成蹊大学理工学研究報告
J. Fac. Sci. Tech., Seikei Univ.
Vol.45 No. 1 (2008) pp.13-19

マルチコアクラスタ向け並列言語 －スレッド/プロセス並列機構の実装－

小山　浩生*[1]，緑川　博子*[2]，甲斐　宗徳*[3]

The Parallel Processing Language for Multicore Clusters - The Implementation of a Thread/Process-Parallel Mechanism -

Hiroki KOYAMA*[1], Hiroko MIDORIKAWA*[2], Munenori KAI*[3],

ABSTRACT： We already proposed a new parallel programming model called “meta process model” and designed a new language MpC and its compiler to realize the model. The MpC compiler converts a MpC program to one of the two different programs according to a target machine’s memory architecture. For distributed memory machines such as clusters, the compiler converts a MpC program to a software DSM program which employs process-level parallelism. For shared memory machines such as SMP, the compiler converts a MpC program into a pthread program which employs thread-level parallelism. This paper presents three topics, 1) Optimization of code generation in the MpC compiler for SMP, 2) Incorporation of a thread parallel mechanism into the in the MpC compiler for clusters, and 3) Implementation of DLM Compiler, new compiler for sequential programs using large data, which is based on the MpC compiler.

Keywords：parallel language, compiler, cluster computing, software DSM, pthread

（Received March 21, 2008）

１．はじめに

現在，分散メモリ並列システムにおける並列プログラミングで，最も広く用いられている移植性のあるソフトウエアインタフェースといえば，MPI(Message Passing Interface)であろう。高性能並列コンピュータシステムのほとんどにおいて，各ベンダーにより高性能な MPI が実装され，並列プログラミングの事実上の標準 API(Application Programming Interface)となっている状況において，最近，MPI 以外のプログラミングモデルや言語にも少し目が向けられるようになってきた。その一つが, GAS 言語とよばれるもので, UPC(Unified Parallel C)[1]などがある。これらは，従来のデータをフラットに扱う純粋な共有メモリプログラミングモデルとは異なり，区分化大域アドレス空間モデル(PGAS：Partitioned Global Address Space)に基づいている。

筆者らは，ソフトウェア DSM(Distributed Shared Memory)の研究開発[2]をもとに，ローカリティに着目したメタプロセスモデル[3]という階層型共有メモリプログラミングモデルを提案し，これに基づくポータブルな API として MpC 言語を開発してきた。すでに，MpC がコモディティクラスタや共有メモリ型 SMP マシンにおいて，UPC や OpenMP と比較し，性能上の優位性があることを示した。さらに，NPB(NAS Parallel Benchmark)ベンチマークセットを MpC と他の言語で記述したプログラムの有効行数の比較では，逐次コードに対し，MPI が平均 1.74 倍に増大するのに比べ，MpC は平均 1.07 倍にとどまっており，プログラムの可読性，生産性の上で優れていることを明らかにした[4][5]。

本報告ではこの MpC 言語への機能の追加, 性能向上として『共有メモリマシン向けコンパイラの生成コードの最適化』,『クラスタ向けコンパイラへのスレッド機構の導入』の 2 項目，また，MpC コンパイラの応用として『逐次大容量データ処理向け DLM コンパイラの構築』の 1 項目，以上の 3 項目についてそれぞれ 3，4，5 で述べる。

*1：工学研究科情報処理専攻修士学生
*2：情報処理専攻助手（midori@is.seikei.ac.jp）
*3：情報処理専攻教授（kai@st.seikei.ac.jp）

２．メタプロセスモデルと MpC 言語

2．1　メタプロセスモデル

メタプロセスモデル(図 2-1)とは，従来の共有メモリモデルに，NUMA マシンなどにも対応できるような各プロセスへの共有分散データの階層構造（スコープ）を取り入れたプログラミングモデルである。メタプロセスとは，1 つのアプリケーションを協力して並列処理する複数のプロセス全体を指し，ユーザにとっての実行単位である。

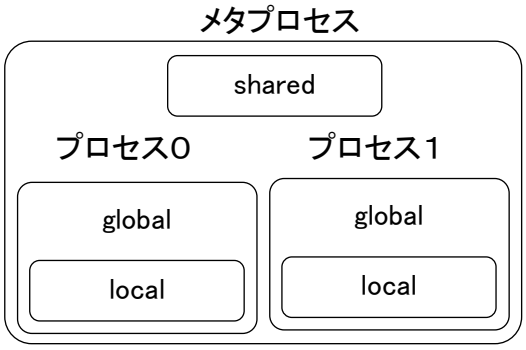

図 2-1 メタプロセスモデル

2．2　MpC 言語

MpC 言語は，メタプロセスモデルをクラスタや共有メモリマシン上で実現するために共有データ型修飾子 shared と共有データ分散マッピング指定子を新しく導入して，C 言語を拡張したものである。UPC も MpC と同様に共有変数に shared(型修飾子)を使用しており，C 言語の拡張であるが，共有データの分散マッピング方式は全スレッドへの均一サイズのマッピングに固定されており，これを前提に CPU アーキテクチャ毎のコンパイラによる最適化を行うことを指向した言語である。これに対し，MpC 言語の共有データの分散マッピング宣言は，スカラ，ベクタを問わず任意のプロセスへ割り付けることが出来る[5]。特に配列データに関しては柔軟性のある割り付けが可能である。MpC 言語ではコンパイル時よりも実行時に割り付け処理を行うことで柔軟性を高めている。また，特定のコンパイラや実装系を前提とせず，アーキテクチャによらず移植性が高いことが特徴である。

現在，MpC コンパイラは 2 通りあり，図 2-2 のようなコモディティクラスタなどの分散メモリマシンではソフトウェア DSM プログラムへ変換しプロセスレベルで並列化し，図 2-3 のような共有メモリマシンでは pthread プログラムへ変換しスレッドで並列実行できるようにして対応している。クラスタ向け MpC コンパイラの例を図 2-4 に示す。MpC コンパイラ内のトランスレータでは C プログラムに変換をしている。

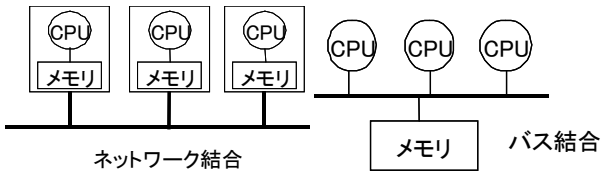

図 2-2 クラスタ

図 2-3 共有メモリマシ

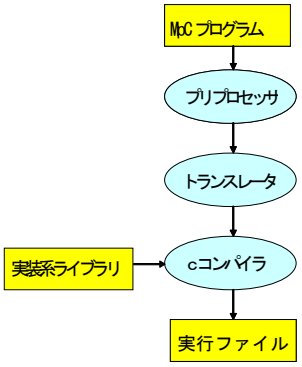

図 2-4 MpC コンパイラ

３．　共有メモリマシン向けコンパイラの生成コードの最適化

MpC コンパイラは共有メモリマシンでは pthread プログラムに変換をしている。その際に，グローバル変数の数，ユーザ定義関数の数に比例してオーバーヘッドが大きくなってしまう変換をしていたので生成コードの効率化を図った。

3．1　共有メモリマシンへの対応

図 2-1 に示したメタプロセスモデルを pthread プログラムに対応させるために，図 3-1 のように shared はグローバル変数へ，local はローカル変数に対応させている。global についてはスコープがスレッド局所になるように，global に対応する変数をユーザプログラムから抽出し，スレッド局所変数(thread specific data)に置き換える。

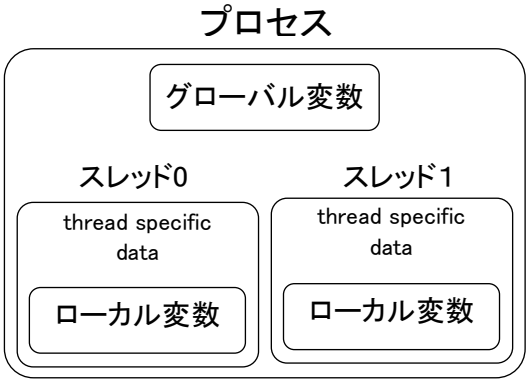

図 3-1 メタプロセスモデルの pthread 向け実装

```
int a, b;
void funcA(){
        a++;
}
void funcB(){
        b++;
}
int main(int argc, char *argv[])
{
        mpc_init(argc, argv);
        a = 0;      b = 0;
        funcA();    funcB();
        printf("%d %d¥n", a, b);
        mpc_exit();
}
```

図 3-2 ソースプログラム

そのために pthread_create された関数で pthread_setspecific 関数を使い，スレッド固有にする変数を各 key に関連付けをしておき，main 以外のユーザ定義関数の先頭において pthread_getspecific 関数により key からスレッド固有データを代入して使用する。MpC コンパイラでは，図 3-2 のソースプログラムを図 3-3 のコードに変換をしている。

```
pthread_key_t key[512];
void funcA(){
        int *a, *b;
        a = pthread_getspecific(key[0]);
        b = pthread_getspecific(key[1]);
        (*a)++;
}
void funcB(){
        int *a, *b;
        a = pthread_getspecific(key[0]);
        b = pthread_getspecific(key[1]);
        (*b)++;
}
void sub_main(){
        int *a, *b;
        a = (int *)malloc(…);
        b = (int *)malloc(…);
        pthread_setspecific(key[0], a);
        pthread_setspecific(key[1], b);
        a = pthread_getspecific(key[0]);
        b = pthread_getspecific(key[1]);
        (*a) = 0; (*b) = 0;
        funcA();    funcB();
        printf("%d %d¥n", *a, *b);
}
int main(int argc, char *argv[]){
…
        pthread_create(…,sub_main, …);
…
}
```

図 3-3 スレッド変換での問題点

例として示した図 3-3 のプログラムにあるように，sub_main 関数内で a に key[0]を，b に key[1]を関連付けしている。そして，main 関数を除くすべての関数の先頭で a には key[0]から，b には key[1]から関連付けされたスレッド固有のデータを返すことでスレッド局所の変数にしている。

3．2　生成コードの改善点

図 3-2 の funcA 関数では a のみ，funcB 関数では b のみを使用しているので，変換後 funcA では b，funcB では a を pthread_getspecific する必要はないが，従来の変換ではすべての変数について pthread_getspecific していたためオーバーヘッドになっていた。関数のコール数が増えると pthread_getspecific のためのオーバーヘッドも増えるのでその関数で使用している global 変数についてのみ pthread_getspecific するように改良した。

3．3　改善前後での性能比較

評価に使用した 2 種類の TSP のプログラムは繰り返し回数の多いヒューリスティックを用いたアルゴリズムである。改善法を用いて，初期値を変えて複数回実行している。for 文を並列処理する Parfor 型と複数回行う実行を並列処理する Trial 型[6][7]を用いてスレッド数を増加させたときの実行時間について改善前後を比べた。実行環境としては，表 4-1 に示す Intel SMP と SunFire の 2 つのマシンで評価を行った。TSP の評価データには TSPLIB[11]の pr1002 を用いた。図 3-4，図 3-5 は各マシンでの実行時間を示したものである。また，図 3-4，図 3-5 の実線は改善前，点線は改善後を示している。

両マシンで改善前に 1 スレッドで 2 つのマシンともに Parfor 型，Trial 型の間に大きな差がある。Trial 型，Parfor 型ともに関数コールの総数は約 112 億回とほぼ同じだが，グローバル変数を Trial 型が 13 変数使用しているのに対して、Parfor 型が 8 変数使用と 5 つ差がある。 1 回 pthread_getspecific を実行すると Intel SMP では約 0.008μsec、SunFire では約 0.0015μsec かかる。このため，改善前は関数の先頭で毎回 pthread_getspecific を実行していたので Intel SMP では 112 億×5×0.008μsec≒450sec の差となった。不要な pthread_getspecific のコールを削減したので改善後は改善前と比べて両方式の差が減り，どちらの方式でも処理時間が 40%～70%削減される成果が得られた。

表 3-1　実行環境

	Intel SMP	SunFireV40Z SMP
CPU	Intel Xeon E5310 1.6Ghz	AMD デュアルコア
CPU 数	Quad コア×2 チップ	Dual コア×2 チップ
OS	Linux FC5 kernel 2.6.19	SunOS 10
メモリ	8GB	16GB

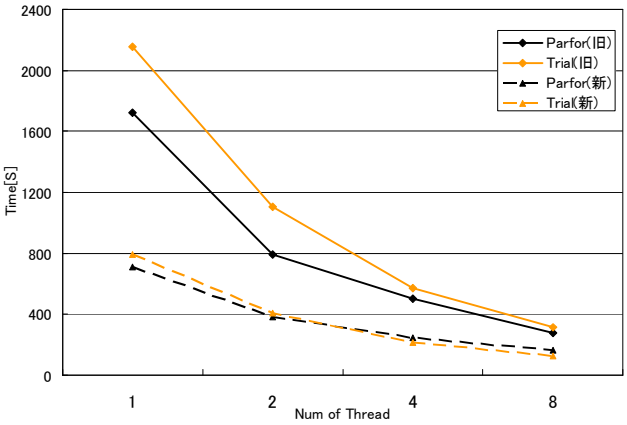

図 3-4 改善前後の実行時間（Intel SMP）

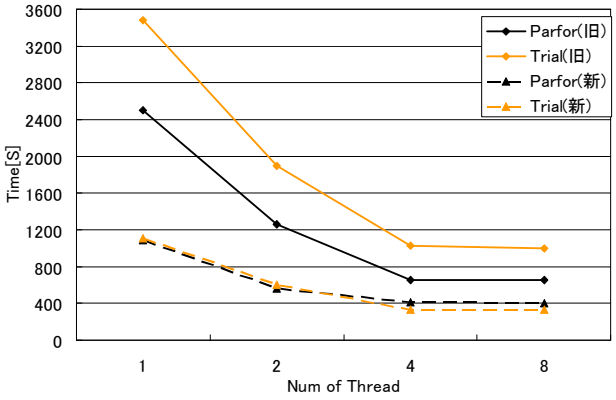

図 3-5 改善前後の実行時間（Sun Fire SMP）

4．クラスタ向けコンパイラへのスレッド機構の導入

最近ではコア数の多いマルチコアクラスタ（図 4-1)が広く普及し始めている。そこで，マルチコアの特性を生かし MpC プログラムの性能をさらに引き出すために，図 4-2 のように，クラスタノードにまたがるプロセスとノード内にあるスレッドを組み合わせ，２レベルの並列処理機構を取り入れた共有メモリ型並列プログラミングモデルを新たに提案する．[8]

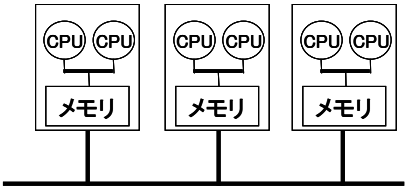

図 4-1 マルチコアクラスタ

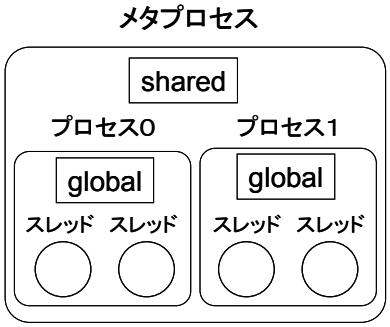

図 4-2 プログラミングモデル

4．1　実　装

今回の実装では，プロセスローカルデータにアクセスする for 文を対象にしたスレッド並列構文 thread_for 文を MpC 文法に追加し，この区間に限ってスレッド並列を行う。あらかじめ MpC コンパイラにより，fork-join 方式でスレッド生成し，for 文におけるイテレーション区間を実行時に指定したスレッド数で等分して並列処理するコードに書き換えておく。

4．2　thread_for 文

現在のところ，thread_for の指定はユーザの責任で行い，for 文内のデータ，フローの依存性はコンパイラは関知しない。MpC プログラムでは thread_for 文以外のスコープで従来通りプロセス共有データ（shared）にアクセスすることが可能であるが，現段階では SDSM(SMS)がスレッド並列に対応していないので thread_for スコープで個々のスレッドが直接 shared データにアクセスすることは禁止している。shared データを global の変数に代入しておき使用することは可能である。また，thread_for スコープでの typedef，enum 変数の使用も禁止している。

4．3　性能評価

実行環境を表 4-1 に示す。性能評価として mandelbrot

を使用し，プロセス，スレッド数を増やして実行したときについての実行時間を比べた。mandelbrot はメモリアクセスに比べ計算量が多く，ノード数，プロセス数，スレッド数に階層的にそれぞれ自由に分け易いアプリケーションになっている。

表 4-1 実行環境

	クラスタ 1	クラスタ 2	
ノード数	4	1	1
CPU type	Dual CPU	Quad core	Dual core
CPU Spec	Intel Xeon 2.8GHz	Intel Xeon E5310 1.6GHz	Intel Xeon E5110 1.6GHz
CPU 数/ ノード数	Single コア ×2 チップ	Quad コア ×2 チップ	Dual コア ×2 チップ
OS	Linux FC5 kernel 2.6.20	Linux FC5 kernel 2.6.20	Linux FC7 kernel 2.6.23
gcc	4.1.1	4.1.1	4.1.2
メモリ	1 GB	8GB	4GB

表 4-2(a)はクラスタ 1 で従来のプロセス並列のみ MpC，表 4-2(b)はクラスタ 1 でスレッド機構を組み込んだ MpC，表 4-3(a)はクラスタ 2 で従来のプロセス並列のみ MpC，表 4-3(b)はクラスタ 2 でスレッド機構を組み込んだ MpC を使用した mandelbrot プログラムの実行時間である。各(a)はノード内をプロセスで並列に，各(b)はノード内をスレッドで並列に処理している。

表 4-2, 表 4-3 は上の欄から Init はプロセスの fork や通信経路の確立などの時間，Calc は計算時間，Mcpy は thread_for スコープでは shared データにアクセスできないのでローカルに必要なデータをコピーしたり，計算した結果を shared データに書き戻したりする時間，Total はプログラムの総実行時間を示す。また，各表の網掛けした値はクラスタ 1，2 においての Total の最小値を示している。なお，時間の単位は全て〔sec〕である。

図 4-3，図 4-4 はプロセス並列の 1 プロセスでの Total を 1 としたときの各部割合を示したものである。図 4-3 はクラスタ 1，図 4-4 はクラスタ 2 での結果である。横軸は，ノード内のスレッドとプロセスの数を示している。

表 4-2 の(a)従来のプロセス並列のみの MpC 内の 4x16 の欄が空いているのは，今回使用している SDSM のバージョン(sms0.0.5)の最大指定プロセス数が 32 になっていることにより実行結果が取れなかったためである。

表 4-2 実行時間比較［クラスタ 1］

(a)従来のプロセス並列のみの MpC〔sec〕

ノード数 x プロセス数

	4x1	4x2	4x4	4x8	4x16
Init	0.376	0.549	1.077	23.355	
Calc	4.219	2.199	1.488	1.420	
Total	4.595	2.748	2.565	24.774	SMS 対応無

(b)スレッド＋プロセス並列の MpC〔sec〕

ノード数 x プロセス数 x スレッド数

	4x1x1	4x1x2	4x1x4	4x1x8	4x1x16
Init	0.368	0.365	0.363	0.359	0.355
Calc	3.642	1.996	1.394	1.277	1.256
Mcpy	0.573	0.183	0.288	0.196	0.181
Total	4.582	2.544	2.045	1.832	1.792

表 4-3 実行時間比較［クラスタ 2］

(a)従来のプロセス並列のみの MpC〔sec〕

ノード数 x プロセス数

	2x1	2x2	2x4	2x8	2x16
Init	0.409	0.431	0.816	2.232	
Calc	6.315	3.679	1.895	1.910	
Total	6.724	4.111	2.711	4.142	エラー

(b)スレッド＋プロセス並列の MpC〔sec〕

ノード数 x プロセス数 x スレッド数

	2x1x1	2x1x2	2x1x4	2x1x8	2x1x16
Init	0.398	0.428	0.390	0.390	0.389
Calc	6.306	3.194	1.746	0.928	0.900
Mcpy	0.015	0.501	0.174	0.468	0.334
Total	6.719	4.124	2.310	1.786	1.624

表 4-3 の(a)従来のプロセス並列のみの MpC 内の 2x16 は原因の特定はできていないがエラーが発生し，実行結果が取れなかった。

図 4-3，図 4-4 にあるように，横軸各左欄の(a)従来のプロセス並列のみの MpC では，プロセス数を増やすと Calc は減少しているが，Init に時間がかかっている。ノード内の実 CPU 数をこえるプロセス数(クラスタ 1 では 4，クラスタ 2 では 8)を指定するとさらに多くの時間がかかる。そのため，プロセス並列の MpC の実行は計算時間を減らすために実 CPU 数をこえるプロセス数での実行は適していない。

横軸各右欄の(b)スレッド機構を組み込んだMpCでは，1 ノードに 1 プロセスずつしか fork していないので，そのプロセス fork や，プロセス同士のコネクションを行っている Init は最小限の時間しかかかっていない。thread_for 文の最後で thread 並列の計算結果を shared データに戻す際に多少の時間はかかっているが，実 CPU 数をこえてもスレッドの起動にはプロセスほどのオーバーヘッドはなく，Calc も減少していっている。そのため，各ノードに 1 プロセスずつ fork しその中でスレッド展開をしたほうが Total で見るといい結果が得られた。

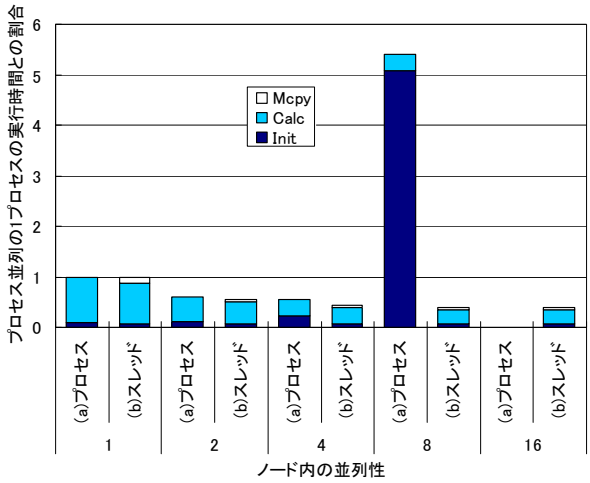

図 4-3 プロセス並列での 1 プロセス使用に対する割合[クラスタ 1]

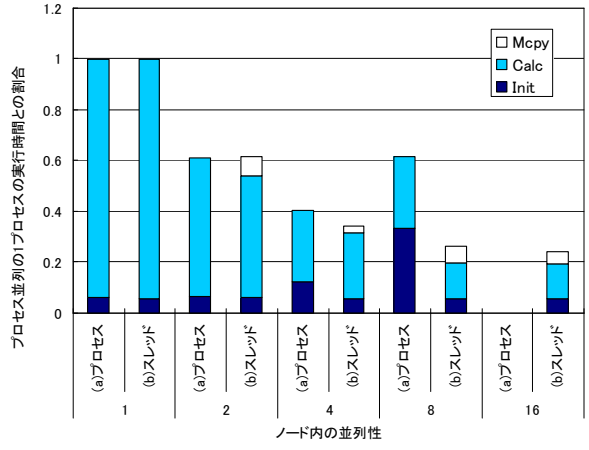

図 4-4 プロセス並列での 1 プロセス使用に対する割合[クラスタ 2]

５． 逐次大容量データ処理向け DLM コンパイラ

大容量のメモリを必要とする逐次プログラムのために，クラスタの各ノードの遠隔メモリを集めて仮想的に大容量メモリとする分散大容量メモリシステム DLM(図 5-1)[9][10]を構築した。DLM システムは一般ユーザが手軽に使えるように，swap システムなどを変更することを必要としないユーザーレベルのソフトウェアになっている。その DLM システムを用いる際に，ユーザの C プログラムに dlm 宣言を加えるだけで，容易にクラスタに分散したメモリを大容量メモリとして用いられるように DLM コンパイラを構築した。

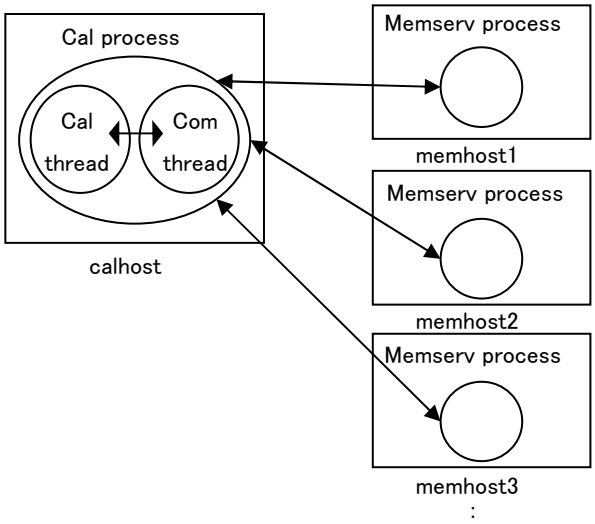

図 5-1 DLM システム構成図

```
calhost     32768   //32GB
memhost1 32768   //32GB
memhost2 65536   //64GB
memhost3 16384   //16GB
          :
```

図 5-2 DLM 設定ファイル例

DLM システムは起動時に図 5-2 に示す DLM 設定ファイルを読み込む。設定ファイルの先頭行は計算ノードホストと，計算ホストでローカルメモリとしてどの程度のメモリを DLM システムに使用させるかということを示す。２行目以降は，メモリサーバとして提供できるホスト名と提供メモリ容量を記述する（メモリ容量は現在，MB 単位で記述)。そして，実行時にメモリ動的割り当てやメモリサーバへの swap 要求を処理する。このため，

DLM コンパイラは, まずメモリサーバプロセスの遠隔起動や計算プロセスの初期設定などを行う初期化関数 dlm_init()をプログラムの開始部分に挿入する。さらに, ユーザにより dlm 指定された静的データ宣言を，動的割り当てとポインタ表現に置き換える処理を行う。すなわち，プログラムスコープを考慮した変数リネーミングを行う。図 5-3 にコンパイル実行例と変換例を示す。

この逐次大容量データ処理向けDLMコンパイラにより，図5-3にあるsample.cのように，逐次CコードにDLM記述を付加するだけでDLMシステムが利用可能になった。

```
//   sample.c
#define MAX 1000

dlm int a[MAX];

int main(int argc, char *argv[])
{
   int i;

   for(i = 0; i < MAX; i++)    a[i] = i;
   for(i = 0; i < MAX; i++)    printf("%d ", a[i]);
   return 0;
}
```

コンパイル　dlmc sample.c -ldlm

```
int (*__dlm_sh_a);

int main(int argc, char *argv[])
{
   int i;

   dlm_init(argc, argv);

   if(MYPID == 0){
      __dlm_dim[0] = 1000;
      __dlm_dim[1] = -1;
      __dlm_div[0] = -1;
      __dlm_sh_a = ( int (*))dlm_mapalloc(__dlm_dim,
__dlm_div, sizeof( int ),0, dlm_nproc);

      for(i = 0; i < 1000; i++)    __dlm_sh_a[i] = i;
      for(i = 0; i < 1000; i++)    printf("%d ", __dlm_sh_a[i]);
   { dlm_exit(); return 0; }

   }
   dlm_exit();
}
```

図 5-3 DLM コンパイラによるプログラムの変換例

６．終わりに

本研究では,『共有メモリマシン向けコンパイラの生成コードの最適化』,『クラスタ向けコンパイラへのスレッド機構の導入』,『逐次大容量データ処理向け DLM コンパイラの構築』の 3 項目について研究を行い以下の成果を得た。共有メモリマシン向けコンパイラの生成コードの最適化により，今回使用した評価プログラムでは，改善後は改善前と比べておよそ 1.5～3 倍の性能向上が得られた。また，クラスタ向けコンパイラへのスレッド機構の導入により従来の MpC 言語に比べ，マルチコアの性能を引き出すことができた。さらに，逐次大容量データ処理向け DLM コンパイラの構築により，dlm 記述のみで DLM システムが利用可能になった。

参考文献

[1] William Carlson, Thomas Sterling, Katherine Yelick, Tarek El-Ghazawi UPC: Distributed Shared Memory Programming, Wiley Series on Parallel and Distributed Computing, 2005

[2] 緑川，飯塚：“ユーザーレベル・ソフトウエア分散共有メモリ SMS の設計と実装”， 情処論文誌 HPC, Vol. 42, No.SIG9(HPS 3), pp.170-190 (2001)

[3] 緑川，片野，飯塚：“メタプロセスモデル －明示的並列処理記述のための分散共有メモリプログラミングモデルー”，情報科学技術フォーラム FIT2003, 情報科学技術レターズ,LB_002, pp. 33-35,（2003, 9）

[4] 小山, 緑川, 甲斐：“並列言語 MpC の高機能化”, 情報化学技術フォーラム FIT2006, FIT 論文集, B－016, pp.101-103,（2006, 9）

[5] 緑川，飯塚：“メタプロセスモデルに基づくポータブルな並列プログラミングインターフェース MpC”, 情処論文誌：ACS, Vol.46 No.SIG4(ACS9), pp.69-85（2005）

[6] 小山, 鈴木, 緑川 : “階層型共有メモリプログラミング言語 MpC を用いた TSP の並列処理”, 情報処理学会第 69 回全国大会論文集,1A-1, pp.1-2,（2007, 3）

[7] 小山, 緑川：“階層型共有メモリプログラミング言語 MpC による TSP 並列処理”, 先進的計算基盤システムシンポジウム 論文集 SACSIS2007 pp.198-199（2007, 5）

[8] 小山, 緑川：“マルチコアクラスタ向け並列言語の提案”, HPCS2008（2008, 1）

[9] 緑川, 小山, 黒川, 姫野：“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 研究会資料, 信学技報 Vol.102, No.398, pp.29-34（2007,12）

[10] 緑川, 小山, 黒川, 姫野：“分散大容量メモリシステム DLM の初期性能評価”, HPCS2008（2008, 1）

[11] TSPLIB：http://elib.zib.de/pub/Packages/mp-testdata/tsp/tsplib/tsplib.html