送の動的な負荷分散を提案する．具体例として，図 2(A) のように，メッセージパッシングにおける broadcast に相当する通信が分散共有メモリ上で発生する場合を考える．このとき，多数のノードがほぼ同時にリード要求を発行することになるが，これら全てのリード要求をオーナーが逐次的に処理してページ転送を行う方法ではオーナーが著しいボトルネックになる（図 2(B)）．特に DMI ではページサイズが任意であるため，1 回のページ転送が巨大化する可能性があり，その影響はより顕著になると予想される．そこで DMI では，図 2(C) のように，オーナーが状況に応じてページ転送の経路を動的に決定することで全体としてのページ転送を木構造化させる（以下，この木構造をページ転送グラフと呼ぶ）．具体的には，オーナーにリード要求が到着した際に，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委譲することによって，ページ転送の負荷分散を実現する．

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 3 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

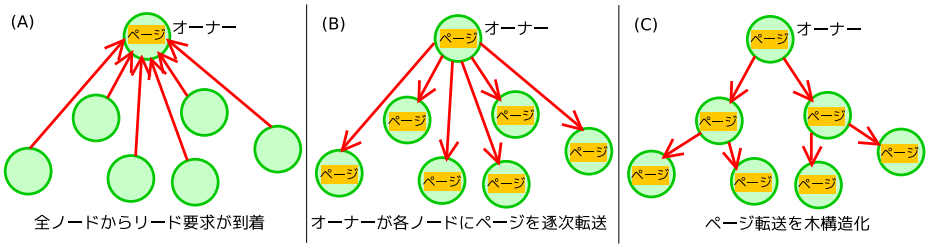

図 2 ページ転送の動的負荷分散（(A)broadcast に相当する通信が発生した場合，(B) ページを逐次転送する場合，(C) ページを木構造転送する場合）.

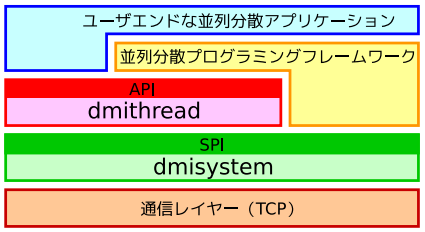

図 3 並列分散プログラミングフレームワークにおける DMI の位置付け.

```
#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 4 ノードの参加/脱退に対応したプログラムの記述例．API の詳細は付録 B を参照.

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．SPI は原則として非同期モードの関数である．一方，dmithread は SPI の上に実装される処理系であり，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．これにより，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 の全容については付録 B を参照されたい．また，ノードの参加/脱退に対応したプログラムの記述例を図 4 に示す．

5 実装

本節では dmisystem の実装について，コンシステンシプロトコルやノードの参加/脱退処理などについて詳述する．DMI の処理系は C 言語で 13000 行程度である．

5.1 ノードの構成要素

DMI における各ノードは，recver スレッド，handler スレッド，sweeper スレッド，ユーザスレッドから構成される．

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 スレッドは，メッセージキューからメッセージを取り出し，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，全てのメッセージはシリアライズされて処理される．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 スレッドは，常に DMI 物理メモリの使用状況を監視しており，DMI 物理メモリの使用量が指定量を超過した場合にページの追い出し処理を行う．ユーザスレッドは，ユーザプログラムの記述に従って生成されるスレッドであり，同一ノード上に任意個生成可能である．

5.2 コンシステンシプロトコル

5.2.1 データ構造

3.4 で述べたように, DMI では, ページの追い出しが可能なように DDMA を修正および拡張した, Single Writer/Multiple Reader 型・オーナー追跡型のプロトコルを採用する．

各ノードは各ページに対して，*owner*，*probable*，*status*，*treeset* の 4 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：

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．

status：ノード i が有効なページを持っていれば $i.status = valid$，持っていなければ $i.status = invalid$ である．

treeset：*treeset* はオーナーによって管理されるデータであり，そのページをキャッシュしているノードの集合，つまり $i.status = valid$ となっているノード i の集合を管理する．DMI ではページ転送グラフを用いた木構造転送を行うため，*treeset* にはページ転送グラフの構造情報も記憶されている．

したがって，DMI では各ノードからオーナーへと向かう上流方向のグラフと，オーナーから各ノードへと向かう下流方向のグラフが存在する．前者がオーナー追跡グラフであり，その形状は各ノードの *probable* によって分散管理される．後者がページ転送グラフであり，その形状はオーナーの *treeset* によって一括管理される．

また，ノード i においてページの read が許可されるのは $i.status = valid$ な場合であり，ページの write が許可されるのは $i.owner = true$ かつ *treeset* に i しか含まれない場合である．それ以外の場合にはリードフォルトもしくはライトフォルトが発生し，オーナー追跡グラフに沿ってオーナーに要求が送信される．

5.2.2 アルゴリズム

(1) リードフォルト，(2) ライトフォルト，(3) ページ追い出しの 3 つの場合に関してプロトコルの概略を図解する．厳密なプロトコル記述は付録 A を参照されたい．

第一に，リードフォルトに関して述べる．ノード x でリードフォルトが発生した場合，リード要求はやがてオーナー v に到達するが，そのときの状況としては以下の 2 通りが考えられる：

(I)$x \notin v.treeset$ の場合（図 5(A1)）：

1. v は，ページ転送の動的負荷分散を考慮して，ページ転送グラフのどの位置に x を追加するか決定した上で，$v.treeset$ に x を追加する．
2. ページ転送グラフにおける x の親ノードを y とすると，v から y までリード応答のメッセージがフォワーディングされ，続い

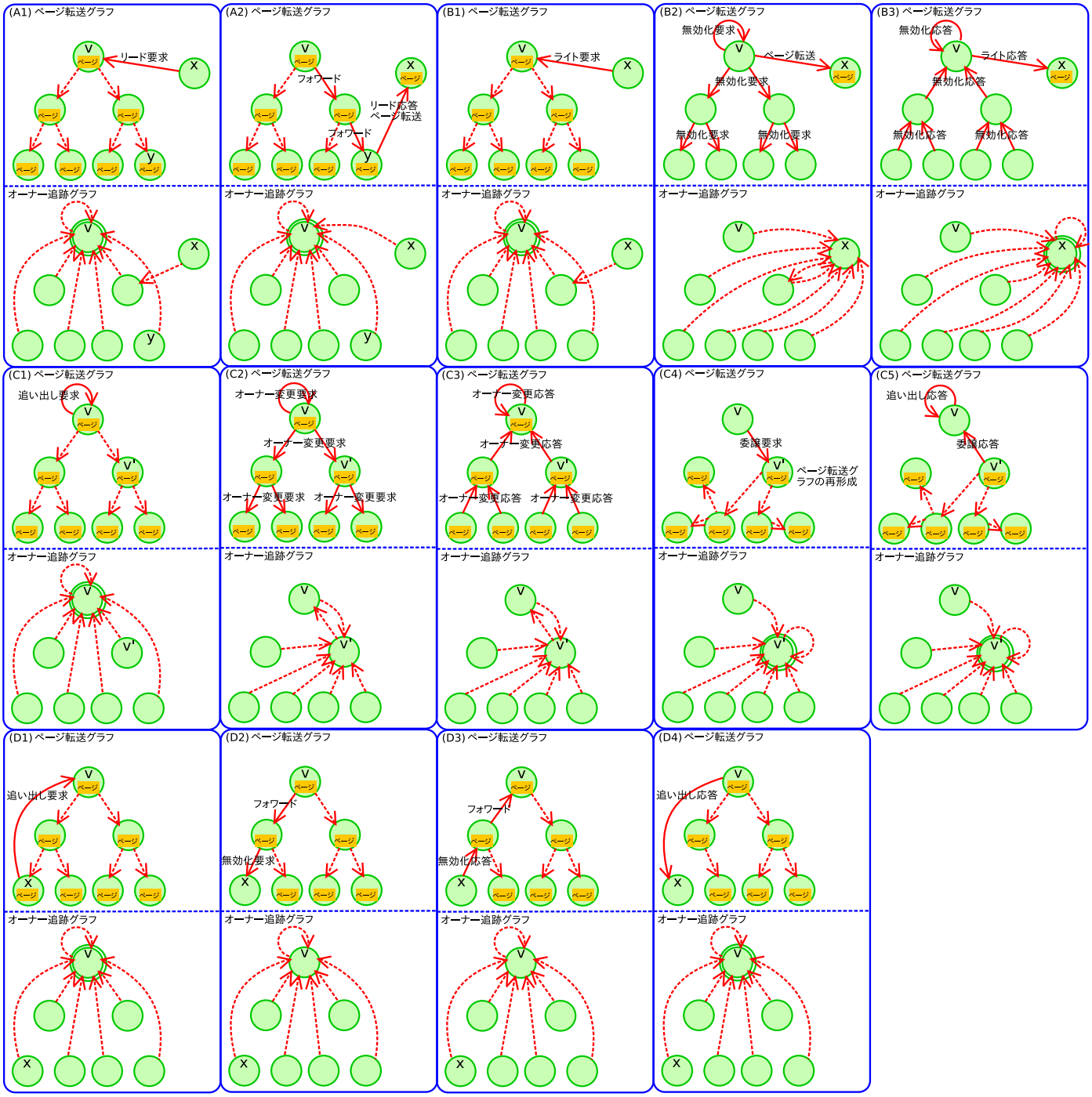

図 5　ページに関するコンシステンシプロトコル（(A1)(A2) リードフォルト，(B1)〜(B3) ライトフォルト，(C1)〜(C5)(D1)〜(D4) ページの追い出し）．各コマにおいて上段がページ転送グラフの状態，下段がオーナー追跡グラフの状態を示す．オーナー追跡グラフではオーナーを二重丸で囲む．グラフの形状を点線矢印で，実際に発生するメッセージ通信を実線矢印で示す．

て y が x へ，ページ転送処理を兼ねたリード応答を送信する（図 5(A2)）．

3. リード応答を受信した x は，$x.status$ を *valid* に，$x.probable$ を v に書き換え，リード処理を完了させる（図 5(A2)）．

(II)$x \in v.treeset$ の場合：

この状況は，過去に x で発行されたリード要求が先に処理されたために，いま着目しているリード要求が v に到達した時点では，すでに x 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．DMI では各ノード上に複数のユーザスレッドが生成されるため，このような状況は十分起こりうる．この場合には，v はページ転送グラフに沿って v から x までリード応答をフォワーディングする．そして，それを受信した x はリード処理を完了させる．

第二に，ライトフォルトに関して述べる．ノード x でライトフォルトが発生した場合，ライト要求はやがてオーナー v に到達し（図 5(B1)），以下のプロトコルが走る：

1. $v.owner$ を *false* に書き換える（図 5(B2)）．この操作により系内から一時的にオーナーが消え，この時点でオーナー追跡グラフを辿っているメッセージは，オーナーが確定するまでオーナー追跡グラフ上をフォワーディングされ続ける．
2. $x \notin v.treeset$ であれば x に最新ページを送信する．最新ページを受信した x は $x.status$ を *valid* に書き換える（図 5(B2)）．

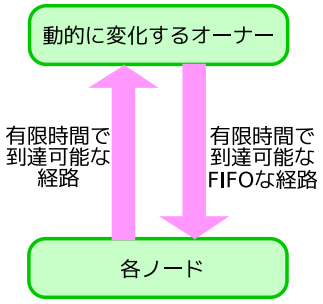

図 6　コンシステンシが維持されるための十分条件.

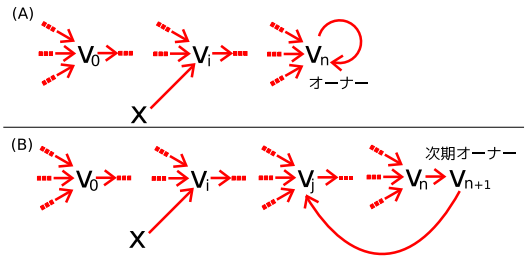

図 7　オーナー追跡グラフの形状 ((A) オーナーが存在する場合, (B) オーナーは存在しないが次期オーナーが存在する場合).

3. v はページ転送グラフ全体に沿って無効化要求をマルチキャストする（図 5(B2)）．この過程で，無効化要求を受信した各ノード $i(\neq x)$ は，$i.status$ を $invalid$ に，$i.probable$ を x に書き換えた上で（図 5(B2)），無効化要求の送信元に対して無効化応答を送信する．オーナーは全ての無効化応答を回収する（図 5(B3)）．
4. v は x にライト応答を送信する．ライト応答を受信した x は，$x.owner$ を $true$ に，$x.probable$ を x に，$x.treeset$ を初期化した上で x を追加し，新たなオーナーとなる（図 5(B3)）．

第三に，追い出しに関して述べる．ノード x がページを追い出す場合，追い出し要求はやがてオーナー v に到達するが，そのときの状況としては以下の 3 通りがある：

(I)$x = v$ の場合（図 5(C1)）：

1. v は $v.owner$ を $false$ に書き換え，系内から一時的にオーナーを消す（図 5(C2)）．
2. 追い出し先のノード v' を選択する．v 以外のノードが $v.treeset$ に含まれる場合には $v.treeset$ の中から v' を選択する．そうでなければ適当に v' を選択し，v' に対して最新ページの送信を行う．最新ページを受信した v' は $v'.status$ を $valid$ に書き換える．
3. v はページ転送グラフに沿ってオーナー変更要求をマルチキャストする（図 5(C2)）．この過程で，オーナー変更要求を受信した各ノード $i(\neq v')$ は $i.probable$ を v' に書き換えた上で，オーナー変更要求の送信元に対してオーナー変更応答を送信する．オーナーは全てのオーナー変更応答を回収する（図 5(C3)）．
4. v は v' に委譲要求を送信する．このとき，v を除く $v.treeset$ の内容も送信する（図 5(C4)）．
5. 委譲要求を受信した v' は，$v'.owner$ を $true$ に，$v'.probable$ を v' に書き換え，新たなオーナーとなる．また，受信した $treeset$ の内容を $v'.treeset$ にコピーし，ページ転送グラフを再形成する（図 5(C4)）．
6. v' は v に対して委譲応答を送信し，それを受信した v は $x(=v)$ に対して追い出し応答を送信する（図 5(C5)）．

(II)$x \neq v$ かつ $x \in v.treeset$ の場合（図 5(D1)）：

1. v は $v.owner$ を $false$ に書き換え，系内から一時的にオーナーを消す（図 5(D2)）．
2. v はページ転送グラフに沿って v から x まで無効化要求をフォワーディングして送信する（図 5(D2)）．
3. 無効化要求を受信した x は，$x.status$ を $invalid$ に書き換え，v に無効化応答を送信する（図 5(D3)）．
4. v は $v.owner$ を再び $true$ に戻し，x に対して追い出し応答を送信する（図 5(D4)）．

(III)$x \neq v$ かつ $x \notin v.treeset$ の場合：

この状況は，x からの追い出し要求が v に到達する前に，他ノードからのライト要求が v に到達し，その時点ですでに x に対して無効化要求が発行されている場合などに生じる．この場合には，v は x に対して追い出し応答を送信するだけでよい．

5.2.3　正しさの証明

コンシステンシを維持するためには，単独のオーナーが全 read/write 操作をシリアライズした上で，そのシリアライズされた順序通りに実際の read/write 操作が発生すればよい．したがって，オーナーが動的に変化する DMI においては，

(I) 各ノードからオーナーへ有限時間内で到達可能な経路が存在する

(II) オーナーから各ノードへ有限時間内で到達可能な経路が存在し，その経路は FIFO である

という 2 条件が満足されるようにプロトコルを設計すれば十分である（図 6）．以下では，**5.2.2** のプロトコルが (I)(II) を満たすことの簡略な証明を記す．

まず (I) を示す．オーナーが確定しない時刻において，次にオーナーが確定した時点でオーナーになるノードを次期オーナーと呼ぶことにすれば，系の状態は任意の時刻において，「オーナーが 1 個だけ存在する状態」もしくは「オーナーは存在しないが次期オーナーが 1 個だけ存在する状態」のいずれかである．また，各ノード間の通信時間が有限時間で抑えられると仮定すれば，プロトコルの性質により，「オーナーは存在しないが次期オーナーが 1 個だけ存在する状態」は有限時間後には必ず「オーナーが 1 個だけ存在する状態」に遷移する．

ここで，任意のノード x が $x' = x.probable$ へとメッセージを送信し，そのメッセージが x' に受理された時点における x' の状況を考えると，以下の 3 通りに限られることがわかる：

(a)x' がオーナーである場合

(b)x' はオーナーではないが，系の状態が「オーナーは存在しないが次期オーナーが 1 個だけ存在する状態」にあり，しかも次期オーナーが x' である場合

(c)x' はオーナーでも次期オーナーでもないが，x' がオーナーであった時刻が過去に存在する場合

(c) の場合をさらに考える．プログラム実行開始からの歴代オーナー系列を $v_0 \to \cdots \to v_{i-1} \to v_i(=x') \to v_{i+1} \to \cdots (\forall j > i : v_j \neq x')$ とすると，プロトコルの性質上，$v_i.probable(=x'.probable)$ は $v_i, v_{i+1}, \cdots$ もしくは次期オーナーのいずれかの値になっている．ここで，v_i がオーナーでないにも関わらず $v_i.probable = v_i$ であるのは「オーナーは存在しないが次期オーナーが 1 個だけ存在する状態」に限り，これが有限時間しか継続しないことに注意すると，$v_i.probable$ は有限時間後には必ず $v_{i+1}, v_{i+2}, \cdots$ のいずれかの値を取る．つまり，$v_i.probable = v_i$ の場合には，メッセージが自分自身へとフォワーディングされ続けることで，やがては歴代オーナー系列をより新しい方向へと辿ることができる．

以上の事実をまとめると，オーナー追跡グラフの形状は任意の時刻において，任意のノード x と歴代オーナー系列 $v_0 \to \cdots \to v_n$ に対して，系の状態が「オーナーが 1 個だけ存在する状態」であれば図 7(A),「オーナーは存在しないが次期オーナーが 1 個だけ存在する状態」であれば図 7(B) のようになることが導ける．したがって，任意のノード x で発行されたメッセージは，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で，オーナーに到達するか，もしくは次期オーナーを含むサイクルに突入する．そして次期オーナーは有限時間内にはオーナーへと確定するため，オーナーの移動速度がメッセージのフォワーディング速度よりも遅いという仮定の下では，サイクルを回る過程でメッセージは必ずオーナーに受理される．

次に (Ⅱ) を示す．オーナーから各ノードへ有限時間内で到達可能な経路が存在することは自明なので，この経路が FIFO 性を満たすことを示す．まずノード v がオーナーである期間を考えると，v から任意のノードに対するメッセージは，ページ転送グラフに沿った同一経路を通って送信されるため，オーナーから任意のノードに対するメッセージは FIFO 性を満たしている．また，オーナーが v から v' に移動する際には，図 5(B2)(B3)(C2)(C3)(D2)(D3) に示すように，オーナー v は，v がオーナーであった時代に v が発行した全メッセージの送信先に対して，ページ転送グラフに沿ってメッセージを送信し，その応答を全て回収するという作業を行った後で，v' にオーナー権を委譲している．すなわち，この応答の回収作業を通じて，v がオーナーであった時代に v が発行したメッセージが全て受理されたことを保証した後で，オーナーを移動している．したがって，オーナーの移動前後においても，オーナーから任意のノードに対するメッセージの FIFO 性は崩れない．

5.3 同期プロトコル

5.3.1 データ構造

DMI では同期操作のために排他制御変数と条件変数を設けており，ともにトークンベースの手法 [30, 29] によって実装している．

各ノードは各トークンに対して，*owner*，*probable*，*locked*，*queue* の 4 つのデータを管理する．以降では，ある特定のトークン t に関するプロトコルを考えることとし，ノード i のトークン t に関する *owner* を $i.owner$ などと表記する：

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．

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．ページの場合と同様，全ノードを通じた *probable* はオーナー追跡グラフを形成する．

locked：オーナーのみが管理するデータで，排他制御変数がロック中ならば $locked = true$，そうでなければ $locked = false$ である．条件変数用のトークンの場合にはこのデータは使用されない．

queue：オーナーのみが管理するデータで，保留中のメッセージがキューイングされる．

5.3.2 アルゴリズム

ノード i が mutex_lock()→cond_wait()→mutex_unlock()，ノード j が mutex_lock()→cond_signal()→mutex_unlock() を実行する場合（図 8(A1)）を具体例に取り，プロトコルの概要を図解する．簡単のため，排他制御変数用トークンのオーナーはノード m，条件変数用トークンのオーナーはノード c に固定であると仮定し，オーナー追跡グラフに関する記述は省略する：

1. i が mutex_lock() を呼ぶと lock 要求 r_1 が発行され，やがて m に受理される．m は $m.locked$ を $true$ に書き換え，i に対して lock 応答を送信する．結果，i の mutex_lock() 呼び出しが返る（図 8(A2)）．
2. ここで j が mutex_lock() を呼ぶと lock 要求 r_2 が m に受理されるが，$m.locked = true$ であるため，r_2 は $m.queue$ に保留される（図 8(A3)）．
3. 続いて i が cond_wait() を呼ぶと，wait 要求 r_3 が c に送信され，c は $c.queue$ に r_3 を保留した上で i に wait 応答を送信する（図 8(A4)）．
4. wait 応答を受信した i は，m に対して unlock 要求 r_4 を送信する．r_4 を受理した m は，i に対して unlock 応答を送信するとともに，保留中の lock 要求が存在するか調べ，存在する場合には適当な 1 個を選んで lock 応答を送信する．今の場合，保留中の r_2 を取り出して r_2 の送信元である j に対して lock 応答を送信する．結果，j の mutex_lock() 呼び出しが返る（図 8(A5)）．
5. j が cond_signal() を呼ぶと，signal 要求 r_5 が c に送信され，r_5 を受信した c は，保留中の wait 要求の中から適当な 1 個を（存在すれば）取り出す．なお，j の呼び出しが cond_signal() ではなく cond_broadcast() である場合には，保留中の wait 要求全てが取り出される．今の場合は r_3 が取り出され，r_3 の送信元である i に対して wake 要求 r_6 が送信されるとともに，j に対しては signal 応答が送信される．結果，j の cond_signal() 呼び出しが返る（図 8(A6)）．

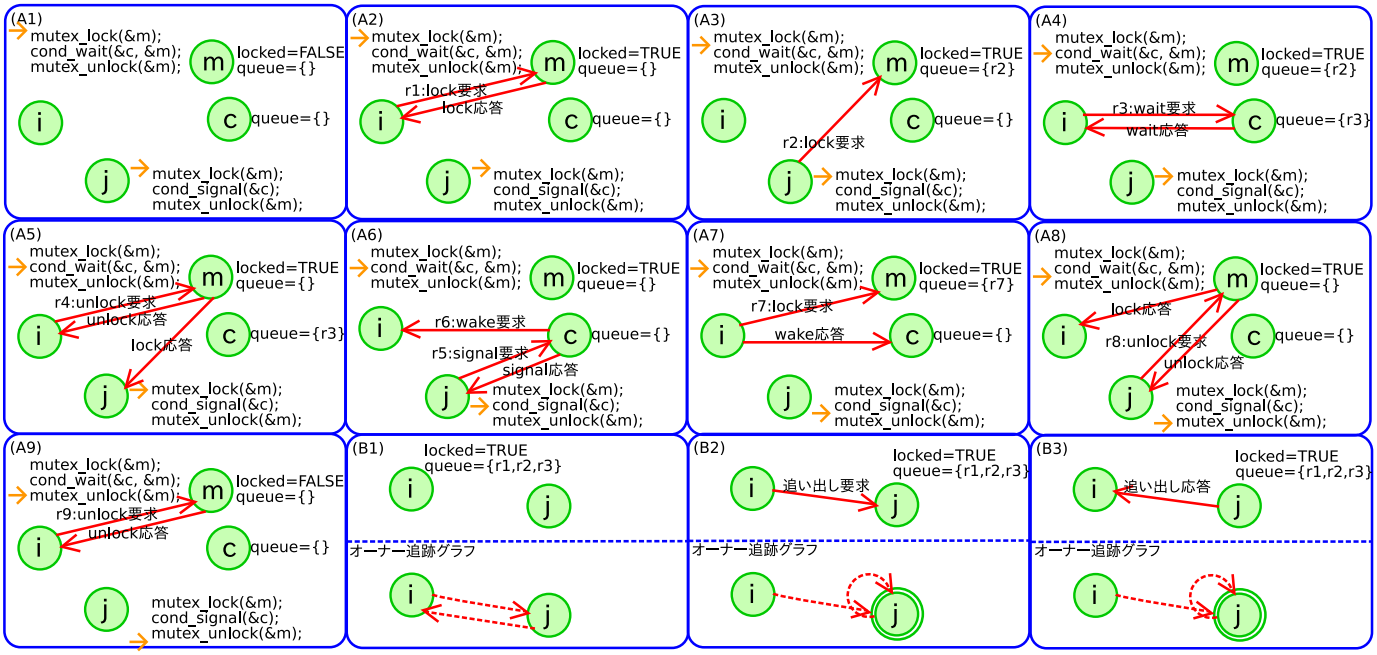

図 8　トークンに関するコンシステンシプロトコル（(A1)〜(A9) トークンによる同期操作，(B1)〜(B3) トークンの追い出し）．

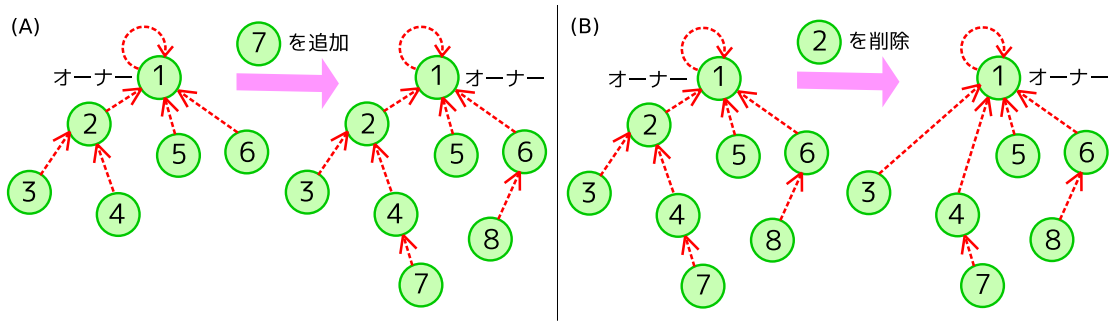

図 9　ノードの参加/脱退時におけるオーナー追跡グラフの再形成（(A) ノードの参加，(B) ノードの脱退）．

6. r_6 を受信した i は，c に対して wake 応答を送信するとともに，m に対して lock 要求 r_7 を送信するが，$m.locked = true$ であるため r_7 は $m.queue$ に保留される（図 8(A7)）．
7. やがて j が mutex_unlock() を呼び出すと，unlock 要求 r_8 が m に送信され，m は j に対して unlock 応答を送信するとともに，保留中の r_7 を取り出して i に lock 応答を送信する．結果，i の cond_wait() 呼び出しと j の mutex_unlock() 呼び出しが返る（図 8(A8)）．
8. 最後に i が mutex_unlock() を呼び出すと，unlock 要求 r_9 が m に送信され，m は i に対して unlock 応答を送信する．この場合には $m.queue$ は空なので，m は単に $m.locked$ を $false$ に書き換える．結果，i の mutex_unlock() 呼び出しが返る（図 8(A9)）．

オーナーが固定であれば上記の方法で十分であるが，ノードの脱退に対応するため，また特定ノードへの負荷集中を避けるためには，オーナーを追い出すプロトコルが必要である．ノード i がオーナーを追い出す場合のプロトコルは以下の通りである：

1. i は，追い出し先のノード j を選択し，$i.owner$ を $false$ に，$i.probable$ を j に書き換える（図 8(B1)）．
2. i は，j に追い出し要求を送信し，$i.locked$ および $i.queue$ を j にコピーする．j は $j.owner$ を $true$ に，$j.probable$ を j に書き換え，新しいオーナーとなる（図 8(B2)）．
3. j は i に対して追い出し応答を送信する（図 8(B3)）．

5.3.3　正しさの証明

以上のトークンのプロトコルは，ページのコンシステンシプロトコルにおいて $treeset$ の要素数を常に 1 に制限した場合（ページのキャッシュを許さない場合）と同等であるため，プロトコルの正しさは **5.2.3** ですでに示されている．

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，メモリの確保/解放，トークンの確保/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．

ノードの参加は，実行中のノード 1 個をユーザプログラム側で指定することによって実現される．ノード x がノード y 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. x が y に参加要求を送信する．参加要求を受理した y はグローバルロックを取得する．
2. y は x に，全ノード情報，全ページと全トークンに対する *y.probable* を送信する．
3. x は，全ページと全トークンを割り当て，全ページに関して *x.owner* を *false* に，*x.probable* を *y.probable* に，*x.status* を *invalid* に初期化する．また全トークンに関して *x.owner* を *false* に，*x.probable* を *y.probable* に初期化する．なお，ここでは *x.probable* を *y.probable* に初期化しているが，プロトコル上は，*x.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 9(A)）．
4. x 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．
5. x はグローバルロックを解放する．

一方，ノード x が脱退する手順は以下の通りである：

1. x はグローバルロックを取得する．
2. x は全ページと全トークンの追い出しを行う．
3. x は全ノードとネゴシエーションを行って終了処理を行う．このとき x は全ノードに対して，全ページと全トークンに関する *x.probable* を送信する．そして各ノード i は，$i.probable = x$ であるようなページまたはトークン全てに関して，*i.probable* の値を *x.probable* に書き換える．この作業により，全ページおよび全トークンのオーナー追跡グラフが，x を含まないオーナー追跡グラフへと再形成される（図 9(B)）．
4. x は全ノードとのコネクション接続を切断した上で，実行中の適当なノード y に脱退要求を送信し，y にグローバルロックを解放させる．

5.5 トランザクション管理

ノード x の脱退処理中に，他ノードが x に対してページを追い出してきたり，x に対してスレッド生成が行われたりすると都合が悪い．そのため DMI では，ページアクセス，ページの追い出し，同期操作，メモリ確保/解放，スレッド管理など，自分以外のノードに影響を及ぼす可能性のある全操作に対して厳格なトランザクションを定義して管理する．これにより，SPI と API の全関数に対して原子性を保証しており，すなわち，ユーザプログラムから呼び出された SPI や API は，操作実行前に失敗を判定してユーザプログラムにエラーを通知するか，もしくは必ず最後まで操作が完了するかのいずれかの結果になる．したがって，DMI では，プロトコルが中途半端に実行されてシステムが未定義状態に陥ることはなく，ユーザプログラム側でのエラーハンドリングが可能である．

5.6 ページ置換

DMI では，sweeper スレッドが DMI 物理メモリの使用状況を常に監視しており，DMI 物理メモリが飽和した場合には他ノードへのページ追い出し処理が実行される．

このページ置換においてまず重要になるのはどのページを追い出すかである．追い出し対象のページは $status = valid$ なページであるが，**5.2.2** で述べた追い出しプロトコルの負荷は，(1) オーナー権を伴わないページ，(2) オーナー権を伴うがこのページをキャッシュしているノードが自分以外に存在するページ，(3) オーナー権を伴いページの実体を保持しているノードが自分以外には存在しないページの順に高くなる．(3) の場合にはページ転送が必要になるが (1)(2) ではページ転送が不要であることに注意されたい．また，ページサイズの大きいページを追い出す方が効果が大きい．以上より，現状の DMI では，ページサイズの大きい順に，(3)→(2)→(1) の優先度順にページの追い出しを行っている．ページへの最終アクセス時刻なども考慮すべきであるが，現状ではまだ実装できていない．さらに，(3) の場合には追い出し先のノードの選択方法も鍵となる．なぜならば，(3) の場合にはページ転送を伴うため，すでに DMI 物理メモリが飽和状態に近いノードに対してページを追い出せば，ノード間でページスラッシングを起こしかねないからである．しかし，現実装では単純に乱数による選択を行っている．

なお，各ノードが提供する DMI 物理メモリ量の値は実行時に指定可能であるが，この値は 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ず，実際の DMI 物理メモリ使用量がその値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ページ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られる．

6 評価

本節では，マイクロベンチマークおよびアプリケーションベンチマークに対する性能評価の結果を示す．使用した実験環境は表 1 の通りである．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 1 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．なお，ページの動的負荷分散機能は現在実装段階にあるため，本節に載せるデータは図 2(B) に示すような逐次転送を行う実装に基づくものである．

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（ローカルリード）と，オーナーに対してリードフォルトを発行して最新ページの転送を要求する場合（リモートリード）の 2 種類がある．ローカルリードの処理の内訳は，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，コンシステンシ管理やトランザクション管理などの処理系のオーバーヘッドである．一方，リモートリードの処理の内訳は，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 10 には，データサイズを変化させたときの，ローカルリード，リモートリード，memcpy の処理時間を示す．図 10 より，リモートリードはローカルリードと比較して 7〜27 倍程度遅いことがわかる．また，データサイズが 1000B 以下では処理系のオーバーヘッドが原因でローカルリードは 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 11 にローカルリード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示す．また図 12 にリモートリード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 11 の結果より，ローカルリードでは 1000B 以下のデータサイズでは処理系のオーバーヘッドが 95% 以上を占めることが読み取れる．また図 12 の結果より，リモートリードでは 200B 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.

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，その時点でどの程度の DMI 仮想メモリが割り当てられているかや，脱退するノードが DMI 物理メモリ内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，(1) ページを何も割り当てない場合，(2) 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 通りに関して，全ノードが同時に参加/脱退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．結果を図 13 および図 14 に示す．まず，参加に関して図 13 を見ると，(1)(2) 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失敗する確率が高くなり，コネクション接続の再試行を行っていることが大きく影響している．次に，脱退に関して図 14 を見ると，(1) は脱退時に何もページを追い出す必要がないため 128 ノードの同時脱退が 1.76 秒で完了しているが，(2) 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，32MB の DMI 仮想メモリを確保し，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 15 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．なお，図 15 中の点線は，指定した DMI 物理メモリ量である 1MB のラインを表す．この結果を見ると，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，**5.6** 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，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

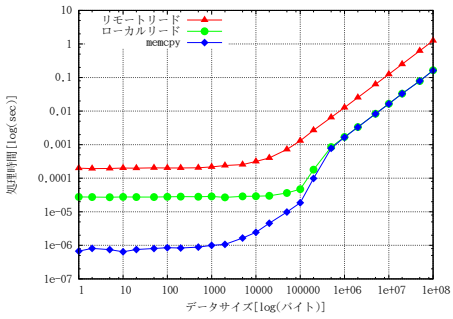

図 10 データサイズを変化させたときのローカルリード, リモートリード, memcpy の処理時間.

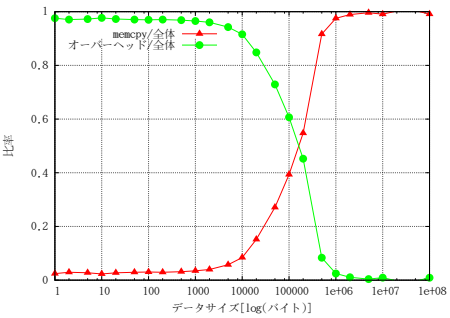

図 11 データサイズを変化させたときのローカルリードの内訳.

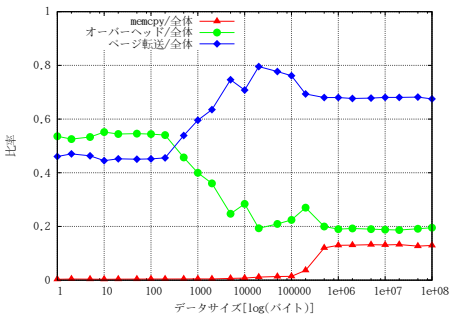

図 12 データサイズを変化させたときのリモートリードの内訳.

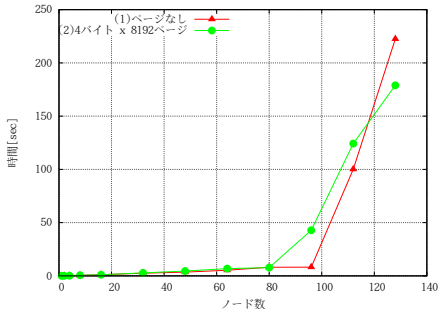

図 13 ノード数と, 全ノードを参加させるのに要する時間の関係.

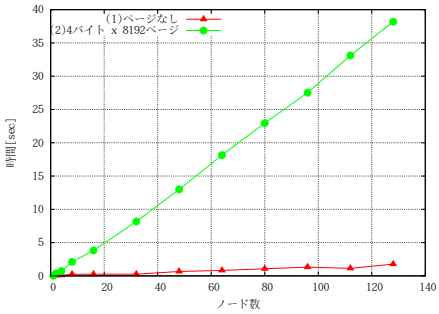

図 14 ノード数と, 全ノードを脱退させるのに要する時間の関係.

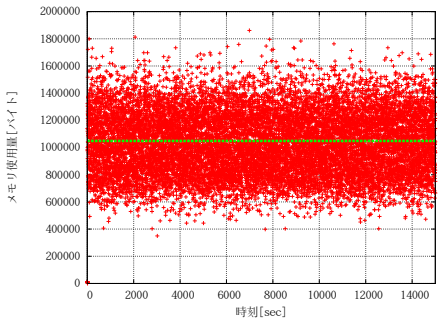

図 15 ページ置換アルゴリズムのもとでの DMI 物理メモリ使用量の時間的変化.

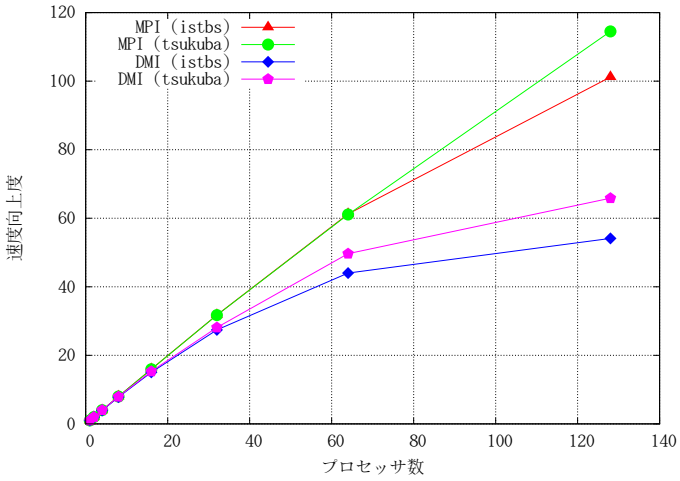

図 16 マンデルブロ集合の描画に関する DMI と MPI のスケーラビリティ.

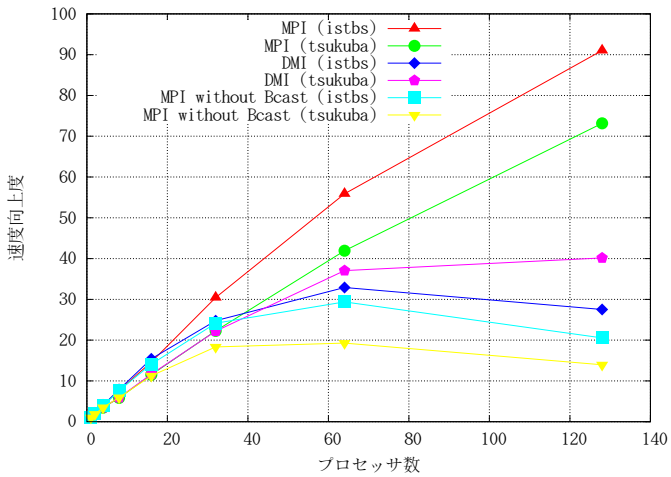

図 17 行列行列積に関する DMI と MPI のスケーラビリティ.

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 Embarassingly 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, 描画領域を縦に N 分割し, この N 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, その結果を描画する.

この実験では描画領域のサイズを 480×480, $N = 480$, 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 16 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 16 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．また，DMI における tsukuba 環境と istbs 環境の性能差は，tsukuba 環境では 1 ノードに 8 スレッドが割り当てられていてそれらが DMI 物理メモリを “共有キャッシュ” として利用可能なため，istbs 環境のように 1 ノードに 1 スレッドを割り当てる場合よりページフォルト回数が少なくなるためである．この結果は，DMI が分散レベルの並列性とともにマルチコアレベルの並列性を活用できていることを示している．

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 のサイズの行列による行列行列積 $AB = C$ の並列演算を行った．アルゴリズムは以下の通りである：

1. スレッド（MPI ではプロセス）0 が行列 A をスレッド数分だけ横ブロック分割し，それを全スレッドに scatter する．
2. スレッド 0 が行列 B を broadcast する．
3. 各スレッド i は自分の担当範囲のブロックに関して行列行列積 $A_iB = C_i$ を計算し，その結果 C_i をスレッド 0 に gather する．

なお，DMI では，各ブロックが 1 ページになるようページサイズを設定し，処理を通じて各ブロックあたり 1 回しかページフォルトが発生しないようにした．

図 17 に DMI と MPI のスケーラビリティを比較した結果を示す．図 17 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の他，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示している．図 17 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データ転送の最適化に起因していることがわかる．以上の結果より，DMI に対して **3.6** で述べたようなページ転送の動的負荷分散を実装することで，DMI の性能を大きく向上できる可能性があると言える．

7 関連研究

本節では，**3** 節で掲げた DMI のコンセプトに関連する技術を中心に取り上げる．

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 [35, 41] 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．また，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．しかし，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かなり複雑である．

研究 [34] 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分散共有メモリとしては，ObFT-DSM [26, 25] 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，耐故障性も考慮されており，故障に起因するノードの参加/脱退に対する対策が提案されている．しかし，DMI とは異なり，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．さらに，ObFT-DSM は WAN 上での複雑な事情を隠蔽するための並列分散ミドルウェア基盤を目標としているのに対して，DMI では，さしあたり均質なクラスタ環境を前提として，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インタフェースを提供できる並列分散ミドルウェア基盤を目標としている．

7.2 コンシステンシプロトコル

DMI では，ページの追い出しに対応できるように DDMA のコンシステンシプロトコルを拡張しているが，同様の拡張は研究 [19] でも行われている．しかし，この研究で提案されているプロトコルは，大規模分散共有メモリに対して DDMA のプロトコルを適用することを目的としており，各ページに対して固定的な帰属ノードを設けるアプローチを取っている．よって，ノードの動的な脱退は考慮されていない．

排他制御変数や条件変数を実現するプロトコルとしては，トークンをベースとしたものが研究 [30] で提案されており，さらに研

究 [29] では排他制御の優先度を考慮した拡張も行われている．

7.3 大規模分散共有メモリ

大規模分散共有メモリの研究事例としては DLM [42] や Cashmere-VLM [12] などがある．DLM におけるネットワークページングは，1Gbit および 10Gbit Ethernet 結合クラスタを用いた性能評価において，ディスクスワップを利用する場合と比較して 5〜10 倍の性能を達成している．DLM では，ページ置換アルゴリズムとして単純なラウンドロビン方式が採用されている．また，Cashmere-VLM は，分散共有メモリである Cashmere を大規模分散共有メモリに拡張したものであり，ページの状態と最終更新時刻に基づいたページ置換アルゴリズムが実装されている．しかし，DLM も Cashmere-VLM も，各ページに対して固定的な帰属ノードを設けることでページの追い出しを実現しているため，これらのプロトコルは，固定的なノードを仮定することが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，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 [15] 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いる．そのため，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追い出しに対処している．

7.4 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例としては DSM-Threads [27, 28, 32] がある．DSM-Threads はコンシステンシプロトコルとして DDMA を採用し，トークンをベースとした同期操作を実装している点などにおいて，DMI と共通する部分が多い．また，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．

7.5 動的負荷分散

分散共有メモリに対して動的負荷分散を図った事例としては Omni/SCASH [40, 39] がある．Omni/SCASH は，page-based・ホーム問い合わせ型の分散共有メモリである SCASH をベースとした OpenMP の分散環境上での実装であり，動的負荷分散技術としてループ再分割手法とページマイグレーション手法が提案されている．ループ再分割手法は，まずループの初期イテレーションの実行時性能をプロファイリングし，その性能差に比例して残りのイテレーションを各計算資源に割り当てることで，計算資源間の性能差に起因するロードインバランスを解消する手法である．一方のページマイグレーション手法は，実行時のページフォルト発生数をカウントし，最も参照数が多かったノードにホームノードを移動することで，通信メッセージ数を削減する手法である．一方，DMI と OpenMP では事情が大きく異なり，DMI では，任意のページサイズを許しているために巨大なページ転送が発生し，オーナーが通信上のボトルネックになるのが問題となっている．そこで DMI では，オーナーからのページ転送を動的に木構造化することによってデータ転送の効率化を図る．

7.6 分散共有メモリとメッセージパッシング

研究 [10] 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，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．DMI の主な特長をまとめると以下の通りである：

- 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/脱退を実現できるようなコンシステンシプロトコルを提案して実装している．
- 動的なスレッド生成/破棄が可能な pthread 型のプログラミングスタイルによって，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．
- マルチコア上の並列プログラムを分散プログラムに移植する際の敷居を下げることを目指し，pthread による並列プログラムとの類似性を重視したインタフェース設計を行っている．
- 並列分散ミドルウェア基盤としての性格を狙っており，ユーザレベルで OS のメモリ管理機構をシミュレートすることによっ

て，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，柔軟性，汎用性，機能拡張性に富んだインタフェースを提供している．

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．

8.2 今後の課題

8.2.1 処理系の改良

第一の課題は，現在実装段階にあるページ転送の動的負荷分散機能の完成である．動的負荷分散を行う際のページ転送グラフの形状としては，完全 m 分木などが考えられる．m を可変パラメータとしてこの手法の有効性を検証する必要がある．

第二の課題は，ページ置換アルゴリズムの改良である．**5.6** で述べたように，現状では，ページサイズとページの状態のみに基づいた優先度順での追い出しを行っているが，ページの参照局所性も考慮するために LRU などを優先度の評価基準に組み込むことが望ましい．また，現状では追い出し先ノードを単純に乱数で選択していることも問題である．そこで，普段飛び交っている多数のメッセージに送信元ノードの DMI 物理メモリの使用状況を付加することで，各ノードが全ノードの DMI 物理メモリの使用状況を大まかに把握できるようにし，その情報を追い出し先ノードの選択時に役立てることも考えている．

第三の課題は，効率的な `DMI_malloc()` の実装である．現状の DMI に実装している `DMI_malloc()` は，呼び出される度にグローバルロックを取得して全ノードへの DMI 仮想メモリ確保を行うという実装になっており，つまり，共有メモリ環境上での brk に相当する操作を行っている．したがって，現状の `DMI_malloc()` をベースとして，「真の `DMI_malloc()`」や `DMI_realloc()` を実装する必要がある．その際には，DMI が DMI 仮想共有メモリ上での “マルチ DMI スレッド環境” であることを意識し，glibc の malloc や Thread-Caching Malloc [6] などのマルチスレッド環境下での効率的な malloc の実装が参考になると考えられる．

第四の課題は，効率的な同期操作の実装である．現状ではトークンによって同期操作を実現しているが，**5.3** および **5.4** で述べたように，トークンがグローバルなリソースであるためにトークンの確保/解放にはグローバルロックが必要であり，したがって同期用変数の初期化/終了処理が重い．また，トークンによる同期操作の実装は，仮想共有メモリアドレス空間とは別の “仮想共有トークンアドレス空間” を作っていることを意味し，全ての操作が共有メモリアドレス空間上で完結する共有メモリ環境とのモデル的な対応性を崩している．したがって DMI では現在，より pthread の実装に近い，効率的な同期プロトコルへの改善を試みている．

第五の課題は，read/write 粒度での invalidate 型と update 型のハイブリッド型プロトコルの実装である．現状では，各ノードにおける各ページの *status* は *valid* と *invalid* の 2 状態で管理され，`DMI_write()` は *status* = *valid* なノードたちに対して無効化要求を送信するという invalidate 型のプロトコルを採用している．これを改善し，*status* の状態を *eager_valid*，*lazy_valid*，*invalid* の 3 状態に増やし，`DMI_write()` は，*eager_valid* なノードたちに対しては最新ページを送りつけ，*lazy_valid* なノードたちに対しては無効化要求を送信するという実装にする．その上で，`DMI_read()` のモードとして，「read 後に *invalid* になりたい read（つまりページをキャッシュしない read）」「read 後に *lazy_valid* になりたい read」「read 後に *eager_valid* になりたい read」の 3 種類を設ける．これにより，read/write の粒度で invalidate 型と update 型のハイブリッド型プロトコルが実現でき，アプリケーションの挙動に合致した，より柔軟なチューニング手段を提供できるようになる．

8.2.2 処理系を基盤とした言語処理系の開発

そもそも本研究の出発点は，大規模な計算環境において計算資源の動的な参加/脱退をサポートできるような分散オブジェクト指向言語処理系を実装しようと考えたことにあった．しかし，言語設計を検討するうちに，オブジェクトのマイグレーションなどを自動で行ってくれるような高度な通信レイヤーが必須であると判断したため，並列分散プログラミングフレームワークを開発するためのミドルウェア基盤として，DMI の開発を始めた．したがって，DMI の適用可能性を探る意味でも，DMI を基盤レイヤーに敷いた並列分散プログラミング言語処理系を構築したいと考えている．具体的に何を主目的とした言語処理系を開発するかは決まっていないが，

- 計算資源の動的な参加/脱退がサポートされているからこそ実現できること
- 大規模な分散共有メモリが提供されているからこそ実現できること
- Java RMI や dRuby [1] のような既存言語に対する分散処理用ライブラリではなく，最初から分散処理を前提として言語を設計するからこそ実現できること

などを念頭に，言語の方向性を検討する必要がある．

8.2.3 処理系の将来像

3.7 で述べたように，現段階の DMI では，主に単一クラスタ環境上での実行しか想定していない．しかし，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，もっと大規模な計算環境である．ところが，DMI をこれらの環境に対応させるには課題が山積している．まず，接続性の問題がある．計算資源数が増加すれば全対全でのコネクション接続はリソース的に不可能であるし，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策も必須である．

また，耐故障性も必須の課題となる．接続性に関しては既存の有能なソケットライブラリを通信レイヤーとして用いることである程度は補えるかもしれないが，耐故障性に関しては実装レベルでの抜本的な対策が必要である．これらの課題に対して具体的にどう対応するかはまだ考慮できていないが，将来的には DMI を大規模な計算環境に適用することも視野に入れつつ今後の開発を進める必要がある．

参考文献

[1] dRuby. http://www2a.biglobe.ne.jp/seki/ruby/druby.html.

[2] GXP. http://www.logos.ic.i.u-tokyo.ac.jp/gxp/index.php.

[3] Java RMI. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.

[4] MPI. http://www-unix.mcs.anl.gov/mpi/.

[5] OpenMP. http://openmp.org/wp/.

[6] TCMalloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[7] Christiana Amza, Alan L.Cox, Hya Dwarkadas, Pete Keleher, Honghui Lu, Weimin Yu Ramakrishnan Rajamony, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 1996.

[8] Rafik A.Salama and Ahmed Sameh. *Potential Performance Improvement of Collective Operations in UPC*. John von Neumann Institute for Computing, 2007.

[9] 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.

[10] Ayon Basumallik and Rudolf Eigenmann. Towards Automatic Translation of OpenMP to MPI. *Proceedings of the 19th annual International Conference on Supercomputing*, 2005.

[11] William Carlson, Thomas Sterling, Katherine Yelick, and Tarek El-Ghazawi. *UPC Distributed Shared Memory Programming*. WILEY INTER-SCIENCE, 2005.

[12] 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.

[13] Tarek El-Ghazawi and Francois Cantonnet. UPC Performance and Potential: A NPB Experimental Study. *Supercomputing 2002*, 2002.

[14] 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.

[15] Erik Hagersten, Anders Landin, and Seif Haridi. DDM — a Cache-Only Memory Architecture. *IEEE Computer*, 1992.

[16] Gerard J.Holzmann. A Stack-Slicing Algorithm for Multi-Core Model Checking. *Electronic Notes in Theoretical Computer Science*, 2008.

[17] Gerard J.Holzmann and Dragan Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, 2007.

[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] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. *In Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

[22] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 1989.

[23] Song Li, Yu Lin, and Michael Walker. Region-based Software Distributed Shared Memory. 2000.

[24] 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.

[25] 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.

[26] 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.

[27] Frank Mueller. Distributed Shared-Memory Threads: DSM-Threads. *Workshop on Run-Time Systems for Parallel Programming*, 1997.

[28] Frank Mueller. On the Design and Implementation of DSM-Threads. *Conference on Parallel and Distributed Processing Techniques and Applications*, 1997.

[29] Frank Mueller. Priority Inheritance and Ceilings for Distributed Mutual Exclusion. *IEEE Real-Time Systems Symposium*, 1999.

[30] 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.

[31] Brian N.Bershad, Matthew J.Zekauskas, and Wayne A.Sawdon. The Midway distributed shared memory system. *Compcon Spring '93, Digest of Papers*, 1993.

[32] 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.

[33] 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.

[34] Weisong Shi. Heterogeneous Distributed Shared Memory on Wide Area Network. *IEEE TCCA Newsletter*, 2001.

[35] 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.

[36] 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.

[37] 高橋慧. Distributed Aggregate with Migration. 東京大学 卒業論文, 2005.

[38] 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.

[39] 栄純明, 松岡聡, 佐藤三久, 原田浩. Omni/SCASH における実行時性能評価に基づく動的負荷分散拡張の実装と評価. 情報処理学会研究報告 [ハイパフォーマンスコンピューティング], 2003.

[40] 栄純明, 松岡聡, 佐藤三久, 原田浩. Omni/SCASH における性能不均質なクラスタ向け動的負荷分散機能の実装と評価. 情報処理学会研究報告 [ハイパフォーマンスコンピューティング], 2004.

[41] 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.

[42] 緑川博子, 小山浩生, 黒川原佳, 姫野龍太郎. 分散大容量メモリシステム DLM の設計と DLM コンパイラの構築. 電子情報通信学会技術研究報告 [コンピュータシステム], 2007.

[43] 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.

謝辞

本研究を進めるにあたって，指導教員である近山隆教授には，発表練習などを通じて数多くの的確なアドバイスをいただいた他，DMI におけるコンシステンシプロトコルの効率化に関して具体的な手法を提案していただきました．同じく指導教員である田浦健次朗准教授には，毎週のミーティングなどを通じて緻密なご指導をいただき，モデル設計から実装の詳細に至るまでテクニカルなアドバイスをたくさんいただきました．DMI が現在の姿を得ているのは，田浦准教授が正しい方向に導いてくださったからこそだと思います．また，横山大作助教，ポスドクの柴田剛志さん，博士課程 3 年の斎藤秀雄さん，修士課程 2 年の弘中健さんには，実装面での有用なアイディアを提供していただいた他，実験環境の構築にも協力していただきました．学部生の藤澤徹さんには，モデル設計から具体的なコーディングに至るまで何度も相談に乗っていただき，抱えていた疑問をいくつも解決することができました．そして，近山・田浦研究室の方々には常に生活面でお世話になっています．

この場を借りて，本研究を支えてくださったみなさまに厚くお礼申し上げます．

付録 A コンシステンシプロトコルのアルゴリズム

ページ p 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，および各ノードにおいてメッセージはシリアライズして処理されることを仮定している．また，簡単のため，ページ転送の動的負荷分散を行う場合に必要となるページ転送グラフに沿ったメッセージのフォワーディングに関するコードは省略している．

以下の記述における *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．また，my id はそのコードを実行するプロセスの id を示す．

```
when read operation on p happened:
  lock p
  if p.status = INVALID
    REQ_READ.src ← my id
    send REQ_READ to p.probable
    unlock p
    wait for ACK_READ to be received
    lock p
    if ACK_READ.buf != NIL
      p.status ← VALID
      p.buf ← ACK_READ.buf
    p.probable ← ACK_READ.src
  execute read operation on p.buf
  unlock p

when REQ_READ is received:
  lock p
  if p.owner = TRUE
    if REQ_READ.src ∉ p.treeset
      ACK_READ.buf ← p.buf
      p.treeset ← p.treeset ∪ { REQ_READ.src }
    else
      ACK_READ.buf ← NIL
    unlock p
    ACK_READ.src ← my id
    send ACK_READ to REQ_READ.src
  else
    forward REQ_READ to p.probable
    unlock p

when write operation on p happened:
  lock p
  if p.owner = FALSE or |p.treeset| != 1
    REQ_WRITE.src ← my id
    send REQ_WRITE to p.probable
    unlock p
    wait for ACK_WRITE to be received
    lock p
    if ACK_WRITE.buf != NIL
      p.status ← VALID
      p.buf ← ACK_WRITE.buf
    p.probable ← my id
    p.owner ← TRUE
    p.treeset ← { my id }
  execute write operation on p.buf
  unlock p

when REQ_WRITE is received:
  lock p
  if p.owner = TRUE
    p.owner ← FALSE
    if REQ_WRITE.src ∉ p.treeset
      ACK_WRITE.buf ← p.buf
      p.treeset ← p.treeset ∪ { REQ_WRITE.src }
    else
      ACK_WRITE.buf ← NIL
    for each s ∈ p.treeset \ { REQ_WRITE.src }
      REQ_INVALIDATE.src ← my id
      REQ_INVALIDATE.next ← REQ_WRITE.src
      send REQ_INVALIDATE to s
    p.treeset ← φ
    unlock p
    wait for all ACK_INVALIDATE to be received
    send ACK_WRITE to REQ_WRITE.src
  else
    forward REQ_WRITE to p.probable
    unlock p

when REQ_INVALIDATE is received:
  lock p
  p.probable ← REQ_INVALIDATE.next
  p.status ← INVALID
  p.buf ← NIL
  unlock p
  send ACK_INVALIDATE to REQ_INVALIDATE.src

when sweep operation on p happened:
  lock p
  if p.status = VALID
    REQ_SWEEP.src ← my id
    send REQ_SWEEP to p.probable
    unlock p
    wait for ACK_SWEEP to be received
  else
    unlock p
```

```
when REQ_SWEEP is received:
  lock p
  if p.owner = TRUE
    if REQ_SWEEP.src = my id
      next ← select new owner arbitrarily
      p.owner ← FALSE
      if next ∉ p.treeset
        REQ_DELEGATE.buf ← p.buf
        p.treeset ← p.treeset ∪ { next }
      else
        REQ_DELEGATE.buf ← NIL
      for each s ∈ p.treeset \ { next }
        REQ_RESIGN.src ← my id
        REQ_RESIGN.next ← next
        send REQ_RESIGN to s
      unlock p
      wait for all ACK_RESIGN to be received
      lock p
      p.treeset ← p.treeset \ { my id }
      p.status ← INVALID
      p.buf ← NIL
      REQ_DELEGATE.src ← my id
      REQ_DELEGATE.treeset ← p.treeset
      send REQ_DELEGATE to next
      p.treeset ← φ
      unlock p
      wait for ACK_DELEGATE to be received
    else
      if REQ_SWEEP.src ∈ p.treeset
        p.owner ← FALSE
        p.treeset ← p.treeset \ { REQ_SWEEP.src }
        REQ_UNSET.src ← my id
        send REQ_UNSET to REQ_SWEEP.src
        unlock p
        wait for ACK_UNSET to be received
        lock p
        p.owner ← TRUE
      unlock p
      send ACK_SWEEP to REQ_SWEEP.src
    else
      forward REQ_SWEEP to p.probable
      unlock p

when REQ_RESIGN is received:
  lock p
  p.probable ← REQ_RESIGN.next
  unlock p
  send ACK_RESIGN to REQ_RESIGN.src

when REQ_DELEGATE is received:
  lock p
  if REQ_DELEGATE.buf != NIL
    p.status ← VALID
    p.buf ← REQ_DELEGATE.buf
  p.probable ← my id
  p.owner ← TRUE
  p.treeset ← REQ_DELEGATE.treeset
  unlock p
  send ACK_DELEGATE to REQ_DELEGATE.src

when REQ_UNSET is received:
  lock p
  p.probable ← REQ_UNSET.src
  p.status ← INVALID
  p.buf ← NIL
  unlock p
  send ACK_UNSET to REQ_UNSET.src
```

なお，上記のプロトコルにおける *ACK_SWEEP* はコンシステンシを維持する上では必要ない．DMI のトランザクション管理のように，ページの追い出しが完了したことを把握する必要がある場合のみ必要となる．

付録 B　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の関数を整備している．API を用いたプログラムでは，`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(...)` を呼び出すと，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．

- `int32_t DMI_id(void);`

  自ノードの id を得る．id は，その時点で実行している全ノードを通じて一意であることが保証されるが，全ノードを通じた id 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, int32_t *num, int32_t capacity);`
  ノードの一覧を最大 `capacity` 個だけ `nodes` に格納する．`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），`id`（id）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` 個になるまで待機する．
- `void DMI_member_init(DMI_member *member);`
  メンバを初期化する．
- `void DMI_member_destroy(DMI_member *member);`
  メンバを破棄する．
- `int32_t DMI_member_poll(DMI_member *member, DMI_node *nodes, int32_t *num, 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状態に変化があったノードのリストを，最大 `capacity` 個だけ `nodes` に格納する．
- `int32_t DMI_member_notify(DMI_member *member);`
  `DMI_member_poll(...)` を強制的に起こす．
- `int32_t DMI_alloc(int64_t *addr_holder, 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 仮想メモリを確保し，そのアドレスを `addr_holder` に格納する．`page_num` 個のページのオーナーノードは，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);`
  DMI 仮想共有メモリアドレス `addr` から `size` バイトを `buf` に読み込む．
- `int32_t DMI_write(int64_t addr, int64_t size, void *buf);`
  DMI 仮想共有メモリアドレス `addr` に `buf` から `size` バイトを書き込む．
- `int32_t DMI_create(int64_t *handle_holder, int32_t id, int64_t addr);`
  id が `id` のノード上に DMI スレッドを生成し，そのハンドルを `handle_holder` 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_holder);`
  スレッドを回収する．
- `int32_t DMI_detach(int64_t handle);`
  スレッドを detach する．
- `int32_t DMI_self(int64_t *handle_holder);`
  自スレッドのハンドラを得る．
- `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_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);`
  排他制御変数を trylock 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_cond_init(int64_t cond_addr);`
  条件変数を初期化する．

- `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);`
  条件変数の wait 操作を行う.
- `int32_t DMI_cond_destroy(int64_t cond_addr);`
  条件変数を破棄する.
- `void DMI_iread(int64_t addr, int64_t size, void *buf, DMI_status *status);`
  非同期モードの `DMI_read(...)` で，`status` が非同期操作のハンドルである．以下省略するが，上記の `DMI_alloc(...)` から `DMI_cond_destroy(...)` までの全ての関数には，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, int32_t *exit_code_holder);`
  `status` をハンドルとする非同期操作が終了したかどうか検査し，終了しているならば非同期操作の返り値を `exit_code_holder` に格納する.
- `void DMI_wait(DMI_status *status, int32_t *exit_code_holder);`
  `status` をハンドルとする非同期操作の終了を待機し，非同期操作の返り値を `exit_code_holder` に入れる.

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．SPI の関数は原則的に非同期モードである．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` のメモリを，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, int32_t *dmi_id_holder, char *ip, uint16_t port, DMI_status *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．`dmi_id_holder` にはこのノードの id が格納される.
- `void dmi_spi_leave_dmi(dmi_t *dmi, DMI_status *status);`
  このノード上での全ての DMI トランザクションを禁止した上で，禁止した時点で実行されている全 DMI トランザクションの完了を待機する．その後，脱退処理を行う.
- `void dmi_spi_open_process(dmi_t *dmi, DMI_status *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_process(dmi_t *dmi, DMI_status *status);`
  このノード上への DMI スレッドの生成を禁止した上で，禁止した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 にそのまま対応する SPI として，`dmi_spi_alloc_vm(...)`, `dmi_spi_free_vm(...)`. `dmi_spi_read_vm(...)`, `dmi_spi_write_vm(...)`, `dmi_spi_create_thread(...)`, `dmi_spi_join_thread(...)`, `dmi_spi_detach_thread(...)`, `dmi_spi_self_thread(...)`, `dmi_spi_mutex_init(...)`, `dmi_spi_mutex_destroy(...)`, `dmi_spi_mutex_lock(...)`, `dmi_spi_mutex_unlock(...)`, `dmi_spi_mutex_trylock(...)`, `dmi_spi_cond_init(...)`, `dmi_spi_cond_destroy(...)`, `dmi_spi_cond_signal(...)`, `dmi_spi_cond_broadcast(...)`, `dmi_spi_cond_wait(...)`, `dmi_spi_check_status(...)`, `dmi_spi_wait_status(...)` が存在する.