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

2004－ARC－157 (12)
2004－HPC－ 97 (12)
2004／3／2

常時転送型分散共有メモリを用いたギガリングクラスタの実現

刑 部 努[†] 松 尾 啓 志[†]

近年, ネットワーク速度の高速化が進み, 1Gbit/s を越えるネットワークインタフェースを容易に入手可能となった. それに伴い PC クラスタ間でのネットワークを用いた通信に要する時間も短縮されてきている. しかし, 従来の分散共有メモリシステムのようにオンデマンドに他計算機のメモリアクセスを行う場合は, 通信の初期レイテンシが依然として大きな問題として残る.

そこで本論文では高速なネットワークを利用し, 共有メモリ領域を PC クラスタ間で常時転送状態とし, 各計算機はその時刻に保持しているメモリ領域を用いた処理を行うという計算モデルに基づく新しい概念の分散共有メモリモデルを提案し, ギガリングクラスタを用いて実装評価を行う.

Giga-ring cluster with continuously transmitted distributed shared memory system

Tsutomu OSAKABE[†] and Hiroshi MATSUO[†]

Recently, transmission speed between computers becomes more and more fast and the network interface which have over 1Gbps speed has come to be used in general. Therefore, interconnected communication overhead in PC cluster environment consisting of computers connected by a interconnected communication network has been reduced. But communication cost with initial connection stage is still high when accessing the memory of other computers like a conventional distributed shared memory on demand.

This paper proposes the new distributed shared memory model that all shared memory contents are always transferred in a cluster system with high transmission speed and no initial communication delay.

1. まえがき

近年, 計算機の高速化に伴い, 従来では困難であった様々な問題を 1 台の計算機でも解くことが可能となりつつある. しかし, 構造解析, 電磁界シミュレーションなどの大規模データ処理を行う問題は依然として必要とする計算量, メモリ量が大きく, 並列処理での計算が有効な場合が多い. また, 計算機の低価格化が進んだことから, 多数の計算機を集めて並列処理を行うクラスタ型並列計算機がコストパフォーマンスに優れた並列処理環境として普及しはじめており, 構造解析などの大規模問題をクラスタを用いて並列処理することが期待されている. しかし, クラスタ環境に代表される分散メモリ環境で並列処理を記述する場合は, 分散されたメモリを意識しなければならない. 従って, データ配置や同期, 計算機間での必要なデータ通信など, プログラマがさまざまな問題を考慮する必要がある. そのため, クラスタ環境での並列処理を対象とするさまざまな手法が, 従来より提案されている.

計算機間の通信のためにメッセージパッシングを利用するためのライブラリとしては, MPI フォーラムにより規格化された API 仕様, MPI[1)] が一般的に用いられている. この API に準拠したライブラリ MPICH などを利用することにより, 比較的容易に分散処理を記述することができるため, 多くの利用例がある. しかし, この方法では必要な通信処理をユーザが記述しなければならず, 依然としてプログラミングのコストが高いと指摘されている.

従来より, 分散メモリ環境でのより容易な並列プログラム開発を可能とするために, IVY[2)] をはじめとする様々な分散共有メモリに関する研究が行われている. 分散共有メモリを利用した手法では, 自計算機が保持していないメモリ領域へもアクセスすることが可能なため, 従来の分散メモリを意識した並列プログラミングに比べて容易に並列プログラムを作成することができる. しかし, 他の計算機に配置されたデータへのアクセスはネットワークを介して行われるため, ローカルメモリへのアクセスに比べ非常に大きな時間を必要とする. そのため, 他計算機に配置されたメモリ領域

† 名古屋工業大学大学院情報工学専攻
Nagoya Institute of Technology, Electrical and Computer Engeneering

へのアクセスを頻繁に行う場合は, 低速なネットワークによる通信遅延が原因となり, 極端に処理性能が低下することになる.

しかし, 近年, 通信速度が非常に高速 (1Gbps〜10Gbps) になりつつあり, 通信遅延の要因である, 1) **バンド幅に依存する転送遅延**と, 2) **通信開始時の初期遅延**の 2 つのうち, 転送遅延は非常に短くなってきている. 将来ますますネットワークの速度は高速化されると予想され, 今後も転送遅延は減少していくと考えられる. しかし, 通信の初期遅延に関しては安価な TCP/IP プラットフォームを用いた場合は劇的な改善を望めない. 従って, 如何に通信の初期遅延を隠蔽するかが重要な問題となる.

この問題を解決するために, 従来の分散共有メモリシステムの研究では, メモリベース通信 (MBCF) 機能 [3] と通信のまとめ上げを行う最適化コンパイラを用いる東京大学の SSS-CORE[4] のように, 計算機間通信にかかるオーバーヘッドを削減している.

しかし, このような手法を用いてもやはりメモリの一貫性制御には, なんらかの同期機構を用い, また排他制御も必要なため, 通信によるレイテンシの影響は避けがたい.

それに対し本研究では, 従来のようにオンデマンドにデータを要求するのではなく, 共有データ全体をクラスタ計算機間で常時移動させ, 必要なデータが自計算機に到着したときにそれを使用する, 通信のレイテンシの影響を受けない新しい概念の分散共有メモリモデルを提案する.

以降, 第 2 章で常時転送型分散共有メモリシステムの概要について示し, 第 3 章では実装した提案システムの構成について述べ, 第 4 章では提案システムの性能評価を行う. さらに第 5 章で本提案手法の考察を行い, 第 6 章でまとめる.

2. 常時転送型分散共有メモリ概要

プログラムで必要となった時点 (オンデマンド) で他計算機の持つメモリ領域へのアクセスを行う場合, 通信の初期遅延によるオーバーヘッドが問題となる. そこで, 通信の初期遅延の問題を解消するため, 常時転送型分散共有メモリを提案する.

常時転送型分散共有メモリは, 従来のようにオンデマンドに他計算機のデータを要求するのではなく,

- 高速な通信速度をもつネットワークを利用し, 共有メモリ空間全体をクラスタ内で常時転送状態とする.
- 各計算機は, その時刻に保持している共有メモリ領域を用いて実行することが可能なタスクを行う.
- 必要な共有変数を得られず参照, 代入処理を実行できないタスクは, 一時キューに格納し, 実行可能時まで処理を遅らせる.

という概念に基づく分散共有メモリモデルである.

このモデルでは, 他計算機に対してオンデマンドにメモリ参照要求を行わないため, 通信の初期遅延の問題を受けない. また, 共有変数の転送と計算は独立に行われるため, 通信と計算をオーバーラップさせることが容易である.

2.1 ギガリングクラスタ

提案モデルの対象環境について述べる. 提案手法では共有メモリ領域全体を常時転送状態とするため, 大量のデータ転送が必要となる. そのため, 対象環境は高速な転送速度を持つ GigaEthernet を用いたクラスタ環境とする. また, 通信時のコリジョンの影響を排除するため, 計算機を図 1 のように GigaEthernet を用いてリング状に直接接続する. さらにリングネットワークとは別に, ハブを用いた FastEthernet ネットワークも利用し, 大量なデータ転送が必要となる共有データの転送には GigaEthernet を使用し, 制御パケットなどの小規模なデータ通信は FastEthernet リンクを使用する. 以降, このクラスタ環境をギガリングクラスタと呼ぶ.

これまでにリング型接続のネットワークで分散共有メモリを実現するシステムとしては, ハードウェア実装の Memnet[5] がある. Memnet は各計算機が共有メモリの複製を持つことを許す複製型分散共有メモリシステムである. トークンパッシング型のネットワークを使用し, 共有メモリへ書き込みを行う際にリングネットワークを 1 周するキャッシュ無効化パケットを送信することでメモリの一貫性を実現する, キャッシュ無効化型プロトコルを用いている.

これに対し本提案手法は, 他の計算機のキャッシュ状態を変更するのではなく, **共有メモリ空間自体が常時ギガリングクラスタ上で転送状態にある**ということが異なる.

クラスタ内の各計算機に共有メモリのキャッシュは存在するが, 共有変数の値を変更することができる計算機は, その時刻に書き込みたい共有メモリセグメントを保持する計算機唯一つに限られ, また, その計算機は他の計算機のキャッシュを無効化することなく共有変数の値を書き換えることができるので高速に代入処理を行うことができる.

2.2 分散共有メモリの実現

提案手法での分散共有メモリの実現方法について述べる. 提案手法では図 2 のように, まず共有変数全体を複数のセグメントに分割し, 各計算機に分散して初期配置する. 共有変数はその後, セグメント単位でクラスタ内を常に巡回する. 各計算機が共有変数へアク

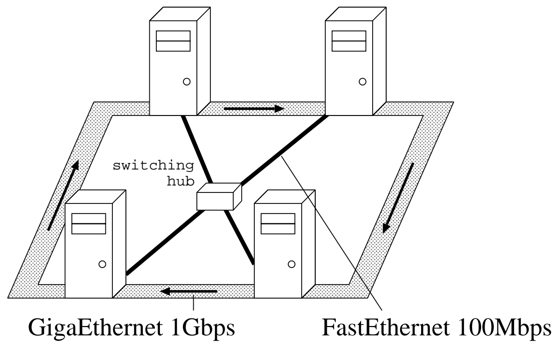

図 1　ギガリングクラスタ構成図
Fig. 1　Overview of Giga Ring Cluster

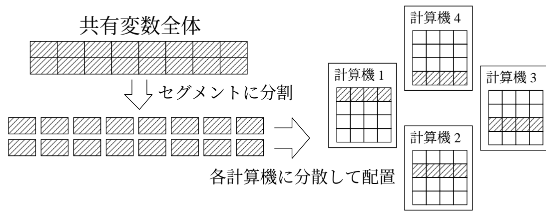

図 2　共有メモリ空間のセグメント化
Fig. 2　Segmentation of shared memory space

セスする際は, 保持しているセグメントに対してのみアクセスを許し, それ以外のセグメントへのアクセスは必要なセグメントの到着を待って非同期に行う (遅延アクセス). 従って, 対象プログラムはある程度の計算領域内で計算順序に依存関係がない SIMD 的なものである必要がある.

また, 各計算機は以前に受信したセグメントの値をキャッシュ値として保存し, 必要に応じてそれを使用して計算を行うことも許す. このように各計算機がアクセスできるのは各自が所持している共有変数領域のみに限られる. このため, 所持している領域は他の計算機からアクセスされることがないので排他制御が必要なく, 高速にアクセスすることが可能となる.

2.3　遅延アクセス

提案手法では, 各計算機に存在しない共有変数領域にアクセスする際には, そのアクセスを遅らせる遅延実行の形態をとる. 必要な共有変数が存在しないために処理を実行することができない状況は, 図 3 に示すように代入対象となる共有変数が存在しない場合と, 読み込みに必要な共有変数 (式の右辺) が存在しない場合の 2 つの可能性があり, それぞれの場合で異なる対応を行う.

図 3(1) のように代入される共有変数だけが存在しない場合は, 右辺の式を評価した結果をバッファに保存し, 次の計算の実行を継続する. 実際の代入は左辺共有変数を受信したときまで遅らせる. (遅延代入)

一方, 図 3(2) のように右辺の共有変数が存在しない場合は, 図 4 のように, その計算のコンテキストをいったんキューにため, 次の計算を継続する. (遅延参照)

キューに留めた計算は, 必要な共有変数が自計算機に転送されてきた時点で実行する.

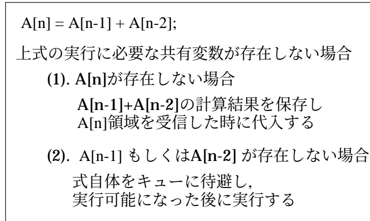

図 3　必要な共有領域がない場合の処理
Fig. 3　The delay write operation

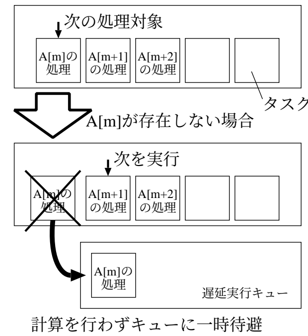

図 4　必要な共有領域がない場合のキュー操作
Fig. 4　The queue operation occurred from access to unusable shared memory segment

3.　常時転送型分散共有メモリの実装

提案手法を実行するランタイムライブラリを, C++ のクラスライブラリとして実装した. そのシステム構成について述べる. システムは図 5 に示すように計算スレッド, 受信スレッド, 送信スレッド, 転送制御スレッドの 4 つのスレッドおよび, 共有変数クラスで構成される. ユーザは共有変数クラスを用いて, 実際の計算を行う計算スレッド部を記述する. また, 共有変数の転送はシステムが提供する残り 3 つのスレッドが自動的に行う. そのため, ユーザは共有変数の分散を意識することなくプログラムを記述することができる.

3.1　プログラミングモデル

ユーザは共有変数クラス (Shared_variable クラス) と制御用関数を用いて, 共有メモリプログラミングと同様の形式で並列プログラムを作成する. 本手法におけるメモリモデルは, エントリ一貫性に基づいた並列プログラミングモデルである.

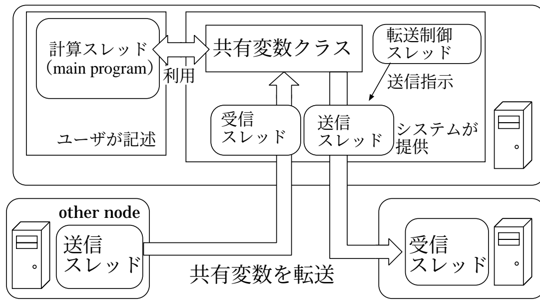

図 5 システム構成
Fig. 5 Overview of the continuously transmitted distributed shared memory system

3.2 共有変数の定義

共有変数を宣言する際の記述を図 6 に示す．共有変数は現時点では 2 次元および 3 次元の整数型, 浮動小数点型を扱うことができる．次元数は引数の数で判別し, 第 1 引数で共有変数で扱うデータ型を指定する．第 2 引数以降で各次元の大きさを指定し, 最終引数 (DIV_TYPE) で共有変数をセグメント化する際の分割方式を指定できる．共有変数は 1 次元目のインデクスでセグメントに分割され, 分割方式はブロック分割, サイクリック分割である．

```
//2次元共有変数
// int型
Shared_variable sv( GR_INT,   ROW, COL, DIV_TYPE);
// float型
Shared_variable sv( GR_FLOAT, ROW, COL, DIV_TYPE);
//3次元共有変数
// int型
Shared_variable sv( GR_INT,   X, Y, Z, DIV_TYPE);
// float型
Shared_variable sv( GR_FLOAT, X, Y, Z, DIV_TYPE);
```

図 6 共有変数の宣言
Fig. 6 declaration of shared variable

3.3 共有変数を用いた計算

共有変数を使用した計算を行う場合, 使用する共有セグメントの有無を調べた後実行する必要がある．また, 共有変数はセグメント単位でクラスタ内を巡回しているため, 実効的な性能向上の点から共有変数へのアクセスはセグメント単位への連続アクセスを行うことを基本とする．図 7 に共有変数を用いた計算の例を示す．図のように, 共有変数を使用する際には isSegment_lock 関数を用いて使用する共有セグメントを獲得する必要がある．

3.4 依存関係判定

実行する処理に順序依存が存在する場合, 並列処理単位を順序依存が存在しない小さな領域にまで分割し,

```
Shared_variable A(GR_INT, Y, X, BLOCK);

for(y=0;y<Y;y++){
  A.isSegment_lock(y);
  for(x=0;x<X;x++){
    A.idata[y][x] = expression;
  }
  A.Segment_unlock(y);
}
```

図 7 共有変数へのアクセス
Fig. 7 The access to shared variable

その領域の実行ごとに同期操作を用いて依存関係を解決する手法も可能ではあるが, 実行効率の観点から良い方法ではない．そこで本システム実装では, 共有変数に被代入回数という付加情報を与え, この情報を参照することで処理の順序依存関係を記述可能とした．図 8 に本手法で提供するセグメント情報調査関数を, 図 9 に依存関係の存在するプログラムの記述例を示す．

この例では, 図 9(1) で計算した結果を, 他の計算機が (4) で使用する可能性がある．そのため, (3) において被代入回数情報を確認し, それが (2) で更新されたものであるかどうかを判定している．

```
Shared_variableクラスのセグメント情報操作関数
 isSegment_lock(int seg_id)   存在調査関数
 Segment_unlock(int seg_id)  ロック解放関数
 check_assignment(int seg_id, int assignment)  被代入数調査関数
 plus_assignment(int seg_id)   被代入回数更新関数
```

図 8 セグメント情報調査関数
Fig. 8 Member functions of shared_variable class

3.5 トランスレータによる遅延アクセスコードの自動埋め込み

共有変数がない場合や依存関係を満たさない場合に遅延実行を行う処理は, ライブラリのみで実現することは不可能である．そのため, 現時点ではこの処理部分をプログラマが記述することで遅延実行を実現している．しかし, 将来的にはトランスレータを用いてこの処理を実現することを予定している．

4. 評 価 実 験

提案手法を実装したシステム上で行列積の計算と FDTD 法を用いた電磁界シミュレーションを行う並列プログラムを作成し, 性能評価実験を行った．

評価環境は CPU:Pentium3 866MHz x2 , Memory:512MByte, OS:Solaris 8 の SMP 計算機を 2 台および 4 台使用したギガリングクラスタを用いた．提案手法では共有変数の通信部に, 計算 CPU に負荷をかけることなく高速な転送を行うことができるハード

```
for(count=0; count<NUM; count++){
    for(y=id*n; y<(id+1)*n; y++){
        B.check_assignment(i-1,n);
        A.isSegment_lock(y);
        for(x=0; x<X; x++){
            A.idata[y][x] = (B.idata[y-1][x]+B.idata[y][x])/2;
        }                                                  (1)
        A.plus_assignment(y);
        A.Segment_unlock(y);                               (2)
    }

    for(y=id*n; y<(id+1)*n; y++){
        A.check_assignment(y-1,n+1);                       (3)
        B.isSegment_lock(y);
        for(x=0;x<X;x++){
            B.idata[y][x] = (A.idata[y-1][x]+A.idata[y][x])/2;
        }
        A.plus_assignment(y);                              (4)
        A.Segment_unlock(y);
    }
}
```

図 9 依存関係の存在する並列プログラム
Fig. 9 A parallel program with a dependency

```
Shared_variableA(GR_FLOAT,ROWS,COLS,BLOCK);

my_id = gr_init();   // ←初期化およびIDを取得
computers = gr_get_np();   //計算機台数を取得
start = (ROWS/computers)*my_id;

for(y=start;calc=0;calc<ROWS;y++,calc++){
    y %= ROWS;
    A.isSegment_lock(y);              // ロック
    for(x=0;x<COLS;x++){
        for(k=0;k<COLS/computers;k++){
            A.fdata[y][x] += B[y][k]*C[k][x];
        }
    }
    A.Segment_unlock(y);              // アンロック
}
```

図 10 行列積演算のプログラム主要部
Fig. 10 The parallel programming for matrix multiplication

```
for(int n=0;n<NSTOP;n++){
    for(int x=start;x<end;x++){
        Hy.check_assignment(x-1,n);  // 被代入回数の判定
        Ex.isSegment_lock(x);
        for(int y=LM;y<JJ-LM;y++){
            for(int z=LM;z<KK-LM;z++){
                Ez.fcube[x][y][z] = Cez[x][y][z]*Ez.fcube[x][y][z]
                    +Cezl[x][y][z]/DX*(Hy.fcube[x][y][z]-Hy.fcube[x-1][y][z])
                    -Cezl[x][y][z]/DY*(Hx.fcube[x][y][z]-Hx.fcube[x][y-1][z]);
            }
        }
        Ez.plus_assignment(x);
        Ez.Segment_unlock(x);  // 被代入回数の更新
    }
・・・・・・省略・・・・・
    for(int x=start;x<end;x++){
        Ez.check_assignment(x+1,n+1);  // 被代入回数の判定
        Hy.isSegment_lock(x);
        for(int y=LM;y<JJ-LM;y++){
            for(int z=LM;z<KK-LM;z++){
                Hy.fcube[x][y][z] = Hy.fcube[x][y][z]
                    -Chz*(Ex.fcube[x][y][z+1]-Ex.fcube[x][y][z])
                    -Chx*(Ez.fcube[x+1][y][z]-Ez.fcube[x][y][z]);
            }
        }
        Hy.plus_assignment(x);// 被代入回数の更新
        Hy.Segment_unlock(x);
    }
}
```

図 11 FDTD 法による電磁界シミュレーションのプログラム主要部
Fig. 11 The parallel programming for electromagnetic simulation using the FDTD method

ウェアアーキテクチャを想定しているが, 本実験ではそのかわりに 2CPU を搭載した SMP 計算機を用い, 1CPU を通信プロセッサとして使用した.

4.1 行列積の計算

1 辺 n の行列積の演算は処理量が $O(n^3)$ と大きい上, 実行順序に依存関係がなく, 並列処理効果が高い問題である. 実験では, 1 辺の長さが 512, 1024, 2048 の正方行列を用いた. 作成したプログラムの主要部を図 10 に示す. プログラムでは, まず gr_init 関数を用いてシステムの初期化を行い, 返り値として自計算機の ID を得ている. さらに, gr_get_np 関数を用いて並列実行する計算機台数を取得し, それらの情報を用いて各計算機の計算範囲を設定している.

作成したプログラムでは共有変数をロックする記述が必要となっているが, 通信の記述などは必要なく, ほぼ逐次プログラムと同じように簡潔に記述することができている.

4.2 FDTD 法による電磁界シミュレーション

1 辺の長さが n の 3 次元空間についての FDTD 法を用いた電磁界シミュレーションも, 行列積演算と同様に処理時間が $O(n^3)$ の問題である. 作成したプログラムの主要部を図 11 に示す. この問題では, 計算の一部に他の計算機が計算した結果を必要とするものがあるため, 依存関係を考慮した記述を行っている. しかし行列積の並列プログラムと同様, 単純な記述を追加するだけでよく, 簡潔にプログラムできていることがわかる.

4.3 実験結果

行列積演算の実験結果を図 12 に, FDTD 法による電磁界シミュレーションの実験結果を図 13 に示す. 本実装では共有セグメントの転送間隔をパラメータとして指定できるが, 行列積演算については 10ms, 電磁界シミュレーションについては最小値を指定し, 可能な限り共有メモリ空間を転送状態において実験を行った. 図中, 横軸は使用した計算機の台数, 縦軸は速度向上率である. 速度向上率は, 「計算機を N 台利用した場合の実行時間 / 逐次プログラムでの実行時間」とした.

行列積の演算では, どの行列サイズにおいても 2 台の計算機使用時で約 1.9 倍, 4 台使用時で約 3.9 倍の

速度向上が得られた．この結果は台数にほぼ比例した良好な結果だといえる．

また FDTD 法を用いた電磁界シミュレーションでは, 2 台の計算機使用時で約 1.5 倍, 4 台使用時で約 2.9 倍の速度向上となった．行列積演算ほどの台数効果が得られなかった原因として, 依存関係を満たすための計算待ちが生じたことが考えられる．

しかし, 計算機台数が 2 台と 4 台の間での速度向上は約 2 倍となっており, 計算機台数をさらに増やした場合での速度向上が期待できる．また, 1 辺のサイズが 100 の場合での速度向上が他のサイズでの速度向上よりも低い値となっているのは, 1 辺が 100 の場合では計算に必要な処理量が小さく, 必要なセグメントを得るまでの待ち時間が生じたためと考えられる．

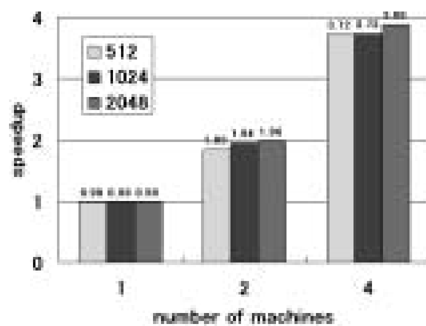

図 12 行列積の並列処理効果
Fig. 12 Speed up rate of execution time for matrix multiplication

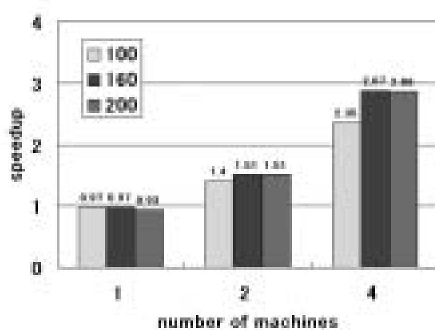

図 13 FDTD 法による電磁界シミュレーションの並列処理効果
Fig. 13 Speed up rate of execution time for electromagnetic simulation using the FDTD method

5. 考　　察

本章では, 現時点における提案手法の問題点について考察する．

提案手法における共有メモリの扱いは従来の分散共有メモリシステムと大きく異る．分散コンピューティングにおいて, データの初期配置は処理性能に大きく関わる重要な問題であるが, 本手法では共有変数を各クラスタに初期配置こそするものの, その後は常にクラスタ内を巡回する．そのため, 初期のデータ配置を正しく行うだけで後の計算をほぼ完全独立に実行できるような問題では, 常時転送は無駄なデータ通信であるだけでなく, 必要な共有変数が得られない場合のキュー操作などのオーバヘッドを生み出す原因となる．

このオーバヘッドを隠蔽するために, より高速な通信環境 (ハードウェア, OS の支援, 通信プロトコル) が必要となる．今後さらなるクラスタ間の通信速度の向上によりこれらの問題も解決するものと考える．

6. ま と め

本論文では, 通信の初期遅延を問題としない, ”共有メモリ空間を常時転送状態とする”分散共有メモリモデルの提案を行った．また, GigaEthernet を用いた高速な転送速度を持つリング型クラスタ (ギガリングクラスタ) 上に C++のクラスライブラリとして本手法を実装し, 性能評価を行った．その結果, 対象とする問題に制限はあるものの, 比較的容易に並列プログラムを記述することができ, また並列処理による速度向上が得られることを確認した．

今後の予定として, より多くの問題での性能評価実験および, トランスレータによる遅延実行処理コードの埋め込みや実装の最適化を行う予定である．

参 考 文 献

1) W. Gropp, E. Lusk, and A. Skjellum: *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, (1999).
2) K. Li: *IVY:A Shared Virtual Memory System for Parallel Computing.* International Conference on Parallel Processing, pp.94-101,(1998)
3) 松本 尚, 駒嵐 丈人, 渦原 茂, 竹岡 尚三, 平木 敬: 汎用超並列オペレーティングシステム：SSS-CORE —ワークステーションクラスタにおける実現—, 情報処理学会研究報告 96-OS-73, Vol. 96, No.79, pp.112-120 (1996)
4) 松本 尚, 渦原 茂, 竹岡 尚三, 平木 敬 : 汎用超並列オペレーティングシステムカーネル SSS-CORE , 第 17 回技術発表会論文集, 情報処理振興事業協会, pp.175-188 (1998)
5) Delp. G. S, Farber, D.J. Minnich, R.G. Smith, J.M. Tam,and M.:*Memory as a Network Abstraction*, IEEE Network Magazine, pp.34-41 (1991)