社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2006－EVA－16（7）
2006／3／20

DIMMnet-2 用 Windows ドライバの実装と評価

金井　遵[†]　森　拓郎[†]　荒木健志[†]
田邊　昇[††]　中條拓伯[†]　並木美太郎[†]

本論文では，大容量バッファを持った高速なネットワークインターフェースである DIMMnet-2 を用い，Windows 上で複数の DIMMnet-2 をまとめて一つの分散ファイルシステム（DFS）や分散共有メモリ（DSM）として利用するドライバとライブラリを設計，実装し，評価を行った．本方式により，OS に手を加えることができないコモディティOS 環境下で DFS や DSM などの PC クラスタ環境を実現することが可能となる．評価では，行列乗算，Wisconsin Benchmark，姫野ベンチ等の分散処理による評価を行った．特に行列乗算による評価では，2 ノードの分散処理において 1.99 倍の性能向上が確認できた．

Implementation and Evaluation of DIMMnet-2 Device Drivers for Microsoft Windows

Jun Kanai ,[†] Takuro Mori ,[†] Takeshi Araki ,[†]
Noboru Tanabe ,[††] Hironori Nakajo[†] and Mitaro Namiki [†]

In this paper, we have designed, implemented and evaluated a Distributed File System(DFS) device driver and a Distributed Shared Memory(DSM) library by plural high-speed network interface cards named DIMMnet-2 with mass buffer for Microsoft Windows. This method realizes implementation of PC Cluster System with DFS and DSM for a non-open source commodity OS. We have evaluated this cluster system by matrix multiplication evaluation, Wisconsin Benchmark, Himeno Benchmark and so on. As a result of matrix multiplication evaluation, up to 1.99 times higher performance has been gained by 2-nodes distributed parallel execution.

1. はじめに

近年，HPC（High Performance Computing）の分野において，多数の PC を相互に接続した PC クラスタシステムの躍進は目覚しい．現在のクラスタシステムの環境はほとんどが Unix をベースとしたものである．しかし，今や Microsoft Windows のシェアは 97%以上に達し，今後の分散処理においても重要な存在になり得ると考えられ，Windows 環境下での分散コンピューティングの可能性を示すことは，ソースコードがなく，OS のカーネルに手を加えることができないコモディティOS 環境での分散処理の可能性を示すことにもなる．

PC クラスタの躍進を背景に，安価にシステムを構築できる DIMM スロットへハードウェアを接続する HPC 用高速ネットワークインタフェース DIMMnet-2[4] が開発された．DIMMnet-2 は従来の汎用バスに接続するタイプの HPC 用 NIC に比べ 1/10 程度のアクセスレイテンシ，最上位レベルの帯域幅を実現している．また，DIMMnet-2 は，SO-DIMM による数百 MB～数 GB の大容量バッファを持ち，通信用バッファやデータ待避用領域として利用することが可能である．

しかし，DIMMnet-2 はシステムソフトウェア面では，Linux 用にコマンド発行のための低レベルなドライバがあるのみで，Windows をはじめとしたコモディティOS 環境下の分散処理環境が整っていない．そこで本稿では，DIMMnet-2 を用い，Windows 向けに DIMMnet-2 用デバイスドライバを設計および実装することで，OS に手を加えることなく，コモディティOS とメモリマップによってクラスタシステムを構築し，評価を行う．

2. DIMMnet-2 プログラミングの問題点

本章では，従来のクラスタシステムで用いられるデータ共有方法であるメッセージパッシングと分散共有メモリ (DSM) について述べ，DIMMnet-2 とコモディティOS 環境下での実現方法について考察する．

メッセージパッシングは，データ共有を行うために明示的にデータのやりとりを行う方法であり，現在，HPC において最も多く利用される分散データによるプログラミング方法である．メッセージパッシングシステムの実装例として，MPICH[1] などが存在する．

MPI をはじめとしたメッセージパッシングでは，一般的に性能が優れるという利点があるが，一方でデータの送受信について明示しなければならないため，プログラマにとって負担が大きい．メッセージパッシングを実現する場合，帯域や遅延が最重要視されるため，OS を介する

† 東京農工大学
Tokyo University of Agriculture and Technology
†† (株) 東芝, 研究開発センター
Corporate Research and Developpment Center, Toshiba

ことによるオーバヘッドを極力減らす必要があり，ユーザランドで処理を行うのが望ましい．そのためには既存の Linux 用 DIMMnet-2 ドライバにおいては各バッファ (ウィンドウ) やレジスタをユーザ空間にマップした上で，DIMMnet-2 のウィンドウを用いた複雑な間接アクセス機構を意識しながら，明示的にコマンド発行を記述する必要があり，プログラミングの手間がかかる．

一方，分散共有メモリ方式 (DSM) は分散メモリ型マシンにおいて，実際に共有メモリがなくても共有メモリが存在するように見せかける技術である．DSM 方式では，明示的にデータのやりとりを記述する必要がないため，プログラミングが容易である．一方で，ページフォールトや通信回数，一貫性制御の負荷の問題によって，メッセージパッシング型よりも性能が劣ることが多い．

DSM 方式では，TreadMarks[2] や SCASH[3] のようにプロセッサの Memory Management Unit (MMU) のページフォールト機能を用いて DSM を実現する例が多いが，UDSM[5] のようにユーザモードのソフトウェアのみで解決した例も存在する．

従来の MMU を利用する DSM での問題は，今回のように OS に手を加えることが出来ない環境下で利用できないこと，ユーザモードのみで DSM を実現する場合の問題は MMU が利用できずオーバヘッドが発生することである．そこで，本稿ではメモリマップトファイルと DFS により，OS に手を加えることができない環境下で間接的に MMU を利用して DSM を実現する方法を提案する．

メモリマップトファイルは仮想アドレス空間にファイルをマップする方法であり，MMU を利用して実装される．DFS 上に共有データを置き，メモリマップトファイルを利用することで，OS に手を加えずに MMU を利用して，DSM の実現ができる．共有メモリを作成した場合に自動的に必要なページのみがロード，ダーティページのみがライトされるため，通信頻度の最適化や DSM 管理用ソフトウェアによるオーバヘッドの削減が可能である．また，各ノードでキャッシュを持ち，速度面で優位である．

また，本方式で DSM を実現する場合，ドライバレベルで DFS を実装するのみで良く比較的実装が容易であると考える．DIMMnet-2 においては大規模な SO-DIMM によるバッファを持ち，ネットワークを利用し，広帯域・低遅延な SO-DIMM 内のデータの送受信が可能である．このバッファを DFS や DSM の記憶領域とする DFS や DSM が本方式に適すると考える．

3. 本研究の目標

本研究では Windows とメモリマップおよび DIMMnet-2 を利用して，クラスタシステムを構築し，評価を行う．

そこで，まず DIMMnet-2 の大容量バッファを RAM ディスクのディスク領域として利用する分散ストレージドライバ AT-dRAM を開発し，DSM を実現し評価を行う．また，従来の主記憶の代替手段として，実用的な速度を実現できるかを検証する．次に，通信をユーザモードのみで完結し，高速なメッセージパッシングシステムを実現や，DIMMnet-2 の拡張機能の利用を可能にするためのドライバ MT-dNET を開発し，有用性について検証する．

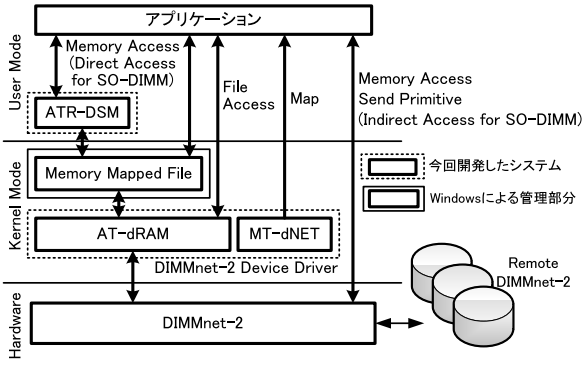

図 1 システム全体構成図
Fig. 1 System Architechture

4. クラスタシステム用ドライバの概要

前章で述べた考察をもとに，DIMMnet-2 を DFS，DSM として利用するための RAM ディスクドライバ AT-dRAM および，DIMMnet-2 の各バッファやレジスタをユーザ空間にマップし，ユーザモードのみで DIMMnet-2 を直接操作するためのドライバ MT-dNET の設計，実装について述べる．今回開発したシステムの全体構成を図 1 に示す．

AT-dRAM は，DIMMnet-2 の大容量バッファをディスク領域とする RAM ディスクドライバであり，DIMMnet-2 の煩雑な間接アクセス機構をプログラマに対して隠蔽するとともに，クラスタシステムで多く利用される DSM，DFS を実現する．AT-dRAM では，複数の DIMMnet-2 を一つのディスクとして仮想化し，リモートの DIMMnet-2 の大容量バッファへのアクセスを隠蔽する．これにより，通常のファイルアクセス手段により，ローカルとリモートの DIMMnet-2 大容量バッファへのアクセスが可能になる．複数のノードにおいてドライバをロードすることで，DFS としての利用も可能である．また，従来の DIMMnet-2 プログラミングでは DIMMnet-2 の間接アクセス機構を意識する必要があったが，メモリマップトファイルの利用により，従来の主記憶を利用するアルゴリズムがそのまま適用可能になり，DIMMnet-2 プログラミングが容易となる．さらに，複数のノードから同一のファイルをマップすることにより，DSM としての利用が可能である．

分散処理を行う場合に，より実用的な DSM とするためには，共有メモリ領域の作成やバリア同期等の機能が必要になる．そのため，これらの機能をサポートするライブラリ ATR-DSM も同時に開発した．厳密な DSM とする場合，一貫性制御が必要になるが，今回の場合は性能を最重視し，ライブラリでの一貫性制御は行わず，明示的にデータのフラッシュ，更新を行う仕組みとした．主な関数を表 1 に示す．

一方で，メッセージパッシングシステムの実装，ストライド命令をはじめとした各種拡張命令の利用時など，OS によるオーバヘッドを減らし，明示的に DIMMnet-2 を操作したい場合が存在する．そこで，応用プログラムからユーザモードのみで DIMMnet-2 の明示的な操作を可能にするドライバ MT-dNET を開発した．

```
mat_a = (CAST)dn_malloc(MATSIZE);
mat_b = (CAST)dn_malloc(MATSIZE);
mat_c = (CAST)dn_malloc(MATSIZE);
//通常の主記憶利用と同じ方法
for (i = 0; i < SIZE; i++)
  for (j = 0; j < SIZE; j++)
    for (k = 0; k < SIZE; k++)
      (*mat_c)[i][j] += (*mat_a)[i][k]
                        * (*mat_b)[k][j];
```

図 2 行列乗算例 (AT-dRAM とメモリマップトファイル利用)
Fig. 2 Example for DIMMnet-2 Programming(AT-dRAM)

```
for (i = 0; i < SIZE; i++){
  //連続ロード命令で列読み込み
  VL(mat_a, i * step, step);
  for (j = 0; j < SIZE; j++){
    //ストライド命令で行読み込み
    VLS(mat_b, MATSIZE + j * szdbl, 3, SIZE, step);
    for (k = 0; k < SIZE; k++)
      mat_c[k] += mat_a[k] * mat_b[k];
  }
  VS(mat_c, MATSIZE * 2 + i * step, step);
}
```

図 3 行列乗算例 (MT-dNET とライブラリによる明示発行)
Fig. 3 Example for DIMMnet-2 Programming(MT-dNET)

MT-dNET では各種ウィンドウやレジスタをプロセスのユーザー空間へマップする機能を提供する．これにより大容量バッファへのアクセスや，リモートノードの DIMMnet-2 へのアクセスをユーザモードで完結することが可能になり，OS 呼出しによるオーバヘッドをなくし，通信の遅延を極力おさえることが可能である．実際に DIMMnet-2 にアクセスするためには，表 2 のようなライブラリを利用し，明示的にアクセスを行う必要がある．

5. プログラミング例

5.1 ローカルでのプログラミング例

AT-dRAM と，MT-dNET およびユーザモードで動作するライブラリを使い，行列の乗算を行う場合のプログラミング例を図 2，3 に示す．

例より，AT-dRAM では DIMMnet-2 の間接アクセス機構を意識することなく，従来とほぼ同じアルゴリズムが適用可能であり，プログラミングが容易である．また，ファイルシステムキャッシュが利用可能なため，DIMMnet-2 へのアクセスが最低限に抑えられ，性能向上が期待できる．

一方で MT-dNET とストライド命令を使った例では，

表 1 DSM ライブラリ関数一覧
Table 1 Functions of DSM Library

関数名	機能
dn_malloc/dn_free	DSM 領域の作成・解放を行う
dn_flush	変更したデータを書き込む
dn_update	データを最新に更新する
dn_barrier	バリア同期をとる

表 2 DIMMnet-2 アクセス関数 (抜粋)
Table 2 Functions of DIMMnet-2 Access Library

関数名	機能
VL/VS	ローカルの SO-DIMM に連続アクセスを行う
VSS/VLS	ローカルの SO-DIMM にストライド (等間隔) アクセスを行う
RVL/RVS	リモートの SO-DIMM に連続アクセスを行う

```
dn_init(argc, argv);
size = MATRIX_SIZ * MATRIX_SIZ * sizeof(double);
if(dn_getrank() == 0){ //親ノードの場合:行列生成
  mat_a = (CAST)dn_malloc("mat_a", size);
  mat_b = (CAST)dn_malloc("mat_b", size);
  mat_c = (CAST)dn_malloc("mat_c", size);
  make_matrix();
  //書き込み
  dn_flush(mat_a); dn_flush(mat_b); dn_flush(mat_c);
  dn_barrier();
}else{
  dn_barrier();
  mat_a = (CAST)dn_malloc("mat_a", size);
  mat_b = (CAST)dn_malloc("mat_b", size);
  mat_c = (CAST)dn_malloc("mat_c", size);
}

step = MATRIX_SIZ / dn_getnodesum();
start = step * dn_getrank();
for(i = start; i < start + step; i++)
  for(j = 0; j < MATRIX_SIZ; j++){
    for(k = 0, s = 0.0; k < MATRIX_SIZ; k++){
      s += (*mat_a)[i][k] * (*mat_b)[k][j];
    (*mat_c)[i][j] = s;
  }
dn_flush(mat_c); dn_barrier();
```

図 4 行列乗算例（分散処理）
Fig. 4 Example for DIMMnet-2 Distributed Programming

DIMMnet-2 とのデータのやりとりを明示的に記述する必要があるが，ストライド命令により，キャッシュを有効に使ったプログラミングが可能である．今回は，プリミティブ発行にライブラリを使用しているが，MT-dNET のみでは DIMMnet-2 のウィンドウサイズの考慮，ステータスレジスタの監視などが必要になり，さらに煩雑になる．

5.2 分散共有メモリプログラミング例

ATR-DSM を用いた行列乗算の分散処理例を図 4 に示す．OS が MMU を用いることにより，AT-dRAM を介して必要なページのみがロードされ，ダーティページのみが書き込まれるため，通信効率の最適化が可能である．

6. 評　　価

本章では，基本データの計測を行うとともに，実際に分散処理による性能測定を行った．評価環境を表 3 に示す．

6.1 OS による各種オーバヘッド

AT-dRAM はストレージシステムという形で実装されているため，ドライバ呼出し等のコストが発生する．このオーバヘッドを計測するため，まず API を利用して 200[MB] のファイルをシーケンシャルにディスクに読み書きする時間を，一度の API 呼び出しで読み書きするサイズを変えて計測した．結果を図 5 に示す。

図 5 より，ディスクアクセスする総サイズが一定の時，総時間 T_{all} は API 呼び出し回数に比例することが分かる．よって，ここでは下記式を用い，明示的に API を利用してファイルアクセスを行った場合の OS によるオーバヘッドを計算した．ここで，T_{os} は OS によるオーバヘッド，N_{call} は API による OS 呼出し回数の実測値，N_{size}

表 3 評価環境
Table 3 Environment for evaluation

CPU	Intel Pentium4 2.4GHz
Memory	512MB(DDR-SDRAM PC1600)
OS	WindowsXP
開発環境	WindowsXP DDK+VisualStudio.NET

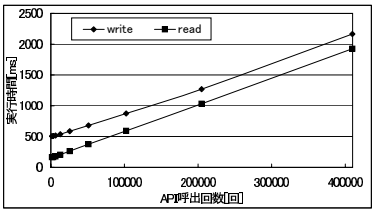

図 5 ディスクアクセス時間
Fig. 5 Disk Access Time

は一度に転送するサイズの実測値，T_{TR} は一度の転送にかかる時間の実測値である．総時間と API 呼び出し回数の差，$\Delta_{T_{all}}$，$\Delta_{N_{call}}$ を求めることにより，1 回の OS 呼出しにかかるコストを計算している．

$$T_{all} = T_{os}N_{call} + N_{size}T_{TR} \quad (1)$$

$$T_{os} = \Delta_{T_{all}}/\Delta_{N_{call}} \quad (2)$$

続いて，メモリマップトファイルを用いてファイルアクセスを行った場合の OS のオーバヘッド T_{mmap} を計算した．読み込みの場合，ページ毎（4KB）にページフォールトが発生し，ストレージドライバが呼び出され，該当ページのデータがディスクから読み込まれる．

$$T_{mmap} = (T_{all} - T_{TR}N_{size})/N_{page} \quad (3)$$

ディスクからの読み込みにかかる時間は (1) 式から算出可能であり，これを利用し，(3) 式からメモリマップトファイルを利用した場合のページ単位の OS のオーバヘッドを計測した．T_{TR} は単位あたりのディスクからの読み込み時間の実測値を，N_{size} の実測値は読み込んだサイズの合計を，N_{page} はページ数の計算値をそれぞれ表す．

メモリマップトファイル利用時ののデータ書き込みは基本的に OS により自動的に行われるが，明示的に書き込みを指示することもでき，ATR-DSM ではこちらを主に利用する．この場合のオーバヘッドを (2) 式により計算した．これらの計算に使用した T_{all} の実測値を表 4 に，オーバヘッドをまとめたものを表 5 に示す．

メモリマップトファイルを利用した場合では，ReadFileAPI を利用した場合に比べ，オーバヘッドが大きくなっているが，メモリマップトファイルでは最初の一回のアクセスのみでこのオーバヘッドが発生するため，明示的に複数回 ReadFile を呼ぶような場合に比べると，性能的に有利である．

表 4 OS オーバヘッドデータ (120MB のファイルアクセス時間)[ms]
Table 4 OS Overhead Parameters for File Access

	1KB	4KB	16KB
Read(ReadFile)	1033	377	205
Write(WriteFile)	1269	681	538
Read(MemoryMappedFile)	—	829	—
Write(MemoryMappedFile)	—	905	737

表 5 ファイルアクセスの OS オーバヘッド
Table 5 OS Overhead for File Access

	time[μ s]
Read(ReadFile)	4.27
Write(WriteFile)	3.75
Read(MemoryMappedFile)	9.87
Write(MemoryMappedFile)	4.39

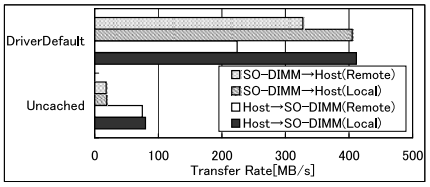

図 6 DIMMnet-2 ホスト間転送速度 (実測値)
Fig. 6 Transfer Rate between DIMMnet-2 SO-DIMM and Host Memory

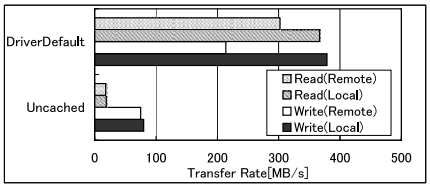

図 7 ディスク性能 (DriverDefault は推定値)
Fig. 7 Transfer Rate of DIMMnet-2 RAM Disk

6.2 DIMMnet-2 ホスト間転送速度

MT-dNET を用いて，OS を介さない場合の DIMMnet-2 の SO-DIMM バッファとホストメモリ間の転送性能の計測を行った．結果を図 6 に示す．

現在，ハードウェアの不具合が若干あり，DIMMnet-2 の各ウィンドウのキャッシュ属性は Uncached として動作している．将来的には Write Window(SO-DIMM 書込み用バッファ) は WriteCombine に，Prefetch Window(SO-DIMM 読込み用バッファ) は Cached☆で動作するようになり，大幅な性能向上が望める．

6.3 分散ファイルシステム性能

キャッシュ属性 Uncached の場合の DFS 性能とドライバを介さない場合の DIMMnet-2 の SO-DIMM バッファとホストメモリ間の転送性能の実測値と下記式を用いて，DriverDefault の場合のディスク性能 T_{DD} を見積った．

$$T_{DD} = T_{os} + T_{DDD} \quad (4)$$

$$= (T_{all} - T_{DUN}) + T_{DDD} \quad (5)$$

ここで T_{os} は，OS 呼出しによるオーバヘッドを，T_{DDD} は実測値で DriverDefault で OS を介さない場合の SO-DIMM への転送時間を，T_{all} は実測値で Uncached で OS を介した場合のファイルアクセス時間の総計を，T_{DUN} は実測値で Uncached で OS を介さない場合の SO-DIMM への転送時間を表す．

結果を図 7 に示す．DriverDefault では，MT-dNET を利用する場合に比べ，ディスク化した場合にはファイルシステム管理や，OS によるオーバヘッドが原因となり，最大 9%の性能低下になっているが，GbE で SMB を用いてディスク共有を行った場合の性能上限 30[MB/s] と比べると最大 12 倍以上になり，DFS として大幅な性能向上を達成できると考えられる．

6.4 ローカルでの評価

AT-dRAM および，MT-dNET を用いて行列乗算を行い，所要時間を計測した．AT-dRAM では，メモリマップトファイルを利用し，SO-DIMM を計算データの保存領域として利用する．また，数値計算におけるストライ

☆ このキャッシュ属性の組み合わせを DriverDefault とよぶ

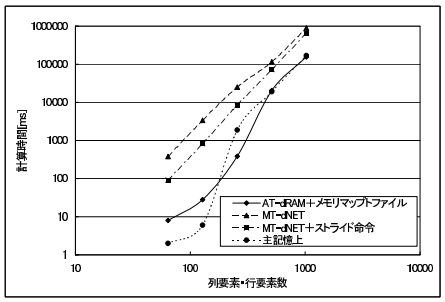

図 8 行列乗算による評価
Fig. 8 A Result of Matrix Multiplication

ド命令の有用性を検証するために，行データをロードする際にストライド命令を利用する場合と，利用しない場合で 2 種類のデータを測定した．測定結果を図 8 に示す．

行列乗算のような計算の総時間に占めるディスクアクセス時間が少ない場合には，AT-dRAM は主記憶上で計算する場合と遜色のない性能となる．また，データが膨大になるにしたがい，ページアウトが発生し，ページアウト先であるディスク性能の影響が大きくなると考えられる．これにより，さらに AT-dRAM を用いた方式が有利になると推測され，主記憶の代替として十分利用可能である．また，ストライド命令を用いた場合と用いない場合の比較では，ストライド命令を用いた場合に約 2.0 倍の性能向上になり，ストライド命令の有用性が証明された．

6.5 分散処理による評価

AT-dRAM および ATR-DSM を用いて，各種分散処理に関する評価を行った．なお前述したとおり，現在 DIMMnet-2 の各ウィンドウのキャッシュ属性が Uncached として動作しており，ハードウェアの不具合の修正により性能が改善した場合の性能予測を下記式を用いて行った．

$$T_{all} = T_{os} + T_{barrier} + T_{tr} + T_{calc} \quad (6)$$

$$T_{os} = T_{mmap}N_{page} + T_{flush}N_{flush} \quad (7)$$

$$T_{tr} = \sum_{i=0}^{N} Tr_i Nr_i + \sum_{i=0}^{N} Tw_i Nw_i \quad (8)$$

$$T_{barrier} = T_{all} - max(T_{os_i} + T_{tr_i} + T_{calc_i}) \quad (9)$$

ここでディスクへの読み込み回数 Nr_i，書き込み回数 Nw_i，および Uncached の場合の読み込み時間 Tr_i，書き込み時間 Tw_i は実測値である．現在，リモート転送に関して，ソフトウェアで再送制御を行っており，i が 1 以上の時は，再送に関しての所要時間および回数である．再送にかかる時間 Tr_i，Tw_i は，再送回数が増えるにしたがい増える．最終的には，DIMMnet-2 ではソフトウェアによる再送制御は必要なくなるため，i が 1 以上の時 Nr_i，Nw_i は 0 となる．キャッシュ属性を DriverDefault にした場合の Tw_i，Tw_i は，図 6 のデータを用い，転送時間の合計 T_{tr} を推定している．T_{os} はページフォールトによる OS 呼出しと，データのフラッシュによる OS 呼出しのオーバヘッドであり，ページフォールトによる OS 呼出しは 1 回あたりのページフォールトによるオーバヘッドの計算値 T_{mmap} とページ数の計算値 N_{page} の積，データのフラッシュによる OS 呼出しのオーバヘッドは，1 回あたりのフラッシュのオーバヘッドの計算値 T_{flush} とフラッシュ回数の計算値 N_{flush} の積から算出される．これ

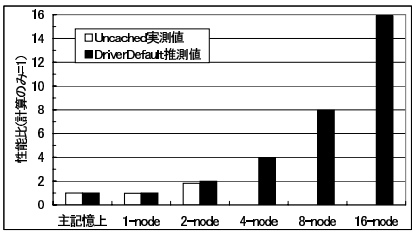

図 9 分散行列乗算結果 (N=1024)
Fig. 9 A Result of Distributed Matrix Multiplication

らにはディスクへの転送時間は含まれない．$T_{barrier}$ はバリア同期に要する時間の合計で計算値である．$T_{barrier}$ は実測により，最低 1[ms] かかることが計測されており，各ノードでの可変値で各ノードでの T_{all} が同一になるような数値になる．T_{calc} は，OS のオーバヘッド，ディスクへの転送時間，バリア同期に要する時間を除いた計算時間の総計の実測値である．

(1) 行列乗算

単純な分散処理による評価として，求める行列 (N*N 行列) を列方向に P 個に分割し，分散処理を行い，行列積を求める所要時間を計測した．DriverDefault の場合の推測に用いたデータを表 6 に，結果を図 9 に示す．なお，ページフォールト回数 N_{page} は下記式で求めた。$size\_dbl$ は double 型のサイズを示す．

$$N_{page} = (N^2 + N^2/P) * size\_dbl/4096 \quad (10)$$

また，今回は 2 ノードまでで実験を行ったが，4，8，16 ノードの場合の性能予測を行った．1 ノードの場合のパラメータから，P 台の場合のパラメータはそれぞれ，T_{calc}/P，$Nr_0/1.5^{(P-1)}$，$Nw_0/2^{(P-1)}$ として計算している．

行列乗算では計算に要する総時間 $O(n^3)$ に占めるディスクアクセスの割合が少なく，また，各ノードでの計算量が同一な上，同期を取る必要がほとんどないため，非常に効率的に並列化が行える．今回の場合においても，2 台での計算では約 1.99 倍の性能を実現している．

(2) Wisconsin Benchmark

Wisconsin Benchmark はデータベースの性能を測定するベンチマークである．図 10 の 6 つのクエリに関して，C 言語で記述し，規定された表形式のデータの格納先を DIMMnet-2 の SO-DIMM およびハードディスクに変えて性能測定を行った．Wisconsin Benchmark を分散処理させた結果を図 11，推測に用いたデータ (Q1 の場合) を表 7 に示す．なお，ページフォールト回数 N_{page} は下記式により求めた．N_{tuple} はタプル数の合計である．

$$N_{page} = \sum_{i=0}^{n} (N_{tuple_i} * tuple\_size)/4096 \quad (11)$$

表 6 行列乗算データ
Table 6 Parameters for Matrix Multiplication

	1-node	2-node	4-node(予測)
$T_{barrier}$	1[ms]	1[ms]	1[ms]
N_{page}	6144[page]	4096[page]	2730[page]
N_{flush}	1[回]	1[回]	1[回]
Nr_0	49152[回]	32768[回]	21845[回]
Nw_0	16384[回]	8192[回]	4096[回]
T_{calc}	136695[ms]	68348[ms]	34174[ms]
T_{all}	136865[ms]	68564[ms]	34309[ms]

```
(Q1) select * from tenk1 where (unique2 > 301) and (unique2 < 402)
(Q2) select * from tenk1 where (unique1 > 647) and (unique1 < 65648)
(Q3) select * from tenk1 where unique2 = 2001
(Q4) select * from tenk1 t1,tenk1 t2 where
     (t1.unique2 = t2.unique2) and (t2.unique2 < 1000)
(Q6) select t1.*,o.* from onek o,tenk1 t1,tenk1 t2 where
     (o.unique2 = t1.unique2) and (t1.unique2 = t2.unique2)
     and  (t1.unique2 != 1000) and (t2.unique2 != 1000)
(Q7) select MIN(unique2) from tenk1
```

図 10 Wisconsin Benchmark クエリリスト
Fig. 10 Queries for Wisconsin Benchmark

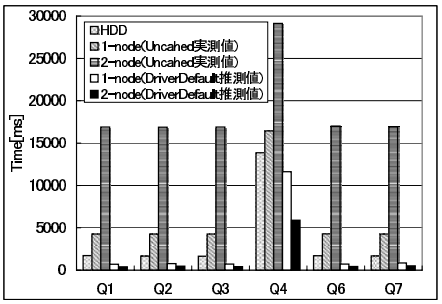

図 11 Wisconsin Benchmark 結果
Fig. 11 A Result of Wisconsin Benchmark

単純な join のないクエリでは，ディスク性能が大きく影響し，Q1 では HDD に対して 2.49 倍の性能向上となっている．逆に分散処理が有用に働くのは join があるクエリ (Q4,Q6) であり，Q4 の場合には，1 ノードの場合に対して 1.97 倍の性能向上となった．データベースのような膨大なデータの格納が要求される分野においても AT-dRAM と DIMMnet-2 は有効である．

(3) **姫野ベンチ**

姫野ベンチは，ポアソン方程式解法をヤコビの反復法で解く場合に主要なループの処理速度を計るものである．今回は AT-dRAM と ATR-DSM を利用するように改変した並列版，および主記憶を利用する単一ノード版 (Original) を用いて性能測定を行った．DriverDefault の計算に用いたデータ (Size=M の場合) を表 8，結果を図 12 に示す．なお、ページフォールト回数 N_{page} は下記式により求めた．

$$N_{page} = N^3/(2*P)*size\_dbl/4096 \qquad (12)$$

姫野ベンチは，計算に要する時間のオーダ $O(n^3)$ に対し

表 7 Wisconsin Benchmark データ
Table 7 Parameters for Wisconsin Benchmark

	1-node	2-node
$T_{barrier}$	1[ms]	1[ms]
N_{page}	18944[page]	9472[page]
Nr_0	152384[回]	76224[回]
T_{calc}	210[ms]	105[ms]
T_{all}	695[ms]	405[ms]

表 8 姫野ベンチデータ
Table 8 Parameters for Himeno Benchmark

	1-node	2-node
$T_{barrier}$	1[ms]	1[ms]
N_{page}	4096[page]	2048[page]
N_{flush}	1[回]	1[回]
Nr_0	33142[回]	16576[回]
Nw_0	32649[回]	16073[回]
T_{calc}	703[ms]	352[ms]
T_{all}	860[ms]	509[ms]

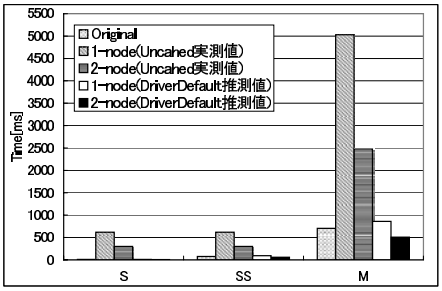

図 12 姫野ベンチ結果
Fig. 12 A Result of Himeno Benchmark

て，ディスクアクセスの量が $O(n^3)$ と総時間に占めるディスクアクセスの時間が多く，主記憶を利用する単一ノード版に比べて 1 ノードの場合は性能が 0.82 倍に低下，2 ノードでの分散処理でも 1.38 倍の性能向上にとどまっている．しかし分散処理の効果を見た場合には，ATR-DSM 環境下で 1 ノードでの計算と 2 ノードでの分散処理を比べると 1.69 倍になっており，分散処理の効果が現れている．

7. おわりに

本研究では，Windows 向けに DIMMnet-2 用ドライバを開発し，クラスタシステムを構築することで，実際に分散処理に関する評価を行った．結果，Windows 上での DIMMnet-2 による分散処理を行うことができ，評価実験からは分散処理により 2 ノードで最大 1.99 倍の性能向上が確認でき，クラスタシステムとしての実用性を達成できた．また，本研究では DIMMnet-2 および，Windows を対象として議論を行ったが，本論文での提案は DIMMnet-2 および，Windows のみに限定されるわけではない．OS のカーネルに手を加えることができないコモディティ OS 環境下で，デバイスドライバとメモリマップトファイルの組み合わせによる分散処理に特化したハードウェアでの分散処理の可能性を示したものであるといえる．

参考文献

1) W. Gropp, E. Lusk: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, Parallel Computing, Vol. 22, No. 6, pp. 789–828 (1996.9).
2) P. Keleher, S. Dwarkadas, A. L.Cox, and W. Zwaenepoel: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, Proc. of the Winter 94 Usenix Conf., pp. 115–131 (1994.1).
3) 佐藤, 原田, 石川: Cluster-enabled OpenMP : ソフトウェア分散共有メモリシステム SCASH 上の OpenMP コンパイラ, IPSJ HPC, Vol.42, No.SIG09(HPS3), pp.158–169 (2001.8).
4) 北村, 濱田, 宮部, 伊澤, 宮代, 田邊, 中條, 天野: DIMMnet-2 ネットワークインタフェースコントローラの設計と実装, IPSJ, Vol.46, No.SIG 12, pp.13–26 (2005.8).
5) 丹羽, 松本, 平木: コンパイラが支援するソフトウェア DSM 機構: ADSM と UDSM の性能評価, IPSJ 99-HPC-77 (SWoPP'99), Vol. 99, No. 66, pp. 95–100(1999.10)