DMIの性能評価と今後の方向性

はらけん

2010.2.15

1　DMIが対象とする分散処理

DMIの最大の目的はノードの動的な参加/脱退のサポートであるため，ノードの動的な参加/脱退に相応して動的に並列度が増加/減少する処理でなければ，わざわざノードを参加/脱退させる意味がない．よって，DMIが対象とすべき分散処理は，数台程度までしかスケールしないような処理ではなく，ある程度のスケーラビリティを発揮できる処理である．したがって，DMIは，細粒度なデータアクセスが頻繁に干渉するような処理ではなく，データアクセスがある程度粗粒度でアクセス干渉の少ないような処理を対象として設計する．当然，このようにデータアクセスが粗粒度で干渉の少ない処理は，分散共有メモリにおけるプログラミングの容易さを持ち出さずとも，メッセージパッシングで十分記述できる場合が多い．しかし，それにも関わらず**本研究があえて分散共有メモリをベースとする理由は，分散共有メモリではデータの送受信が仮想共有メモリへのアクセスに抽象化されるためノードの参加/脱退を容易に記述できるからである**，という点を強調したい．

2　DMIのシステム概要（何を性能評価したのか）

スライド参照.

3　DMIの性能評価

■ 3-1　概要

3-3-1ではDMIにおけるread操作のオーバーヘッドを，3-3-2ではマルチモードread/writeの有効性を，3-3-3では遠隔スワップシステムとしての性能と非同期read/writeの有効性を，3-4-1，3-4-2，3-4-3では各種アプリケーションに対するDMIとMPIとの性能比較およびノードの動的な参加/脱退を行うことの有効性を，3-4-5では任意のページサイズを設定できることの有効性を検証する．3-4-4では，NAS Parallel Benchmarkに対する文句をまとめる．なお，DMIが採用するSequential ConsistencyやSingle Writer型のコンシステンシプロトコルは，既存の緩和型コンシステンシモデルやMultiple Writer型のコンシステンシプロトコルよりも並列性が絞られるため，データアクセスの競合が頻発する処理に対しては性能上不利になると考えられるが，これに関する評価は行えていない．ただし，前述のように，そのような処理の多くは台数効果が出にくくノードの参加/脱退による効果が期待できないため，DMIが主に対象とする処理ではない．

その他，DMIの特性を分析するために，ぜひ取得すべきデータがあれば教えてください．

■ 3-2　実験環境

実験環境としては，kyutechの16ノードを用いた．具体的な構成は，Intel Xeon E5410 2.33GHz（4コア）×2のCPU，32GBのメモリ，カーネル2.6.18-6-amd64のLinuxで構成されるマシン16ノードを1Gbitイーサネット×2でネットワーク接続した，合計128プロセッサのクラスタ環境である．以降の実験では，DMI/MPIをnプロセッサで実行する際には，8本のDMIスレッド/MPIプロセスを$\lfloor n/8 \rfloor$台のノードに立て，残りの$n - 8 \times \lfloor n/8 \rfloor$本のDMIスレッド/MPIプロセスを別の1

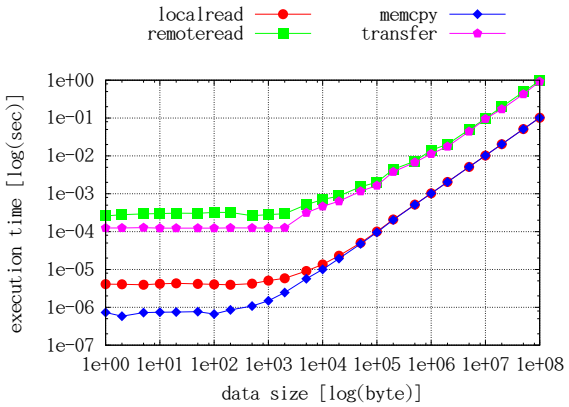

Fig.1　データサイズを変化させたときのローカル read（localread），リモート read（remoteread），memcpy（memcpy），データ転送（transfer）の実行時間.

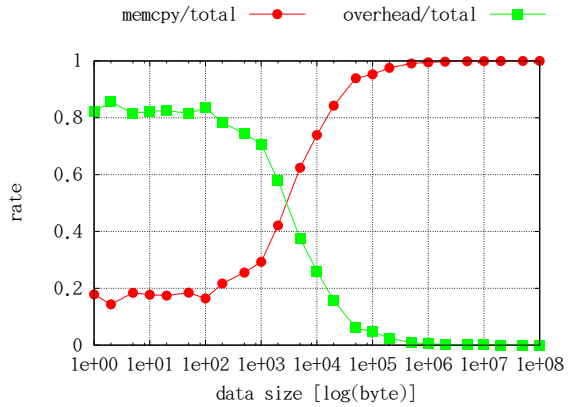

Fig.2　データサイズを変化させたときのローカル read の内訳.

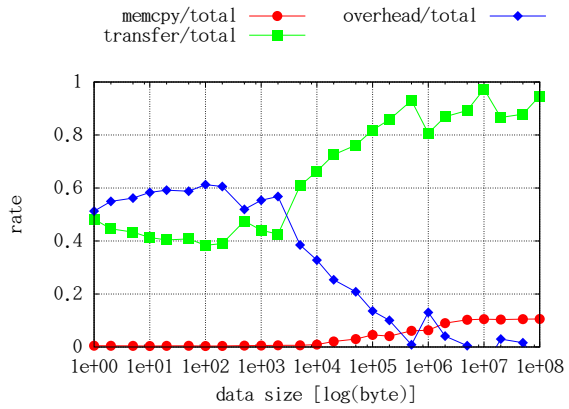

Fig.3　データサイズを変化させたときのリモート read の内訳.

つのノードに立てるプロセス構成とした．また，コンパイラには gcc 4.1.2，MPI には OpenMPI 1.3.3，最適化オプションには-O3 を使用した.

■ 3-3　マイクロベンチマーク

3-3-1　read 操作の性能

DMI における read のオーバーヘッドを評価した．DMI における read には，そのノードがすでにキャッシュを保有していてローカルに完了する場合（ローカル read）と，オーナーに対して read フォルトを送信して最新ページの転送を要求する場合（リモート read）の 2 種類がある．ローカル read の処理の内訳は，DMI 物理メモリのメモリ空間からユーザプログラムのメモリ空

間への memcpy と，ユーザレベルによるコンシステンシ管理などの処理系のオーバーヘッドである．一方，リモート read の処理の内訳は，オーナーからのページ転送と，memcpy と，処理系のオーバーヘッドである．なお，DMI の read/write には，複数ページにまたがってもよく，いかなるタイミングで呼び出しても安全に処理される `DMI_read(...)`/`DMI_write(...)` と，単一ページにしかアクセスできず，良からぬタイミング（たとえばメモリ解放中など）で呼び出すとシステムが未定義状態に陥る可能性がある一方で高速に実行可能な `DMI_oneread(...)`/`DMI_onewrite(...)` が存在する．これらの実行速度としては，8 バイトのローカル read に関して，`DMI_oneread(...)` は `DMI_read(...)` よりも約 20 倍高速だった．以下の実験は，`DMI_oneread(...)` を用いて行ったものである．

Fig.1 には，さまざまなデータサイズ x に関して，ページサイズ x のページ 1 個をローカル read するのに要する時間，およびリモート read するのに要する時間，サイズ x のデータの memcpy に要する時間，サイズ x のデータの転送に要する時間を比較した結果を示す．また，ページサイズを変化させたときに，ローカル read 全体の実行時間に占める memcpy の比率およびオーバーヘッドの比率を Fig.2 に，リモート read 全体の実行時間に占めるページ転送の比率，memcpy の比率，およびオーバーヘッドの比率を Fig.3 に示す．なお，これらのグラフには 100 回測定した場合の平均値をプロットしているが，1KB 以下の場合の memcpy やページ転送はあまりに高速過ぎるため，測定ごとのばらつきが非常に大きく，正確な数値を読み取ることにはあまり意味がない．

Fig.1 より，リモート read はローカル read よりも，2MB 以上では 9.52 倍〜9.96 倍遅く，10KB 以下では 50.8 倍〜81.4 倍遅いことがわかる．ローカル read に関して Fig.2 の内訳を見ると，500KB 以上では処理系のオーバーヘッドが占める比率が 1% 以下となるものの，100 バイト以下では 80% 以上を占めることがわかる．しかし，ローカル read では，(1) ページテーブルから該当ページの管理情報を取り出し，(2) そのページに対する他の操作を排他し，(3) アクセスチェックを行い，(4) memcpy し，(5) 排他を解除するという操作を行うのみであり，これらはユーザレベルでコンシステンシ管理を行う以上は最低限必要になる操作であるため，やむを得ない．また，リモート read に関して Fig.3 の内訳を見ると，100KB 以上ではオーバーヘッドの比率が 15% 以下に抑えられるものの，2KB 以下では 50% 以上を占めており，今後ネットワーク性能が向上すれば，さらにこの比率は増加すると予測される．以上より，現状の DMI は細粒度なアクセスが頻発する処理には弱くオーバーヘッドの低減が重要な課題と言えるが，最初に述べたように，そのような処理の多くは台数効果が出にくくノードの参加/脱退による効果が期待できないため，DMI が主に対象とする処理ではない．

3-3-2 マルチモード read/write の性能

DMI では，read/write/fetch-and-store/compare-and-swap をベースとした，共有メモリベースの Permission Word アルゴリズムに基づいて排他制御を実装している．Permission Word アルゴリズムを pthread 型のインタフェースに従って書き換えたものを以下に示す：

```
01: struct mutex_t {
02:   int *head;
03:   int *next;
04:   int *p1;
05:   int *p2;
06: };
07:
08: void init(struct mutex_t *mutex) {
09:   mutex->head = NULL;
10:   mutex->next = NULL;
11:   mutex->p1 = NULL;
12:   mutex->p2 = NULL;
13: }
14:
15: void lock(struct mutex_t *mutex) {
16:   int flag;
17:   int *prev, *curr;
18:
19:   flag = 0;
20:   curr = convert(&flag, mutex->p1, mutex->p2);
                          /* address conversion */
21:   prev = fetch_and_store(&mutex->head, curr);
22:   if(prev == NULL) {
```

```
23:     mutex->p1 = curr;
24:   } else {
25:     while(flag == 0); /* spin */
26:   }
27:   mutex->next = prev;
28: }
29:
30: void unlock(struct mutex_t *mutex) {
31:   int *curr;
32:
33:   if(mutex->next == NULL
         || mutex->next == mutex->p1) {
34:     if(mutex->next == mutex->p1) {
35:       mutex->p1 = mutex->p2;
36:     }
37:     if(!compare_and_swap(&mutex->head, mutex->p1, NULL)) {
38:       mutex->p2 = mutex->head;
39:       curr = revert(mutex->p2); /* address reversion */
40:       *curr = 1;
41:     }
42:   } else {
43:     curr = revert(mutex->next); /* address reversion */
44:     *curr = 1;
45:   }
46: }
47:
48: void destroy(struct mutex_t *mutex) {
49: }
50:
51: int* convert(int *curr, int *p, int *q) {
52:   int v1, v2, d;
53:
54:   v1 = (intptr_t)p & 0x3;
55:   v2 = (intptr_t)q & 0x3;
56:   d = 0;
57:   if(d == v1 || d == v2) {
58:     d = 1;
59:     if(d == v1 || d == v2) {
60:       d = 2;
61:     }
62:   }
63:   return (int*)((intptr_t)curr + d);
64: }
65:
66: int* revert(int *curr) {
67:   return (int*)((intptr_t)curr - ((intptr_t)curr & 0x3));
68: }
```

この実験では，上記の排他制御アルゴリズムを題材にして，マルチモード read/write の有効性を評価した．アルゴリズムの詳細は（あまりにややこしいので）理解する必要はないが，重要なポイントだけ抑えておくと，構造体 `mutex_t` のメンバ変数に対して，23 行目，27 行目，35 行目，38 行目で write を，21 行目で fetch-and-store を，37 行目で compare-and-swap を行い，20 行目の直前，33 行目の直前，38 行目の直前で read を行っている．そして，DMI ではアクセス競合時の性能低下を防ぐために，全ての write と fetch-and-store と compare-and-swap を WRITE_REMOTE モードで，全ての read を READ_ONCE モードで行っている．この理由は，write などを WRITE_LOCAL モードで行うと，排他制御の度にオーナーが変化するため，アクセス競合時にオーナーの頻繁な移動が発生してオーナー追跡のための通信が多量に発生するためである．また，全ての read を READ_ONCE モードで行う理由は，アクセス競合時にはデータを read してから次に read するまでの間に他ノードによってそのデータが更新される可能性が高いため，READ_INVALIDATE モードや READ_UPDATE モードによってデータをキャッシュすることに意味がない上に，多量の invalidate 要求や update 要求が発生してしまうためである．

以上のようなマルチモード read/write の使い分けの効果を検証するため，全ての write と fetch-and-store と compare-and-swap を X モードで，全ての read を Y モードで行う場合の性能を，(I) X =WRITE_REMOTE，Y =READ_ONCE,

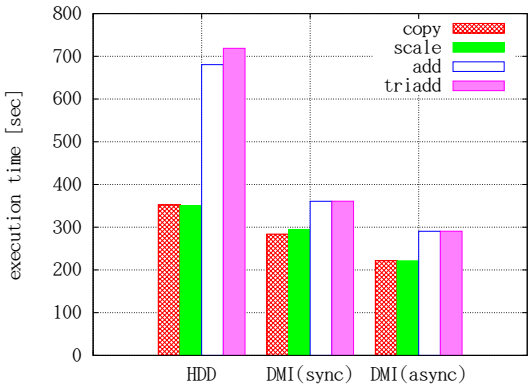

Fig.4 HDD アクセスの実行時間，STREAM ベンチマークにおける `DMI_read(...)/DMI_write(...)` の実行時間（DMI(sync)），非同期 `DMI_read(...)/DMI_write(...)` の実行時間（DMI(async)）．

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

このマルチモード read/write は，この排他制御の例のようにある限られた場面においてのみ効果が出るような最適化手段ではなく，**かなり広い場面において実用的に有効な最適化手段**である．論文の査読者からも，このマルチモード read/write には一定の独自性があると思われるので，もう少し強調してもいいのではないかというコメントをいただいた．

3-3-3 非同期 read/write を利用する遠隔スワップシステムの性能

STREAM ベンチマークで採用されている copy，scale，add，triadd の各処理に関して，非同期 read/write を利用する遠隔スワップシステムの性能と HDD アクセスの性能を比較した．STREAM ベンチマークは，本来 CPU のメモリバンド幅を測定するためのベンチマークであり，キャッシュに乗り切らない程度に大きい 3 つの配列 A, B, C と定数 d を用意して，copy（$C = A$），scale（$B = dC$），add（$C = A + B$），triadd（$A = B + dC$）の演算を行う時間を計測する．つまり，主記憶への連続アクセスを通じて，CPU の byte/flops を求めるのが STREAM ベンチマークの目的である．

DMI の遠隔スワップシステムにおける評価では，16 ノードを使用し，各ノードが提供する DMI 物理メモリ量を 2GB に設定し，3 つの 8GB 配列 A, B, C をページサイズ 1MB で DMI 仮想共有メモリ上に確保した．そして，各配列 A, B, C に関して，先頭から $i \times 512$MB 以上 $(i+1) \times 512$MB 未満 $(0 \leq i < 16)$ の領域が，ノード i にはオーナー権を伴った DOWN_VALID 状態で存在し，他のノードには INVALID 状態で存在するように配列領域を分散配置した上で，配列 A から配列 C への 8GB の copy をノード 0 が逐次で行う実行時間を測定した．同様にして，測定の度に前述の分散配置をやり直し，ノード 0 が逐次で scale，add，triadd を行う実行時間を測定した．この実験では，各配列の read 操作には READ_ONCE モードを，各配列の write 操作には WRITE_LOCAL モードを使用した．なお，各ノードが提供する DMI 物理メモリ量を 2GB に設定しているため，この処理では絶えずページの追い出し処理が発生する．

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

ただし，この非同期 read/write に関しては，有効利用できる場面がそれほど多くないと思われる．MPI の非同期 send/recv にも言えることだが，非同期操作は，計算と通信がちょうど良いバランスになるような処理でないとその効果が見えにくいため，有効性が確認できるアプリが限られる．事実，今回の性能評価で試した処理の中で，非同期操作の有効性が明らかに見えたのはこの実験だけである．

また，今のところ，既存の遠隔スワップシステムとの相対比較は行えていないが，おそらく（というか間違いなく）既存の遠隔スワップシステムと比較すると DMI の性能は著しく低いと予想される．その理由は主に 2 点ある．第一の理由は，DMI のように，OS のメモリ保護機構を利用することなくユーザレベルで遠隔スワップを行うのは，遠隔スワップの本来の目的とずれており，あまりに遅いからである．遠隔スワップの本来の目的は，ネットワークの高速化を背景として，メモリ階層における主記憶とディスクスワップとの間に，遠隔メモリを挟み込むことにある．よって，もっとも正道な遠隔スワップの実現方法は，カーネルモジュールとしてデバイスを実装し，アプリケーションからは完全に透過的に遠隔スワップを実現する方法である．事実，Anemone や Nswap など，性能を意識した遠隔スワップシステムではこのような実装を取るのが主流であり，最近では，Infiniband や Myrinet 環境での効率的な遠隔スワップの実装に関する研究が盛り上がっているようである．これらの研究に対して，DLM は，「わざわざデバイスを作ることなくユーザレベルで実装しても性能はあまり変わらない」という主張のもとで，OS のメモリ保護機構を利用して実装しているが，評価結果を見る限り，やはり遅い印象がある（相対評価を行っていないので実際のところはよくわからない）．いずれにせよ，遠隔スワップはこういうレベルで論じられるべき話なので，DMI のようにユーザレベルでごちゃごちゃ行うような実装では，性能面ではまず対抗できない．第二の理由は，遠隔スワップシステムでは，高性能な大規模メモリの提供が最大の焦点であるため，主に逐次プログラムの実行が対象とされる．つまり，逐次プログラムに特化することにより，分散共有メモリのような複雑なコンシステンシ管理を一切排除し，プロトコルを単純化することで効率化が図られている．よって，分散共有メモリの実現を主目的とする DMI における遠隔スワップシステムが，それら逐次プログラム用に最適化された遠隔スワップシステムと比較して遅いのはやむを得ない（当然，DMI では並列実行環境が提供されるので，処理を並列化することで「全体として見れば」性能向上できるような場面はあるのかもしれないが，少なくとも遠隔スワップシステムとしての read/write 速度では太刀打ちできない）．分散共有メモリに遠隔スワップシステムの機能を付加させた既存研究には，JIAJIA や Cashmere-VLM などがあるが，これらの研究も遠隔スワップシステムとしての read/write の性能を競っているわけではなく，分散共有メモリ + 遠隔スワップシステムという処理系が実現できたことを強調している．以上を踏まえると，DMI における遠隔スワップシステムは，JIAJIA や Cashmere-VLM などの既存研究の二の舞に過ぎず，性能面で有利なアプローチも特に入れていないため，これ以上踏み込まない方が無難だと考えている．

■ 3-4 アプリケーションベンチマーク

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

3-4-1 ヤコビ反復法による 3 次元熱伝導方程式の求解

この実験は Himeno ベンチマークを都合よく改造したものである．

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

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

の更新を行い，$\sum_{x=1}^{n}\sum_{y=1}^{n}\sum_{z=1}^{n}|T_i(x,y,z) - T_{i-1}(x,y,z)|/n^3 < 10^{-5}$ を満たすまでイテレーションを繰り返した．DMI では，各イテレーションの先頭でノードの参加/脱退を処理し，その時点で参加しているノードたちで z 軸方向に領域を均等に分割して，そのイテレーションを実行するようなプログラムを記述した．各イテレーションでは，各プロセッサが自分の担当領域の境界面の温度を計算する際に，直前のイテレーションで隣のプロセッサが更新した領域を参照する必要が生じるため，この境界面が 1 ページになるようにページサイズを設定した．また，自分の担当領域の更新には WRITE_LOCAL モードを利用し，境界面の計算時に直前のイテレーションで隣のプロセッサが更新した領域を参照する際には READ_INVALIDATE モードを使用した．この理由は，仮に担当領域の更新に WRITE_REMOTE モードを使用してしまうと，第 i イテレーションの先頭でノードの参加/脱退が発生して領域に対するプロセッサ割り当てが変化した場合に，第 i イテレーション以降では他のプロセッサがオーナー権を持つページに対する更新作業が常に必要になり，多量の通信が生じてしまう（Fig.5(A)）．これに対して，WRITE_LOCAL モードを使用すれば，第 i イテレーションの先頭でノードの参加/脱退が発生して領域に対するプロセッサ割り当てが変化したとしても，第 i イテレーション終了時には，その時点でのプロセッサ割り当てに一致したページの

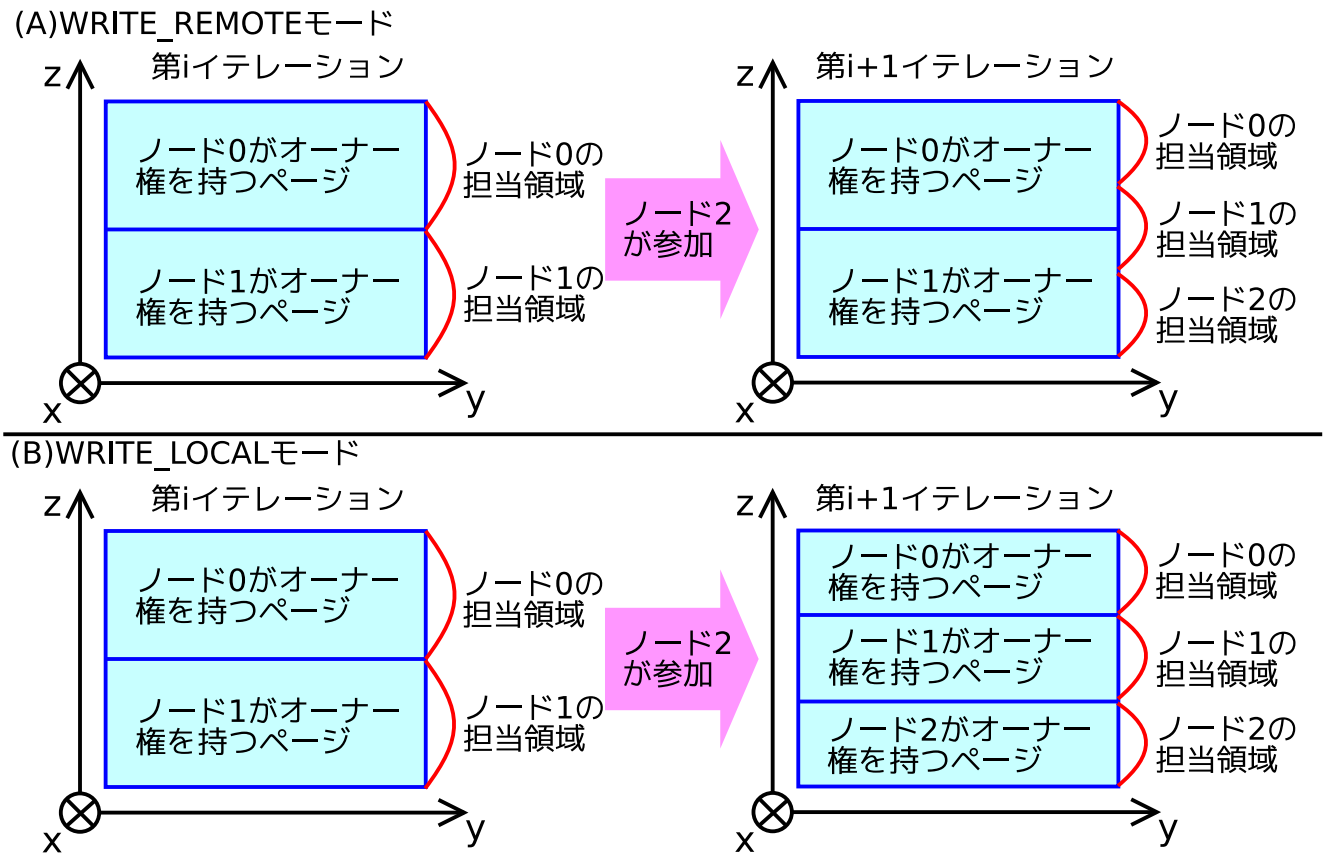

Fig.5 ヤコビ反復法においてプロセッサ数を増減させた場合における，領域に対するプロセッサ割り当ての変化（(A) 領域更新に WRITE_REMOTE を使用した場合，(B) 領域更新に WRITE_LOCAL を使用した場合）.

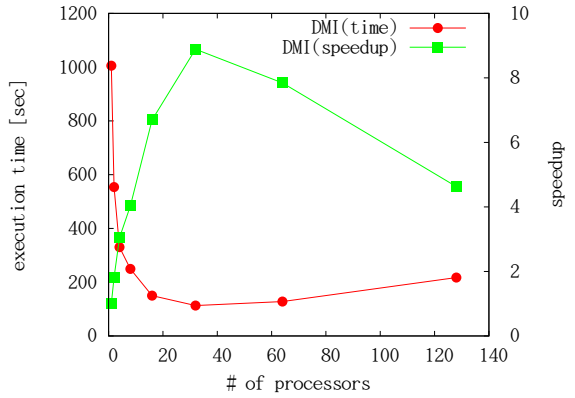

Fig.6 ヤコビ反復法に関する，DMI の実行時間とスケーラビリティ.

オーナーの割り当てが実現されるため，第 i イテレーション以降における更新作業の対象は自分がオーナー権を持つページになる（Fig.5(B)）．このように，マルチモード read/write をうまく利用することで，**データへのプロセッサ割り当てが重要となるデータパラレルなアプリケーションに関して，動的な参加/脱退に伴うプロセッサ割り当ての変化にロバストなプログラムを記述することができる**.

まず，Fig.6 には，さまざまなプロセッサ数に関する DMI の実行時間（DMI(time)）と速度向上度（DMI(speedup)）を示す．Fig.6 より，32 プロセッサまでスケールしており，32 プロセッサで 8.90 の速度向上度を達成している．次に，ノードの動的な参加/脱退の効果を調べるため，最初は 8 プロセッサで実行し，第 50 イテレーション終了時に 56 プロセッサを追加し，第 85 イテレーション終了時に 48 プロセッサを脱退させた場合の，各イテレーションの実行時間を Fig.7 に示す．Fig.7 より，ノードの動的な参加/脱退に伴って動的に並列度を増減させられていることがわかる．また，第 51 イテレーションと第 86 イテレーションの実行時間が長いのは，更新領域に対するプロセッサ割り当ての変化に伴って，WRITE_LOCAL モードによる更

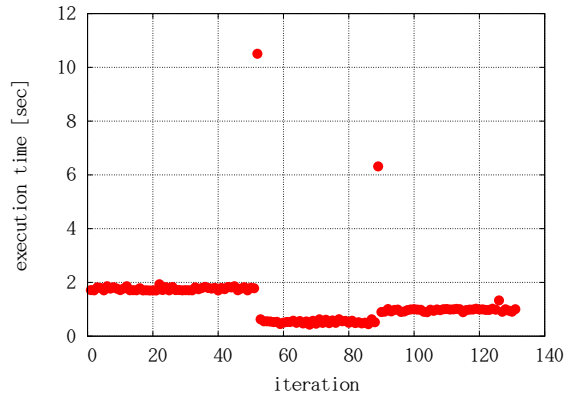

Fig.7 ノードを動的に参加/脱退させる場合の，ヤコビ反復法の各イテレーションの実行時間.

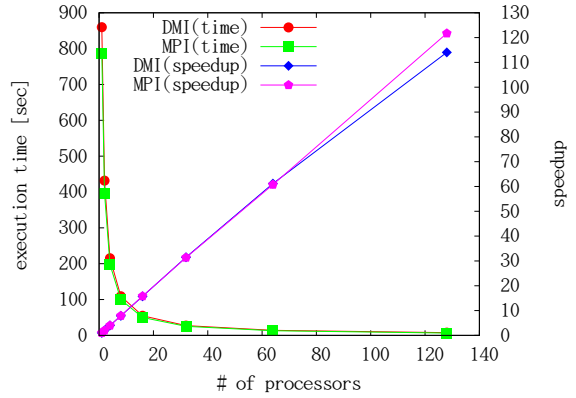

Fig.8 EP に関する，DMI と MPI の実行時間とスケーラビリティ.

新時に，最新のページ転送を伴ったオーナー権の移動が多量に発生するためである.

現段階ではこの実験が最大のウリで，「以上のように，単純なクライアント・サーバ方式では記述できないような，プロセッサが密に協調しながら動作するアプリケーションに対しても計算資源の動的な参加/脱退をサポートし，参加/脱退に相応して動的に並列度を増減できる処理系は新規性の高いものである．この結果は，従来の処理系では計算資源の動的な参加/脱退をサポートできなかったアプリケーション領域に対しても，DMI によるアプローチが応用できる可能性を示唆している.」というのが主張である.

3-4-2 NAS Parallel Benchmark（EP）

NAS Parallel Benchmark における EP（Class C）を題材にして，DMI と MPI のスケーラビリティを比較するとともに，DMI においてノードの動的な参加/脱退に伴って並列度を動的に変化させる効果を評価した．なお，NAS Parallel Benchmark は，本家サイトに置いてある MPI プログラムをそのまま実行させて，システムアーキテクチャや MPI の性能評価に使う場合が多いが，本来はシステムアーキテクチャの評価を目的としたベンチマークであり，「どのように実装すると効率的かはシステムアーキテクチャに依存するため，ベンチマークの仕様は “paper and pencil” としてしか規定しない（規定すべきではない）．ただ，それだけだと実装者が困るだろうから，あくまでもサンプルとして MPI などのプログラムも参考までに提供しておく」というスタンスのベンチマークである．たとえば，IS ならば，「要するにソートできればいい」という程度しか仕様として規定されておらず，ソートのアルゴリズムなどは実装者の判断に委ねられている．つまり，仕様を満たす限り，DMI にとって都合がいいアルゴリズムで実装した DMI プログラムと本家サイトで配布されている MPI プログラムを比較することは，妥当な評価と言える.

EP（Class C）は，$n = 2^{32}$ 個の乱数の組 (x_i, y_i) を生成し，$t_i = x_i^2 + y_i^2 \leq 1$ なる (x_i, y_i) に関して $\max(|x_i\sqrt{(-2\log t_i)/t_i}\,|, |y_i\sqrt{(-2\log t_i)/t_i}\,|)$ の分布を求める処理である．DMI では，ノードの動的な参加/脱退に対応させるため，$n = 2^{32}$ を 128 個の均等なタスクに分割し，単純なマスタ・ワーカ方式によってプログラムを記述した．一方，MPI では，NPB3.3-MPI のプログラムをそのまま評価に用いた.

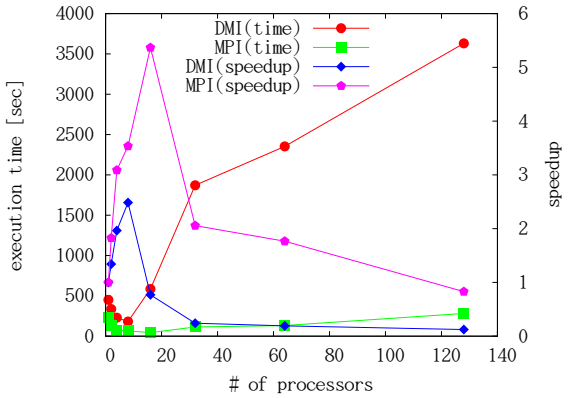

Fig.9 CG に関する，DMI と MPI の実行時間とスケーラビリティ．

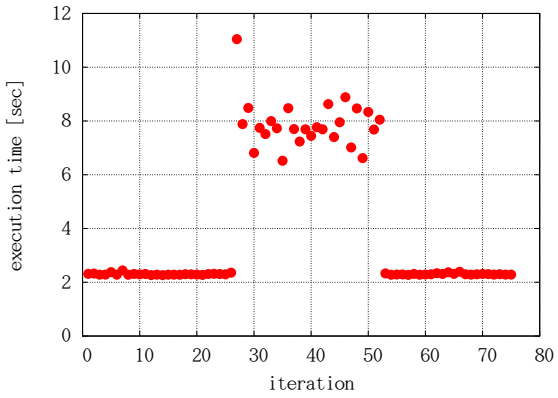

Fig.10 ノードを動的に参加/脱退させる場合の，CG の各イテレーションの実行時間．

まず，Fig.8 には，さまざまなプロセッサ数に関する DMI の実行時間（DMI(time)）と速度向上度（DMI(speedup)），MPI の実行時間（MPI(time)）と速度向上度（MPI(speedup)）を示す．この処理はほとんど通信を伴わないため，DMI は MPI とほぼ同等のスケーラビリティを達成している．DMI の実行時間が MPI よりも長いのは，演算部分に関するコンパイラの最適化などが原因で，MPI のプログラム（Fortran）では，DMI のプログラム（C 言語）よりも効率的な実行バイナリが生成されたためと思われる．次に，ノードの動的な参加/脱退の効果を調べるため，(I) 最初から最後まで 32 プロセッサで実行する場合，(II) 最初は 32 プロセッサで実行するが，約 10 秒後に 96 プロセッサの参加を開始させる場合，(III) 最初は 32 プロセッサで実行するが，約 10 秒後に 24 プロセッサの脱退を開始させる場合について実行時間を計測した．なお．この実験では 10 秒後に参加/脱退を開始させただけであって，実際に参加/脱退が完了するまでには数秒を要しており，10 秒を境目として一気にプロセス数が増減したわけではない．その結果，(I) が 27.3sec，(II) が 17.5sec，(III) が 47.8sec となり，このような embarrassingly parallel な処理では，ノードの参加/脱退に伴って，効果的に並列度を増減させられることがわかった．

3-4-3 NAS Parallel Benchmark（CG）

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

手順 1 $\boldsymbol{x} = {}^t(1, 1, \cdots, 1)$

手順 2 $A\boldsymbol{z} = \boldsymbol{x}$ を CG 法で解く．具体的には，$\boldsymbol{z} = 0$，$\boldsymbol{r} = \boldsymbol{x}$，$\rho = (\boldsymbol{r}, \boldsymbol{r})$，$\boldsymbol{p} = \boldsymbol{r}$ とした上で，以下の処理を 25 回繰り返す．

$$\boldsymbol{q} = A\boldsymbol{p}$$
$$\alpha = \rho/(\boldsymbol{p}, \boldsymbol{q})$$
$$\boldsymbol{z} = \boldsymbol{z} + \alpha\boldsymbol{p}$$
$$\rho_0 = \rho$$

$$\boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{q}$$
$$\rho = (\boldsymbol{r}, \boldsymbol{r})$$
$$\beta = \rho/\rho_0$$
$$\boldsymbol{p} = \boldsymbol{r} + \beta\boldsymbol{p}$$

手順 3 $\boldsymbol{x} = \boldsymbol{z}/||\boldsymbol{z}||$

手順 4 手順 2 と手順 3 を 75 イテレーション繰り返す.

DMI では，手順 2 と手順 3 を構成する各イテレーションの先頭でノードの参加/脱退を処理し，その時点で参加中のノードたちで行列とベクトルを均等に横ブロック分割し，そのイテレーションを行うようなプログラムを記述した．また，この処理で最大のボトルネックになるのは手順 2 の CG 法内部で行われる行列ベクトル積 $A\boldsymbol{p}$ であるが，この行列ベクトル積を，行列 A の横ブロック分割によって最大 128 プロセッサで並列演算することを想定し，行列 A のページサイズは 75000 × 75000/128 × sizeof(double) バイトとし，各ベクトルのページサイズは 75000/128 × sizeof(double) バイトに設定した．一方，MPI では，NPB3.3-MPI のプログラムをそのまま評価に用いた.

まず，Fig.9 には，さまざまなプロセッサ数に関する DMI の実行時間（DMI(time)）と速度向上度（DMI(speedup)），MPI の実行時間（MPI(time)）と速度向上度（MPI(speedup)）を示す．Fig.9 を見ると，DMI は 8 プロセッサまでしかスケールしておらず，および 16 プロセッサ以上では MPI よりも著しく実行時間が長い．この原因は，プロセッサ数を p，行列サイズを n としたとき，行列ベクトル積に関して，MPI では総通信量 $O(\sqrt{p}\,n)$ の洗練されたアルゴリズムが採用されているのに対して，DMI では簡単化のため，行列の横ブロック分割による総通信量 $O(n^2)$ のアルゴリズムを採用してしまったためである．次に，ノードの動的な参加/脱退の効果を調べるため，上記手順の手順 2 と手順 3 の 75 イテレーションのうち，最初は 8 プロセッサで実行し，第 26 イテレーション終了時に 8 プロセッサを参加させ，第 51 イテレーション終了時に 8 プロセッサを脱退させた場合の，各イテレーションの実行時間を Fig.10 に示す．Fig.10 ではノードの参加によって性能が落ちており，当然ながら，このようなスケーラビリティの悪い処理は参加/脱退に適さない.

そして，当然ながら，MPI と DMI という 2 つの処理系の比較を行いたいにも関わらず，違ったアルゴリズムの実装で比較してしまったのは，明らかな失敗である（詳しくは次節）.

3-4-4 NAS Parallel Benchmark の問題点（やや余談）

本当は，NAS Parallel Benchmark に数種類取り組んでみたかったのだが，不都合な点が多すぎて，かなりの時間を費やしたものの EP と CG しか評価できなかった．また，そもそも DMI と MPI の処理系の比較を行いたいのならば，アルゴリズムや言語は極力統一する必要があるが，配布されている MPI プログラムが Fortran で書かれていたり，アルゴリズムが理解できなかったりしたため，結局 EP も CG も統一できない状態での評価しかできなかった.

NAS Parallel Benchmark には以下のような問題点がある：

- CG では疎行列 A が定義される必要があるが，NAS Parallel Benchmark の仕様書には疎行列 A の生成アルゴリズムが最後まで説明されておらず，「詳しくは配布されている Fortran を読め」と書かれている．結局アルゴリズムはわからなかったが，Fortran を改造して，生成された疎行列 A をファイルに書き出して使うことで解決した.
- CG 法を解く部分の Fortran のアルゴリズム（前述の「総通信量 $O(\sqrt{p}\,n)$ の洗練されたアルゴリズム」）が理解できず，DMI に真似できていない.
- IS に関して，仕様書に書かれている結果が間違っている.
- MG に関して，アルゴリズムの肝心な部分が仕様書に書かれておらず（僕には）実装できない.
- EP の要求している解精度が不適切なように思われる．現状の DMI の EP は仕様書の解精度を満たしていないが，見なかったことにした.
- CG を mpich-1 で 32 プロセッサ以上で実行すると，途中で p4_error が出てエラーになってしまう．ネットで調べると，同様の現象は他でも起きているようだが，明確な原因はわからなかった．OpenMPI を使うと問題が起きないため，今回の性能評価は全て OpenMPI で行った．なお，藤澤くん，中島くん，加辺くんに協力してもらった実験により，これは NAS Parallel Benchmark の問題ではなく，mpich-1 における非同期 recv 周辺の実装のバグの可能性が高いと判断している（気になる方はあとで聞いてください）．中島くんからも mpich-1 の挙動不審が指摘されている．mpich-1 は Debian の apt-get で入るものだが，mpich-1 は 2005 以降更新されておらず，バグフィックスが行われていない可能性がある.

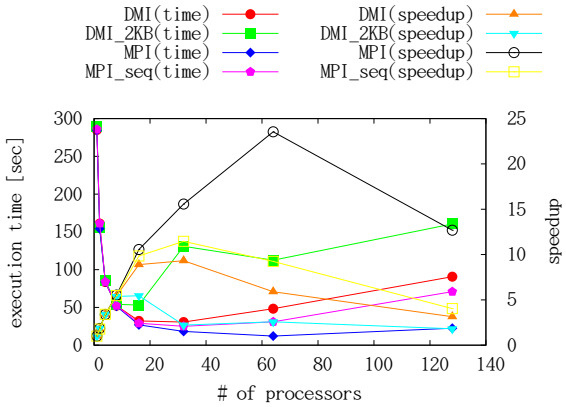

Fig.11 行列行列積に関する，DMI と MPI の実行時間とスケーラビリティ．

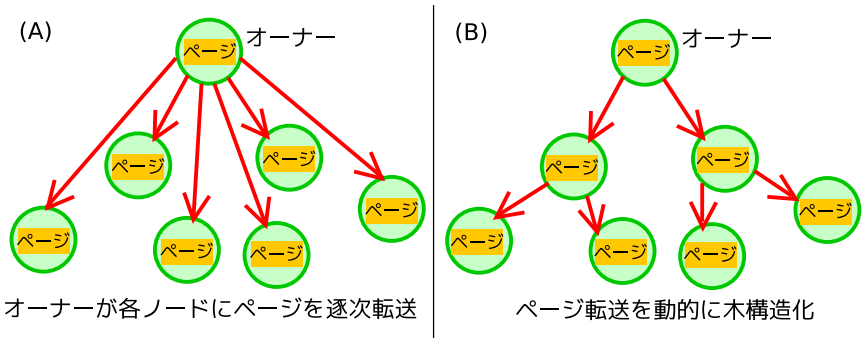

Fig.12 ページ転送の木構造化（(A)read 要求に対してオーナーがページを逐次転送する場合，(B) ページを木構造転送する場合）．

3-4-5 行列行列積

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

(1) プロセス 0 が行列 A をプロセス数分だけ横ブロック分割し，それを全プロセスに scatter する．
(2) プロセス 0 が行列 B を broadcast する．
(3) 各プロセス i はプロセス 0 から送信された横ブロック部分行列 A_i と行列 B を用いて，部分行列行列積 $A_iB = C_i$ を計算する．
(4) 各プロセス i は部分行列 C_i をプロセス 0 に gather する．

DMI においても，read/write ベースで記述することを除けば，MPI と同様のデータ操作が起きるアルゴリズムで記述した．まず，Fig.11 には，さまざまなプロセッサ数に関して以下を測定した結果を示す：

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

Fig.11 より，DMI_2KB の実行時間およびスケーラビリティが DMI_normal より大きく劣っており，任意のページサイズ指定によってページフォルト回数を大幅に削減できることの有効性がわかる．ここで，通常の OS のページサイズである 4KB

でなく 2KB を採用した理由は，なぜか，4KB 以上だと顕著な性能低下が見られなかったためである．2KB 以下にすると，著しい性能低下が見られるようになり，1KB にすると 10 分以上待っても処理が終わらなかった．また，DMI_2KB の実行時間のグラフは，3 回の実行時間の平均値としてプロットしているが，通信があまりにも多発するせいか，測定の度に実行時間が大きく変化するため，グラフの線が上下している．これらの現象の詳細な解析はまだできていない．

この任意のサイズによるコンシステンシ維持は，**粗粒度なデータアクセスを伴うデータパラレルなアプリ一般に対して有用な最適化手段**と言える．現状の DMI の場合，オーバーヘッドの低減を度外視してしまっているため，ページ 1 個を管理するための構造体のサイズが最小でも 200 バイトあり，パケット 1 個が最小でも 104 バイトあるため，コンシステンシ維持の単位を大きくすることの効果が表れやすい．

また，Fig.11 より，DMI_normal のスケーラビリティは MPI_normal よりも大きく劣るが，DMI_normal と MPI_normal と MPI_seq の結果を比較すると，その原因の大部分が MPI の集合通信に起因しているとわかる．特に影響が大きいのは行列 B の broadcast であり，現状の DMI では，Fig.12(A) のように，ページをキャッシュするノード（DOWN_VALID または UP_VALID なノード）を常にオーナーの直下に配置する構造になっているため，read 要求を発行してきたノードたちへの最新ページの転送がオーナーによって逐次化され性能低下につながっている．これに対する解決策としては，Fig.12(B) に示すように，オーナーに read 要求が到着した際には，オーナーが，すでにページ転送を完了したノードに対して実際のページ転送処理を動的に委譲することによって，ページ転送を全体として木構造化させるなどの工夫が考えられる．

4 今後の方向性

■ 4-1 「実用的な」アプリをクラスタ間でマイグレーション

現状の DMI は，全対全のコネクションを張ったり，参加/脱退時にグローバルロックを取得するなど，各ノードが「大局的な」知識に基づいて動作するため，まだ大規模な環境で動作する設計にはなっていない．これを改善し，**「局所的な」知識に基づいて動作するプロトコルを作り上げるのが DMI の最終目標**であるが，現状のままでも，効率は悪いかもしれないが，計算を継続したまま，参加/脱退を通じてクラスタからクラスタに処理をマイグレーションすることは原理的に可能である．もちろん，処理系を評価する上では NAS Parallel Benchmark の EP で評価しても良いのだが，やはりインパクトに欠ける．そこで，**従来のクライアント・サーバベースの処理系では，クライアントは自由に参加/脱退できてもサーバは自由に参加/脱退できなかった「実用的な」アプリケーションに対して，DMI ではそれもできる**ということを示したい．

具体例としては，ユーザからの重いリクエストを受け付けるサーバノード数個と，そのリクエストを処理する計算ノード多数個から構成される，クラスタ A 上の Web サーバシステムが考えられる．クラスタ A 内の計算ノードを動的に参加/脱退させると Web サーバシステム全体のスループットが増減するのが観測でき，さらにクラスタ A で計画停電が起きる際には，Web サーバシステムの運用を継続したまま，サーバノードも含めてクラスタ B に移動できたらおもしろいと思う．（当然，サーバノードの IP アドレスが変わった場合に，ユーザがどうやって新しいサーバノードに接続するかなどの現実的な問題はあるが，Web サーバシステムとしての運用が継続するという部分が実現できればそれで良いとする．）

何か，他におもしろそうなアイディア（画像処理？，自然言語？，・・・etc）があれば教えていただけるとうれしいです．以下の条件を満たすことが理想です：

- （動的な並列度変化が見れないとおもしろくないので,）本質的に embarrassingly parallel な処理であること
- （ファイルに頼っては意味がないので,）メモリ上の処理で完結する処理であること
- （DMI のプログラムは，既存プログラムを少し改造することでは得られず，まず pthread で記述して動作を確認し，それを `DMI_read(...)`/`DMI_write(...)` に翻訳するという非常に面倒な作業を行う必要があることを踏まえ,）DMI で記述できるレベルに簡単な処理であること
- 少しは実用的と思える処理であること

■ 4-2 粗粒度なデータアクセス中心の処理に対する最適化手法の評価

DMI は，従来の多くの分散共有メモリとは少し性格が違う部分がある．

僕が知る限り，従来の多くの分散共有メモリにおける最適化手法では，データアクセスが細粒度でアクセスが頻繁に干渉するような処理に対する研究が中心である．多様な緩和型コンシステンシモデルや Multiple Write 型のプロトコルなどは，細粒度なアクセスが競合する際に並列性を確保するための工夫と捉えられる．また，分散共有メモリならではのプログラム記述の容易さも重要視され，page-based な分散共有メモリであれば，OS のメモリ保護機構を利用することで，通常の変数代入文で仮想共有メモリにアクセス可能とするのが主流である．

これに対して DMI は，そもそもの目的は参加/脱退のサポートであり，**分散共有メモリを利用する理由は，参加/脱退を容易にサポートするためという部分にしかない**．また，参加/脱退に相応して動的に並列度を増減させることが目的なので，データアクセスが粗粒度なスケーラブルな処理が対象である．それゆえ，プログラム記述の容易さも重視しておらず，`DMI_read(...)`/`DMI_write(...)` によるプログラム記述は，まさに MPI 関数的なノリであって，プログラマに対して明示的なチューニング手段を提供して性能を引き出すことを目指している．要するに，（かなり大げさな言い方だが,）DMI は，従来の分散共有メモリがあまり相手にしてこなかったようなスケーラブルな処理の性能向上を意図した分散共有メモリと言え，その意味で，DMI をベースとして**データアクセスが粗粒度な処理に対する分散共有メモリの最適化手法というシナリオ**を考えることには多少の新規性があるように思う．具体的には，任意粒度でのコンシステンシ維持，非同期 read/write，マルチモード read/write，ページ転送の木構造化の 4 つを提案している．このうち，任意粒度でのコンシステンシ維持はすでに CRL や HIVE などで提案されており，region-based な手法として定着しているので新規性はない．しかし，非同期 read/write とマルチモード read/write は，`DMI_read(...)`/`DMI_write(...)` という関数呼び出し型のプログラム記述を採用しているからこそ実現できる機能である（非同期 read/write に似たものはあるが）．また，ページ転送の木構造化に関しても，調べた限りでは例を見ない．

以上を踏まえて，データアクセスが粗粒度な処理に対する分散共有メモリの最適化手法というシナリオで，もう少し評価を進めてみたいと思う．上記の 4 手法のうち，まだ未実装なのはページ転送の木構造化なので，とりあえず木構造が m 分木になるように実装してみて，m による効果を調べる予定である．現在の DMI は，$m = \infty$ としたものと捉えられる．なお，現状の DMI では，オーナーからのメッセージを受信側で順序制御しており，オーナー発のメッセージの転送経路は問題にならないため，ページ転送の木構造化は現在のプロトコルに単純に組み込める（予定である）．