情報処理学会論文誌　コンピューティングシステム　Vol. 2　No. 4　15–36 (Dec. 2009)

クラスタをメモリ資源として利用するための MPI による高速大容量メモリ

緑川博子[†1]　斉藤和広[†2]
佐藤三久[†3]　朴泰祐[†3]

64 bitOS の普及により，飛躍的に大きなアドレス空間が利用可能となった．筆者らはローカル物理メモリサイズに制限されず，クラスタの各ノードの遠隔メモリを集めて，仮想的に 1 つの大容量メモリとして逐次処理に提供するシステム，分散大容量メモリシステム DLM を提案してきた．DLM は，OS スワップシステムに組み込む他の多くの遠隔ページング手法とは異なり，OS のスワップシステムとは独立にユーザレベルソフトウエアとして実装されている．すでに，汎用の TCP のみを用いた DLM が，ブロックデバイス構築，専用 NIC，低レベル高速通信プロトコルなどを併用した他手法に比べ，より高い性能と動作安定性を示すことを明らかにした．本論文では，従来の DLM では TCP で実装してきたノード間通信機構を，標準的なクラスタ間通信機構である MPI で実装し，より可搬性・可用性を高め，様々な最新高性能通信機構にも対応可能とした．あわせて，クラスタや並列処理の知識のない一般ユーザにも容易にクラスタを利用できる API も構築した．これにより，自分専用のクラスタやメモリ豊富なハイエンドマシンを持たないユーザであっても，少ない資金で，MPI バッチシステムで運用される多くのオープンクラスタをメモリ資源として利用することが可能になった．Myri-10G/bonding=4 のネットワークを持つオープンクラスタでの実験では，遠隔メモリバンド幅 613 MB/s を達成し，241 GB のデータに対する Himeno ベンチマーク処理を 20 GB メモリ/ノードを複数用いて，稼動できることを示した．また NPB の 6 種のプログラムについて，ローカル/遠隔メモリサイズ比と性能の関係などについて明らかにした．

A Fast and Portable Virtual Memory System Utilizing Memory Resource of a Cluster

Hiroko Midorikawa,[†1] Kazuhiro Saito,[†2]
Mitsuhisa Sato[†3] and Taisuke Boku[†3]

Today, a 64 bitOS provides ample memory address space that is beneficial for the applications using a large amount of data. This paper proposes a new idea of using a cluster as a memory resource for such applications requiring large amount of memory. We already proposed the Distributed Large Memory System: DLM, which enables the users to make use of large virtual memory by using remote memory distributed over nodes in a cluster. It achieves better performance than other remote paging schemes using a block swap device to access remote memory. It is designed for sequential programs accessing larger amount of data beyond local physical memory. In this paper, we propose the newly designed MPI-based DLM, which uses only MPI for inter-node communication, to exploit higher performance and portability than the former socket-based DLM. It achieves 613 MB/sec of remote memory bandwidth in STREAM benchmark on Myri-10G/bonding=4 and high performance of applications in NPB and Himeno benchmarks. It has opened the door to use a cluster for the people who are not familiar to the parallel programming. All experiments here were done on the public open cluster with MPI batch queuing system.

1. はじめに

64 bitOS の普及により，飛躍的に大きなアドレス空間がプログラムから利用可能となり，多量のデータを扱う様々な応用にとって恩恵が得られるようになってきた．通常，OS における仮想メモリシステムでは，プログラムの使用メモリ量が，用いるコンピュータの搭載物理メモリサイズを超えると，ローカルハードディスクにあらかじめ設けられたスワップ領域（ファイル）にメモリページを必要に応じて出し入れ（スワップアウト・イン）して，物理メモリサイズに制限されない仮想メモリを実現するのが典型的手法である．しかし近年，ローカルハードディスクへの I/O 性能を超える通信性能を持つネットワークが出現しており，ローカルハードディスクに代えて，遠隔コンピュータのメモリを利用するという研究がいくつかなされるようになってきた[7)–15)]．

筆者らはすでに，ローカル物理メモリサイズに制限されず，クラスタの各ノードの遠隔メモリを集めて仮想的な大容量メモリとして逐次処理用に提供するシステム，分散大容量メモリシステム DLM（Distributed Large Memory）を提案している[1)–4)]．DLM は，OS ス

†1 成蹊大学理工学部
Faculty of Science and Technology, Seikei University
†2 成蹊大学大学院工学研究科
Graduate School of Engineering, Seikei University
†3 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

ワップシステムに組み込む他の多くの遠隔ページング手法とは異なり，OS のスワップシステムとは独立にユーザレベルソフトウエアとして実装されている．

他の研究のほとんどは，遠隔メモリアクセスのためのドライバを新たに構築し，従来のスワップデバイス（ローカルハードディスク）に代えて，OS スワップシステムからこの遠隔メモリデバイスを用いるという手法をとっている．多くはカーネルの一部変更，専用 NIC，専用高速通信プロトコル，RDMA 機能，通信前のデータの事前メモリ登録なども併用し，高速化を図っている[7),12)]．しかし，これらの研究では，用いた通信媒体や機構の性能に比べ，非常に低い遠隔メモリアクセス性能を得るにとどまっており，実際の応用レベルのプログラムに対しても安定的に動作する実験結果を得ておらず，様々な点で十分に成功しているとはいいがたい．

筆者らは，OS のスワップ機構とは独立にユーザレベルソフトウエアとして構築することにより，このような特別なハードウエアや手法を用いずに汎用 TCP のみを用いただけで，上述の他手法に比べ，より高い性能と動作安定性が得られることをすでに明らかにした[1)–4)]．10 Gbps Ethernet における初期性能評価では，応用プログラムレベルでのメモリ読み書き性能を測定する STREAM ベンチマークにおいて，380 MB/s の遠隔メモリバンド幅を達成し，NPB や Himeno ベンチマークなどの応用ベンチマークにおいても，特殊 NIC，高速通信プロトコル，RDMA などを用いた他の手法に比べ，高い性能を得ている[4)]．

このような遠隔メモリを逐次プログラムに利用する研究には，カーネルレベル実装とユーザレベル実装の 2 つの方式があり，それぞれ長所・短所を持つ．前述のような遠隔メモリを OS のスワップシステムから利用する場合は，カーネルレベル実装となり，ユーザに完全な透過性を提供できるという利点がある．反面，様々な OS やクラスタ上において一般ユーザが自由にインストールして実行できる環境を提供するのはむずかしく，現状では，可搬性や可用性の点で制限をうける．

ユーザレベル実装の場合には，一般ユーザでも容易に利用でき，可搬性，可用性は高いが，応用プログラムの書き換えなどが必要になる点でユーザ透過性は劣っているといわざるをえない．この点を改良をするために，ユーザレベル実装である JumboMem[15)] では，malloc などのメモリ関連関数コールを，LD_PRELOAD 環境変数を利用し，本来の OS ライブラリ関数に代えて，JumboMem 独自の共有ライブラリ関数コールと交換してしまう方式を提案している．これにより，ユーザによるプログラムの書き換えが不要になり，バイナリプログラムも実行時に動的に JumboMem の関数をリンクして処理されるため，高いユーザ透過性を得ることができる．しかし，一方で，この方式は動的メモリ確保関数にしか適用されないために，大規模データ利用プログラムでよく用いられる静的な大規模配列宣言データなどには適用できない．したがって，JumboMem を利用する際には，このようなデータ宣言は malloc 形式にプログラムを変換する必要がある．さらに，もう 1 つの問題点は，すべての動的メモリ確保関数が JumboMem の関数に置き換わってしまうため，確保されたデータが，ローカルメモリではなく遠隔メモリへスワップアウトされてしまう危険性をはらむことである．これは，入出力のバッファリングや多くのシステムソフトウエアなどでデータをローカルメモリにおいておくことが必要なプログラムにとっては，望ましくない副作用を生むことになる．

この点に関して，同じくユーザレベル実装である DLM では，ユーザによるプログラム書き換えの負担を最小限に抑えるために，静的配列宣言や動的メモリ確保の両方に対応できる単純な API を提供し，専用コンパイラを構築してきた[4)]．静的配列データ宣言の先頭に **dlm** という指定子を付加するだけで，遠隔メモリにも展開可能なデータ（DLM データ）としての宣言が可能で，ユーザは，プログラム中の静的データ宣言を DLM 動的メモリ確保関数にいちいち変換する必要がない．また，DLM データか，通常データ（ローカルメモリにつねにあるデータ）なのかを明示的にユーザが指定できるため，上述の JumboMem のような問題が起きず，効率の良い利用が可能である．

性能や動作安定性の点では，ハードディスク用にチューニングされた現在の OS スワップデーモンから遠隔メモリを利用するという，従来のカーネル実装方式が採用している手法は，現状では十分な成果を得るのが難しい[1),4),14)]．このため，遠隔メモリアクセス用のドライバを構築してカーネルに組み込むものの，OS スワップデーモンからの遠隔メモリ利用をあきらめてユーザ透過性を犠牲にし，ユーザレベル実装と同様なプログラム改変を前提とするカーネル実装システムも提案されている[14)]．

ユーザレベル実装が，カーネルレベル実装に比べて性能上，なんら遜色がないばかりか，きわめて高い性能を得られるということが明らかになった現在，本研究では，ユーザレベル実装の最大の利点である可搬性・可用性をさらに高めることを目指した[5)]．本論文では，クラスタを逐次処理応用のメモリ資源として利用するという新しい概念・恩恵を，より多くの人が享受できるように，ノード間通信として MPI のみを用いた DLM を新たに設計し，汎用性・可用性を高め，一般ユーザが高性能通信環境を利用できる機会を増やす．前述した DLM の実装では，ノード間通信として，シグナル割込みによる TCP ソケット通信を用いていた[4)]．このソケット通信を用いる DLM（DLM-socket）では，ユーザが指定したホスト名（あるいは IP アドレス）を利用して遠隔ホストにメモリサーバプロセスを自動生成し，

このプロセスと通信を行うことにより，遠隔メモリの利用を可能にする．このため，通常はインタラクティブな利用を許されたクラスタでの実行を前提としていた．しかし，インタラクティブな利用を許さない MPI バッチシステムなどで運用される多くの汎用オープンクラスタ上での実行には適さないことも多く，汎用性・可用性が制限されていた．今回設計した MPI による DLM システム（DLM-MPI）は，このようなバッチスケジューラにより自動的に割り当てられるノード上においても実行可能で，インタラクティブなクラスタ上での利用にとどまらず，MPI バッチシステム運用のクラスタにおける利用も可能にした．

DLM を MPI で実装する利点は 2 つある．第 1 は，通信媒体を選ばず，利用環境が提供する最大級の通信性能を得られる点である．MPI は下層通信媒体などに依存しない抽象通信記述であるため，各クラスタが用いる通信環境によらず DLM の可搬性が高まる．さらに，最近の MPI ライブラリ関数の多くは，従来の TCP/IP 上に実装されているものだけでなく，Myrinet や Infiniband など，多くの最新の高速通信媒体上に直接実装されているものが少なくない．下層の通信媒体を最大限に生かす実装が各通信媒体ベンダなどにより構築・提供されている．また複数の物理チャネルを 1 つの通信で用いるネットワークボンディングなど，利用環境が提供する最新ネットワーク技術による様々なユーティリティも MPI から容易に利用できることが多い．

第 2 は，ユーザの運用利便性，利用可能性としての利点である．多くの汎用オープンクラスタが現在 MPI バッチシステムで運用されており，DLM がユーザレベルの MPI プログラムとして実行できることにより，自分専用のクラスタを持たないユーザでも，予算に応じて，必要なときだけ，汎用オープンクラスタを用いて大容量データを使う逐次処理応用を走らせることが可能になる．このような MPI バッチ運用のオープンクラスタで DLM-MPI を利用する場合，ユーザには MPI ソフトウエアの設定や MPI による並列プログラミングの知識が不要である．

DLM は，大容量データを扱う逐次プログラム実行のためのシステムで，MPI や OpenMP を直接用いた並列プログラムのためのものではない．並列化が難しいプログラムや，むしろ逐次処理のまま実行したいプログラムもあり，すべてのプログラムは並列プログラムに変換する必要性があるとはいえない．また並列化のための知識を持たないユーザも存在する．もともと並列化技術を活かせる応用やユーザであれば，それぞれの工夫をもとにクラスタに分散したメモリだけでなく CPU も有効利用できる．しかしすべての応用が並列化できるとは限らず，また多くのユーザにとってそれは容易なことではない．DLM-MPI は，従来はクラスタに縁のなかった，大容量データ逐次処理応用を持つユーザが，並列プログラミングの知識なしに，MPI バッチシステムで運用される多くのオープンクラスタを，少ない資金で，メモリ資源として利用することを可能にする．

本論文では，まず 2 章で MPI による DLM の概要を説明する．3 章では，従来のソケットを用いた DLM との違いを中心に MPI による DLM の実装を述べる．また MPI のスレッドサポートレベルに応じたスワッププロトコルについても触れる．4 章では，MPI バッチキューイングシステムで運用されているオープンクラスタにおける DLM-MPI の性能を示す．STREAM ベンチマークを用いた遠隔メモリバンド幅性能や，応用ベンチマークとしては，Himeno ベンチマーク，6 種の NPB プログラムについて，ローカルメモリデータサイズ/プログラム使用データサイズ比の違いによる性能への影響などを示す．また，異なる通信性能を持つネットワークにおけるスワップ処理時における時間成分などを示し，現在採用しているプロトコルについても考察する．また応用の持つメモリアクセス局所性と DLM で用いるページサイズとの関連などにも触れる．5 章では，本論文の成果をまとめる．

2. MPI による DLM システム（DLM-MPI）の概要

DLM は，逐次プログラム実行において，実行するクラスタノードの搭載物理メモリサイズ以上のサイズのメモリを利用したい場合に，クラスタの他の遠隔ノードのメモリも利用できるようにするユーザレベルのソフトウエア（ライブラリを提供）である．これは，クラスタにおけるページベースのソフトウエア分散共有メモリにおけるページ書き込み検出やページ転送と基本的に同じ手法を用いている．すなわち，ローカルノードメモリにマップされていないページにユーザプログラムがアクセスした場合には，遠隔ノードに展開してあった該当ページを要求して受け取り（DLM スワップイン），代わりにローカルメモリ上にあったページを遠隔ノードに送付（DLM スワップアウト）してページを交換（**DLM スワップ**）する．これにより，逐次プログラムには，ローカル物理メモリサイズを超えた容量のメモリが仮想的にあるかのように見せる．

今回，新たに構築した DLM-MPI は，InfiniBand や Myrinet といった最新の高速通信リンクとその上に直接実装された MPI ライブラリを提供する MPI バッチ運用オープンクラスタにおける稼動も可能である．これにより，ユーザは個人的なクラスタ環境ではなかなか得ることが難しい高性能通信環境を利用することができる．DLM-MPI システムは，従来，クラスタには無縁であった大容量メモリを消費する逐次処理応用を持つユーザが，クラスタを大容量仮想メモリとして容易に利用できるように設計されている．MPI で運用されるオープンクラスタにおいて，ユーザレベルソフトウエアである DLM-MPI ライブラリを用

いることで，並列プログラミングの知識なしに，1 台のコンピュータの搭載物理メモリサイズを超えた大きなデータを扱うような逐次プログラムを，従来のローカルハードディスクを用いた OS スワップシステムとは比較にならないほど高速に実行させることができる．

2.1　MPI バッチキューイングシステムにおける DLM

従来のソケットベースの DLM（DLM-socket）では，DLM システム起動関数である dlm_init() 関数の中で，hostfile と呼ばれるユーザの記述したホスト名のリストファイルをもとに，指定された各ホストノードにメモリサーバプロセスを自動的に生成（遠隔 fork）する[3]．MPI ベースの DLM（DLM-MPI）の新しい起動関数 dlm_startup() では，このプロセス生成部分の処理を削除し，すでに MPI バッチキューイングシステムによって自動的に割り付けられたノードに生成済みの MPI プロセスをそのまま利用する．

DLM-MPI のランタイムシステムの基本構成は，従来の DLM-socket とほぼ同様の構成で，**図 1** に示すように，ローカルホストにある 1 つの計算プロセス（rank0）と，1 つ以上の遠隔ホスト（メモリサーバホスト）にあるメモリサーバプロセス（rank1〜）からなる．ユーザプログラムは，計算プロセスの中の計算スレッド calThread（MPI バッチシステムで自動生成されたスレッド）において実行される．計算プロセス（rank0）は，通常，dlm_startup() 内で，メモリサーバプロセスとの通信を行うための通信スレッド（comThread）を生成し，プログラム処理（calThread）との並列処理性を確保する．メモリサーバプロセスでは通信スレッドは生成しない．

ただし，後述するように，用いる MPI ライブラリのスレッドサポートレベルによっては，計算プロセスは，通信スレッドを生成せずにシングルスレッドのまま，MPI バッチシステムで初期に自動生成された計算スレッドのみだけで，構成することも可能である．通信プロトコルも様々な方式がとれるが，用いる MPI ライブラリのスレッドサポートレベルやその性能との兼ね合いで計算プロセスの構成と通信プロトコルを選択することもできる．

ここでは，図 1 に示すマルチスレッド構成の計算プロセスの場合を中心に説明する．図 1 は，4 ノードを用いて各ノードに 1 プロセスを生成し，そのうちの 3 プロセスをメモリサーバプロセスとして利用する例である．ただし，ユーザのアプリケーションの処理は計算プロセス（rank0）上でのみ，逐次的に行われる．MPI バッチキューイングシステムにおける，典型的なバッチ用のシェルスクリプト例を図 1 の上部に示す．このシェル例では，実行するバッチキューの名前指定（– q），利用ノード数（– N），1 ノードあたりの生成プロセス数（– J T），打ち切り実行時間（– lT）などを指定した後，実行プログラムのあるディレクトリ（programs）に移動し，mpirun コマンドで，ユーザプログラム（prog）をプログラム引

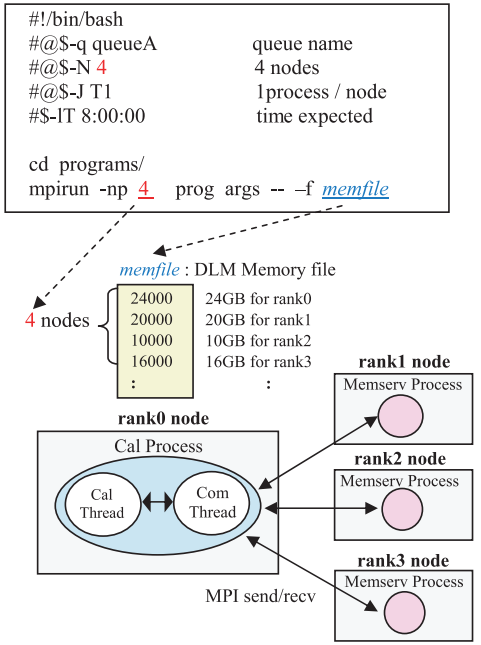

図 1　DLM-MPI ランタイムシステムとメモリ設定ファイル，バッチシェル例
Fig. 1　The DLM runtime system, a memfile and a batch shell example.

数（args）とともに実行している．実行コマンドの最後にある – – から続くパラメータは DLM システムへ与える引数で，この例では，– f memfile によって，各ノードで DLM が利用するメモリサイズ指定のファイルを加えている．このファイルは省略可能で，省略時には DLM システム構築時に定めたデフォルト設定に従い，全ノードとも同一の規定メモリサイズを使用する．多くの場合，クラスタシステムの各ノードメモリサイズは同一の場合が多いため，指定が不要となる．この例では，図 1 の中ほどに示すような memfile というファイルで，各ノードで DLM が利用するメモリサイズを（MB 単位で）指定している．先頭行が計算ノードで利用可能なメモリサイズで，2 行目以降は，rank1 以降のメモリサーバノードで利用可能なメモリサイズである．DLM はこのサイズ以上のメモリを各ノードで利用しないようにする．この値を各ノードの搭載する物理メモリサイズに見合った余裕のあるサイズに設定しておくことにより，各メモリサーバ上で OS によるスワップ処理（すなわち OS

スワップデーモン起動とこれによるハードディスクへのスワップ処理）が二次的に発生することを抑制し，安定で高速な稼動が可能になる[4]．

DLM システムにおけるクラスタでの複数ノード利用とは，従来の並列処理のように多くの計算パワー（CPU）の利用を目的にするのではなく，より多量のメモリ利用を目的としているので，ノードあたり 1 つのメモリサーバプロセス生成して（– J T1），ノードに搭載された最大限のメモリをそのメモリサーバプロセスが利用できるように設定する．より多量のメモリを利用したければ，メモリサーバノード数（– N）を増やすこともできる．

原理的に，プログラムから利用可能なメモリサイズは，各ノードで利用可能なメモリサイズ × ノード数である．すなわち，DLM の現実装は，前述の JumboMem のようなキャッシュ実装ではない．たとえ同一物理メモリ量を搭載する 2 ノードのみのマシン利用の場合でも，プログラムから利用できるメモリサイズを 1 ノードのローカルメモリサイズより増やすことができるようにするため，ローカルノードにあるページのコピーをメモリサーバノードは保持しない．大規模ノード数クラスタのみでの実行を前提にするなら，キャッシュ方式による実装のほうが，ローカルノードで書き込みのないページをメモリサーバに戻す必要がないので，さらに性能向上ができる可能性もある．

また，すでに文献 4) の調査から，用いるメモリサーバ数が増えることによる性能低下は基本的にはないことが分かっている．すなわち大容量メモリを持つサーバを 1 つ利用するか，小容量のメモリを持つノードを複数利用するかによって，性能の違いはない．1 度にメモリサーバと通信するのは計算プロセス 1 つだけなので，並列処理などとは異なり，使用ノード数が増えることによって通信が混みあうことが原理的に起きないためである．

ユーザプログラムからのデータ割付けの際に，計算ノードのローカルメモリが不足し遠隔メモリが必要なときには，計算プロセスの中の通信スレッドが，メモリサーバプロセスと通信し，遠隔メモリ上にデータを展開する．また，ユーザプログラムからのアクセスなど，必要に応じてメモリサーバプロセスと通信スレッドが，データ（**DLM ページ**）を交換し，計算プロセスからはローカルメモリを超えるサイズのメモリがあるかのように実現する．すなわち，メモリサーバノードと計算ノード間で，**DLM ページサイズ**という単位で，メモリページの DLM スワッピングを行い，仮想的に大容量メモリを実現している．

DLM ページサイズは，OS の管理するページサイズ（たとえば 4 KB）の倍数であれば，DLM 構築時に自由に設定できる．すでに文献 4) で明らかにしたように，DLM は OS スワップ処理とは独立に設計することにより，OS によるスワップ処理時のパラメータ設定などに縛られることなく，ネットワーク通信性能などに合わせて通信効率の良い大きなページサイズを用いることができる．ただし DLM ページサイズは大きいほど通信効率はあがるが，1 度に交換する DLM ページサイズを大きくしすぎると，応用プログラムからみて，ページ内の利用しないデータも含めて通信することになり，頻繁にページ交換を繰り返す場合にはオーバヘッドになる可能性もある．また後述するように応用プログラムのメモリアクセスローカリティによっては小さいページサイズが有利に働く場合もある．ページ転送サイズを環境に合わせて自由に設定できる柔軟性は，カーネルレベルではなくユーザレベルソフトウエア実装である DLM の利点である．文献 4) の性能評価プログラム例では，応用プログラム性能上でも 10 Gbps のネットワークにおいては，比較的大きな 64 KB 以上 1 MB 程度の DLM ページサイズを用いることが効果的であることが示された．DLM では，現在 1 Gbps ネットワークに対しては 128 KB，10 Gbps 以上のネットワークに対しては 1 MB のページサイズを通常用いている．

すべてのノード間通信には，従来のシグナル割込みによるソケットの read/write に代えて，MPI_Recv，MPI_Send を用い，同期には MPI_Barrier を利用している．また，計算プロセス内の計算スレッドと通信スレッドとのやりとりには，SIGUSR シグナルと状態フラグ変数を使っている．

2.2　DLM-MPI システムのプログラムインタフェース

DLM では，並列プログラミングの知識を持たないユーザであっても，既存の逐次プログラムに最小限の変更を加えて利用できるようにしている．DLM は，並列プログラム用の MPI ライブラリのように，ユーザレベルソフトウエア（ライブラリ）で実装されているため，ユーザレベル権限で容易に導入が可能（ライブラリファイルをコピーするだけ）で，コンパイル時に-ldlmmpi とライブラリ指定するだけで用いることができる．また MPI ライブラリとは異なり，用いる関数は基本的に 3 あるは 4 種と非常に少なく，並列プログラムに変換する必要もない．現在，逐次 C プログラムのみを対象にしており，malloc，free 関数と同様のインタフェースを持つ関数で **DLM データ**（DLM の管理下で，必要に応じて遠隔メモリへの書き出し/読み込み対象となるデータ）の割付けや解放を行う．ユーザは，逐次 C プログラムの中で，どのデータを DLM データとしてクラスタノードに展開するかを指定する．この情報を加えた逐次プログラムを **DLM プログラム**という．

この DLM プログラムを作成するには，大きく 2 つの手法がある．1 つは，文献 4) で示したのと同様な DLM 用のコンパイラを利用する方法である（実際には，dlm_init() と dlm_statup() の機能が異なっているために，変換後のコードには違いがある）．**図 2** に示すのは，行列とベクトルの積を計算する単純なプログラム例であるが，データの静的宣言に

```
//DLM Program example :  Matrix Vector Multiply
#include <stdio.h>
#define N 16384 // example: mem 2048MB + 32KB
dlm double a[N][N], x[N], y[N];  // DLM data declare

main(int argc, char *argv[])
{   int i,j;
    double temp;
    for(i = 0; i < N; i++)        //  Initialize array a
          for(j = 0; j <N; j++) a[i][j] = i;

    for(i = 0; i < N; i++) x[i] = i;     // Initialize array x
    for(i = 0; i < N; i++){           // a[N][N]*x[N]=y[N]
          temp = 0;
           for(j = 0; j <N; j++)  temp += a[i][j]*x[j];
          y[i] = temp;
    }
    return 0;
}
```

図 2　DLM-MPI コンパイラ変換用の DLM プログラム例
Fig. 2　The sample of DLM program for DLM-MPI compiler.

dlm という予約語を付加するだけで，このデータを DLM データとして扱うことが指定できる．もし，動的割付け malloc を用いているプログラムであれば，関数名を dlm_alloc に変更するだけでよい．これにより，ユーザはどの部分のデータをメモリサーバに展開するかを指定でき，それ以外のデータはつねに計算ノードのローカルメモリにおかれることになる（ただし OS カーネルスワップ起動時以外）．また dlm 指定されたデータであっても，ローカルホストのメモリに余裕があれば，ローカルメモリから順に割り当てていくので，遠隔メモリにデータを展開したりスワップしたりするのは，ローカルノードの DLM 利用可能メモリサイズを超えた場合だけである．図 2 では，従来の逐次 C プログラムからの変更部分が太字の dlm 宣言のみで，きわめて容易な変更で DLM が利用可能であることを示す．

2 番目の手法は，DLM 用のコンパイラを用いずに，dlm_startup()，dlm_shutdown()，dlm_alloc()，dlm_free() のうちの 3，4 種の dlm 関数だけを用いて，逐次プログラムを手で書き換える方法である．**図 3** に示すのは，図 2 と同じプログラム内容であるが，ユーザが dlm_startup() と dlm_shutdown() をプログラムの先頭と最後にそれぞれ挿入し，DLM データとして確保したいデータは dlm_alloc() を用いて動的に割り付けている．1 次元配列を用いるプログラムであれば，dlm_alloc を用いるだけで容易に変換が可能である．多次元配列を用いる場合は，図 3 に示すように各次元サイズを含む多次元配列へのポインタを宣言すれば，多次元配列を用いるプログラム処理記述部分にはいっさい変更がいらない．この

```
//DLM Program example :  Matrix Vector Multiply
#include <stdio.h>
#define N 16384 // example: mem 2048MB + 32KB
double (*a)[N];  //  a pointer for 2-D array
double  *x, *y;   //  pointers for 1-D arrays

main(int argc, char *argv[])
{  int i,j;
   double temp;
   dlm_startup(&argc, &argv);

   a = (double (*)[N]) dlm_alloc(  N * N * sizeof(double) );
   x = (double *) dlm_alloc( N * sizeof(double) );
   y = (double *) dlm_alloc( N * sizeof(double) );

   for(i = 0; i < N; i++)       //  Initialize array a
         for(j = 0; j <N; j++) a[i][j] = i;

   for(i = 0; i < N; i++) x[i] = i;     // Initialize array x
   for(i = 0; i < N; i++){           // a[N][N]*x[N]=y[N]
         temp = 0;
         for(j = 0; j <N; j++)  temp += a[i][j]*x[j];
         y[i] = temp;
   }
   dlm_shutdown( );
   return 0;
}
```

図 3　dlm 関数を用いて手動変換した DLM プログラム例
Fig. 3　The sample of DLM program for hand-translating.

```
Compile command for hand-translating programs
        mpicc   prog.c  -o prog  -ldlmmpi
```

図 4　手動変換した DLM プログラムのための DLM-MPI コンパイルコマンド
Fig. 4　The compile command for hand-translating programs.

第 2 の手法では，特別なコンパイラを必要とせず，**図 4** に示すように，dlm 関数ライブラリ（dlmmpi）を指定して，汎用の MPI コンパイラ mpicc でコンパイルが可能である．

DLM システムでは，本来，MPI バッチキューイングシステムで動的に割り当てられたノード上に，実行時に動的にメモリを割り付ける．したがって，第 1 の手法であるコンパイラによる変換とは，この第 2 の手法で作成したようなコードを自動生成しているにすぎない．ただし，実際には，DLM コンパイラは，DLM データの変数名と通常データの変数名スコープチェックやリネーミング，静的なデータ宣言や配列アクセス記述を，ポインタ変数宣言やポインタベースアクセスの記述に変換するなどの処理を行っている．

しかし，いずれの DLM プログラムにおいても，並列処理のための記述や MPI 記述はな

く，依然，逐次処理プログラムのままで，並列プログラミングに縁のないユーザにとっても理解しやすいという点は重要である．実際には上述の 3 種の dlm 関数を挿入することにより，この DLM プログラムは，複数プロセス（場合によっては通信スレッドを含むマルチスレッド）の並列 MPI プログラムとして実行される．しかし，ユーザにとっては，特別な並列プログラミングの知識なしに，クラスタにおいて複数ノードを利用した大容量のメモリを使用できる恩恵がもたらされる．

3. DLM-MPI システムの実装

3.1 DLM システムの起動と終了処理

DLM-MPI システムでは，MPI システムにより起動された MPI プロセスを用いているので，SPMD 型ですべてのプロセスが同じプログラムを実行する．したがって，全プロセスは最初に起動関数 dlm_startup() を呼ぶことになる．この関数では，MPI_Init(), MPI_Comm_size(), MPI_Comm_rank() を呼び，各プロセスの rank 番号と実行時の MPI プロセス数を取得する．

ソケットベースの DLM（DLM-socket）で用いている起動関数 dlm_init() の実装とは異なり，DLM-MPI の起動関数 dlm_startup() では，メモリサーバプロセス（rank1〜）が，処理開始時にこの関数を呼ぶと，計算プロセスで実行されるユーザプログラムが終了するまで，この関数から戻ってこない仕組みになっている．メモリサーバプロセスは，この起動関数内で，使用する内部管理データ構造を初期化後，計算プロセスからのページ要求やスワップ，データ割付けなどのサービスをするループサーバプロセスとして稼動しはじめ，計算プロセスからプログラム終了のメッセージを受け取ると，dlm_startup() から戻ることなく，プロセスを終了する．

一方，dlm_startup() を呼んだ計算プロセス（rank0）の計算スレッド（calThread）は，mpirun のコマンドライン引数の解析を行い，必要なパラメータ（たとえば，各ノードの DLM による利用可能メモリサイズなど）をメモリサーバプロセスへ送付する．次に，不要なシグナルをブロックして通信スレッド（comThread）を生成し，その後 calThread に SIGSEGV シグナルハンドラの設定を行う．これは，ローカルメモリにマップされておらずメモリサーバノードにマップされている DLM データ（DLM ページ）にユーザプログラムがアクセスしたことを検知するためののものである．SIGSEGV ハンドラ内で，メモリサーバへの DLM ページ要求や，必要に応じてページスワップを行う．

計算プロセス（rank0）の計算スレッドは，シグナル設定とスレッド生成が終了すると，DLM 設定ファイル（図 1 の memfile）に指定された（あるいは省略時には規定の）利用可能なメモリサイズ分の DLM ページ数のエントリを持つ **DLM ページ表** [4] や動的メモリ管理のための空き領域管理用のデータ構造[6] などを割り付けて初期化する．この後，dlm_startup() から戻り，この関数コールに続くユーザプログラムコードの実行を開始する．通信スレッドは，メモリサーバからのメッセージ待ちの状態で稼動しはじめる．

DLM ページとは DLM システムにおけるメモリ管理単位で，DLM スワップの単位でもある．**DLM ページ表**の各エントリは，そのページが割り付けられている rank 番号，ページの先頭アドレスやページ内の現在使用している最後のアドレスなどの情報を保持する．DLM ページ表は，DLM 設定ファイルに指定された計算ノード（ローカルホスト）の DLM 提供メモリサイズの中から最初に割り当てられ，DLM ページ表を除いた残りのローカルメモリサイズをユーザのデータの割付けに用いている．また DLM ページ表は遠隔メモリへのスワップ対象とせずに，計算ノードのローカルメモリに常駐するようにしている．

ユーザプログラムコードの最後に呼ばれる終了関数 dlm_shutdown() は，前述のように計算プロセス（rank0）の計算スレッド（calThread）のみが呼ぶ関数で，この中で，ユーザプログラム終了を全メモリサーバプロセスへ送付する．終了処理のためのメッセージを交換後，計算プロセス，メモリサーバプロセスの両方で MPI_Finalize() が呼ばれて，MPI システムが終了する．ただし，前述のようにメモリサーバプロセスは dlm_startup() 関数内部でこれを行う．

3.2 DLM データの割付けと DLM ページ管理

ユーザプログラムから dlm_alloc などのメモリ割当て要求が起こると，計算ノード内（rank0）のローカルメモリへの割付けを最優先とし，ローカルメモリに空きがない場合には，足りないサイズ分を rank1 以降のメモリサーバノードのメモリに，rank 番号順に順次割り付ける．

計算スレッドで実行されるユーザプログラムが，ローカルメモリ以外にあるデータにアクセスした場合は SIGSEGV で検知し，そのページを保持するメモリサーバに DLM ページ要求を起こす．その際にローカルメモリに余裕がない場合は，ローカルにある DLM ページからスワップアウトする DLM ページを選び munmap し，該当メモリサーバとの間で要求ページとスワップし，新しいページを mmap する．DLM-MPI では，dlm_alloc() だけでなく，初期実装の DLM-socket ではサポートしていなかった dlm_free() も実装しており[6]，より柔軟で動的なメモリの利用を行う応用プログラムにも対応している．このため，dlm_free() などにより，ローカルメモリに余裕が新たに生まれている場合には，メモリサー

バプロセスにスワップアウトするページを選ぶ必要がなく，メモリサーバに必要なページを要求するだけで，受け取ったページをそのまま mmap する場合もある．

一方，メモリサーバでは，dlm_alloc 時にそのノードにページ割当てが要求された場合にのみ，mmap で必要ページ数分のページスロット（DLM ページ 1 つ分を格納する領域をここではページスロットと呼ぶ）を確保する．メモリサーバの持つ DLM ページ表は，DLM ページ番号に対応したエントリに，その DLM ページの格納場所（スロット先頭アドレス）を記録しているにすぎない．スロット先頭アドレスは，格納した DLM ページが計算ノードで使われるときの DLM ページ先頭アドレスとは無関係である．すなわち，メモリサーバでは，計算ノードから要求された DLM ページがどこにあるのかを把握できるようにしているだけである．各メモリサーバの持つ DLM ページ表（というよりスロット対応表）は，自分の持つページの格納場所のみを保持するもので，他のページがどのメモリサーバにあるのかなども関知しない．計算プロセスへの要求ページの送信や，スワップアウトされてきたページの受け取りのたびに，このスロットとして確保されたメモリ領域を mmap や munmap をすることはしない．計算ノードの要求によりページを送って空いたスロットに，計算ノードからスワップアウトされたきたページを格納するという単純な方式で，運用している．したがって，計算プロセスの持つページ表と，各メモリサーバプロセスの持つページ表では，その内容も異なる．計算プロセスで管理するページ表は，DLM 起動時に設定された DLM 利用可能メモリサイズ（全ノード分）に応じたページ数分のページ表であり，各ページがどのメモリサーバにあるか，ページ内のどこまでを利用しているかなどの情報を網羅するが，各メモリサーバが保持するページ表は，自分の持つページのスロット位置情報のみを持つだけで単純である．

また，メモリサーバに初期時に割付されたページを最初に計算プロセスが要求する場合，ページ内データ値は不定であり実際にメモリサーバからページを転送する必要がないので省略し，通信オーバヘッドを省いている．プログラムの最初の部分などで，大容量データの初期値設定などを行う多くの応用にとっては，有用である．

3.3　DLM ページスワッププロトコルと MPI スレッドサポートレベル

この節では遠隔メモリページのスワッププロトコルついて整理する．スワップ処理とは，計算ノードがメモリサーバにページを要求し，メモリサーバからスワップインページを受け取り，その代わりにスワップアウトページをメモリサーバに送り返すという交換処理である．前述のように DLM はキャッシュとして実装されておらず，DLM ページを複数ノードに分散して保持する分散メモリシステムで，1 つのページはいずれかのノードにしか存在しない．

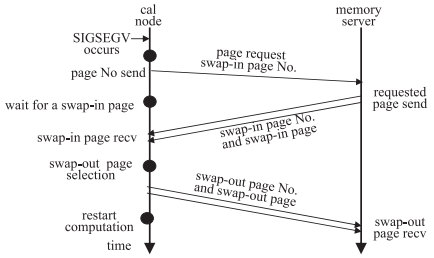

図 5　スワップ処理におけシリアルプロトコル
Fig. 5　serial protocol.

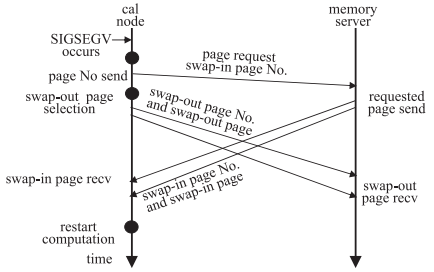

図 6　スワップ処理におけるクロスプロトコル
Fig. 6　cross protocol.

このため，ローカルメモリに利用可能なスペースがないときには，スワップインの際に必ずスワップアウトが発生する．少なくともスワップインページが無事計算ノードに受信されるまでは，ユーザプログラム実行は一時中断されることから，いかにこの中断時間を短くして，効率良くプログラムを実行できるかが性能上のポイントである．この 3 つの作業（ページ置換アルゴリズムによるスワップアウトページ選択作業も入れるなら 4 つ）を行うプロトコルとして，大きく 2 つのタイプが考えられる．1 つは，**図 5** に示すように，(1) ページ要求，(2) スワップインページ受信，(3) スワップアウトページ送信，の順に計算ノードが逐次的に処理を進めるプロトコルで，保守的であるが 1 度に 1 つの通信しか行わず安全なプロトコルである．もう 1 つは，**図 6** に示すように，(1) ページ要求，(2) スワップアウトページ送信，(3) スワップインページ受信，の順に計算ノードがスワップインページが届くのを待つ時間を節約し，先にスワップアウトページの送信をはじめるもので，同時に双方向の通信が行われることを特徴とし，多重通信による効率化を期待するプロトコルである．

ここでは便宜上，シリアルプロトコルとクロスプロトコルとそれぞれを呼ぶことにする．

DLM では，ページスワップの際に出すページと入るページが一時的に両方共存できるように，物理ページスロットには余裕を持たせてある．したがって，クロスプロトコルも利用可能で，DLM-socket では，このクロスプロトコル型のスワッププロトコルを採用していた．ソフトウエア分散共有メモリなどで行われる並列処理と違って，本来，DLM では，計算プロセスがプログラム実行中に他のプロセスからページ要求を受け取る可能性はない．したがって，計算スレッドが 1 つであれば，必ずしも通信スレッドは必要ではない．すなわち，DLM のような用途であれば必ず計算スレッドが中断した状態で通信を行うことになるので，計算スレッド自らが通信する方式であっても，通信スレッドとのやりとりがない分，効率が良い可能性もある．しかし，将来の DLM における計算スレッドのマルチスレッド化や並列処理用の分散共有メモリシステムへの応用のために，DLM-socket 実装では通信スレッドを使ってクロスプロトコルを実装した．DLM-socket では，SIGSEGV で中断した計算スレッドのプログラム実行ができるだけ早く再開できるように，必要なページの要求を計算スレッドが通信スレッドを介さずに直接メモリサーバに送る．この新しいページ要求のメッセージに，スワップアウトするページ本体も添付して，1 度の通信ですむようにベクトル型のソケット通信で送信し，効率化を図っていた．メモリサーバはまず最初のメッセージヘッダ部にある要求ページ番号を読み取り，すぐに要求ページを計算プロセスに送った後，2 番目のメッセージデータであるスワップアウトページデータの受信を行う．このため，計算プロセスではすぐにページ要求のメッセージと一緒に送れるように，スワップアウトするページを選択するアルゴリズムは極力簡単化していた．一方でページ表の更新や mmap, mprotect などを安全に行うために，送られてきたページの受け取りはすべて通信スレッドが一本化して逐次的に行うようにしていた．

DLM-MPI では，ページ置き換えアルゴリズムも積極的に取り入れることも考慮して，スワッププロトコルを変更し，**図 7** に示すように，ページ要求のメッセージとスワップアウトページの転送を分離し，計算スレッドと通信スレッドにおけるそれぞれの処理を並行して進められるようなシリアルプロトコルを試作，実装してみた．手順的には，シリアルプロトコルであるが，スワップページを受信した時点で計算スレッドは計算を再開し，その間に通信スレッドがスワップアウトページを選択して送信するという並列性を持ったプロトコルである．ここでは，AS プロトコル（積極的シリアルプロトコル：Aggressive Serial Procotol）と呼ぶことにする．すなわち，計算スレッドは SIGSEGV 発生直後すぐにメモリサーバプロセスに直接，要求するページ番号を送る．その後，該当メモリサーバが要求したページ

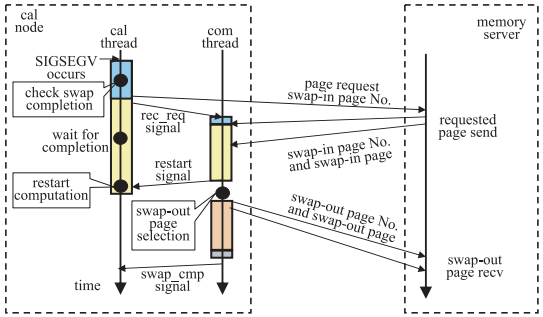

図 7　MPI_THREAD_SERIALIZED で利用可能な積極的シリアルプロトコル AS1
Fig. 7　Aggressive serial protocol for MPI_THREAD_SERIALIZED, AS1.

を送ってくると，通信スレッドが受け取り，それを mmap してから，計算スレッドに知らせる．計算スレッドは中断していたユーザプログラムの実行をすぐに再開する．その間に，通信スレッドは，スワップアウトするページを選択し，このページをさきほどのメモリサーバプロセスへ返信する．したがって，計算スレッドのプログラム実行中に，通信スレッドがページスワップアウト処理をするようにしており，スレッドの処理並列性を生かす機能を組み込んだのが AS プロトコルである．もちろん，通信スレッドがスワップページを選択して送付が完了する前に，再開した計算スレッドのプログラムが次の SIGSEGV 発生による新しいページ要求をする場合も考えられる．この場合には，前回のページ要求にかかわるスワップアウトページ送信が終了するまで，計算スレッドは新しいページ要求の送信を待たせる．

従来の DLM-socket では，ページ要求をすばやくメモリサーバに送れるように，コストのほとんどかからないアドレス順のラウンドロビン方式（ここでは便宜上アドレスラウンドロビン置き換え方式，ARR と呼ぶ）でスワップページを選択するページ置換アルゴリズムを用いてきた．スワップアウトページの選択は，遠隔ページングの通信性能と並んで大きく性能に影響を及ぼすので，今回の DLM-MPI の実装では，スワップページを選択に多少時間がかかっても許されるように，ページ要求とスワップアウトページを一緒に送るのではなく，分離する方式とした．これにより少し時間をかけてスワップアウトページを選択するための余裕ができたので，ページ置換アルゴリズムについても工夫する余地がある．しかし，本論文の実験・性能評価では，従来の単純な ARR 方式をそのまま用いている．

次に MPI とスレッドを併用するうえで重要な MPI スレッドサポートレベルについて触れる．スレッドサポートレベルは MPI ライブラリの実装により様々であるが，MPI の仕様

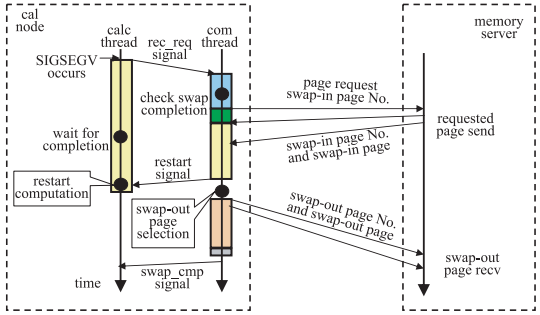

図 8　MPI_THREAD_FUNNELED で利用可能な積極的シリアルプロトコル AS2
Fig. 8　aggressive serial protocol for MPI_THREAD_FUNNELED, AS2.

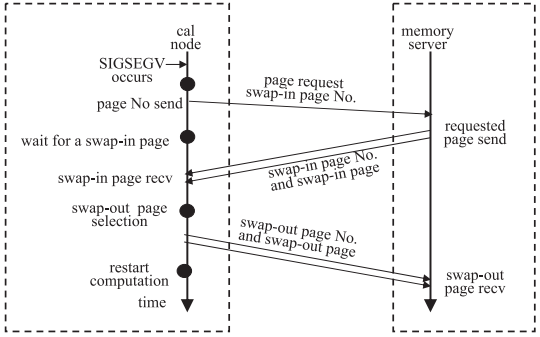

図 9　MPI_THREAD_SINGLE 用の保守的シリアルプロトコル CS
Fig. 9　conservative serial protocol for MPI_THREAD_SINGLE, CS.

では次の 4 つの段階が定義されている．

1. MPI_THREAD_SINGLE：単一のユーザスレッドのみ．
2. MPI_THREAD_FUNNELED：複数スレッド可能，ただし MPI 呼び出しはメインスレッドのみ．ただしメインスレッドとは MPI_Init，あるいは MPI_Init_thread を呼び出したスレッドと定義されている．
3. MPI_THREAD_SERIALIZED：複数スレッドからの MPI 呼び出しは可能であるが，いちどきには 1 つのスレッドからの呼び出しに限る．
4. MPI_THREAD_MULTIPLE：複数スレッドが，自由に，MPI 呼び出しを行える．

利用する MPI システムのスレッドサポートレベルを調べるには，ユーザが希望するスレッドサポートレベルを MPI システムに伝え，それに対して提供できるスレッドサポートレベルを返すような MPI_Init_thread() 関数が提供されている．現状では，最上位レベルのサポートができている MPI 実装は限られており注意が必要である．したがって可搬性を重視するならば，MPI スレッドサポートレベルとして下位 3 つのレベルを考慮しておくことが現実的となる．たとえば，シリアルプロトコルを採用するとすれば，図 7，**図 8**，**図 9** のプロトコルが考えられる．図 7 と図 8 の違いは，必ず 1 つの通信スレッドが MPI 通信を行うように，計算スレッドからの直接の通信をさせない点である．DLM-MPI は，MPI 通信方式を一義に決めるものではないので，用いる環境と MPI ライブラリに合わせたプロトコルを採用することが可能であるが，今回は，図 7 の AS1 プロトコルを試験的に実装して試してみた．現在の DLM-MPI では，計算スレッドが 1 つであることを前提にしているので，このプロトコルは通信スレッドで MPI 通信を逐次化する必要がなく，計算スレッドからメモリサーバへ直接通信を行うことができ，最も効率が良い可能性のあるシリアルプロトコルである．2 つのスレッドからの通信が行われるが，プロトコル上，MPI 通信は逐次化されるので，MPI_THREAD_SERIALIZED をサポートする MPI 実装では稼動可能である．

さらに，通信手順上，最も時間がかかるが，安全である計算スレッド 1 スレッド（通信スレッドを生成しない）による CS プロトコル（保守的シリアルプロトコル，conservative serial protocol）も実装してみた．図 9 に示すように，CS プロトコルは上述の 3 つのスワップ処理作業が順番に行うので同時処理性がなく，性能は期待できないが，どのような MPI 実装でも稼動可能であるという点で汎用性・可搬性が最も優れたプロトコルともいえる．また CS プロトコルは，AS プロトコルと違って，前のスワップが終了していないためにおこる次のスワップ処理の待ち時間を含まないため，応用プログラムによるスワップ要求のタイミング，すなわち応用プログラムの遠隔メモリアクセス頻度などによらず，ほぼ一定の時間でスワップ処理を終了し，これ以上長く時間のかかるスワッププロトコルはないことから，用いる環境の最低限の性能を見積もる指標にもなる．

4. DLM-MPI のオープンクラスタにおける性能評価

4.1 実験環境

性能評価に用いたクラスタは，**表 1** に示すような MPI バッチキューイングシステムで運用されている汎用のオープンクラスタ（東京大学 T2K Open Supercomputer）である．本実験では，2 ノードから最大 16 ノードを用いた．ノード間を結ぶネットワークは Myri-10G [17]

表 1　バッチキューイングシステム運用のオープンクラスタ
Table 1　Open cluster with batch queuing system.

	T2K Open Supercomputer, HA8000
Machine	HITACHI HA8000-tc/ RS425
CPU	AMD QuadCore Opteron 8356(2.3GHz) 4 CPU/ node
Memory	32 GB/ node (936 nodes), 128 GB/ node 16nodes)
Cache	L2 : 2 MB/ CPU (512 KB/ Core), L3 : 2 MB/ CPU
Network	Myrinet-10G x 4, (5 GB/s full-duplex) Myrinet-10G x 2, (2.5 GB/s full-duplex)
OS	Linux kernel 2.6.18-53.1.19.el5　x86_64
Compiler	gcc version 4.1.2 20070626 cc: Hitachi Optimizing C mpicc for 1.2.7
MPI Lib	MPICH-MX (MPI 1.2)

で，1 リンク 1 方向あたり 1.25 GB/s のバンド幅を持ち，双方向同時通信が可能である．DLM の初期評価[4] では，10GEthernet（Myri-10G における 10GEthernet プロトコル利用：Myri10GE software）を TCP/IP で利用していたが，本実験環境で用いる通信リンクも Myri-10G なので，単純比較はできないものの，リンク 1 本あたりのピーク性能はほぼ同じオーダであると考えてよい．用いるクラスタには，各ノード間に 4 本の Myri-10G リンクがあるノード群と，各ノード間に 2 本のリンクがあるノード群とがあり，それぞれ最大 5 GB/s（MX_bonding=4）と最大 2.5 GB/s（MX_bonding=2）の性能がある．実験では，この 2 種の通信性能を持つネットワーク上での性能結果を調べた．用いた MPI ライブラリは，Myrinet 上に実装された MPICH-MX [17] で，スレッドサポートレベルは現状では MPI_THREAD_FUNNELED である．各ノードには，32 GB のメモリが実装され，ユーザには 28 GB までの利用が許されている．

1 ノードに，4 CPU（16 コア）の CPU があるが，MPI バッチシステムによるユーザごとのプロセス割付けはノード単位で，たとえ 1 プロセス，1 CPU コアしか利用しない場合であっても，同一ノードに他のユーザのプロセスが同時に実行されることはない．通信は，フルバイセクションバンド幅が確保される方法でマルチポートスイッチによる接続がなされており，他ユーザのプロセスによる通信の影響がまったくないということはないが，比較的少ない環境にある．このため，同一条件で複数回の測定を行った場合には，他のプロセスの通信トラフィックの影響を考慮し，最良の値を結果として用いるようにしている．また，本

表 2　STREAM ベンチマーク
Table 2　STREAM benchmark.

	Kernel	Code
STREAM	COPY	a(i) = b(i)
	SCALE	a(i) = q*b(i)
	ADD	a(i) = b(i) + c(i)
	TRIAD	a(i) = b(i) + q*c(i)

論文での評価実験では，特に記述のない限りは 2 ノード（計算ノードとメモリサーバノード）を用い，初期評価[4] の結果をふまえ，DLM ページサイズ 1 MB を使用している．ここでいうローカルメモリ率とは，応用プログラムの使用する総データサイズに対するローカルメモリにおかれたデータサイズの比を意味する．

4.2　STREAM ベンチマークによるプログラムレベルでの遠隔メモリアクセス性能

STREAM ベンチマーク[16] を用いて，DLM-MPI システムの遠隔メモリバンド幅を測定した．ここでの計測には AS1 プロトコルを用いている．スレッドサポートレベルにおいては，必ずしも動作を保障されていないが，少なくとも STREAM においては，値の verification なども通っており，通信動作が不安定になるようなことも観測されていない．STREAM は，**表 2** に示すような典型的かつ単純な算術操作などをともなう 1 次元配列に対する連続アクセスを複数回行い，様々な影響をうける 1 回目の計測結果を除外した複数回の実行からの最良値を，アプリケーションプログラムレベルでのバンド幅として出力する．もともとの STREAM ベンチマークでは，配列データのメモリ確保には静的宣言を用いているが，今回の実験のような大容量の静的データをコンパイラが許していないため，すべて動的メモリ確保の形式 malloc に変換し，これを通常プログラムにおけるローカルメモリバンド幅として計測に用いた．DLM の性能測定では，この malloc を dlm_alloc に変換して計測している．

配列のサイズを一定レベル以上に大きくしておくことにより，キャッシュの影響を無視することができる．本測定では，2 種のパラメータセット，（1）100 M 個の要素を持つ 3 つの配列（2.4 GB）と（2）1 G 個の要素を持つ 3 つの配列（24 GB）について，バンド幅を測定した．**図 10** に，2.4 GB の場合についてのローカルメモリバンド幅と遠隔メモリバンド幅の性能を示す．左端から，すべてをローカルメモリ上に malloc して測定した通常実行のローカルメモリバンド幅，DLM システムを利用するが，ローカルノードにおける DLM 利用可能サイズを十分大きくしてメモリサーバノードへのデータ展開をせずにすむ場合のローカルメモリバンド幅，5 GB/s（bonding=4）の通信リンク上での遠隔メモリバンド幅，2.5 GB/s（bonding=2）の通信リンクにおける遠隔メモリバンド幅を示す．

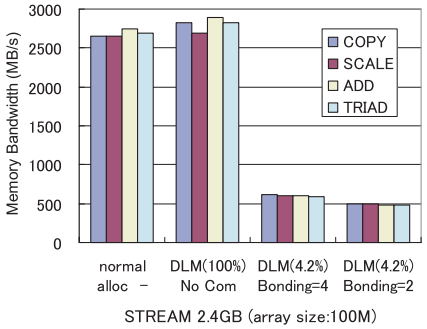

図 10　STREAM（10 M 要素配列・2.4 GB）におけるメモリバンド幅
Fig. 10　Local and remote memory bandwidth (STREAM 100 M elements array size (2.4 GB)).

図 10 の横軸 DLM() 内の%値が示すのは，プログラムで利用するデータサイズのうち，どのくらいの割合のサイズがローカルメモリに収まるかを示した**ローカルメモリ率**である．遠隔メモリの測定では 4.2%のローカルメモリ率が示されているが，STREAM における 2 回目以降の連続アクセスでは，DLM で用いたページ置き換え方式（ARR）の特性により，アクセスするデータはすべてスワップアウトされており，遠隔メモリからの連続的なページスワップインを起こすアクセスになるため，遠隔メモリバンド幅として扱える．

これによると，通常プログラムのローカルメモリバンド幅は 2.7 GB/s 程度で，DLM においてローカルノードメモリのみを利用した場合の性能とほぼ同じ性能である．一方，遠隔メモリバンド幅は，bonding=2 の場合で 493 MB/s，bonding=4 の場合で 613 MB/s に達している．これは，10GbpsEthernet 上で TCP を用いた場合の遠隔メモリバンド幅 380 MB/s [4] に比べれば高い性能を示しているが，通信リンクハードウエアが，理論上提供するはずの通信性能どおりに 2 倍，4 倍というふうに高速化はされないことが分かる．

ただし，STREAM によるメモリバンド幅とは応用プログラムレベルのデータを読み書きしたバイト数から換算したメモリアクセス性能で，DLM の場合には，sigsegv ハンドラ呼び出し，ページ要求送信，スワップアウトページ選択，スワップアウトページの送信と munmap，受信スワップインページの mmap など，多くのオーバヘッドを含んだ値であり，通信リンクが提供するバンド幅性能値とは直接比較できない．遠隔メモリにあるデータを 1 回アクセスするには，書き込み・読み出しを問わず，通常 DLM ページ 2 ページ分の通信（送受信各 1 ページ）が行われるため，DLM ページ 1 MB のすべてにアクセスする応用プログラム（STREAM）の場合であっても，応用プログラムにおけるアクセスバイト数から換算したバンド幅の 2 倍程度の通信バンド幅を内部では使用していることになる．

表 3　STREAM ベンチマークにおけるローカルメモリと遠隔メモリのバンド幅性能
Table 3　Local and remote memory bandwidth on STREAM benchmark.

Memory Size	Array Size	Network Bonding	Local mem Ratio %	COPY MB/sec	SCALE MB/sec	ADD MB/sec	TRIAD MB/sec	Memory Configuration	Memory Bandwidth Type
2.4GB	100M	-	100.00	2657.95	2655.63	2746.98	2695.07	normal malloc	Local mem bandwidth
2.4GB	100M	4	100.00	2829.35	2692.67	2893.66	2820.04	2.4GB-0.1GB 2nodes	DLM　Local mem bandwidth
2.4GB	100M	4	4.20	613.81	607.38	596.70	593.98	0.1GB-2.4GB 2nodes	DLM Remote mem bandwidth
2.4GB	100M	2	4.20	493.70	492.42	483.29	481.99	0.1GB-2.4GB 2nodes	DLM Remote mem bandwidth
24GB	1G	-	100.00	2660.21	2660.27	2750.64	2696.38	normal malloc	Local mem bandwidth
24GB	1G	4	4.35	593.79	578.25	579.82	579.42	1GB-24GB 2nodes	DLM Remote mem bandwidth
24GB	1G	2	4.35	479.03	466.90	470.25	469.86	1GB-24GB 2nodes	DLM Remote mem bandwidth
96GB	4G	2	27.30	443.17	495.43	464.78	463.87	25GB x 4nodes	DLM Remote mem bandwidth
144GB	6G	4	14.56	524.88	520.75	514.33	519.05	20GB x 8nodes	DLM Remote mem bandwidth

また遠隔メモリバンド幅を左右するのはページの送受信時間であるため，通常のメモリアクセスとは異なり，リードが速くライトが遅いというような傾向は隠され，リード・ライトに差は現れない．さらに，STREAM では COPY 以外の操作では，表 2 に示すように単純な算術演算も測定時間に含まれる．バンド幅の計算は，配列全体をスキャンするループ 1 回あたりにかかった時間を，アクセス配列要素数と 1 要素操作あたりのアクセスバイト数（COPY なら double のリード・ライト各 1 回（16 B），ADD ではリード 2 回・ライト 1 回（24 B））で乗じた値で，除算して求める．

表 3 には，2.4 GB と 24 GB の場合の，2 種の通信性能の違う通信リンクでの結果，利用ノード数とメモリ構成などが示されている．また，大きなデータに対し複数ノードを用いて計測したバンド幅も測定した．96 GB データ利用時の遠隔バンド幅は，通信 bonding=2 のリンクで結ばれた 25 GB メモリ/ノードを 4 ノード使用して 443 MB/s であった．144 GB データ利用時の遠隔メモリバンド幅は，通信 bonding=4 のリンクで結ばれた 20 GB メモリ/ノードを 8 ノード使用して 524 MB/s であった．

4.3　Himeno ベンチマークにおける性能

Himeno ベンチマーク [20] は，非圧縮流体解析処理の性能評価のために，ポアッソン方程式解法をヤコビの反復法で解く場合の主要ループの処理速度を計るものである．メモリアクセス負荷の高いベンチマークで多重ループ処理で配列全体をスキャンする．ここでは C プログラム版の ELARGE サイズ（513×513×1025 サイズ，15 GB）を用いた．このベンチマークの性能は MFLOPS で出力されるが，相対実行時間に換算し，通常プログラム（ローカルメモリのみを使用）の場合と DLM の場合を比較した．

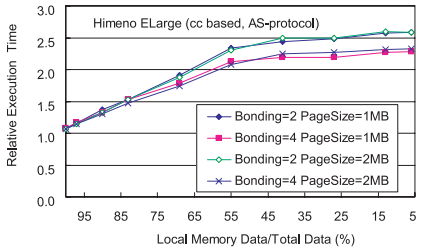

図 11　2 種の DLM ページサイズにおける Himeno ベンチマーク相対実行時間
Fig. 11　Himeno benchmark relative exec time for different DLM pagesizes.

図 11 は，AS1 プロトコルを用いて，2 つの通信リンク（bonding=2 と bonding=4）と 2 つの DLM ページサイズ（1 MB と 2 MB）をそれぞれ用いた場合の通常プログラム実行に対する相対実行時間を示している．横軸はプログラム使用データサイズに対するローカルメモリデータの割合，すなわちローカルメモリ率である．Himeno ベンチマークでは通信 bonding の違いによる性能差があまり大きくないことが分かる．通常，DLM ページサイズとして，10GbpsEtnernet で最適であった 1 MB を用いているが，ネットワーク性能を十分活かしきれていない可能性もあると考え，DLM ページサイズを 2 倍の 2 MB にした場合の性能も合わせて調べている．しかし 1 MB のページサイズとの差はほとんどなく，ページサイズを大きくすることによるメリットはないということが分かった．ローカルメモリ率 6.9%のとき（いい換えると 93.1%のデータが遠隔メモリにあるとき）であっても，通常実行（ローカルメモリのみを利用）の場合の 2.3 倍（bonding=4）から 2.6 倍（bonding=2）程度の速度低下で DLM プログラムが実行できることが分かる．このとき，bonding=4 は bonding=2 に比べ 13%程度性能が高い．

図 12 は，同じサイズの DLM データを利用する場合にメモリサーバプロセス・ノードの数の違いが影響するのかを示したものである．これによると，ELARGE（15 GB）の Himeno ベンチマークを処理する際に，2，4，8 ノードとノード数を変えても，利用するメモリ総サイズと，ローカルメモリ率が同じならば，性能に変化はないということが分かる．これは，前述のようにすでに文献 4) でも確認されているが，大きなデータサイズの場合においてもメモリサーバ数が性能に影響しないことが確認された．ここで用いた ELARGE は，15 GB と非常に大きく，たとえ 100%ローカルメモリを利用する場合であっても，比較的低い値となっている．これは利用している T2K クラスタでは 1 ノードあたり 32 GB メモリを搭載しているが，4 CPU がメモリを NUMA 型で用いる構成であるため，計算を行った CPU の直

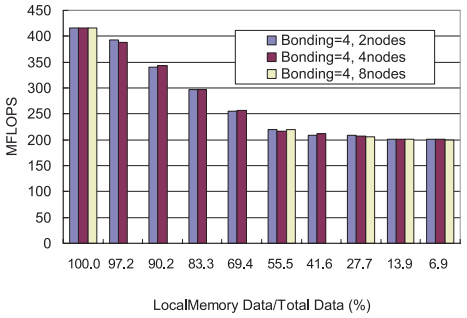

図 12　複数ノード上における Himeno ベンチマークの性能
Fig. 12　Himeno benchmark performance on multiple nodes.

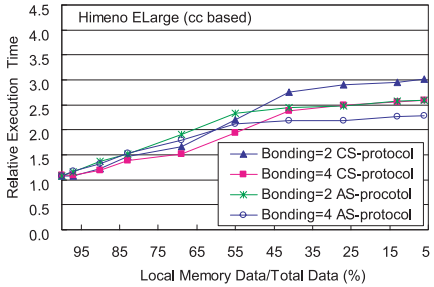

図 13　スワッププロトコルの違いによる Himeno ベンチマーク性能差
Fig. 13　Himeno benchmark relative exec time for AS and CS protocols.

近のメモリ（8 GB）にはデータが入りきらない状態が起きている影響もあると考えられる．

次に，スワッププロトコルの性能比較を行った．**図 13** には，1 MB の DLM ページサイズで 2 つのスワッププロトコル（AS1 と CS），2 つの通信リンク（bonding=2 と bonding=4）をそれぞれ用いた場合の性能を示す．ローカルメモリ率が低くなると，CS プロトコルは AS1 プロトコルよりも性能が低下するが，スワップ頻度の低いローカルメモリ率が高い領域では差は少ない．ローカルメモリ率 6.9%のとき，CS プロトコルは，bonding=2 で 17%，bonding=4 で 19%，AS1 プロトコルよりも性能が低い．

最後に，もともとの Himeno ベンチマークには定義されていないが，ELARGE データよりも大きい XLARGE（1025×1025×2049，112 GB）を定義して，bonding=4 の通信リンクを持つ 20 GB メモリ/ノードを 6 ノード利用して実行してみた．このときのローカル

表 4　6 種の通信環境
Table 4　6 Communication environments.

Name	Machine	Network	Protocol
sock-10G	T2K	Myri-10G, IP on Myrinet	socket
sock-HP	HP	1GbitEthernet	socket
MPI-bon2	T2K	Myri-10G bonding=2	MPICH-MX
MPI-bon4	T2K	Myri-10G bonding=4	MPICH-MX
mpich-HP	HP	1GbitEthernet	MPICH
openmpi-HP	HP	1GbitEthernet	Open MPI

メモリ率は 17.4%で，性能は 179.8 MFLOPS，相対実行時間は 2.32 倍であった．さらに，もともとの Himeno ベンチマークは float 配列であるが，これを double 配列にした処理を dXLARGE（1025×1025×2049 double，241 GB）を定義し，bonding=4 の通信リンクを持つ 20 GB メモリ/ノードを 12 ノード利用して実行した．使用メモリサイズは 241 GB でローカルメモリ率は 8.1%であるが，性能は 88.77 MFLOPS，相対実行時間は 4.68 倍であった．

このように，Himeno ベンチマークは，メモリアクセス負荷の高いプログラムだといわれているが，高速通信ネットワークと 1 MB 程度の DLM ページサイズを用いると，多次元配列とはいえ近傍処理を行う計算ループを繰り返すため，メモリアクセス局所性があり，ローカルメモリ率が低く，大きなサイズのデータを処理する場合でも，現実的な時間で実行ができることが分かる．

4.4　各種通信リンクにおける AS1 スワッププロトコルにおける処理時間成分

前述の STREAM では応用プログラムレベルでのアクセス性能を求めたが，AS1 プロトコルにおける 1 回のスワップ処理においてどのような時間がそれぞれの処理にかかっているのかを，**表 4** に示すような 6 種類の通信リンクと 2 種のホスト（T2K と**表 5** に示す HP）において，それぞれ 2 種類のローカルメモリ率の場合を持つ Himeno ベンチマーク MIDDLE サイズを例に調査した．**図 14** と**図 15** は，2 種のホストと 2 つの通信方式（ソケットと MPI），3 つの MPI ライブラリ（TCP/IP 用の MPICH，TCP/IP 用の OpenMPI，Myri-10G 用の MPICH-MX），4 種の通信性能リンク（1GbpsEthernet，10GbpsEthernet（IP on Myrinet），Myri-10G bonding=2，Myri-10G bonding=4）において，計算スレッドと通信スレッドでどのような処理時間成分があるのかを示している．表 4 の上 2 行に示すソケット利用 DLM とは，初期に MPI バッチシステムの起動した MPI プロセスを利用するのみで，その後の通信にはソケット通信を用いる．この 2 つの図のうち高速通信リンク

表 5　HP クラスタ　1 Gbps の Ethernet クラスタ
Table 5　HP cluster with 1GbpsEthernet.

	HP
machine	HP ML150 G2
CPU	Xeon 2.8GHz x 2CPU　HyperThread
memory	1GB
Cache	L2：1MB/CPU
OS	Linux kernel2.6.20-1.2320.fc5 x86_64
NIC	Broadcom 5721 PCI-Express Gigabit NIC
Network	1GbitEthernet
MPI	MPICH-1.2.7, Open MPI 1.3.2

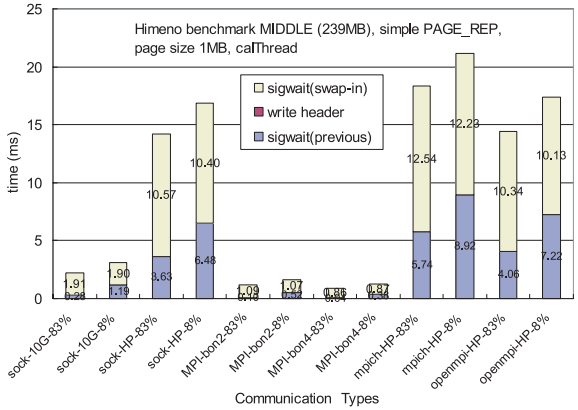

図 14　HimenoM　6 種の通信リンクにおける AS プロトコルにおける計算スレッドの時間成分
Fig. 14　HimenoM time compornents of AS protocol in calthread.

利用の場合のみを**図 16** に示す．また 15 GB を用いる HimenoELARGE における高速通信リンクにおける成分を**図 17** に示す．いずれの図においても，それぞれの成分は，図 7 に示す色成分に対応している．計算スレッドの下の薄紫部分が前のスワップアウトが終了しておらず待たされる時間，その上の黄色部分が今回のスワップインページを待つ時間である．この黄色の部分は，通信スレッドにおける黄色の部分にほぼ対応しており，メモリサーバから受け取ったページを読む時間に対応する．

図 15 が示すように，ネットワークの速度が 1 Gbps から 10 Gbps になると，ページの送受

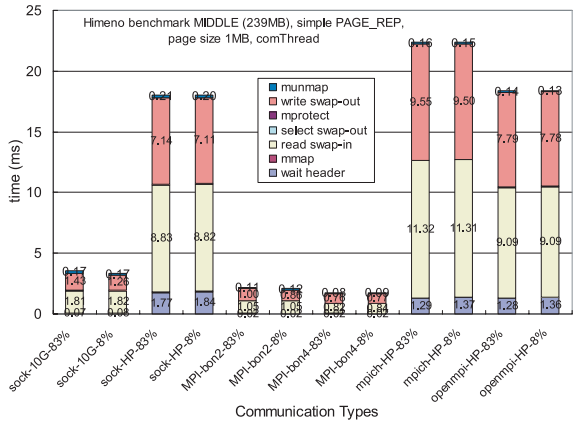

図 15　HimenoM　6 種の通信リンクにおける AS プロトコルにおける通信スレッドの時間成分
Fig. 15　HimenoM time compornents of AS protocol in comthread.

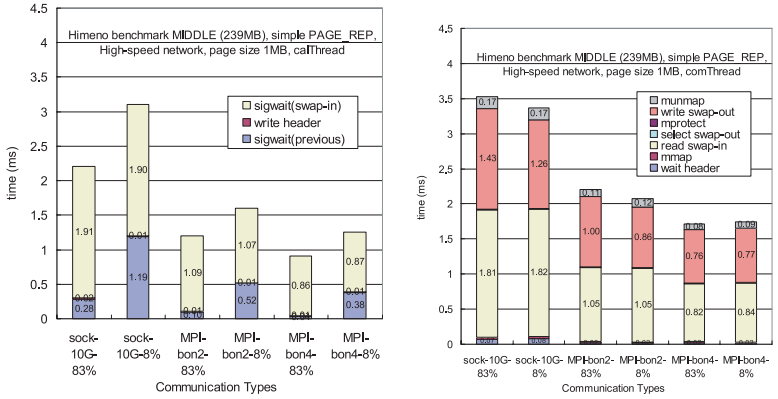

図 16　HimenoM　高速通信リンクにおける AS1 プロトコルにおける計算・通信スレッドの時間成分
Fig. 16　HimenoM time compornents of AS1 protocol.

信（write swap-out と read swap-in）の通信速度は約 5 倍になった．図 16，図 17 の通信スレッドの図が示すように，IP on Myrinet の 10GbpsEthernet から Myri-10G の bonding=2 になると，性能は約 1.8 倍になり，bonding=2 から bonding=4 になると，性能は約 1.3 倍になった．ここで，通信ライブラリの違いによる性能差を見ると，図 14 のように，HP にお

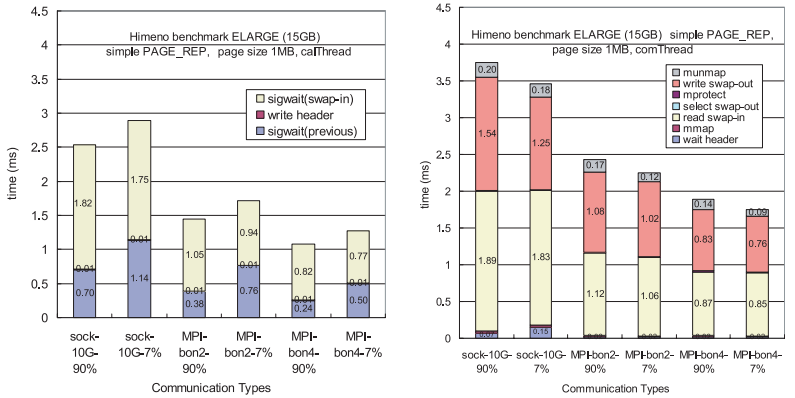

図 17　HimenoEL　高速通信リンクにおける AS1 プロトコルにおける計算・通信スレッドの時間成分
Fig. 17　HimenoEL time compornents of AS1 protocol.

けるソケットと MPICH と Open MPI のページの送受信の通信時間の比は 1：1.28：1.03 と，ソケットが最も短い．MPI でも，MPICH よりも Open MPI の方が短く，実装により性能差があることが分かった．

また図 16，図 17 の計算スレッドの図が示すように，sigwait(previous)（計算スレッドにおける前回のスワップ待ち）の時間も，ネットワーク速度が速くなるほど短くなる．特にローカルメモリ率が高いと，スワップ頻度が小さいために前回のスワップから次のスワップ発生までの間隔が長くなり，計算スレッドが前回のスワップを待つ時間が非常に短くなる．しかし今回計測で用いた AS1 プロトコルでは，同時性を期待してのプロトコルではあったが，sigwait(previous) の結果から当初の予想よりも計算スレッドが待たされる割合が大きいことが分かった．

図 18 に，HimenoMIDDLE の処理を CS プロトコルを用いた際の 1 スワッププロトコルの時間成分を示す．図 16 の計算スレッドと比較してみると，同一通信リンクを用いた場合，ローカルメモリ率が大きいとき，AS1 は CS の 2 倍の性能で，ローカルメモリ率が小さいとき，AS1 は CS の 1.4 倍の性能であることが分かる．

図 16，図 17 の通信スレッドでは，ローカルメモリ率が高いと ELARGE(16 GB) に比べ MIDDLE(239 MB) の write swap-out の時間が短いことが分かる．ELARGE でローカルメモリ率が高いとローカルメモリを多く使用するため，NUMA 型の影響で利用 CPU から遠いローカルメモリにアクセスすることが多くなる影響も考えられる．

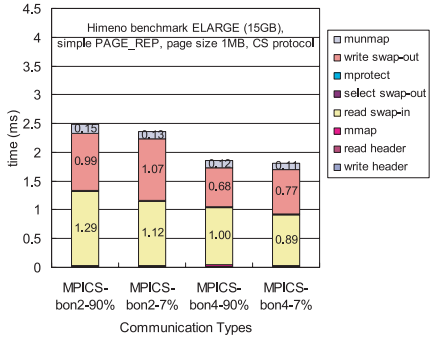

図 18　HimenoM　高速通信リンクにおける CS プロトコルにおける計算プロセスの時間成分
Fig. 18　HimenoM time compornents of CS protocol.

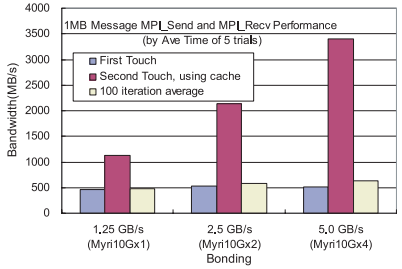

図 19　実験環境（ha8000）における MPI_Send・MPI_Recv による 1 MB メッセージ転送時のバンド幅
Fig. 19　Bandwidth of 1 MB message between 2 nodes in ha8000 (MPI_Send and MPI_Recv).

また，計算スレッドにおけるスワップイン待ち時間（黄色の sigwait(swap-in)）が，DLM ページ（1 MB）の通信時間に対応するので，この時間から計算したバンド幅が T2K・Myri-10G での DLM の実質通信リンク性能に近い．図 16 の計測値から計算した通信リンク性能値は，ソケットによる 10GbpsEthernet(IP on Myrinet) では 526 MB/s，MPI の bonding=2 では 935 MB/s，MPI の bonding=4 では 1149 MB/s になる．

実際に，実験環境（ha8000）上で，2 ノードを用いた MPI_Send と MPI_Recv による単純なループバックテストを行い，1 MB メッセージのバンド幅を調べたところ，**図 19** のようになった．各通信リンク性能（bonding1，2，4）において，3 種類の測定条件において 5 回の試行を行いその平均を測定した．同じアドレスへの読み書きが生じる同じページの送受信を繰り返す実験での，初回の性能（First Touch）と，2 度目の送受信時の性能（Second Touch，すなわち初回でデータはキャッシュに入っている）を示している．これによると，同じページを 2 度目に送る際には，キャッシュ効果でメモリアクセスオーバヘッドが小さくなり，bonding=1 で 1,129 MB/s，bonding=2 で 2,142 MB/s，bonding=4 で 3,395 MB/s という NIC の通信リンクの性能に近い値が得られる．3 つ目の測定値（100 iteration averagge）は，送信データアクセス時のキャッシュの効果を避けて，1 MB データを毎回違うアドレス領域に受け取り，違うアドレスの 1 MB のデータを送り出すというような送受信を行ったときの性能を示す．100 回ループバック送受信の平均値である．この場合は，ほぼ First の場合と同様の性能で，bonding=1 で 483 MB/s，bonding=2 で 580 MB/s，bonding=4 で 637 MB/s の性能にとどまる．すなわちメモリバンド幅にほぼ制限され，通信リンク性能の恩恵がほとんど得られていない．さきの AS1 でのページリード時間によって計算したバンド幅は，なんらかのキャッシュ効果を得ているためか，通信リンク性能（bonding の違いなど）によって差が出ている．

本論文で用いた MPI のスレッドサポートレベルは，MPI_Init_thread 関数で調べたところ，MPICH-TCP/IP と T2K の MPI-MX では，MPI_THREAD_FUNNELED，OpenMPI-TCP/IP では，MPI_THREAD_SINGLE であった（MPICH2-TCP/IP では MPI_THREAD_MULTIPLE をサポートしている）．しかしいずれのクラスタにおいても，AS1 プロトコルを用いて，通信が不安定になったり，実行時間が変動したりすることはなく，ベンチマーク中 verify を行うものについてはほぼ成功であった．しかし，詳しく調査すると，まれに扱うデータサイズの大きさによらず，verify が通っていないときがあることが分かった．これはある特定の応用における特定のローカルメモリ率などの条件で再現性のあることから，MPI によるスレッドサポートレベルの影響である可能性がある．したがって，現状では AS2 または CS プロトコルを用いたほうが，MPICH_MX では安全といえる．本実験で MPI による AS1 プロトコルで問題のあったパラメータセットでは，MPI による CS プロトコルやソケット通信（1 Gbps や 10 Gbps Ethernet）では問題が起きなかった．

4.5　NAS Parallel Benchmark における性能

ここでは応用プログラムの例として，多くの人に内容的になじみのある NAS Parallel Benchmark（NPB）[18] プログラムの逐次 C プログラム版（NPB2.3-omni-C [19]）を用いて，DLM-MPI の性能を調べた．本来 NPB は並列処理性能を調べるベンチマークであり，DLM が対象とする並列化がしにくい逐次処理プログラムではなく，むしろ並列処理応用に適した処理応用であるが，サイズや処理の種類などが豊富であることから，今回はこれを用いた．

表 6　用いた NPB における各パラメータ
Table 6　NAS Parallel Benchmark used in the experiment.

	CG-C	IS-C	MG-C	BT-C	SP-C	FT-C	IS-D
Size Parameter	15000	2**27	512**3	162**3	162**3	512**3	2**31
DLM Data Size (GB)	1.156	1.610	3.581	5.080	1.317	7.014	51.538
Data Type in original	static	static	malloc	static	static	static	static
Num of Data and malloc	14	4	974,107	16	20	8	4
Iteration Shrink Ratio	10/75	-	-	10/200	10/400	-	-
Normal Exec Time (static) (sec)	88.01	43.58	-	225.51	-	496.98	-
Normal Exec Time (malloc) (sec)	87.82	42.27	190.90	226.66	85.55	497.19	1809.03

NPB ベンチマーク結果出力は，応用プログラム中，繰返し部分などの並列処理コア部分の時間のみに注目している．通常，その並列処理コア部分の前に，乱数発生やファイル読み込みによるデータの初期化があったり，コア部分終了後に，結果の検証部分がついていたりする．このため，実際のプログラム全体の実行時間は，出力されるコア部分の時間よりも非常に長くなっている場合も多い．今回の性能評価では，本来のベンチマーク出力値をそのまま用い，コア部分のメモリアクセス性能にかかわる時間になっている．

各ベンチマークにおいて，NPB のデータサイズクラスを決める中心的データ（配列や，malloc）を DLM データに置き換えて計測を行った．ベンチマークによっては，コア部分の処理にすべての中心的データを利用しない場合もある（初期化や検証に用いる）．NPB における性能結果のローカルメモリ率は，全体の DLM データのうちどの程度がローカルメモリにあるかを示しているため，見かけ上，ローカルメモリ率が低いにもかかわらず，性能が悪くならないように見える場合もある．実際の応用処理においても，初期化時間に比べコア部分処理時間が長いが，全体データサイズに対してコア部分で用いるデータが局所的であるということもあるので，そのままとしている．

NPB2.3-omni-C [19]（OpenMP の pragma は無効）から，IS，CG，MG，FT，BT，SP の 6 種について DLM 利用時の性能を調べた．用いたデータサイズは，この NPB2.3-omni-C が提供する最大サイズであるクラス C を用いた．最新の NPB3.3 には，大きなデータクラスであるクラス E やクラス D が導入されているが，もともとあった 8 種の NPB プログラムのうち IS を除きすべてが FORTRAN で書かれており，DLM が対象とする C プログラムのベンチマークとしては利用できない．ここでは，NPB3.3 で用いることのできる C プログラムのうち，IS のクラス D（NPB3.3 の IS 最大クラス）についても実験を行った．**表 6** に，用いたプログラムのデータサイズやパラメータ，データの個数，データ割付けのタイプ（静的・動的），ローカルメモリのみを利用する通常実行の場合の実行時間などを示す．プログラムによっては，計測時間を短縮するために，繰返し回数を削減している．

図 20〜図 26 に各プログラムの DLM-MPI 性能を示す．図はいずれも，横軸はローカルメモリ率，縦軸は表 6 に示した通常実行の実行時間に対する相対実行時間を表している．

4.5.1　IS における性能

IS は整数ソートで，クラス C（IS-C）は 2^{27}，クラス D（IS-D）では 2^{31} のサイズの 3 つの整数配列を用い，それぞれ 1.6 GB，52 GB のデータを使用する．IS-C は他のプログラムに比べてデータサイズが小さいが，IS-D では IS-C の int 配列に代えて long 配列を使用し，サイズも int の限界まで増加させており，IS-C に比べデータサイズが巨大化している（なお，IS-D のコードの一部に long に対応していないバグがあったので修正している）．**図 20** は NPB3.3 の IS-D の結果であるが，IS-C も同様な傾向を見せる．ローカルメモリ率が 41%以上では速度低下はなく相対実行時間は 1.0 倍を維持している．IS と後述する CG では，コア計算部分で用いるデータが，プログラム全体で用いるデータの一部であるため，ローカルメモリ率が一定の値まではまったくスワップがおきておらず，通常実行と差がない．ローカルメモリ率が 30%〜4%のとき，相対実行時間は 1.45 でスワップ回数は 174 K 回，ローカルメモリ率が 2%のとき，相対実行時間は 4.34 でスワップ回数は 2,475 K 回になる．しかし用いるデータサイズ（51 GB）に比べれば，IS のスワップ回数は少ないといえる．

4.5.2　MG における性能

MG-C は，512×512×512 のサイズのデータで 3.6 GB を使用する．このプログラムの特徴的なことは，非常に多くのデータをすべて動的割付けで確保することである．クラス C の MG プログラムで呼ばれる malloc（あるいは dlm_alloc）は，約 97 万回にも達している．**図 21** に示すように，相対実行時間はローカルメモリ率に応じて増加し，ローカルメモリ率 100%で 1.19，6%のときには 3.55 倍（bonding=4）にまで低下する．ローカルメモリ率の低下にともないスワップ回数は増加し，ローカルメモリ率が 73%のとき 134 K 回，44%のとき 204 K 回，15%のとき 310 K 回になる．

この応用においては，通信ネットワークの性能差が DLM 性能差として現れており，bonding=4 通信では，bonding=2 通信の場合に比べ，最大で 19%性能が高い．IS や CG に比べ，ローカルメモリ率が高い領域から，性能が低下するということは，遠隔スワップが頻繁におきており，メモリアクセスの負荷も高いことを示している．このような状況では，用いる通信ネットワークの性能差が比較的大きく DLM 性能に現れてくる．

4.5.3　CG における性能

CG-C は，14 の異なる型とサイズの配列を用い，全部で 1.2 GB のデータを使う．ローカルメモリ率が 100%から 40%までは，相対実行時間は IS と同様に 1.1 とほとんど変化せず，コア部分ではスワップは起きていない．しかし**図 22** に示すように，ローカルメモリ率が 40%をきると，遠隔メモリへのスワップが始まり，相対実行時間が 3〜4 倍まで増加する．CG はインデックス配列を介したデータアクセスが行われるため，遠隔メモリにデータが出た場合には，性能が低下しはじめる．同様の傾向が 10GbpsEthernet における CG（クラス B）の性能評価[4] でも見られている．ローカルメモリ率が 36%のときスワップ回数は 55 K 回，ローカルメモリ率が 9%のときスワップ回数は 111 K 回になる．CS プロトコルは測定にノイズを含んでいるが，ローカルメモリ率が 4%のとき，AS1 プロトコルに比べ，20%程度性能が劣る．

CG はジョブが長いため繰返し数を 10/75 に短縮しているが，全体ジョブの時間（10 回分）は**図 23** のようになる．したがってこの応用では，ローカルメモリ率 43%程度まではジョブ実行時間は通常実行とほぼ同等であるが，ローカルメモリ率 3%になると，コア部分は 4 倍程度であってもジョブ全体では 90 倍の時間がかかることになる．繰返し回数を増やした場合には，相対的にコア部分の処理が支配的となりジョブ全体時間との差は少なくなる．

4.5.4　FT における性能

FT-C は，512×512×512 のサイズ 3 つの複素数配列を使う 3 次元 FFT 処理で全体で 7 GB のデータを使用する．繰返し処理では異なる次元方向に離散ギャップでアクセスされるため，多くのスワップを引き起こす負荷の高いプログラムである．FT のスワップ回数は，コア部分だけでもローカルメモリ率 92%で 64 K 回，76%〜46%で 130 K 回である．FT はデータのほとんどをコア処理部分で利用する．**図 24** に示すように，AS1 プロトコルでの相対実行時間は，ローカルメモリ率 100%で 1.16，ローカルメモリ率 19%で 1.69（bonding=4）と 1.86（bonding=2）を示す．したがって，この範囲では MG や CG に比べ緩やかな性能低下を示す．しかし，**図 25** に示すように，ローカルメモリ率が 15%になると，相対実行時間は 54.1（bonding=4）と 71.2（bonding=2）にまで急激に増加する．これは後述するように 1 回の繰返し内でアクセスするデータのワーキングセットが，ローカルメモリサイズを超えてしまったためと考えられる．通信ネットワークの性能差もこの時点で現れてくる．

4.5.5　BT・SP における性能

BT-C は，162×162×162 サイズ，5 GB のデータを使用する 3 重対角方程式を解くプログラムである．各次元について順次処理する．200 回の繰返しを 10 回に短縮して計測して

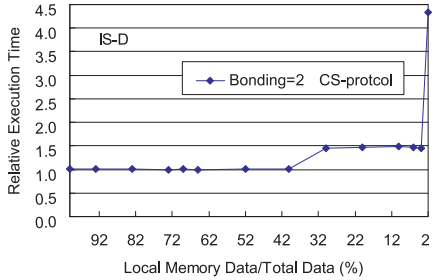

図 20　NPB IS-D 相対実行時間（CS-prot）
Fig. 20　NPB IS-D relative exec. time (CS-prot).

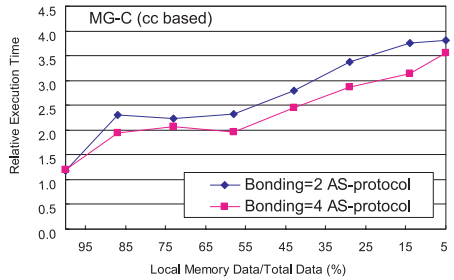

図 21　NPB MG-C における相対実行時間（AS-prot）
Fig. 21　NPB MG-C relative exec. time (AS-prot).

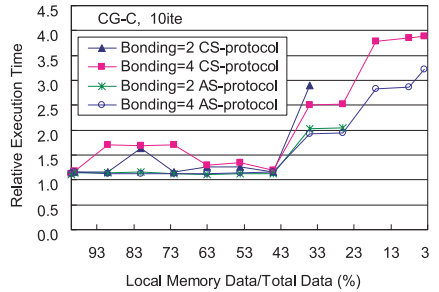

図 22　NPB CG-C 相対実行時間（AS，CS prot）
Fig. 22　NPB CG-C relative exec. time (AS, CS-prot).

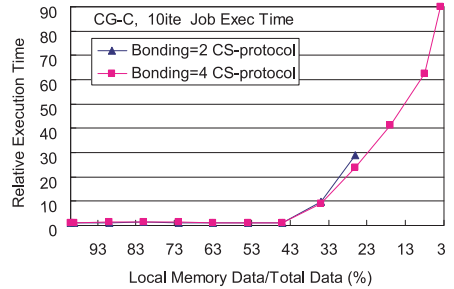

図 23　NPB CG-C ジョブ相対実行時間（CS-prot）
Fig. 23　NPB CG-C relative job time (CS-prot).

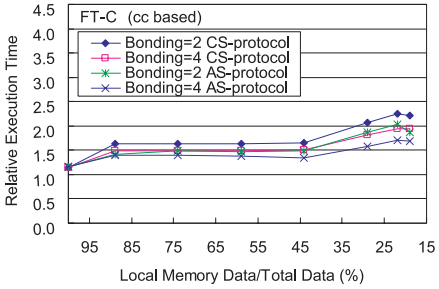

図 24　NPB FT-C 相対実行時間（AS，CS-prot）
Fig. 24　NPB IS-C relative exec. time (AS, CS-prot).

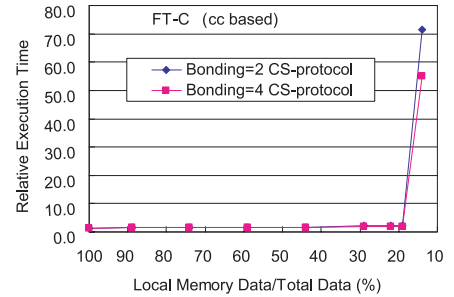

図 25　NPB FT-C 相対実行時間（CS-prot）
Fig. 25　NPB FT-C relative exec. time (CS-prot).

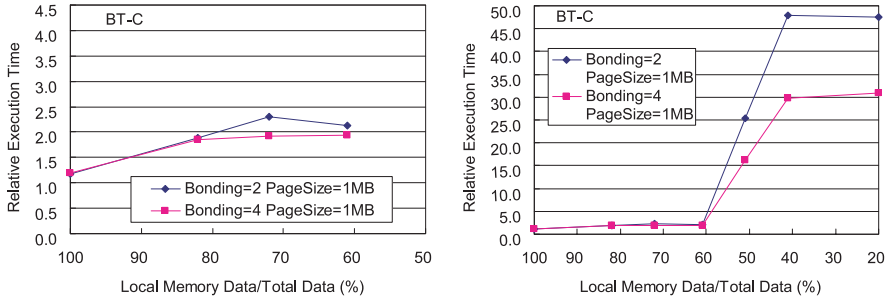

図 26　NPB BT クラス C における相対実行時間（AS-prot）
Fig. 26　NPB BT-C relative exec. time (AS-prot).

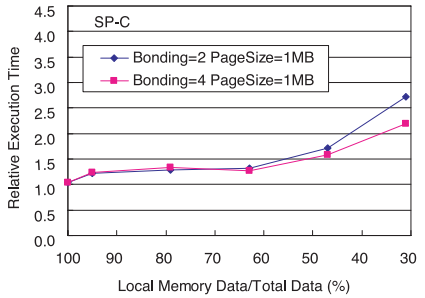

図 27　NPB SP-C における相対実行時間（AS-prot）
Fig. 27　NPB SP-C relative exec. time (AS-prot).

いる．FT と同様に，BT でも，ローカルメモリ率が 60%を切るまでは，非常に穏やかな性能低下を示す．**図 26** の左に示すように，相対実行時間は，ローカルメモリ率 100%で 1.18，ローカルメモリ率が 62%で 1.94（bonding=4）と 2.14（bonding=2）を示す．しかし，図 26 の右に示すように，ローカルメモリ率が 21%では，31.00（bonding=4）と 47.46（bonding=2）まで低下する．

この原因は，コア処理部分の繰返し処理でアクセスするデータのワーキングセット（アクセスするデータの個数）が，ローカルメモリサイズを超えてしまうことにある．すなわち 1 回の繰返し処理中に，通常ならば次回の繰返し時に利用できたはずのページも 1 度スワップアウトして他のページをスワップインする必要が出てくるためと考えられる．BT は，3 次元〜6 次元の多次元配列を 16 個利用しており，1 回の繰返し中で多くの配列要素にアクセスする．これに対し，Himeno ベンチマークでは 4 次元配列に対して 4 重のループで近傍処理を繰り返すものであるが，1 回の繰返し中では，1 つの値の更新計算に対し 28 個の要素しかアクセスしない．Himeno ベンチマークは，すべての DLM データをコア処理部分で利用し，計算とメモリアクセス比の点ではメモリ負荷の高い応用ということになるが，1 回の繰返し処理中でのデータアクセスワーキングセットが比較的小さいために，低いローカルメモリ率においても，DLM プログラムの速度低下が小さい．

SP-C は，162×162×162 のサイズ配列を用い全体で 1.3 GB のデータを使用する．本実験では 400 回の繰返し回数を 10 回に短縮して時間を計測している．**図 27** に示すように，ローカルメモリ率 32%程度までは，2.19 倍（bonding=4）と 2.72（bonding=2）の性能低下にとどまっているが，それを過ぎると，爆発的に実行時間が増加し，計測自体が難しい．bonding=4 でローカルメモリ率が 16%のときには，427 倍という桁違いの速度低下が計測されている．これも，先の BT と同様の理由で，1 回の繰返し処理内でアクセスされるワーキングセットが BT 以上に大きいことが原因であると考えられる．

4.5.6　データアクセスワーキングセットと DLM ページサイズ

NPB の中で，BT，SP，FT のコア処理部は，いずれも複数の多次元配列を扱い，異なる次元方向にアクセスする関数を 1 回のループに複数個含んでいる．このため，1 回の繰返しの中でアクセスされるデータワーキングセットが他の応用に比べ大きく，利用できるローカルメモリサイズがそれよりも小さくなると，その時点から極端な速度低下が観測される．特に今回の実験では，1 MB の DLM ページサイズを用いているため，ローカルメモリ内に入るページ数が少なくなり，大きいデータワーキングセットに含まれる不連続アドレスデータのすべてを格納することが難しくなる．BT および SP では，DLM ページサイズを 128 KB〜16 KB 程度に小さくすると，スワップ回数が非常に低減し，性能低下を抑えられることが実験で確かめられている．たとえば，クラス B（BT-B）の実験では，ローカルメモリ率が 25%のとき，1 MB の DLM ページサイズで相対実行時間は 684.5 であるが，DLM ページサイズ 16 KB を用いると 5.5 にまで低減する．さらに，ローカルメモリ率が 5%では，DLM ページサイズが 512 KB のとき 1,037 であった相対実行時間を，DLM ページサイズを 32 KB 以下にすると 17.5〜22.9 程度に抑えることができる．どの程度のローカルメモリ率から極端な性能低下が生じるかは，全体プログラムで利用するデータサイズに占める応用の持つワーキングセットサイズの比率で決まるため，扱うクラスによって変化する．NPB のような処理では，1 回の繰返しにおけるワーキングデータセットは，全体サイ

ズに比例して増加することはなく，応用の種類によってほぼ同程度なので，小さいデータクラスであるほど（すなわちクラス C よりクラス A のほうが），ローカルメモリ率が高い時点で速度低下が発生することが多い．

すなわち，ワーキングセットに含まれるデータが多く，さらにそれらが不連続アドレスにある応用では，DLM ページサイズが大きいと，1 ページ内に含まれる利用しない無駄なデータを転送をすることよる通信オーバヘッドが増加するだけでなく，限られたサイズのローカルメモリの中に入れられるワーキングセットのデータ個数を少なくしてしまい，頻繁なスワップを引き起こす．したがって，1 回の繰返しにおけるデータアクセスのワーキングセットサイズとローカルメモリサイズが同程度になる場合には，DLM ページサイズの選択にも注意を払いバランスをとることが必要になる．

4.5.7　DLM 利用の効果

計算に比べメモリアクセスを多く含む応用では，当然ながら，性能の低下やスワップ発生頻度が高まることが多い．しかし，たとえデータサイズ，メモリアクセス頻度が同じでも，データアクセスパターンに局所性が高い場合，すなわち，前述のように，全体の処理データサイズに比べ，コアの処理部分でのデータアクセスワーキングセットが小さい応用では，ページスワップの回数は高くならず，大きな DLM 性能低下は起きにくい．また，同じワーキングセットを持つ応用であっても，連続アドレスのアクセスが多い応用では，大きな DLM ページサイズを用いることで通信オーバヘッドを減らすことができ効果的である．

一方で，最も望ましくないメモリアクセスパターンの 1 つは，インデック配列による間接データアクセスで，1 つのデータ取り出しに複数回のメモリアクセスが必要であるというだけでなく，このような場合の多くは，インデックス配列そのもののアクセスが連続であっても，中に含まれるインデックスがランダムに近く（こういう場合にこそ，インデックス配列アクセスを用いることが多い），局所性のほとんどないメモリアクセスが発生することになる．たとえば，NPB の IS の検証処理部分はその典型的例である．IS は，ランキング値を結果として求めるソート処理なので，最終検証部分の処理は以下の式のような，最悪な状況を発生させる．

```
for( i=0; i<NUM_KEYS; i++ )
    key_array[--key_buff_ptr_global[key_buff2[i]]] = key_buff2[i];
```

むしろコア処理部分にはそういう部分はないため，先に示したように性能低下は少ないが，この検証部分にはあまりに時間がかかるために，ローカルメモリ率が小さくなった場合にはスキップして計測を行った．たとえば，クラス B の IS の検証付き（402 MB，ローカルメモリ率 20%）実行では，1 MB の DLM ページサイズの場合 8 時間を要している．このときのスワップ回数はコア部分が 2 K 回で，検証部分 15,840 K 回と 8,000 倍にもなる．

このような応用に対しては，少なくともインデックス配列は，dlm 指定をせずにすむようなサイズなら，ローカルメモリに置く，あるいや前述のように小さい DLM ページサイズを用いるなどの工夫が必要となる．

5. おわりに

本論文では，クラスタを逐次処理応用のメモリ資源として利用するという新しい概念・恩恵を，より多くの人が享受できるように，ノード間通信として MPI のみを用いた DLM を新たに設計し，汎用性・可用性を高めた．これにより，MPI バッチ運用のオープンクラスタにおいても DLM が利用可能になった．従来，クラスタには無縁であった大容量メモリを消費する逐次処理応用を持つユーザが，オープンクラスタが提供する最新の高速通信環境の恩恵を享受して，大容量メモリを容易に使えるようになった．

本研究で得られた成果を以下にまとめる．

- DLM のようなユーザレベルソフトウエアによるメモリシステムにおいて，高速通信媒体に直接実装された様々な MPI ライブラリを用いることが可能になった．これにより様々な通信リンク上で，提供されている MPI のスレッドレベルに合わせたスワッププロトコルを選択することで，高い性能と可搬性を得ることができる．

  複数の Myri-10G リンクを持つ本実験環境で，STREAM による遠隔メモリバンド幅は，Myri-10G/bonding=2 の場合で 493 MB/s，Myri-10G/bonding=4 の場合で 613 MB/s に達している．これは従来の 10GbpsEthernet（Ethernet on Myrinet・Myri-10G 使用）における遠隔メモリバンド幅 380 MB/s の 1.4 倍と 1.6 倍の性能にそれぞれ匹敵し，これまでに報告されている関連研究の性能に比べ高性能な環境を実現している．

  また，可搬性の高い MPI 通信を用いたことにより，Infiniband などのさらに高速な他の通信媒体上での稼動も可能で，通信媒体の高速化とともに，DLM の性能を今後も高めることができる．
- Himeno ベンチマークや NPB の 6 種の応用プログラムについて，様々なローカルメモリ率と，異なる通信性能を持つネットワークにおいての性能と問題点を明らかにした．

  この結果，Himeno ベンチマーク，IS，CG，MG では，ローカルメモリ率が 5%程度であっても，通常実行（ローカルメモリ 100%）に比べ 1.5〜4 倍以内の実行時間であることが分かった．同様に，FT，SP，BT では，ローカルメモリ率の低い場合には，

DLM ページサイズを小さくすることが効果的であることも分かった．

- 実際に非常に大きなデータを扱う応用プログラムを，複数のノードに分散した遠隔メモリを利用して実行可能であることを示した．

  本論文では，241 GB のデータに対する Himeno ベンチマーク処理を 20 GB メモリ/ノードを 12 ノード用いて，稼動できることを示した．これは単に，ローカルメモリサイズの 12 倍のメモリサイズが使えるというだけでなく，241 GB という大きなサイズの物理メモリを持つコンピュータシステムを利用できない環境にいるユーザにとって，本来ならプログラム実行が不可能であったにもかかわらず，DLM によって実行が可能になったという点で，意義が大きい．
- 従来の DLM で提供していた DLM コンパイラに加え，DLM-MPI では，ユーザレベルソフトウエアとして dlm-mpi ライブラリを提供し，dlm_startup()，dlm_shutdown()，dlm_alloc()，dlm_free() などの関数のみを用いて，容易に従来の逐次プログラムを DLM で実行可能な DLM プログラムに変換することができる．これにより，大容量データを扱う逐次処理応用を持つユーザが，並列プログラミングの知識なしに，MPI バッチシステムで運用される多くのオープンクラスタを，メモリ資源として利用することが可能になった．

今後の研究として，(1) 応用におけるアクセス局所性・ワーキングセットサイズと適正な DLM ページサイズの自動選択，(2) ユーザレベル実装方式に適した低コストで効果的なページ置き換えアルゴリズムの構築，評価，(3) 用いる通信プロトコルや媒体を生かす高性能なスワッププロトコルの構築・評価，(4) マルチスレッドユーザプログラムへの対応などを考えている．

謝辞　東京大学計算センター松葉浩哉氏には，T2K オープンスパコンと MPICH-MX の利用に関して多くの技術的助言と支援をいただきましたことを深謝いたします．なお，本研究の一部は，科研費基盤研究（C）（No.21500062）「大規模データ処理のための高速仮想メモリシステムの研究」の助成を受けています．

参 考 文 献

1) 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 CPSY 研究会報告，信学技報，Vol.102, No.398, pp.29–34 (2007).
2) 緑川博子，黒川原佳，姫野龍太郎：遠隔メモリを利用する大容量メモリシステム DLM とコンパイラ，情報処理学会 HPC 研究会資料 HPC115-7，pp.37–42 (2008).
3) Midorikawa, H., Kurokawa, M., Himeno, R. and Sato, M.: DLM: A Distributed Large Memory System Using Remote Memory Swapping over Cluster Nodes, *Proc. IEEE International Conferences on Cluster Computing* (*Cluster2008*), pp.268–273 (2008).
4) 緑川博子，黒川原佳，姫野龍太郎：遠隔メモリを利用する分散大容量メモリシステム DLM の設計と 10GbEthernet における初期性能評価，情報処理学会論文誌　コンピューティングシステム，Vol.1, No.3, pp.136–157 (2008).
5) Midorikawa, H., Saito, K., Sato, M. and Boku, T.: Using a Cluster as a Memory Resource: A Fast and Large Virtual Memory on MPI, *Proc. IEEE International Conferences on Cluster Computing* (*Cluster2009*) (2009). slides: (2009) Cluster2009 [Online]. available from http://www.cluster2009.org/ (accessed 2009-10-19)
6) 斉藤和広，緑川博子，甲斐宗徳：遠隔メモリを用いた大容量仮想メモリ DLM におけるメモリ管理機構の導入，電子情報通信学会 CPSY 研究会 CPSY2008-47，信学技報，Vol.108, No.361, pp.25–30 (2008).
7) Liang, S., Noronha, R. and Panda, D.K.: Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device, *Proc. IEEE International Conferences on Cluster Computing* (*Cluster2005*) (2005).
8) Mache, P.: Linux Network Block Device [Online]. available from http://www.xss.co.at/linux/NBD/, http://nbd.sourceforge.net/ (accessed 2009-10-19)
9) Anemone web site [Online]. available from http://ww2.cs.fsu.edu/mhines/anemone/ (accessed 2009-10-19)
10) Newhall, T., et al.: Nswap: A Network swapping Module for Linux Clusters, *Proc. EuroPar03* (2003).
11) Newhall, T., et al.: Reliable Adaptable Network RAM, *IEEE cluster2008*, pp.2–12 (2008).
12) 後藤正徳，佐藤　充，中島耕太，久門耕一：10GbEthernet 上での RDMA を用いた遠隔スワップメモリの実装，CPSY 研究会報告，信学技報告，Vol.106, No.287 (2006).
13) 北村裕太，松葉浩也，石川　裕：大規模メモリ空間の利用を支援する遠隔スワップメモリシステム，情報処理学会研究報告 2007-HPC-111(21)，pp.121–126 (2007).
14) 山本和典，石川　裕：テラスケールコンピューティングのための遠隔スワップシステム Teramem，情報処理学会論文誌　コンピューティングシステム，Vol.2, No.3, pp.142–152 (2009).
15) Pakin, S. and Johonson, G.: Performance Analysis of a User-level Memory Server, *Proc. IEEE cluster2007*, pp.249–258 (2007).
16) STREAM Benchmark web site [Online]. available from http://www.cs.virginia.edu/stream/ref.html (accessed 2009-10-19)
17) Myri-10G, Myricom web site [Online] (2009). available from

http://www.myri.com/ (accessed 2009-10-19)
18) NPB (NAS Parallel Benchmarks) web site [Online] (2009). available from http://www.nas.nasa.gov/Resources/Software/npb.html (accessed 2009-10-19)
19) NPB2.3-omni-C web size [Online] (2009). available from http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html (accessed 2009-10-19)
20) Himeno Benchmark web site [Online] (2009). available from http://accc.riken.jp/HPC/HimenoBMT/index.html (accessed 2009-10-19)

(平成 21 年 5 月 11 日受付)
(平成 21 年 9 月 7 日採録)

緑川　博子（正会員）

慶應義塾大学工学部電気工学科卒業．筑波大学システム情報工学研究科コンピュータサイエンス専攻後期博士課程修了．博士（工学）．日本電気（株）C&C システム研究所にて，データフロー型プロセッサ，マルチプロセッサの研究開発，パターン認識，並列処理応用研究開発に従事．現在，成蹊大学理工学部情報科学科助手．並列処理，システムソフトウエア，並列アルゴリズム，並列プログラミングモデル等に興味を持つ．IEEE，電子情報通信学会各会員．

斉藤　和広（学生会員）

昭和 60 年生．平成 20 年成蹊大学工学部経営・情報工学科卒業．現在，同大学院工学研究科博士課程前期情報処理専攻に在籍．ハイパフォーマンスコンピューティングの研究に従事．

佐藤　三久（正会員）

昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年同大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年通産省電子技術総合研究所入所．平成 8 年新情報処理開発機構並列分散システムパフォーマンス研究室室長．平成 13 年より筑波大学大学院システム情報工学研究科教授．平成 19 年より同大学計算科学研究センターセンター長．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術，グリッドコンピューティング等の研究に従事．IEEE，日本応用数理学会各会員．

朴　　泰祐（正会員）

昭和 35 年生．昭和 59 年慶應義塾大学工学部電気工学科卒業．平成 2 年同大学院理工学研究科電気工学専攻後期博士課程修了．工学博士．昭和 63 年慶應義塾大学理工学部物理学科助手．平成 4 年筑波大学電子・情報工学系講師，平成 7 年同助教授，平成 16 年同大学院システム情報工学系助教授，平成 17 年同教授，現在に至る．超並列計算機アーキテクチャ，ハイパフォーマンスコンピューティング，クラスタコンピューティングに関する研究に従事．平成 18 年度〜21 年度情報処理学会 HPC 研究会主査．平成 14 年度および平成 15 年度情報処理学会論文賞受賞．日本応用数理学会，IEEE-CS 各会員．