ハイパフォーマンス
コンピューティング 62－2
（1996. 8. 28）

分散メモリ型並列計算機における共有オブジェクト空間の実現

丹羽純平† 稲垣達氏†
松本尚† 平木敬†

分散メモリ型計算機において、メッセージパッシングライブラリを使用して、動的で複雑なデータ構造を扱うアプリケーションを記述することは多大な労力を必要とする。それを減少させるためには、言語やランタイムシステムが共有名前空間を提供する必要がある。

本稿では、ソフトウェアによりオブジェクトベースの共有名前空間を提供する方法を述べ、既存のシステムがユーザーに委ねてきた低レベルの通信に関する記述を、コンパイラがコードを解析して生成することを提案する。更にこの記述を複数個用意して、コンパイラがそれらを状況に応じて使い分けることにより性能の向上を試みる。我々はAP1000+上に作成したプロトタイプシステムを用いて、本方式により高速化が達成されることを確認した。

Compiler Oriented Implemntation of Shared Object Space on Distributed Memory

JUNPEI NIWA,† TATSUSHI INAGAKI,† TAKASHI MATSUMOTO† and KEI HIRAKI†

On a distributed memory parallel machine, it needs much effort to write applications which deal with dynamic and complex data structures by using message passing library. To reduce the difficulty, it is necessary for a language or a runtime-system to provide shared name space.

In this paper, we describe how to provide the software shared name space based on objects. Existing systems entrust users the description of low level communication. We propose that the compiler analyzes the code and supports the description of low level communication. Furthermore we propose that the compiler generates many descriptions of the communication, and the compiler uses the suitable one as the case may be, which results in speedup. We develop the prototype system runnning on AP1000+, and evaluate our approach, which exhibits good speedup.

1. はじめに

動的に生成されるオブジェクト☆を用いた並列処理は、不規則なデータ構造による計算負荷の分配や、動的な負荷分散への有力な手段である。しかし、分散メモリ型計算機においてこれらのアプリケーションを記述するためには、共有されるデータに対して、通信を行なうために明示的にメッセージパッシングを使用する必要がある。更に、コンシステンシの管理やプロセス間の同期が必要になる場合もある。したがって、シングルプロセッサ用のプログラムを大幅に修正する必要があり、多大な労力を必要とする。

上記の労力を減少させるためには、共有データの変更がすぐに全プロセッサに反映される共有名前空間というインターフェイスを与えることが必要になる。このインターフェイスを与えるソフトウェアのシステムは既に数多く開発されており、大別すると次の二種類に分類される。

(1) 『共有アドレス空間を実現してページ単位でコヒーレンスの管理を行なう』

この方式はIVYのShared Virtual Memory [6](SVM)に代表される。しかし、false sharingの問題が避けられない。

false sharingの問題に対処するために様々な方式が提案されている。例えばTredmark [4]では、Lazy Release Consistencyを使用し、複数のプロセッサが同じページに書き込むことを許し、次の同期点で変化をマージする方式を採用している。SVMではAutomatic Update Release Consistency [3]を使用し、homeは常にupdateされ、そのcopyに関しては各同期点

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Faculty of Science, University of Tokyo

☆ オブジェクト指向のオブジェクトではなく、いわゆるユーザー定義のデータ構造 (Tree, Linked List, Graph)

で変化がマージされる方式を採用している。

(2) 『オブジェクトベースの共有名前空間を実現する』
false sharing の問題を防ぐため、ページ単位ではなくユーザー定義のオブジェクト単位で通信／コンシステンシの管理を行なう。完全に動的な共有名前空間を提供したシステム (SAM [7]) から Inspector/Executor というスキーム [5] を動的なオブジェクトにまで拡張したもの (Chaos++ [1]) まで様々なシステムが提案されている。いずれも、コンシステンシの管理をランタイムライブラリや言語のシステムにより仮想化している。

我々は false sharing の問題を回避するために、自動並列化コンパイラがユーザー定義のオブジェクト単位の大域的な名前空間を提供する方針を取る。

分散メモリ環境においては、オブジェクトの複製並びに移動は全て通信によって行なわれる。既存のシステムでは、基本型 (int や double 等) に関してはランタイムライブラリや言語のシステムが、低レベルの通信に関する記述をサポートする。一方、ユーザー定義の動的なオブジェクトに関してはユーザーに解放され、最適化の可能性はユーザーに委ねられていた。その理由は、階層的構造を有するオブジェクトの場合には簡単にメモリからメッセージにコピーするだけではうまくいかず、もっと洗練された方法が必要とされるからである。

我々は、自動並列化コンパイラに力点を置いたアプローチにより、ユーザー定義の動的なオブジェクトに関して、これまでユーザーに解放されてきた低レベルの通信に関する記述をコンパイラが自動生成することを提案する。これだけでは性能の観点から不十分なので、更に低レベルの通信に関する記述を複数個用意して、コンパイラがコードを解析して適切なものを選択することで、性能の向上を図る。

もちろん並列化にあたって、性能向上のためには、グローバルポインタの実装や、複製の管理方法が密接に関係してくる。また実行時の効率の良いサポートが必須になる。

2. 共有名前空間の実現

2.1 オブジェクト単位

ユーザー定義のオブジェクトで大域的に複数のプロセッサにアクセスされるオブジェクトを**大域オブジェクト**と呼ぶ。各大域オブジェクトには**大域 ID** が割り振られる。この大域 ID はたとえ大域オブジェクトが移動しても変わらない [7]。

大域オブジェクトにはホームとオーナーが存在する。

- オーナーは現在そのオブジェクトを所有しているプロセッサである。
- ホームはオブジェクトのオーナーがどのプロセッサか知らせてくれるプロセッサであり、大域 ID とオーナーの組を管理している。
- オブジェクトが移動してオーナーが変わったらホームに知らせる。

2.2 実行時のサポート

実行時のデータアクセスの度に送信／受信プロセッサを確定することは、高オーバーヘッドにつながる。動的で不規則なデータ構造を持つ問題の多くは、ループにおけるデータのアクセスパターンがその計算を実行する前に知ることが可能で、しかもループ不変である。したがって、我々は、ループ本体を実行する前にプロセッサ間通信を伴うデータ参照を調べる Inspector を実行し、引き続いてループ本体の計算を行なう Executer を実行するコードを生成する方式 (Inspector/Executer スキーム [5]) を使用する。

Inspector/Executer

(1) Inspector
ループを実行する前にデータの参照を調べて、受信しなければならないデータの集合を列挙し、受信集合を元に送信するべきデータの集合を列挙し、通信のスケジューリングを生成する。

(2) Executer
Inspector で生成したスケジューリングを利用して実際のループ本体を計算する。
- スケジューリングにしたがって送信
- 局所的なデータを使用した計算
- スケジューリングにしたがって受信
- 局所的ではないデータを使用した計算

3. ポインタベースのオブジェクトを扱う時の注意

- ポインタとはシングルプロセッサの場合ではアドレスに他ならない。分散メモリ環境では、あるプロセッサにおけるアドレスは他のプロセッサーにとっては無意味なものであるから、あるオブジェクトを通信によって他に送る時に、メンバーにポインタがある場合は、そのまま中身をコピーして送るだけでは意味がない。
- 動的で不規則なデータ構造を持つ問題では、実行時に動的にオブジェクトを生成し、オブジェクトはポインタを介してアクセスされる。分散メモリ環境でポインタベースのオブジェクトを分散させるためには、従来の (局所的) ポインタを使用することはできない。その理由は、同一のプロセッサにおけるアドレス空間においてのみ有効な従来の (局所的) ポインタは、別々なプロセッサが所有するオブジェクトをつなげることができないからである。

グローバルポインタ

グローバルポインタは他のプロセッサが所有するオブ

ジェクトを指すことが可能である。グローバルポインタは (オーナー, オーナー内の局所アドレス) の組であらわされる [2] [1]。上記の実装は、確かにオブジェクトの移動の回数が少ない場合は効率的だが、頻繁に移動する場合には、その都度そのオブジェクトへのポインタを全て修正する必要がある。

そこで我々はグローバルポインタを大域 ID で表現することにより、大域オブジェクトが移動する時に、そのオブジェクトへのポインタを全て修正する手間を省くことができた。大域オブジェクトが移動した場合、オーナーが変化したことをホームに一回通知するだけで良い。

Inspector/Executer スキームを使用する場合 [1]、Inspector では、グローバルポインタを介してオブジェクトを参照していて、かつ自分がオーナーでないと判断したら、そのオブジェクトを自分の所に複製するようにスケジューリングを生成する。Executer においては、スケジューリングに従って通信を実行し、オブジェクトを自分の所に複製し、実際の計算では全てこの複製を使用する。

4. Pack/Unpack

オブジェクトのメンバーが部分オブジェクトへのポインタを含む階層的構造を持つ場合を例として用いる (図 1)。このオブジェクトへリモートからのアクセスが

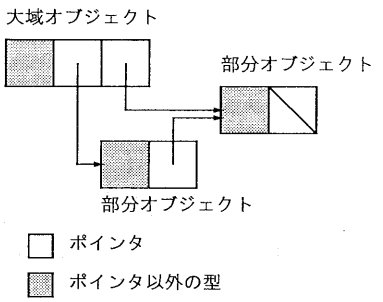

図1 階層的構造を有するオブジェクト

起こったとしよう。オーナーはこのオブジェクトを一個のメッセージに Pack して送信する必要がある。ところが、このオブジェクトのメンバーの中にはポインタが存在して、ただ値をメッセージにコピーするだけではうまくいかない。同様に、オブジェクトへのアクセスを起こした側が送られてきたメッセージを Unpack した時に、オブジェクトのメンバーが正しく部分オブジェクトを指すようにうまくメモリをアロケートしてやる必要がある。特に図 1のようにメンバーが指す部分オブジェクトが disjoint でない時は、Pack/Unpack の関数を書くことは容易ではない。

既存のシステム [7] [5] では、基本型ならびにその配列に関してはランタイムライブラリや言語のシステムがサポートする。一方、階層的構造を持つオブジェクトを Pack/Unpack する部分に関しては、ユーザーに解放し、最適化の可能性をユーザーに委ねるという方針である。もちろん最適化の観点からいえば、オブジェクトを如何に使用するかを知っているユーザーが Pack/Unpack のコードを書くのがベストである。しかし、せっかく共有名前空間を提供してプログラミングをしやすい環境を提供したにもかかわらず、共有メモリ型マシン上でプログラミングする場合には現れない部分をユーザーに委ねることは、ユーザーに負担をかける。我々は自動並列化コンパイラの立場からコンパイラがコードの解析を行ない、オブジェクトを Pack/Unpack するコードを自動生成する方式を提案する。

4.1 Pack/Unpack の自動生成

コンパイラが実際の参照を元に、ポインタを介して参照される階層的構造を持つオブジェクトを、メモリからメッセージに Pack し、メッセージからメモリに Unpack するコードを生成する。そのために必要な情報、並びにコードを生成する手順を述べるとともに、コンパイラがサポートできない場合を例をあげて説明する。

Pack/Unpack のコードを自動生成するのに必要な情報は、

(1) 大域オブジェクトとその部分オブジェクトの型宣言
(2) 参照しているオブジェクトのメンバー
(3) アロケートされたメモリの大きさ
ポインタが指す領域をどれだけメッセージにコピーすれば良いのか判断するために必要である

である。(1) と (2) はコードを解析することでコンパイラが収集できる情報である。(3) はコンパイラだけでは判断できず、ランタイムライブラリのサポートが必要である。C のライブラリ関数 `malloc` と `free` を修正して、アロケートされたメモリの大きさを記憶できるようにすることでこの情報は動的に収集できる。

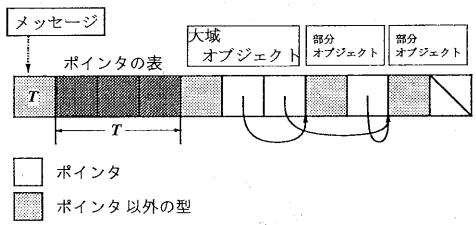

図2 オブジェクトをメッセージにパッキングする例

Pack のコード生成の手順を以下に簡単に述べる。オブジェクトをバッファにコピーするコードを生成する。オブジェクトのアドレス (ポインタの値) をテーブルに登録するコードを生成する。次に参照リストに含ま

れる各メンバーに対して

- 基本型かその配列の場合には何もしない
- グローバルポインタの場合には何もしない
- 局所ポインタの場合には
  - 既に登録済みの時は何もしない
  - 未登録の時は、アロケートされたメモリ領域の大きさを調べて、メンバーが指す内容をバッファにコピーし、ポインタをテーブルに登録する。参照の仕方によっては (メンバーが指すものがオブジェクトやポインタの場合には) 再帰になる

  というコードを生成する
- オブジェクトかその配列の場合には、このオブジェクトの各メンバーに対して同様のことをする

Unpack のコード生成はこの逆の操作をすれば良い。

実際にはオブジェクトをメッセージに Pack する前に、メッセージ領域を確保するためにメッセージの長さを知る必要がある。そこで Pack を行なう前に、Pack と良く似た手順でメッセージの長さを調べる関数を走らせる必要がある。同様に Unpack を行なう前に、メッセージ内のポインタのテーブルを参考にして、先にメモリアロケーションを行なう必要がある。いずれも Pack/Unpack 関数と同様にコンパイラが自動生成する。

もしオブジェクトが指す部分オブジェクトが全て disjoint ならば、上述のポインタをテーブルに登録する手間が省け、高速化が可能になる。残念ながら、これはコンパイラが参照のリストや宣言のリストから判断できない。現時点ではユーザーに annotation をつけてもらうことで対処する。

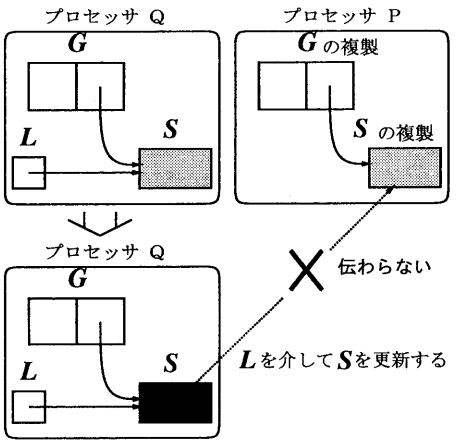

図3　コンパイラがサポートできない例

もちろん、コンパイラが自動的にデータを Pack/Unpack するコードを生成するにしても限界がある。大域オブジェクトからたどれる部分オブジェクトが全く別な方法でアクセス可能な場合を例として用いる (図 3)。あるプロセッサ P がこの大域オブジェクト G とその部分オブジェクト S の複製を必要とした時、そのオーナーのプロセッサ Q は G だけではなく S も一緒に P に送る。その後で、Q が大域オブジェクト G を介してではなく、局所的なオブジェクト L を介して S を更新した時、S が更新されたことを P の持つ S の複製に伝えることができない。この後、P が G の複製を介して S の複製を参照した場合、古い値が参照される。

コンパイラがサポートできる条件は、『大域オブジェクトからたどれる部分オブジェクトはその大域オブジェクトからのみアクセス可能である』とまとめられる。

4.2　アップデートされた時の Pack/Unpack

オブジェクトの構造が複雑で大きい時、オブジェクトのメンバーの一部しか変更されない場合には、通信の都度オブジェクトを全部送っていてはオーバーヘッドが大きくなる。大域オブジェクトの一部のみが変更された時には、その複製を更新するために変更箇所のみを送ることで、高速化が可能になる。

性能の向上のためには、同じオブジェクトに対しても複数の Pack/Unpack が存在して、状況に応じて使い分けをすることが必要である。既存のシステム [7] [5] はこのようなカスタマイズはユーザーに委ねている。本稿ではプログラミングをしやすい環境を与える自動並列化コンパイラの立場からコンパイラがサポートすることを提案する。

我々は コンパイル時に Pack/Unpack 関数をそれぞれ二種類生成する。

(1)　オブジェクト全体の Pack/Unpack
これは上述のオブジェクト全体を Pack/Unpack するものである。コンパイラがコードを解析して、初めてオブジェクトの複製を作成する時、オブジェクトを移動させる時、オブジェクトの構造が変化した時 (つまり、各オブジェクトのポインタ型のメンバーの値が変化した場合)、これらの関数を呼びだす。メッセージにはポインタの情報を正しく復元するのに必要な情報が含まれている。
Pack する前にはメッセージ長を計算する関数を走らせ、Unpack する前にはメモリアロケーションを行なう関数を走らせる必要がある。

(2)　アップデート方式の Pack/Unpack
コンパイラがコードを解析して、オブジェクトの構造が変わらずに一部分のみが修正されたと判断した時に、これらの関数を呼び出す。
オブジェクトの構造が変わらないためポインタの情報をメッセージにつめる必要はなく、オブジェクトのポインタ以外のメンバーで使用されるもののみをメッセージにつめる。これによりメッセージ長が短くなる。更に Pack/Unpack する前にメッセージ長を計算する関数 / メモリアロ

ケーションを行なう関数を走らせる手間を削減できる。

例えば、ループの中で大域オブジェクトにリモートアクセスをして、更にこの大域オブジェクトの構造がループ不変であるとすれば、最初のイテレーションだけオブジェクト全体の Pack/Unpack 関数を呼び、それ以降はアップデート方式の Pack/Unpack 関数を呼べば良い。

5. 実　　装

当研究室では現在、自動並列化コンパイラとランタイムライブラリを開発中である。現在の実装ではコンパイラは共有オブジェクト空間で計算を行なう SPMD コードを入力とし、分散メモリ計算機用の通信コードが入った SPMD コードを生成する。現在、ターゲットマシンとして AP1000+ を使用している。同一のプロセッサへのメッセージは集約して送信／受信する。バックエンドコンパイラは gcc を使用している。

以下ではこのプロトタイプシステムを用いて実験を行ない、我々の提案した方式の有効性を確かめる。

5.1 EM3D

不規則なメッシュ格子点の値を隣接点の値を用いて反復計算するアプリケーション EM3D [2] のコード生成を行なった。

```
typedef struct e_node {
  double         value;
  int            edge_count;
  double         *weights;
  struct h_node * global (*h_nodes);
  /** 隣接点のリスト **/
  struct e_node *next;
} e_node_t;
```

h_nodes というメンバーが隣接するノードへのポインタの配列である。e_node_t に関する計算を以下に示す。

```
int          t,i;
e_node_t     *e;
for(t = 0; t < TIME_STEPS; t++)
  for(e = e_nodes; e; e = e->next)
     for (i = 0; i < e->edge_count; i++)
       e->value -=
        e->h_nodes[i]->value *
        e->weights[i];
```

このコードを入力として出力されたのが以下のコードである (宣言は省略している)。

```
for (e = e_nodes; e; e = e->next)
  for (i = 0; i < e->edge_count; i++)
     e->h_nodes[i] =
        AcquireGhostObject
        (e->h_nodes[i],"struct h_node");
/** スケジューリングの生成 **/
GenerateSendReceiveList(sh);
/** ここまでが Inspector で
    これ以降が Executer **/
for(t=0;t<TIME_STEPS;t++){
  PackAndSend(sh);
  RecvAndUnpack(sh);
  e_nodes に関する計算;
}
```

AcquireGhostObject という関数では求めるオブジェクトのオーナーが自分でなければ複製を生成し、そのことを記録する。次の GenerateSendReceiveList という関数で 上述の記録をもとに受信集合と送信集合を生成し、スケジューリングを生成する。

この EM3D のコードを AP1000+ 上で実行して実行時間 (Executer の部分) を計測した。問題の設定は頂点数 16384、辺数 36656(リモートを参照する辺の割合は 10%)、TIME_STEP = 30 で、オブジェクト全体を Pack/Unpack する方式を使用した。

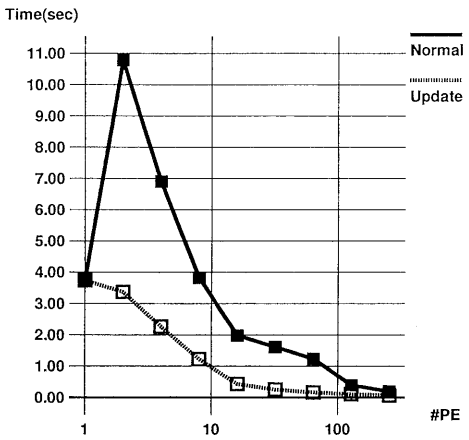

図4　EM3D の実行時間 (秒) と台数の関係

結果は図 4の Normal である。縦軸は Executer の実行時間 (秒) で、横軸はプロセッサ台数の 10 を底にした対数である。スケーラビリティはあるものの、Pack/Unpack のコストが大きく 8 台で 1 台と同程度の性能しか得られない。

t = 1 以降ではオブジェクトの構造は変わらない、かつデータのアクセスパターンは変わらない。従って、PackAndSend と RecvAndUnpack の代わりに 3.2.2 で提案したアップデート方式の Pack/Unpack 関数を使用した PackNoAllocateAndSend と RecvAndUnpackNoAllocate を用い、実行時間を計測

したのが図4のUpdateである。

NormalとUpdateでは3倍から8倍近くの性能の差が見受けられる。Updateのメッセージ長はNormalのメッセージ長の3分の1倍であり、UpdateのPack/Unpack関数はNormalのものと比較して約3倍の高速化が達成されている。それだけではなく、前もってメッセージの長さを計算する関数が走らなくても良いという点や、メモリアロケーション時の情報を引き出す手間を省くことができたという点が、この高速化の要因として挙げられる。

次に、Pack/Unpack並びに通信にかかる時間とリモートデータを参照する量との関係を調べてみる。問題の設定はTIME_STEP = 30で、台数は256台、頂点数は16384、辺数は36656で固定する。アップデート方式のPack/Unpackを使用し、リモートを参照する辺の割合を変化させて実験を行なった(図5)。図5のExecはExecuterにかかった時間でCompは実際の計算にかかった時間である。

リモートを参照する辺の割合が増化しても、計算時間はほとんど変わらないはずである。よって実行時間の増加のほとんどはPack/Unpack並びに通信時間の増加である。図5からはPack/Unpack並びに通信時間の手間はリモートを参照する辺の割合、つまりリモートのデータを参照する量に対して線形に増加していると判る。

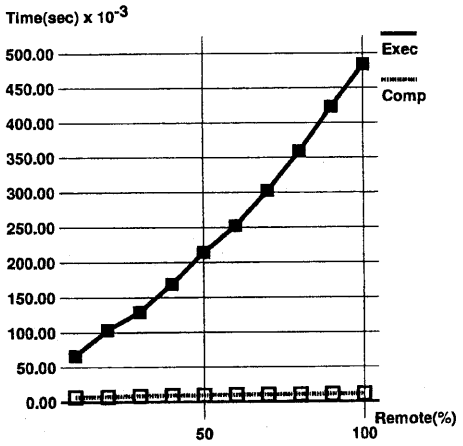

図5 リモート参照をするデータ量と実行時間の関係

6. まとめと今後の課題

本稿では、自動並列化コンパイラという観点から、分散メモリ環境におけるオブジェクトベースの共有名前空間を実現し、今までユーザーに委ねられてきた低レベルの通信に関する記述(Pack/Unpack)をコンパイラが自動生成することを提案した。更に性能の向上のために、低レベルの通信に関する記述を複数用意し、状況に応じて適切なものを使用することを提案した。その提案が正しいことを実験によって確かめた。今回用いた例はPack/Unpack関数が非常に簡単になるので、残念ながらコンパイラがコードを自動生成してもその効果を十分に享受できる例ではない。

今後は、更にオブジェクト構造が複雑なアプリケーションを走らせて、この方式の有意義性を更に確かめる予定である。

このPack/Unpack関数の自動生成並びにその使い分けは今回使用したInspector/Executerスキームに限らず、ポーリングベースのシステム[7]においても利用可能であるし、一般に分散メモリ環境においてメッセージでオブジェクトを転送する場合に利用可能な方式である。

参考文献

1) Chialin Chang, et al. Object-Oriented Runtime Support for Complex Distributed Data Structures. Technical Report CS-TR-3428, University of Maryland, March 1995.

2) David E. Culler, et al. Parallel Programming in Split-C. In *Proc. of Supercomputing '93*, pp. 262–273, November 1993.

3) L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proc. of the 2st IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, February 1996.

4) P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pp. 115–131, January 1994.

5) Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. on Parallel and Distributed Sys.*, Vol. 2, No. 4, pp. 440–451, October 1991.

6) K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, Vol. II, pp. 94–101, August 1988.

7) Daniel J. Scales and Monica S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proc. of 1st Symp. on Operating Systems Design and Implementation*, November 1994.