成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.48 No.1 (2011) pp.15-22

大容量データを扱うプログラムのための遠隔メモリ利用を容易にするCコンパイラ

吉村　礎[*1]，緑川　博子[*2]

A C compiler for Large Data Sequential Processing using Remote Memory

Shiyo YOSHIMURA[*1], Hiroko MIDORIKAWA[*2]

ABSTRACT : Prevailing 64bit-OS enables us to use a large memory address space in our programs. When a program requires more memory than physical memory in a computer, a traditional virtual memory system does page swap between a local hard disk. However, recent high-speed networks realize better performance of accessing remote memories in network-connected computers, rather than accessing a local hard disk. In this background, the Distributed Large Memory System, DLM, was designed as a user-level software for high portability. The DLM provides very large virtual memory using remote memories distributed over cluster nodes. This paper proposes a newly designed C compiler for DLM. It provides an easy programming interface to use rich memory of DLM with existing sequential programs, instead of developing parallel programs.

Keywords：compiler, memory, cluster

（Received April 6, 2011）

１．はじめに

近年，64bit OS の普及により，プログラムから大容量のアドレス空間が利用可能になった。通常，実行するプログラムのメモリ量がそのコンピュータに搭載されている物理メモリ量を超えるとローカルハードディスクにスワップされる。しかし，ネットワークの高速化が進んできたため，ローカルハードディスクを利用するよりも，高速なネットワークに接続された遠隔コンピュータのメモリを利用する方がより高速に処理できるようになってきた。

科学数値シミュレーションなどでは，大規模な問題を解く前に，通常，逐次プログラムを作成して問題をモデル化し，小規模の問題で実行を検証する。次に，問題の大規模化を図るが，多くの場合，大容量のメモリが使用できるクラスタなどで，作成した逐次プログラムを MPI などの並列プログラムに変更し，実験を行う。これによって，クラスタ上の多数ノードにあるメモリが利用できるだけでなく複数の CPU により，プログラム実行時間も高速化することができる。しかし，逐次プログラムの並列化，作成した並列プログラムの実行におけるモデルの正当性のチェックやデバックを行うには，非常に多くの人的，時間的なコストがかかってしまう。また，扱うシミュレーションのモデルによっては，プログラム自体が複雑で並列化が困難なことも多い。同様に，他人が作成した既存のプログラムやライブラリ関数を利用したプログラムの場合にも並列化が困難な場合が存在する。すなわち，並列プログラム開発を専門としない多くの科学者にとって，このようなプログラム並列化とそれに伴う開発コストは，煩雑であるばかりか，本来，シミュレーション対象のモデル化や解析に費やすべき時間を大幅に損失することにもなっている。そこで，筆者らは，多くの科学者が問題の大規模化の際に直面する，煩雑かつ高コストであるプログラム並列化を行わずに，クラスタの複数ノードに分散するメモリだけを利用して，逐次プログラムコードのままで，大容量データを扱えるようにした仮想大容量メモリシステムを開発している。実行するプログラムは逐次コードのままなので，クラスタノードの複数 CPU を用いないため，実行時間を高速化することはできないが，煩雑な並列プログラムの開発やデバッ

[*1]：理工学研究科理工学専攻情報科学コース修士学生
[*2]：情報科学科助教(midori@st.seikei.ac.jp)

グに要する時間的コストを削減することができる。これを用いることで，多少実行時間がかかっても，今まで並列化困難であったモデルやプログラムを容易に実行させることが可能となり，多くの研究者に恩恵がある。

DLM(Distributed Large Memory)[2][3] とは，プログラムの使用メモリ量がコンピュータのローカル物理メモリ量を超えた時に，遠隔コンピュータ(クラスタの遠隔ノードなど)のメモリを通信によって利用し，大容量メモリとして逐次処理用に提供するシステムのことである。

本論文では，逐次ユーザプログラムを容易に DLM システム上で実行できるようにするため，DLM 用のコンパイラを開発した[1]。

逐次プログラムを利用する際，遠隔メモリを使用する研究には，カーネルレベル実装[5][6]とユーザレベル実装[2][3][4]がある。一般的にカーネルレベル実装では，OS のスワップシステムを利用して遠隔メモリにアクセスするため，OS の書き換えなどが必要になる。そのため，可搬性・可用性は低いが，ユーザに完全な透過性を与えてくれる。しかし，従来の OS ではハードディスクをスワップデバイスとして用いることを前提にチューニングされているため，スワップデバイスとして遠隔メモリを利用するように変更すると，メモリ枯渇時に遠隔メモリへのネットワーク通信などが発生し，通信動作が不安定になるだけでなく，十分な通信性能をあげることができないことがわかっている[2]。DLM は，ユーザレベル実装で，OS のスワップシステムとは独立に，ユーザレベルのソフトウェアとして， MPI や socket 通信などの汎用プロトコルを用いて遠隔メモリにアクセスするようなライブラリを提供している。OS の変更をしないため，可搬性・可用性は高く，さらに，スワップシステムに組み込むカーネルレベル実装と比べ，性能が高い[2]。しかし，ユーザのプログラムをこれらのライブラリ関数を利用するように変更することが必要となり，ユーザ透過性が低くなる。そのためユーザ透過性を向上させることを目的として，DLM 用コンパイラを開発した。

２．関連研究

大容量データを用いる逐次プログラムのために遠隔メモリを使用する関連研究について述べる。

Teramem[5]は，カーネルレベル実装の 1 つであるが，性能の観点から前節に述べた OS のスワップシステムに組み込まずに，独立した遠隔メモリアクセス用のデバイスドライバを構築し，このドライバを使って遠隔メモリにアクセスするライブラリを提供している。これにより，ユーザがライブラリ関数を使用して遠隔メモリにアクセスができるようになり，従来のカーネルレベル実装の性能上の欠点を克服している。しかし，カーネルレベル実装の利点であったユーザ透過性が失われ，ユーザにプログラム書き換えの負担を増やすことになっている。また，プログラム書き換えを軽減するためのサポートはおこなっていない。

一方，ユーザレベル実装の 1 つの，JumboMem[4]では，ユーザ透過性を実現するため，動的ライブラリのロードパスを変更することにより，OS のメモリ関連関数コール(malloc や free など)を，JumboMem 独自の遠隔メモリに展開するメモリ割り付け関数コールに置き換えてしまう方式をとっている。これにより，ユーザによるプログラムの書き換えは不要になるばかりか，バイナリプログラムも変更なく実行することができる。しかし，この方式には 2 つの問題点がある。1 点目は，動的メモリ確保関数にしか適用されないため，数値計算プログラムなどで多用される静的な大規模配列宣言データには遠隔メモリ利用による大規模化が適用できない点。2 点目は，ローカルメモリに確保されるべき入出力時のバッファ領域なども遠隔メモリに取られてしまう可能性があり，実用上問題がある点である。

DLM システムもユーザレベル実装であるが，DLM ではユーザによるプログラム変更を最小限にするため，動的メモリ確保と静的配列宣言の両方に対応できる API を提供する。本研究では，ユーザ透過性を高め，ユーザへの負担を軽くするため，この API を実現するコンパイラを作成した。

３．DLM システム

DLM システムは，逐次プログラムを実行する際に，クラスタのノードのローカル物理メモリサイズ以上のメモリを利用したいときに，クラスタの他のノードの遠隔メモリを使用できるようにしている，ユーザレベルソフトウェアである。

DLM システムは，図 1 に示すように，計算ホストノード(Cal Host node)でユーザの逐次プログラムを動かす。データの通信は DLM ページサイズ(OS ページサイズの倍数の大きさ)という単位で行っている。遠隔メモリが必要なときにのみ，メモリサーバノード(Mem Server node)のメモリに割り付ける。メモリサーバノード(Mem Server node)に割り付けたデータにアクセスするときは，アクセスしたいデータを持つメモリサーバ(Mem Server node)からそのデータを含む DLM ページを計算ホ

ストノード(Cal Host node)へスワップインし，すぐに利用しないと思われるDLMページをメモリサーバ(Mem Server node)へスワップアウトする。通信は，TCP/IP またはMPI を用いており，それらで動かすことのできる高速通信媒体(10Gbps Ethernet，Infiniband，Myri10G など)で利用可能である。ユーザにとっては逐次プログラムの実行として見えるが，実際には計算プロセス(Cal Process)とメモリサーバプロセス(Mem Server Process)の複数の並列プロセスを実行している。

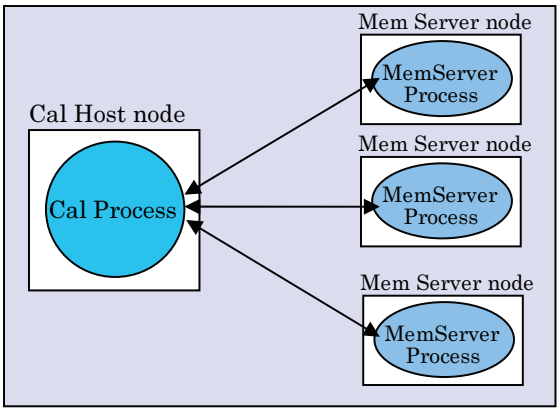

図１　DLM システム

４．DLM システムにおけるプログラムインターフェース

このインターフェースでは，ユーザのプログラム書き換えの負担を極力減らすことを念頭に置いて設計している。具体的には，プログラムを並列化する知識がなくても，DLM を使用できるようにしている。

そして，前述の JumboMem での欠点である，動的メモリ確保関数のみにしか適用できない点については，静的な大規模配列宣言データにも適用できるように設計されている。ローカルメモリに確保されるべきバッファ領域なども遠隔メモリに取られてしまう可能性がある点は，ユーザがローカルメモリ不足時に遠隔メモリへ展開したいデータ変数(DLM データと呼ぶ）と必ずローカルメモリにおくデータ変数を区別して指定できるようにして，解決を図っている。

通常，静的変数(グローバル変数や static 変数)では静的データメモリ領域に，関数内部自動変数(ローカル変数)ではスタックメモリ領域に，動的メモリ割り当て(通常は malloc 関数を利用)はヒープメモリ領域に取られる。しかし，この DLM システムでは，ユーザが DLM データとして指定した変数は全て遠隔メモリも含む動的割当て（ヒープ領域）に展開されるように構築されている。

これにより，ローカルメモリサイズやコンパイラなどの制限を受けずに，データの大規模化を図ることが可能になっている。

DLM プログラムは，通常の C 言語プログラムの文法に「dlm」を記憶指定子(extern，static など)として組み込んだ文法を利用して書かれている。DLM の API の特徴は以下の 3 点である。

・ローカルメモリのみを使用する通常変数とローカルメモリ不足時に遠隔メモリにも展開可能な変数(DLM データ)を，ユーザが「dlm」を付加するか否かで指定可能である。図 2 に例を示す。

```
int a[100][10] ;          →ローカルメモリにのみ展開
dlm int b[1000][1000] ;   →遠隔メモリにも展開
```

図２　DLM データの区別

・静的データ宣言に「dlm」と付加するだけで DLM データとして使用可能になる。図 3 に例を示す。

```
dlm double c[1000][1000] ;
```

図３　DLM データの静的データ宣言

・動的データ割り付けの malloc 関数を dlm_alloc 関数に書き換えるだけで DLM データとして使用可能になる。図 4 に例を示す。

```
d = (int(*))dlm_alloc(1000*1000*sizeof(int));
```

図４　DLM データの動的データ割り付け

図 5 は，DLM データを利用して配列の中央値を求めるプログラム例である。配列 a(①)は，10 個の整数配列（各 10G 個）を用意し，main 関数でランダムに整数を代入し，10 個の数列を作成する。median 関数では 1 つの整数数列分（10G 個）の配列(配列 b②)を用意し，そこに元の整数配列 a のうちの 1 つの配列の値をコピーし，この配列 b をソートすることにより，中央値を求めて返す。そうすることにより，元の配列 a(①)の数列を順序を保持しつつ，10 個のデータ列のそれぞれの中央値が求められるプログラムになっている。

通常のプログラムであれば，①の変数は，グローバル変数なのでローカルメモリの静的データ領域に，②の変数は，ローカル変数なのでローカルメモリのスタック領域に割り当てられるが，今回は dlm 変数（DLM データ）となっているため，すべてローカルメモリのヒープ領域か，ローカルメモリが足りない場合には遠隔メモリのヒープ領域に取られる。

```
#include <stdlib.h>
#include <dlmm.h>
#define NUM 10
#define LENGTH (10*(1L<<30))   //10G
dlm int a[NUM][LENGTH]; // 400GB   ①
int median(long int num) {
      dlm   int b[LENGTH]; // 40GB    ②
      long int j;
      for ( j = 0; j < LENGTH; j++)
         ③   b[j] = a[num][j]; ④
      qsort(b, LENGTH, sizeof(int), compare_int); ⑤
      return b[LENGTH/2]; ⑥
}
int main ( int argc, char *argv[])
{
      long int i, j;
      for ( i = 0; i < NUM; i++)
          for ( j = 0; j < LENGTH; j++)
            ⑦ a[i][j] = rand();
      for ( i = 0; i < NUM; i++){
          printf("median[%d] = %d¥n", i, median(i));
      }
      return 0;
}
```

図５　DLM プログラム例

５．DLM コンパイラの構造

DLM コンパイラは，移植性を高めるため，図 6 に示すように二段構成になっている。前段では独自に作成したトランスレータに通し，後段で汎用の C コンパイラを利用する構成にしている。その二つを合わせて DLM コンパイラとして作成している。DLM コンパイラ(dlmc)では，DLM データが宣言されているプログラム(DLM プログラム)をトランスレータ(dlmpp)に通して，通常の C 言語プログラムに変換する。次に，gcc コンパイラにより，dlm ライブラリをリンクし，実行ファイルを作成する。図 7 は，dlm コンパイルコマンドの例を示している。

前段のトランスレータの中では，最初に C プリプロセッサに通し，その後，3 つの作業を行っている。DLM のライブラリ関数の挿入，DLM データ宣言をポインタ宣言に変換，DLM データにあたる変数をスコープを考慮してリネーミングする。後段では C コンパイラに，dlm のライブラリをリンクさせて，実行ファイルを作成している。

DLM システムは, DLM コンパイラを使用せずとも，ユーザが dlm ライブラリの関数などをプログラムに書き加え，dlm 専用のライブラリ(dlmmpi)をリンクして，通常の gcc コンパイラを用いて実行ファイルを作成することも可能となっている。しかし，DLM コンパイラを使用すると，開発時間を短縮することができる。

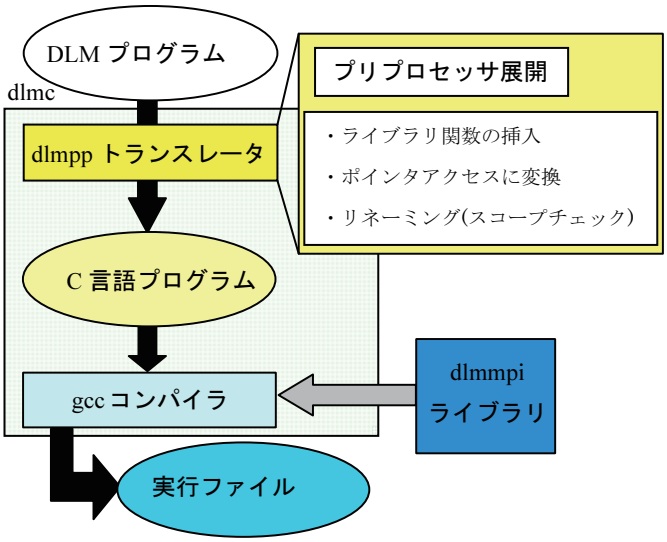

図６　DLM コンパイラの構造

```
dlmc　“dlm プログラム名.c” –o “実行プログラム名” -ldlm
```

図７　DLM プログラムコンパイルコマンド

６．dlmpp トランスレータ

6．1　dlm 関数の挿入・ポインタアクセスへの変換

dlmpp トランスレータにより図 5 のプログラムは，図 8 の C プログラムに変換される。まずは C プリプロセッサによりヘッダファイルなどが展開されるが, 図 8 では，展開された部分を省略し，図の上部に「・・・」と記している。DLM プログラムをこの dlmpp トランスレータに通すと，次のように挿入・変換が行われる。

```
　　・・・・・・・・・・・・・
int (*__dlm_a_0)[(10*(1L<<30))];　①

int median(long int num) {
   int (*__dlm_b_1);　　②
   long int j;
__dlm_b_1 = ( int (*))dlm_alloc((10*(1L<<30))*sizeof( int ));　③
   for ( j = 0; j < (10*(1L << 30)); j++)
          ④__dlm_b_1[j] = __dlm_a_0[num][j];　⑤
   qsort(__dlm_b_1,(10*(1L<<30)),sizeof(int),compare_int);　⑥
   dlm_free(__dlm_b_1);　⑦
   return __dlm_b_1[(10*(1L << 30))/2];　⑧
}

int main ( int argc, char *argv[])
{
      long int i, j;
      dlm_startup(&argc, &argv);　⑨
__dlm_a_0 =　　⑩
(int(*)[(10*(1L<<30))])dlm_alloc(10*(10*(1L<<30))*sizeof( int));
for ( i = 0; i < 10; i++)
      for ( j = 0; j < (10*(1L << 30)); j++)
            ⑪__dlm_a_0[i][j] = rand()
      for ( i = 0; i < 10; i++)
 printf("median[%d] = %d¥n", i, median(i));
            dlm_shutdown();　⑫
            return 0;
            dlm_shutdown();　⑬
}
```

図８　変換後の C 言語プログラム

- main 関数の変数宣言部分の後に dlm_startup 関数を挿入する。この関数は DLM システムの起動関数であり，これにより，あらかじめ設定した複数のメモリサーバノードにメモリサーバプロセスを必要に応じて生成し，計算プロセスとの間に通信を確立する。　(図 8－⑨)
- main 関数や return 文の前に，プログラムを終了する前に dlm_shutdown 関数を挿入する。この関数は DLM システムの終了関数であり，これにより，メモリサーバとの通信を閉じて，メモリサーバプロセスを終了する。(図 8－⑫，⑬)
- DLM データの静的宣言の記述があれば，その変数をポインタ変数に変換し(図 5－①, ②→図 8－①, ②)，dlm_startup 関数の後に，その変数のメモリを確保するため dlm_alloc 関数を挿入し(図 8－③, ⑩)，以降その変数名を「_dlm_変数名_ブロック番号」という変数名にリネーミングする。(図 5－③, ④, ⑤，⑥，⑦→図 8－④，⑤，⑥，⑧，⑪)リネーミングは次節で詳しく説明する。
- 関数やブロック内部の局所変数に dlm が宣言されている場合には，ブロックや関数の最後に dlm_free 関数も挿入する。(図 8－⑦)

6．2　リネーミング

dlmpp トランスレータでは，DLM データにあたる変数をスコープを考慮してリネーミングを行っている。

図 9 は，リネーミングを説明するためのプログラム例である。図 9 では，先頭の関数外に dlm データ a(①)が宣言され，main 関数の中に通常の変数 b(②)と dlm データ c(③)が宣言されている。さらに main 関数の中の if 文の中に，通常の変数 a(⑦)が宣言され，test_print 関数に前述とは別の dlm データ a(⑭)と通常の変数 c(⑮)が宣言されている。このように図 9 では，複数同じ変数名が使用されている。これをトランスレータに通すと DLM データにあたる変数のみを宣言のあるブロック番号を付与して，リネーミングしていく。

④：この変数は DLM データ(①)のため，ブロック番号 0 を付与してリネーミングされる。

⑤：この変数は通常の変数(②)のため，リネーミングされない。

⑥：この変数は DLM データ(③)のため，ブロック番号 1 を付与してリネーミングされる。

⑧：この変数は通常の変数(⑦)のため，リネーミングされない。

⑨：この変数は通常の変数(②)のため，リネーミングされない。

⑩：この変数は DLM データ(③)のため，ブロック番号 1 を付与してリネーミングされる。

⑪：この変数は DLM データ(①)のため，ブロック番号 0 を付与してリネーミングされる。

⑫：この変数は DLM データ(③)のため，ブロック番号 1 を付与してリネーミングされる。

⑬：この変数は通常の変数(②)のため，リネーミングされない。

⑯：この変数は DLM データ(⑭)のため，ブロック番号 1 を付与してリネーミングされる。

⑰：この変数は通常の変数(⑮)のため，リネーミングさ

れない。

図 9 をトランスレータに通した結果が図 10 である。

```
#define M 256
#define N 1024
dlm int a[M][N]   ①

void test_print ();
int main ( int argc, char *argv[]){
      int   b, i;   ②
      dlm double c[M];   ③

      a[0][0] = 0;      ④  //リネームする
      b = 0;            ⑤  //リネームしない
      c[0] = 0;         ⑥  //リネームする

      if( ・・・・・・ ) {
          int a[5][5];   ⑦
          a[0][0] = 5;  ⑧ //リネームしない
          b = 7;         ⑨ //リネームしない
      ⑩  c[0] = a[0][0];   //リネームする
      }
      printf("a[0][0] = %d ¥n", a[0][0]);   ⑪
      printf("c[0] = %d ¥n", c[0]);   ⑫
      test_print(b);   ⑬
      //変数 b 以外の 3 つリネームする
      return 0;
}
void test_print(int c) {
      dlm double a[M][N];   ⑭
      int i, c;   ⑮

      for ( i=0; i<M; i++) {
            a[i][0] = i+1;    ⑯  //リネームする
            c++;   ⑰     //リネームしない
      }
}
```

図 9　リネーミング用 DLM プログラム例

```
   ・・・・・・・・・・・・・・・・・・・・・・・・・
int (*__dlm_a_0)[1024]   ①

void test_print ();
int main ( int argc, char *argv[]){
      int   b, i;   ②
      double (*__dlm_c_1);   ③

      dlm_startup(&argc, &argv);
__dlm_a_0=(int(*)[1024])dlm_alloc(256*1024*sizeof(int));
__dlm_c_1=(double(*))dlm_alloc(256*sizeof(double));

      __dlm_a_0[0][0] = 0;   ④
      b = 0;   ⑤
      __dlm_c_1[0] = 0;   ⑥

      if( ・・・・・・ ) {
          int a[5][5];   ⑦
          a[0][0] = 5;   ⑧
          b = 7;   ⑨
       ⑩__dlm_c_1[0] = a[0][0];
      }
      printf("a[0][0] = %d ¥n",__dlm_a_0[0][0]);   ⑪
      printf("c[0] = %d ¥n", __dlm_c_1[0]);   ⑫
      test_print(b);   ⑬
       dlm_free(__dlm_c_1);
      dlm_shutdown();
      return 0;
      dlm_shutdown();
}
void test_print(int c) {
      double (*__dlm_a_1)[1024];   ⑭
      int i, c;   ⑮

 __dlm_a_1=
   (double(*)[1024])dlm_alloc(256*1024*sizeof(double));

      for ( i=0; i<256; i++) {
            __dlm_a_1[i][0] = i+1;   ⑯
            c++;   ⑰
      }
      dlm_free(__dlm_a_1);
}
```

図 10　リネーミング用 DLM プログラム例の変換後

７．DLM プログラムの実例

科学数値計算のプログラムでは，静的な配列が使用されていることが多い。ここでは，姫野ベンチマークのソースプログラム[7]を例にあげる。

静的領域の限度数を越える領域を宣言するときは，動的に malloc 関数でメモリを確保することもできるが，多次元配列で記述されているプログラムをすべてポインタベースアクセスに変更するのは煩雑な作業を必要とする。しかしこの DLM コンパイラを使用すれば，「dlm.h」のヘッダを挿入し，図 11 の姫野ベンチマークプログラムのように，配列変数宣言の前に「dlm」と付加するだけで，それ以外の部分を変更せずに，既存プログラムが扱う問題の大規模化を図れる。

```
・・・・・・・・・・
dlm float p[MIMAX][MJMAX][MKMAX];
dlm float a[MIMAX][MJMAX][MKMAX],
          b[MIMAX][MJMAX][MKMAX],
          c[MIMAX][MJMAX][MKMAX];
dlm float bnd[MIMAX][MJMAX][MKMAX];
dlm float wrk1[MIMAX][MJMAX][MKMAX],
          wrk2[MIMAX][MJMAX][MKMAX];
・・・・・・・・・・
```

図 11　姫野 DLM プログラム変更箇所

実験は以下の表 1 で示す東京大学情報基盤センターオープンクラスタ T2K[8]で行った。

表 1　T2K オープンクラスタの性能

	T2K Open Supercomputer, HA8000
Machine	HITACHI HA8000-tc/RS425
CPU	AMD QuadCore Opteron 8356(2.3GHz) 4CPU/ node
Memory	32GB/node (936 nodes)，128GB/node (16nodes)
Cache	L2 : 2MB/CPU (512KB/Core)，L3 : 2MB/CPU
Network	Myrinet-10G x 4, (40Gbps) bonding4 Myrinet-10G x 2, (20Gbps) bonding2
OS	Linux kernel 2.6.18-53.1.19.el5 x86_64
Compiler	gcc version 4.1.2 20070626，Hitachi Optimizing C mpicc for 1.2.7
MPI Lib	MPICH-MX (MPI 1.2)

姫野ベンチマークの ELARGE(float 型の 513x513x1025 で計 14GB のメモリサイズ)を 1 ノード最大 20GB ずつ利用可能な 2 つのノードで実験を行った。図 12 は，横軸がローカルメモリ率，縦軸が津城実行(ローカルメモリ率 100%)に対する相対実行時間を示している。この結果，ローカルメモリ率が 6.9%(遠隔メモリに 93.1%展開している)の状態でも，実行時間はノード間通信が 40Gbps で通常の 1.78 倍, 20Gbps で通常の 1.98 倍で実行できることを確認した[3]。

さらに，既存の姫野ベンチマークのプログラムサイズを変更し，XLARGE(float 型の 1025x1025x2049 で計 112GB のメモリサイズ)を 1 ノード 20GB ずつ使用し，合計 6 ノードで実験を行った。この結果，ノード間通信が 40Gbps，ローカルメモリ率 17.4%(遠隔メモリに 82.6%展開)で 179.34MFLOPS の性能であった。さらに XLARGE-d(double 型の 1025x1025x2049 で計 225GB のメモリサイズ)を 1 ノード 20GB ずつ使用し，合計 12 ノードで実験を行った。結果，ノード間通信が 40Gbps でローカルメモリ率 8.1%(遠隔メモリに 91.9%展開)で 88.8MFLOPS の性能であった。

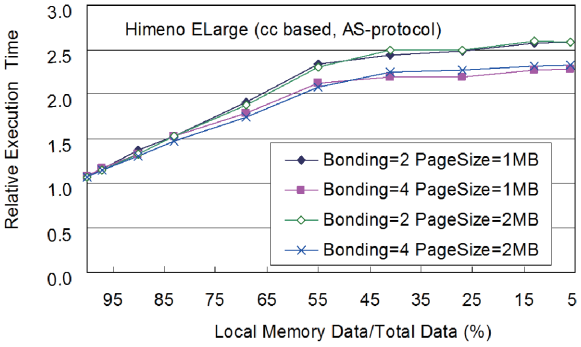

図 12　姫野ベンチマークにおける DLM の性能

８．おわりに

DLM システムを用いて大規模データを使用する場合，従来のプログラムでは，dlm_alloc 関数により動的にメモリを確保する必要があった。しかし，科学数値計算などで大規模データ利用の際に多用される静的な配列宣言は，dlm_alloc 関数とポインタアクセスに書き直すなどプログラムを変更しなければならなかった。しかし，今回の DLM コンパイラを使用することにより，静的な配列宣言であっても，dlm を付加するだけでユーザの変更をほとんど必要とせずに，遠隔メモリに展開する大容量データを利用することが可能となった。

通常のプログラムでは，関数内部変数はスタック領域に確保されるが，たとえローカルメモリサイズに収まる範囲であっても，スタック領域サイズの制限などを受けて，大規模な配列データを宣言することはできなかった。しかし，DLM データ宣言を利用することで，関数内部で宣言された配列データも，大域変数(関数外部)と同様に，ローカルメモリサイズに制限されず(ローカルメモリ

が不足するときは遠隔メモリに展開し), 大規模データを扱うことが可能になった。

今回の DLM コンパイラにより，パソコンなどで小さいサイズでモデルやアルゴリズムを設計し，小さな問題に適用してモデルの正当性などを検証した後，ほとんど変更を加えずに，逐次プログラムコードで並列プログラム化することなく，クラスタを利用して，問題の大規模化が可能になった。

参考文献

1）吉村，緑川，甲斐,「ローカルメモリを越える大容量データを扱う逐次処理のための C コンパイラ」，FIT 論文集 B-026 pp.335-336，Nov.2010

2）緑川，黒川，姫野,「遠隔メモリを利用する分散型大容量メモリシステム DLM の設計と 10Gb Ethernet 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, Dec. 2008

3）緑川，斉藤，佐藤，朴,「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」，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, pp.15-36，Dec.2009

4）Scott Pakin, Greg Johnson,「Performance Analysis of a User-level Memory Server」， Proc. of IEEE International Conference on Cluster Computing, pp.249-258, 2007

5）山本，石川,「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，Nov.2009

6）Liang， S.， Noronha， R.， and Panda， D.K.，「Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Decice」, IEEE Cluster Computing, Sep. 2005

7）Himeno Benchmark website [Online], http://accc.riken.jp/HPC/HimenoBMT/downloadtop.html 2011 年 3 月

8） 東京大学情報基盤センタースーパーコンピューティング T2K-TOKYO, http://www.cc.u-tokyo.ac.jp/service/ha8000/ 2011 年 3 月