卒業論文中間報告

DMI：ノードの動的な増減に対応した大規模分散共有メモリインターフェース

平成 20 年 9 月 18 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

目次

1 はじめに

1.1 背景

近年，高性能プロセッサの低価格化やネットワークの高バンド幅化，大規模計算資源を必要とするアプリケーションの要請などを背景として，多数の汎用計算資源をネットワーク接続したクラスタ環境上での並列分散コンピューティングの重要性がますます高まっている．

並列分散コンピューティングのプログラミングモデルとしては，メッセージパッシング，DSM（Distributed Shared Memory），RPC（Remote Procedure Call），ソケットなどさまざまな形態が存在するが，プログラムの記述性，スケーラビリティ，多様なネットワーク構成への柔軟な対応性，動的な計算資源の増減への適応性などの観点から比較すると，各モデルには長所短所が存在する [1]．本研究対象である DSM は，他のプログラミングモデルと比較してプログラミングの容易さという利点を持つ [1, 2, 3, 4]．たとえばメッセージパッシングではデータのフローを意識して送信側と受信側の両方を記述する必要があるのに対して，DSM では共有メモリ型アーキテクチャ上のプログラミングと同様の記述が可能であり，逐次プログラムからの飛躍が小さい．また，DSM は通常の仮想共有メモリとしての利用以外にも，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幅広い応用を持つ [2]．その反面，DSM はスケーラビリティの点ではメッセージパッシングに劣る [1, 2, 3]．これは，DSM がメッセージパッシングよりも抽象度の高いプログラミングモデルであることに依る部分が大きいが [2]，より具体的な要因としては，メッセージパッシングではプログラム上に通信パターンを明示的に記述するため効率的な集合通信を実現可能であるのに対し，DSM では通信パターンを把握できないためにデータ転送の最適化が難しいことなどが挙げられる．これらのトレードオフを背景として，DSM で記述されたプログラムをメッセージパッシングに変換する試みも成されている [3]．

一方，並列分散コンピューティングを実現するミドルウェアに求められる機能も多様化している．近年ではプロセッサのマルチコア化が加速し，クラスタ環境やスパコンの構成要素として 8 コアや 16 コアのプロセッサが用いられる場合も多い．そのため，分散レベルでの並列性とマルチコアレベルでの並列性を同時に考慮した並列分散ミドルウェアの開発がますます重要視されている [1]．MPI などでも，ノード間通信にはソケット通信を利用するものの，同一ノード内では共有メモリ経由のプロセス間通信を行うなど，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効率的に取り扱う実装が施されている場合が多い．また，動的な資源の追加や削除への対応も重要な課題である．動的な資源の増減に対しては，ロードバランシングのために動的負荷分散を図りたいというアプリケーション側からの要請と，クラスタ環境やスパコン環境の運用ポリシーや課金制度の都合上，利用する計算資源を動的に移動させたいという資源面からの要請がある．現在，高い記述性と性能を確保しつつノードの動的な参加・脱退を実現するためのプログラミングモデルや実行形態が研究されている [1]．

1.2 本研究の貢献

本研究では，大規模分散共有メモリのインターフェースとして DMI（Distributed Memory Interface）を提案および実装し，その評価を行う．本研究の貢献は以下の通りである：

- DMI では，OS のページフォルトをシグナルハンドラで捕獲する方式ではなく，DMI に専用の論理アドレス空間やアドレス空間記述テーブル，ページテーブル，ページの実体を格納するためのメモリ領域などを，全てユーザレベルで自前で実装してメモリ管理を行う．これにより，任意のページサイズの仮想共有メモリの構築が可能となる他，OS のアドレッシング範囲を超えた大規模分散共有メモリが実現できるなど，アプリケーションプログラムに対して柔軟性の高いインターフェースを提供できる．
- DSM の長所であるプログラミングの容易さと明快さを維持しつつ，ノードの動的な増減を実現するようなプログラミングスタイルと実行形態を整備する．
- 特定のノードへページ要求が集中した場合には，それを動的に負荷分散することで転送効率を向上させる．
- マルチコア環境への適応を図るため，DMI のメモリ空間管理用のリソースをプロセス間共有メモリ上に配置す

ることで，同一ノード内に存在する複数の DMI のプロセスがリソースを共同利用できるようにする．

DSM は過去 20 年以上にわたり盛んに研究されている分野であり，多数の実装例が存在し，ネットワークページングや動的負荷分散の研究事例も多い．しかし，DSM 用のメモリ空間をユーザレベルで実装したり，動的なノードの増減に対応した例はほとんどない．

DMI は，アプリケーションプログラムを実際に記述するためのインターフェースであるとともに，特に，並列分散ミドルウェア開発の基盤レイヤーとしての利用を意識している．たとえば，DMI の上に分散オブジェクト指向の言語処理系を構築することで，オブジェクトのマイグレーションや動的なプロセス管理などの機能の実現が容易になるなどの応用が考えられる．

1.3 本稿の構成

2 節では関連研究について述べる．**3** 節では，DMI が実現する大規模分散共有メモリモデルについて論じる．なお，**3** 節の内容は現状の DMI の実際の実装に基づいているが，**3.5** 項などをはじめ，今後実装予定の機能に関する記述も一部含む．**4** 節では，MPI と現状の DMI との予備的な性能比較を行う．**5** 節では，本研究が今後取り組むテーマである，ノードの動的な増減，ページ転送の動的負荷分散，マルチコア環境への適応についてその実現手法を提案する．

2 関連研究

2.1 DSM のコンシステンシモデル

DSM のコンシステンシモデルとしては，Sequential Consistency（SC），Weak Consistency（WC），Release Consistency（RC），Entry Consistency（EC）などが一般的であり，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通信量が低減されパフォーマンスが向上するため，RC の Munin，Lazy Release Consistency（LRC）の TreadMarks，EC の Midway など，従来の DSM ではコンシステンシの緩和が積極的に試されてきた [5]．しかし，コンシステンシの緩和はプログラミング上の負担を増やす上，プログラムの挙動が直感的にわかりにくくなる．

DMI では，ページサイズを任意に指定可能とすることで性能上の問題をある程度補えると考えたため，動作が直感的に明快な SC を採用した．

2.2 DSM におけるオーナーノードの管理

コンシステンシ維持のためのプロトコル設計に際しては多様なデザインイシューが存在するが，ページフォルト発生時の挙動に関して主に以下の 3 種類がある [2]：

- オーナー固定型：ページの最新状態やノード間でのページの共有状況などを一括管理するオーナーノードをページごとに固定する．ページフォルト時には，該当ページのオーナーノードにリクエストを送る．
- ホーム問い合わせ型：オーナーノードは変化するが，その位置を常に把握するホームノードをページごとに固定して設ける．ページフォルト時に，オーナーノードを見失っている場合にはホームノードに問い合わせれば良い．
- オーナー追跡型：オーナーノードは変化し，ホームノードも設置しない．ページフォルト時には，後述するオーナー追跡グラフを辿ることで，リクエストを真のオーナーノードへとフォワーディングする．

まず，データのローカリティを考慮した場合，特定ノードへの負荷集中を回避する意味では，オーナーノードは固定ではなく移動させる方が望ましい．しかし，ホームノードの設置の是非に関してはコンシステンシモデルに依るところが大きく，たとえば SC ではホームノードを設置しない方がメッセージ総数が減ると考えられるが，LRC の場合

にはホームノードを設置する方が効率が良いという報告もある [6].

オーナー追跡型のプロトコル [7] では，各ノードが各ページに対して，そのオーナーノードの位置情報を管理する．ただし，通信量を抑制するため，オーナーノードが変化する度に全ノードにおけるオーナーノードの位置情報を厳格に更新するわけではない．代わりに，全ノードを通じたオーナーノードの位置情報の参照関係が，任意のノードが必ず真のオーナーノードへと到達可能なグラフを形成するように，必要な更新のみを施す．すなわち，オーナー追跡型のプロトコルでは，任意のページ p に関する有向グラフ G_p：

$$
\begin{aligned}
G_p &= (V, E_p), \\
V &= \text{the set of nodes}, \\
(i,j) \in E_p &\ \text{if node } i \text{ thinks the owner node of } p \text{ is node } j
\end{aligned}
$$

に関して，到達可能条件：

任意のノードは，有向枝を辿ることで真のオーナーノードに到達可能である

が維持されるように，各ノードにおけるオーナーノードの位置情報を適宜書き換えていく．ページフォルト発生時には，このオーナー追跡グラフに沿って，リクエストが真のオーナーノードへとフォワーディングされる．

SC を採用する DMI では，ホームノードを設けないオーナー追跡型のプロトコルを，Li らの提案するアルゴリズム [7] を改善した上で実装した．

2.3 大規模分散共有メモリ

1 台のマシンが提供できる物理メモリにはハードウェア制約があること，大容量物理メモリを搭載したマシンは非常に高価なこと，ディスクアクセスを伴うローカルな swap 処理よりも他ノードのメモリを利用するネットワークページングの方が高速なことなどを根拠として，クラスタ環境の各ノードのメモリを集めた大規模分散共有メモリの技術が出現している [8]．大規模分散共有メモリの一実装である DLM（Distributed Large Memory） [8] は，1Gbit/10Gbit Ethernet 結合クラスタを用いた実験で，swap ファイル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．また，ネットワークページングにおいては，遠隔メモリから転送してきたページを格納するための空き領域が不足した場合に，効率的なページ置換アルゴリズムを働かせることが重要である．Cashmere-VLM [9] では，大規模分散共有メモリの一実装である Cashmere に対して，ページ状態と最終更新時刻に基づいた優先度順でのページ置換を実装し，その有効性を報告している．

これら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するため 64bit OS を前提としているが，DMI では，DMI 用のメモリ空間をユーザレベルで実装してメモリ管理を行うため，OS のアドレッシング範囲は問題にならない．また，ページ状態のみに基づいたページ置換アルゴリズムを実装する．

2.4 DSM における動的負荷分散

ホーム問い合わせ型の DSM である SCASH 上の OpenMP の実装である Omni/SCASH では，動的負荷分散技術としてループ再分割とページマイグレーションが施されている [10, 11]．ループ再分割は，ループの初期イテレーションの実行時性能を計測し，その性能差に基づいて残りのイテレーションを各プロセスに負荷分散する手法であり，ページマイグレーションは，ページフォルト発生数をカウントし，最も参照数が多いノードにホームノードを移動する手法である．

DMI では，特定ノードへのページ転送要求の集中を検知した場合に，実際のページ転送処理を他ノードに任せることで，データ転送のボトルネックを動的に解消することを試みる．

2.5 動的な資源の増減

Phoenix [1] は，動的な資源の増減に対応したメッセージパッシングベースの並列計算プラットフォームである．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，各計算ノードにこ

```
int main(int argc, char **argv) {
    int rank, id, pnum, i, *pid;
    ull addr;

    DMI_init(&argc, &argv, 100 * 1024, &rank, &id, &pnum);
    DMI_malloc(sizeof(int), pnum, sizeof(int), &addr);
    DMI_write(addr + rank * sizeof(int), sizeof(int), getpid());
    DMI_barrier();
    if(rank == 0) {
        pid = (int*)malloc(pnum * sizeof(int));
        DMI_read(addr, sizeof(int) * pnum, pid);
        for(i = 0; i < pnum; i++) {
            printf("rank %d : %d\n", i, pid[i]);
        }
        free(pid);
    }
    DMI_free(addr);
    DMI_final();
    return 0;
}
```

Fig.1　DMI のプログラム例

の部分集合を重複なく割り当てる．ノードが参加する場合にはすでに参加中の適当な計算ノードが持つ集合の一部を分け与え，脱退する場合には他の計算ノードに集合を委譲することで，計算を通じて，計算ノード全体で仮想ノード名空間を重複なく包むよう管理する．この管理の下では，ノードの増減を局所的な変更操作のみで実現可能であるとともに，仮想ノード名を用いてメッセージの送受信を行えば，ノードの参加・脱退が生じてもメッセージの損失は起こらない．また Phoenix は，メッセージパッシングで記述された各種プログラムを容易に広域分散化できる記述力を有し，スケーラビリティにも優れる．

DMI では，SPMD のプログラミング形態を採用し，各プロセスに対してランクとは別に ID を割り振ることで，動的な資源の増減に対して高い記述性を確保するとともに，外部コマンドからノードの増減を容易に実現できる実行形態を整える．

3　DMI の大規模分散共有メモリモデル

3.1　プログラムインターフェース

DMI では，Fig.1 [*1] に示すような SPMD 型のプログラミングを行う．次節以降で詳説するが，DMI ではユーザプログラムのメモリ空間とは独立に DMI のメモリ空間をユーザレベルで構築する．Fig.2 のモデル図が示すように，ユーザプログラムから DMI のメモリ空間をポインタなどで直接操作することはできず，`DMI_read(...)` や `DMI_write(...)` などの API を利用しなければならない．このように DMI は，ユーザプログラムのメモリ空間と DMI のメモリ空間の操作を明確に区別したプログラミングを強制する．そのため，仮想的な共有メモリをあたかも物理的な共有メモリと同様に扱えるという DSM 本来の透過性を損なっており，プログラミング上の負担も大きいため，インターフェースとしては従来の DSM に劣ると言わざるを得ない．しかし，DMI のプログラミングは作業的に手間はかかるが論理的に難しいわけではない．また，メモリ空間の明確な区別ゆえ，必要なデータ通信しか起きない効率的なプログラムが自然と開発される可能性が高い．

Fig.3 に DMI の提供する API を示す．API の設計に当たっては主に UPC [4] を参考にした．これらの API はス

[*1] 以降，`unsigned long long` を `ull` と示す．

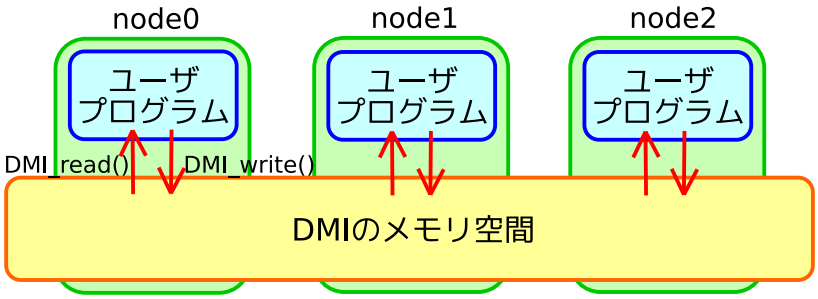

Fig.2　ユーザプログラムから見た DMI のモデル

`int DMI_init(int *argc, char ***argv, int size, int *rank, int *id, int *pnum)`：初期化関数
`int DMI_final(void)`：終了関数
`int DMI_polling(void)`：ノードの増減の有無を検査
`int DMI_rebuild(int *id, int *pnum)`：ノード増減時の再編関数
`int DMI_malloc(ull block, ull num, ull unit, ull *address)`：DMI 仮想メモリ確保
`int DMI_free(ull address)`：DMI 仮想メモリ解放
`int DMI_read(ull address, ull size, void *buf)`：該当メモリ領域をリード
`int DMI_write(ull address, ull size, void *buf)`：該当メモリ領域をライト
`int DMI_iread(ull address, ull size, void *buf, int *req)`：該当メモリ領域を非同期リード
`int DMI_iwrite(ull address, ull size, void *buf, int *req)`：該当メモリ領域を非同期ライト
`int DMI_itest(int req)`：非同期操作の完了を検査
`int DMI_icomplete(int req)`：非同期操作の完了を待機
`int DMI_barrier(void)`：バリア
`int DMI_notify(void)`：バリアの notify フェーズ
`int DMI_wait(void)`：バリアの wait フェーズ
`int DMI_fence(void)`：以前に発行したメモリ操作の大域的完了を待機
`int DMI_lock(ull address, ull size)`：該当メモリ領域の占有ロック取得
`int DMI_unlock(ull address, ull size)`：該当メモリ領域の占有ロック解放

Fig.3　DMI の API

レッドセーフである．また，プログラムからの明示的な制御によってパフォーマンスを引き出せるようにするため，チューニングの自由度を与えるような柔軟性の高いインターフェースを提供する一方で，長大な連続アクセス発生時の自動プリフェッチ，複数メッセージのコンバイニング，データの差分転送など，複雑で高度な暗黙的機構は DMI には実装しない方針とした．

3.2　実行モデル

DMI プログラムの実行モデルは以下の通りである（Fig.4）：

1. `libdmi` をリンクして DMI プログラムをコンパイル後，ノードファイルを指定して `dmirun` コマンドで実行すると，そのノード上で coordinator が生成される．
2. coordinator は，予め各ノード上で起動してある daemon に対して，coordinator の位置情報と（デフォルトでは）DMI プログラムの実行バイナリを送りつける．
3. 各 daemon は DMI プロセスを生成し，各 DMI プロセス上で，coordinator の位置情報を与えて実行バイナリが実行される．

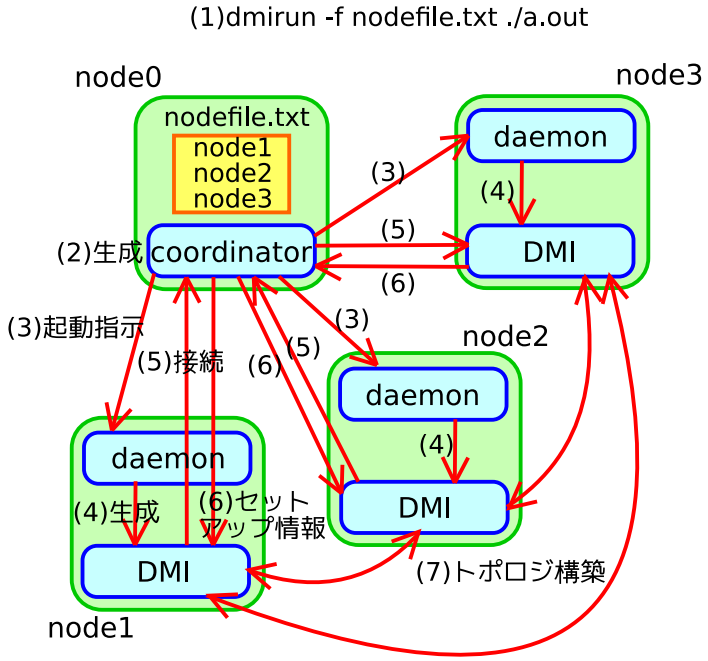

Fig.4　DMI プログラムの実行モデル

4. 各 DMI プロセスは `DMI_init(...)` を呼ぶことで coordinator への接続を確立し，coordinator から全 DMI プロセスの情報を受信し，初期化処理を行う．
5. 全 DMI プロセスが全対全のコネクションを確立する．

MPI や各種 DSM の実行形式としては rsh を用いるものが多いが，DMI では，将来的にセキュリティ上の対策を講じたいため，各ノードで daemon を起動しておくモデルを採用した．また，（デフォルトでは）実行時にプログラムバイナリを直接送りつける形式を取るため，実行ノードのアーキテクチャがそのバイナリを実行できる必要はあるものの，ホームディレクトリが共有されていないノードに対して事前に実行バイナリを配置する手間が不要である．

3.3　大規模分散共有メモリの管理

DMI メモリ空間の管理は，OS が行うメモリ管理機構に類似の原理で行う．その模式図を Fig.5 に示す．DMI では，DMI 論理アドレス空間，DMI アドレス空間記述テーブル，DMI 仮想メモリのページテーブル，DMI 物理メモリをユーザレベルで実装している．DMI 論理アドレス空間とは，全 DMI プロセスに共通の論理アドレス空間であり，この上に DMI 仮想メモリが割り当てられる．DMI 論理アドレス空間にはメモリ領域としての実体は存在しない．DMI アドレス空間記述テーブルとは，DMI 論理アドレス空間と DMI 仮想メモリのマッピングや各 DMI 仮想メモリのページサイズなどを記述した表であり，これは全 DMI プロセスが同一内容のものを保持する．DMI 仮想メモリはページの集合から構成されるが，これらページの実体は，いずれかの DMI プロセスの DMI 物理メモリ上に割り当てられている．なお，コンシステンシ維持のプロコトル上，あるページが複数の DMI プロセスの DMI 物理メモリに存在することはありうるが，どの DMI プロセスの DMI 物理メモリにも存在しないといういわゆるマッピング不在の状況は許さない．ページテーブルには，ページの状態やページのオーナープロセスなどのコンシステンシ管理のための情報の他，ページの実体がその DMI プロセス上の DMI 物理メモリに存在するならばその位置情報が記述されている．当然，ページテーブルの内容は全 DMI プロセスで異なる．DMI では，ページの状態を「リードもライトも可能」「リードのみ可能」「リードもライトも不可」の 3 状態で管理しているが，「リードもライトも可能」「リードのみ可能」の場合には，そのページの実体は必ずその DMI プロセス上の DMI 物理メモリに存在する．「リードもライトも不可」の場合には存在するとは限らない．

具体的な DMI のメモリ管理の動作を Fig.1 のプログラムに沿って説明する．まず，全 DMI プロセスが

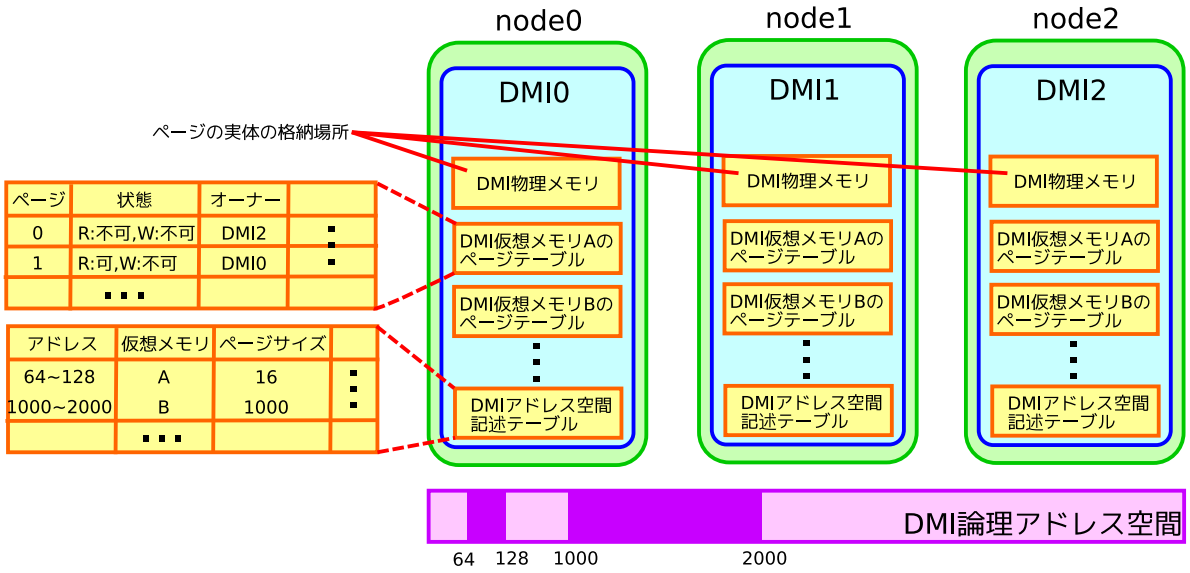

Fig.5　DMI のメモリ管理機構

`DMI_init(...,int size,...)` を呼ぶと，各 DMI プロセス上に `size` バイトの DMI 物理メモリが生成される．この `size` は DMI プロセスごとに異なって良い．

次に，全 DMI プロセスが `DMI_malloc(ull block, ull num, ull unit, ull *address)` を呼ぶと，ページサイズが `block` バイトでページ数が `num` 個の DMI 仮想メモリが生成される．この時点で実際に起きる作業は以下の通りである：

1. 全 DMI プロセスの DMI アドレス空間記述テーブルにこの DMI 仮想メモリを記録する．
2. 各 DMI プロセスの DMI 物理メモリ上に `unit` 個ずつのページをラウンドロビン方式で分散割り付けする．この分散割り付けは初期的なコールドミスに影響するに過ぎない．
3. DMI 仮想メモリのページテーブルを全 DMI プロセス上に生成し，適切な初期化を施す．

`address` には，生成した DMI 仮想メモリの DMI 論理アドレス空間におけるアドレスが入る．詳細は **3.5** で述べるが，DMI プロセス上の DMI 物理メモリにページを割り当てるだけの空き領域が存在しない場合，ページ置換のアルゴリズムを働かせて不要なページを追い出す．それでも割り当てられない場合には `DMI_malloc(...)` は失敗する．なお，各 DMI プロセス上の DMI 物理メモリ領域は `size` バイトの連続領域として確保する必要はなく，合計が `size` バイト以下ならばどこに確保しても良い．したがって，ページ置換を繰り替えす際にデフラグの問題は考えなくて良い．

`DMI_read(ull address, ull size, void *buf)` を呼ぶと，DMI 論理アドレス空間におけるアドレスが `address` の位置から `size` バイトの領域をバッファ `buf` に読み込む．この時点で実際に起きる作業は以下の通りである：

1. DMI アドレス空間記述テーブルを参照して操作対象の DMI 仮想メモリを特定する．
2. 特定した DMI 仮想メモリのページテーブルにおける該当ページの状態を参照し，「リードもライトも不可」ならば，コンシステンシプロトコルに基づいて他ノードにページ要求を発行し，やがて最新のページを受け取る．受け取ったページを DMI 物理メモリの空き領域に割り当てるとともにページテーブルを更新する．
3. 該当ページを `buf` にコピーする．

割り当てた DMI 仮想メモリを解放するには，全 DMI プロセスから `DMI_free(ull address)` を呼び出す．`DMI_final()` を全 DMI プロセスから呼び出すと DMI が確保した全リソースを解放する．

以上のような DMI のメモリ管理方式を，ほとんどの DSM が採用している OS のページフォルトをシグナルハン

ドラで捕捉する方式と比較した場合，利点としては，DMI 仮想メモリごとに任意にページサイズを設定可能なため，プログラムによってはページフォルトの発生回数を非常に抑制でき高い独立性を確保できる点，OS のアドレッシング範囲を意識しない記述が可能な点，非同期操作などの柔軟な API を提供可能な点などが挙げられる．その反面，`DMI_read(...)` および `DMI_write(...)` を発行する度に DMI アドレス空間記述テーブルやページテーブルへの参照が必要であり，また DMI のメモリ空間とユーザプログラムのメモリ空間の間でのメモリコピーが生じるため，アクセス時のオーバーヘッドが大きいのが欠点と言える．

3.4 コンシステンシ管理

DMI では，ページサイズを任意に指定可能とすることで性能上の問題をある程度補えると考えたため，プログラミング上のわかりやすさと負担減を優先して，コンシステンシモデルとして SC を採用した．DMI におけるコンシステンシ管理はページ単位で独立なため，スレッドや DMI の非同期操作を利用すれば複数のページに対するアクセス要求を並列に処理可能である．ページの状態の管理に関しては，「リードもライトも可能」な CLEAN，「リードのみ可能」な SHARED，「リードもライトも不可」な INVALID の 3 状態で管理する．オーナープロセスが変化中の状況を除けば，任意のページに対してオーナープロセスがちょうど 1 つ存在し，「CLEAN が 1 個，残りが INVALID」または「CLEAN が 0 個，SHARED が 1 個以上，残りが INVALID」を不変条件として保つ．CLEAN は必ずオーナー権を伴い，INVALID はオーナー権を伴わないが，SHARED はオーナー権を持つ場合と持たない場合が存在する．

DMI におけるプロトコルの実装は，Li らの提案する Dynamic Distributed Manager（DDM）のオーナー追跡型アルゴリズム [7] に近いが，あるページのオーナープロセスをプロセス i からプロセス j に変化させる際の挙動に関して，以下の点でより優れていると言える（Fig.6）：

- DDM では，プロセス i は常に最新のページをプロセス j に転送する．これに対して DMI では，プロセス j がすでに最新ページを保持している場合にはページの転送処理を省く．DMI ではページサイズが巨大になりうるため，これは有効と予想される．
- DDM では，そのページに関して SHARED な状態を持つプロセスリストをプロセス i がプロセス j に転送する．その後プロセス j がリスト内のプロセスたちに対して INVALID 要求を発行し，それらの応答を待つ．これに対して DMI では，プロセス i は，そのページに関して SHARED な状態を持つ（プロセス j 以外の）プロセスたちに対して，応答はプロセス j に返すようにという指示を付けて INVALID 要求を発行すると同時に，プロセス j に対しては届くはずの応答の数を伝える．これにより，プロセス i がプロセス j にプロセスリストを送信する遅延時間が省略される．

3.5 ページ置換アルゴリズム

一般的に，DSM ではページの状態に基づく優先度順でページの追い出しを行うのが効果的であるため [2, 9]，DMI では，INVALID，オーナー権を持たない SHARED，オーナー権を持つ SHARED，CLEAN の優先度順で以下のように追い出しを行う：

1. INVALID なページを追い出すには，単純にメモリを解放するだけで良い．
2. オーナー権を持たない SHARED なページを追い出すには，プロセス i はオーナープロセスに通知を送る．通知を受け取ったオーナープロセスは，そのページに関して SHARED な状態を持つプロセスリストからプロセス i を除外し，応答を返す．プロセス i は，応答を受け取った後でページの状態を INVALID に変えメモリを解放する．
3. オーナー権を持つページを追い出すには，適当なプロセス j を選び，プロセス j にプロセス i へ向けてライト違反要求を発行させることで，オーナー権をプロセス i からプロセス j に移動させる．その後ページの状態を INVALID に変えメモリを解放する．

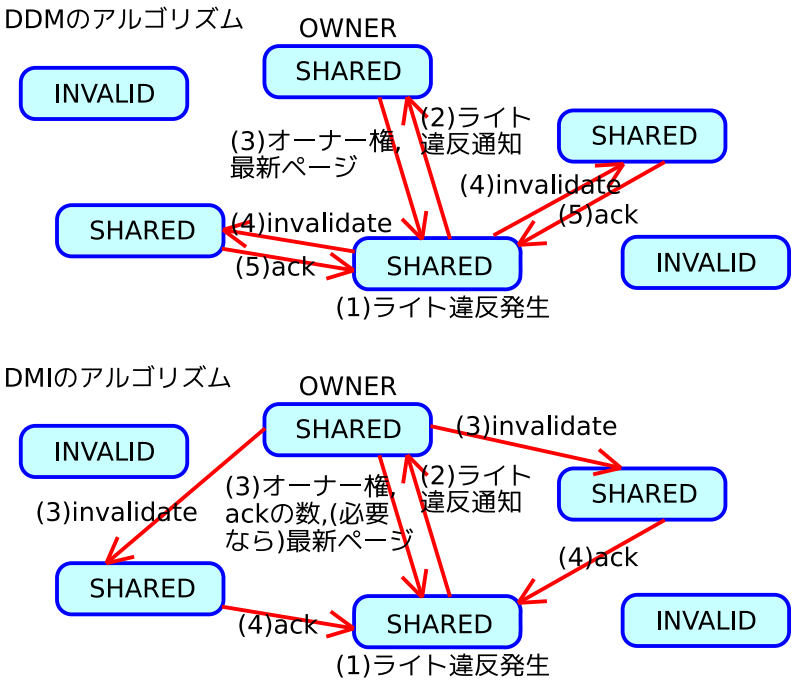

Fig.6　DDM と DMI における，オーナープロセス変化時の挙動の比較

4　予備的性能評価

4.1　実験

現状の DMI は，一部のインターフェースとページ置換を未実装であるが，予備的性能評価として，サイズが 2048×2048 の行列を用いた行列行列積 $AB = C$ を題材にして，MPI と DMI の処理性能を Xeon 2.40 GHz 2 コアノードを構成要素とする 1Gbit Ethernet 結合のクラスタ環境で比較した．OS は Linux 2.6.18-6-686，コンパイラは gcc 4.2.2 である．用いたアルゴリズムは以下のような単純な行分割法である：

1. プロセス 0 が A と B を初期化
2. 時間計測を開始
3. プロセス 0 が全プロセスへ A を（MPI 用語で言えば）scatter
4. プロセス 0 が全プロセスへ B を（MPI 用語で言えば）broadcast
5. 各プロセスは担当行に関して演算
6. 全プロセスからプロセス 0 へ C を（MPI 用語で言えば）gather
7. 時間計測を終了

DMI においては，A と C のページサイズを行ブロックのサイズに，B のページサイズは行列丸ごと 1 個に設定した．すなわち，各ノードの各ページに対してページフォルトは 1 回しか発生しない．DMI における各行列のページ状態の遷移図を Fig.7 に示す．

4.2　結果と分析

MPI と DMI のスケーラビリティの比較結果を Fig.8 に示す．Fig.8 には，MPI の結果（MPI）と DMI の結果（DMI）の他，MPI で B を broadcast する際に，`MPI_Bcast()` ではなく `MPI_Send()` による各プロセスへの逐次転送を使った場合の結果（MPI SEND）も載せている．

この結果より，DMI は 30 台弱までしかスケールせず，スケーラビリティの点で MPI に大きく劣っているが，その

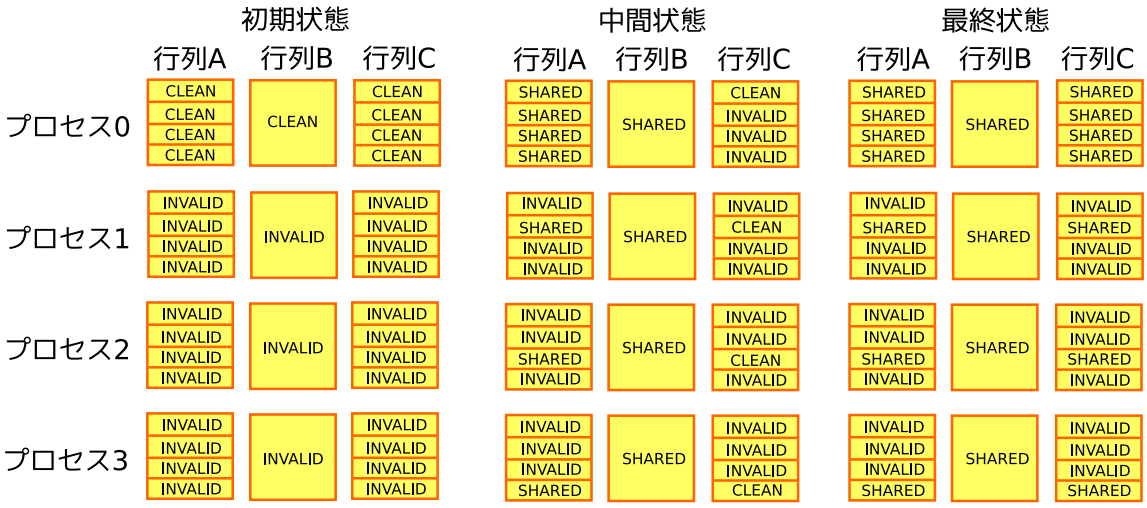

Fig.7　4 プロセスで行列行列積を実行した場合のページ状態遷移図

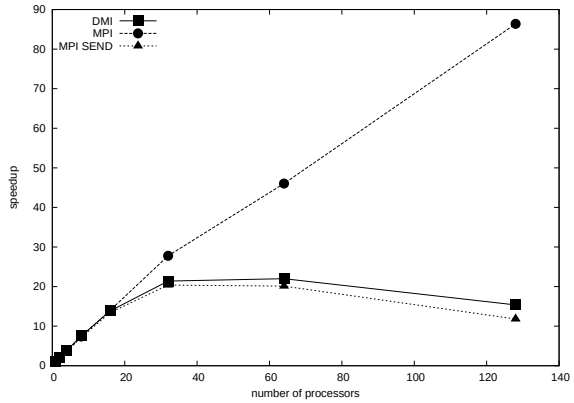

Fig.8　行列行列積における，MPI と DMI のスケーラビリティ

原因のほぼ全てが B の broadcast の実装に起因することが読み取れる．つまり，MPI では効率的な集合通信による `MPI_Bcast()` が行われる一方で，DMI ではプロセス 0 が全プロセスからのページ要求に対して逐次的に 32 MB のデータ転送を行うため，著しいボトルネックが生じている．事実，DMI で 50 プロセス以上の場合には，B が全プロセスに届き終わる前にすでに計算を終了するプロセスが出現した．以上より，DMI には，ページ転送時のボトルネックを回避するための動的負荷分散の導入が必須と言える．

5　DMI の機能拡張および最適化への提案

5.1　ノードの動的な増減

DMI プログラムはノードファイルを指定して `dmirun` を実行することによって起動されるが，DMI におけるノードの動的な増減は，このノードファイルの内容を DMI プログラムの実行中に外部コマンドを通じて編集することで実現する．また，DMI プロセスが増減した場合の挙動をプログラムに記述しやすくするため，各 DMI プロセスには，DMI プログラムの起動から終了までを通じてその DMI プロセスに一意なランクとは別に，その時点で実行されている DMI プロセスの集合の中でその DMI プロセスに一意な ID を割り振る．ID は常に，0 以上全 DMI プロセス数未満の値を取る．ノードの動的な増減に対応した DMI プログラムの例を Fig.9 に，増減時の DMI プログラムの挙動を Fig.10 に示す．

具体的な動作手順は次の通りである．`dmirun` を実行して DMI プログラムを起動すると，まず coordinator が立ち上がり，coordinator がノードファイル中の全ノードの daemon に対して DMI プロセスの生成を命じるが，こ

```
int main(int argc, char **argv) {
    int rank, id, pnum;

    DMI_init(&argc, &argv, 0, &rank, &id, &pnum);
    while(1) {
        if(DMI_polling()) {
            DMI_rebuild(&id, &pnum);
        }
        printf("rank = %d, id = %d, pnum = %d\n", rank, id, pnum);
        sleep(10);
    }
    DMI_final(); /* never reached */
    return 0;
}
```

Fig.9　ノードの動的な増減に対応したプログラム例

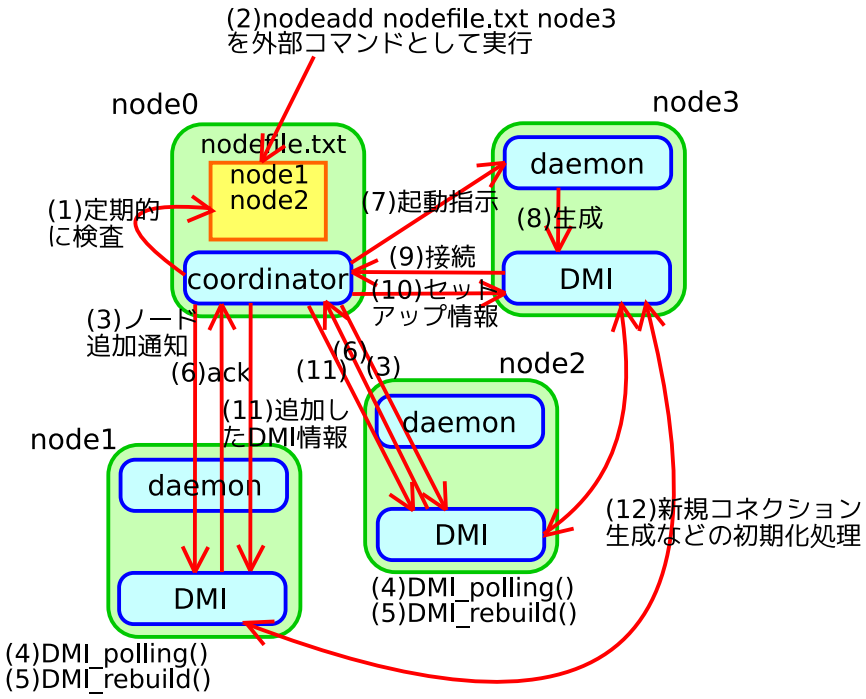

Fig.10　ノード追加時の DMI プログラムの挙動

の後 coordinator はノードファイルに変更がないかを定期的に検査する．一方，このノードファイルは，ノード追加用の `nodeadd` コマンドあるいはノード削除用の `nodedel` コマンドを外部から発行することで，coordinator のノードファイル検査とは排他的に書き換えることができる．たとえば，DMI プログラムの実行中にコマンドラインから `nodeadd nodefile.txt istbs[[000-191]]` を発行すれば，ノードファイル `nodefile.txt` に対して，`istbs000`，`istbs001`，…，`istbs191` の各ノードを追加する操作を，coodinator の検査とは排他的にアトミックに実現できる．さて，coordinator がノードファイルへのノードの追加または削除を検知した場合，coordinator は，その時点で走っている全 DMI プロセスに向けてノード変更要求を送信する．一方で，各 DMI プロセスは coordinator からノード変更要求が届いているか否かを `DMI_polling()` によって検査し，真が返されたならば `DMI_rebuild(int *id, int *pnum)` を呼び出す．全 DMI プロセスが `DMI_rebuild(...)` を呼び出すと，ノードの追加または削除を実現するための再編処理が開始される．

ノード追加時の再編手順は以下の通りである：

1. その時点の DMI プロセスの集合 X でメモリフェンスおよびバリアを張る．

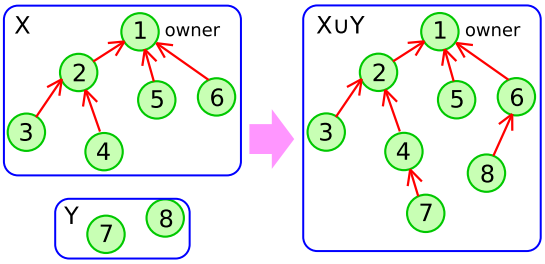

Fig.11　ノード追加時におけるオーナー追跡グラフの再構築

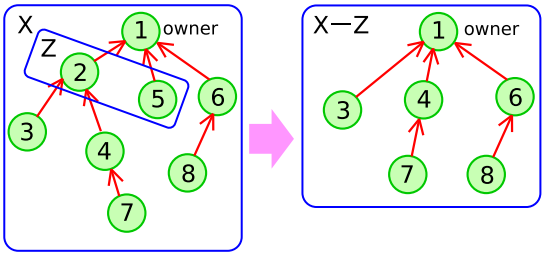

Fig.12　ノード削除時におけるオーナー追跡グラフの再構築

2. coordinator は，追加する各ノード上に新たな DMI プロセスを生成する．これら新たに生成された DMI プロセスの集合を Y とする．
3. 集合 Y 内の全 DMI プロセスが `DMI_init(..., int *rank, int *id, int *pnum)` を呼び出す．
4. 集合 $X \cup Y$ の間で新たに必要なコネクションを張るとともに，集合 Y 内の DMI プロセスの初期化処理を行う．具体的には，DMI 物理メモリのアロケートを行い，集合 X 内の適当な DMI プロセスから DMI アドレス空間記述テーブルとページテーブルをコピーした後，ページテーブル中の全ページの状態を INVALID に設定し，オーナープロセスの位置情報としては集合 X 内の任意の DMI プロセスを書き込む．これは，集合 X ではすでに到達可能条件を満たすオーナー追跡グラフが形成されているため，新規参入した集合 Y 内の DMI プロセスは集合 X 内の任意のプロセスへの有向枝を張ることで，集合 $X \cup Y$ で到達可能条件を満たすオーナー追跡グラフが形成可能だからである（Fig.11）．
5. 集合 $X \cup Y$ 内の DMI プロセスの `pnum` を $|X \cup Y|$ に更新する．集合 Y 内の DMI プロセスの `id` には $|X|$ 以上 $|X \cup Y|$ 未満の値を割り振り，`rank` には，その DMI プロセスが DMI プログラムが起動してから何番目に生成された DMI プロセスかを入れる．集合 X 内の DMI プロセスの `id` と `rank` は変化しない．
6. 集合 $X \cup Y$ でバリアを張る．

ノード削除時の再編手順は以下の通りである：

1. その時点の DMI プロセスの集合 X でメモリフェンスおよびバリアを張る．
2. ページ置換の要領で，削除する各ノード上の DMI プロセスの DMI 物理メモリ上にある全ページを追い出す．追い出し時にメモリ領域不足が発生した場合はエラーとする．これら削除対象の DMI プロセスの集合を Z とする．
3. この時点では集合 X で到達可能条件を満たすオーナー追跡グラフが形成されているため，ここで単純に集合 Z 内の DMI プロセスを脱退させるとグラフが崩れてしまう．そこで，集合 $X - Z$ 内の各 DMI プロセスは，オーナープロセスの位置情報として集合 Z 内の DMI プロセスが記述されている全てのページに関して，真のオーナープロセスを問い合わせ，位置情報を正しく更新する．これにより，集合 $X - Z$ で到達可能条件を満足するオーナー追跡グラフの再形成が完了する（Fig.12）．
4. 不要になるコネクションを切断した後，集合 Z 内の DMI プロセスを終了する．
5. 集合 $X - Z$ 内の DMI プロセスは `pnum` を $|X - Z|$ に更新し，`id` に `0...pnum-1` の値を割り振る．`rank` は変化しない．
6. 集合 $X - Z$ でバリアを張る．

以上の動的なノードの増減の応用として，クラスタ U のノードを記述したノードファイルを指定して `dmirun` を実行した後，ノードファイルにクラスタ V のノードを追加し，そしてノードファイルからクラスタ U のノードを削除すれば，クラスタ U からクラスタ V への計算資源のマイグレーションを実現できる．当然，DMI プロセス固有の変数や資源はマイグレーション時に失われるが，DMI のメモリ空間はマイグレーション前後で保存されるため，計算に必要なデータを DMI のメモリ空間経由で利用するようにプログラムを記述しておけばマイグレーションは成功する．

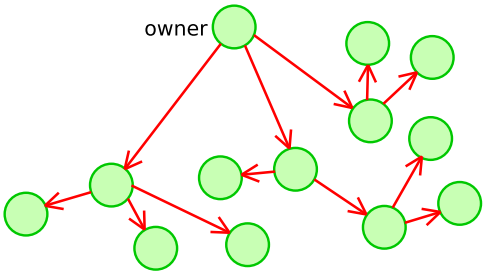

Fig.13　動的負荷分散を適用した場合の owner からのページ転送

5.2　ページ要求の動的負荷分散

DMI では，MPI と異なりその時点で発生している通信形態を把握できないため，ページサイズが比較的大きなページ要求がオーナープロセスに集中する場合，動的な負荷分散を行うことが望ましい．

たとえば，次のような手法が考えられる．オーナープロセスに多数のページ要求が到着した場合，オーナープロセスは最初に届いた log(全 DMI プロセス数) のページ要求だけを自分で処理する．つまり，要求元 DMI プロセスに対して逐次的にページ転送を行う．それ以降のページ要求に対しては，すでにページ転送を行った DMI プロセスのいずれかにその要求をフォワーディングし，実際のページ転送はその DMI プロセスに任せる．以上のアプローチを各 DMI プロセスに対して再帰的に適用すれば，結果的に，Fig.13 に示すような，構造化されたページ転送が実現できると予想される．具体的にどのようなアルゴリズムとパラメータ設定で負荷分散を行うのが最適なのかを，性能評価を通じて吟味したい．

5.3　マルチコア環境への適応

現状の DMI では，ノードファイルに同一ホスト名を複数記述することで，マルチコアノード上で複数の DMI プロセスを起動することが可能であるが，2 つの DMI プロセスが同一ノード内にあるか別ノードにあるかの区別をしていないため，マルチコアノードにおける物理的な共有メモリ機構を活かせていない．よって現状の DMI では，同一ノード内の DMI プロセス同士であってもソケット通信を行っている．そこで，DMI 物理メモリやページテーブルなどの DMI のメモリ管理用リソースを各ノードのプロセス間共有メモリ上に配置し，同一ノード内の複数の DMI プロセスから共同利用できるよう改善する．8 コアマルチプロセッサで構成されるクラスタ環境を用いて，この手法の有効性を評価したい．

6　おわりに

現状の DMI は，C 言語で 3000 行程度であり，一部のインターフェースとページ置換アルゴリズムを未実装である．今後，本研究は以下の順序で進めたい：

1. **3** 節で述べた DMI のモデルを完全に実装する．
2. ノードの増減に対応させ，計算環境の動的なマイグレーションを実現する．
3. ページ転送の動的負荷分散を実装し，典型的な通信パターンに対する効果を評価する．
4. プロセス間共有メモリを利用してマルチコア環境への適応を図り，その有効性を 8 コアノードで構成されるクラスタ環境上で検証する．

参考文献

[1] 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.

[2] Pradeep K.Sinha. *Distributed Operating Systems.* IEEE COMPUTER SOCIETY PRESS,IEEE PRESS, 1996.

[3] Ayon Basumallik and Rudolf Eigenmann. Towards automatic translation of openmp to mpi. *Proceedings of the 19th annual International Conference on Supercomputing*, 2005.

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

[5] 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.

[6] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. *In Proceedings of the Operating Systems Design and Implementation Symposium*, 1996.

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 1989.

[8] 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.

[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] 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.

[11] 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.