OpenC++のリフレクション機能を用いた分散共有メモリの実現

早田恭彦† 小川宏高† 松岡聡†

近年の計算機構成技術の発展に伴って多様化するソフトウェア実行環境の上でプラットフォームポータビリティを確保することは希求の課題である．特にPCクラスタ/WSクラスタ等の分散メモリ並列機上の並列言語処理系に関しても同様のことが求められる．つまり，汎用性・保守性を維持しつつ，実行時環境に最適化されたプログラムを生成できなければならない．このような要求を満たす処理系を実現する一つの方法はリフレクションを用いることである．本稿では，C++言語のOpen Compilerである OpenC++2.5 の Compile-time MOP を用いて，SPMD スタイルで書かれた C++の並列プログラムに対して，共有分散メモリ機能を付加する並列言語処理系を実現した．具体的には Compile-time MOP を用いたプログラム変換によって，共有変数に対応する共有メモリ管理オブジェクトの生成・初期化，共有変数へのアクセスの管理オブジェクトを介したアクセスへの変換などを行って MPC++のプログラムを生成する．Myrinet で接続された 8 台構成の PC クラスタ上で性能評価し，リフレクションを用いた分散共有メモリ実現の有効性を確認した．

Implementation of DSM Using OpenC++ Reflection

Yukihiko Sohda,† Hirotaka Ogawa† and Satoshi Matsuoka†

Platform portability is one of the utmost demanded properties of a system today, due to the diversity of runtime execution environment of wide-area networks, and parallel programs are no exceptions. However, parallel execution environments are VERY diverse, could change dynamically, while performance must be portable as well. As a result, techniques for achieving platform portability are sometimes not appropriate, or could restrict the programming model, e.g., to simple message passing. Instead, we propose the use of *reflection* for achieving platform portability of parallel programs. As a prototype experiment, a software DSM system was created which utilizes the compile-time metaprogramming features of OpenC++ 2.5 to generate a message-passing MPC++ code from a SPMD-style, shared-memory C++ program. The translation creates memory management objects on each node to manage the consistency protocols for objects arrays residing on different nodes. Read- and write- barriers are automatically inserted on references to shared objects. We evaluated this system on a PC cluster linked by the Myrinet gigabit network.

1. はじめに

近年の計算機構成技術の発展によって，ソフトウェアの実行環境はますます多様化する一方である．しかしながら，肝心のソフトウェアの構成技術はその発展に追従できているとは言いがたい．計算機構成技術の発展は，ハードウェアの急速なコモディティ化と対応するものであり，PCクラスタ・WSクラスタという形態での分散メモリ型並列計算機や，高性能な組み込み機器を安価かつ容易に実現した．これらの動きにはエンドユーザレベルのソフトウェア開発についても影響を免れない．従って，保守性や可搬性を重視して汎用的に書かれたソフトウェアをこれらの実行環境に適合させることが望まれるが，そのための枠組が十分でないのが現状である．例えばオブジェクト指向技術は確かにソフトウェアの効率の良い構築フレームワークを与えるが，実行環境の変化に適合した最適化を行うには十分ではない．

従来，こうした多様化する計算機環境，特に並列実行環境でのプラットフォームポータビリティを実現するために，HPFに代表されるような並列プログラミング言語や自動並列化コンパイラ，あるいは言語機能を拡張せずに済むメッセージ通信ライブラリが用いられてきた．しかし，この種の固定的な処理系ではプログラムの記述性が損なわれたり，実行時環境を強く意

† 東京工業大学 情報理工学研究科 数理・計算科学専攻
Tokyo Institute of Technology

識しない限り高い性能を発揮することが困難であったり，そもそも処理系自体の可搬で高効率な実装や言語拡張が難しいという問題があった．

リフレクションは，この種の問題を整合的かつ柔軟に解決する方法の一つである．基本的には，通常の計算を表す従来からのベースレベルのコードに加え，自己の計算系を表すメタレベルのコードを記述することができるため，従来例外的かつ固定的に扱わざるを得なかった例外処理や資源管理を整合的に処理できる．特に，Open Compilerと呼ばれる，コンパイラのコンパイル時の挙動を自己反映的に変更する機能を持つシステムでは，これらの機能を容易に実現できることが知られている．Open Compilerではコンパイラをオブジェクト指向設計に基づきモジュール化してユーザから変更可能にし，更にコンパイル時に静的にメタ計算を行うこと（Compile-time MOP）によって言語の拡張やアプリケーション固有の最適化などを行うことができる．

そこで，我々は様々な並列実行環境上でプラットフォームポータビリティを実現する並列プログラミング環境の実現を，従来の逐次の最適化コンパイラとOpen Compilerの組合せで試みた．プログラム中に明示的に記述する必要のあった並列実行環境向けの機能拡張のコードをメタコードとして分離し，Open Compilerによって機能拡張させることで，プラットフォームの違いをメタコードで吸収できる．

本稿では，その予備段階として，C++言語のOpen Compilerである OpenC++2.5[1),2)] のリフレクション機能を用いて，SPMDスタイルで書かれたC++の並列プログラムに対して，OpenC++のCompile-time MOPを用いて元のプログラムをプログラム変換することにより，共有分散メモリ機能を付加する並列言語処理系を実現した．なお，ベースレベルの実行環境としてRWCPで開発されたMPC++[3)]およびSCoreを利用している．このシステムはOpenC++，MPC++，SCoreといずれも汎用性・可搬性に優れた要素によって構成されおり，異なるプラットフォームにおいても比較的容易に実現可能であると考えられる☆．

本稿ではまた，このシステムをMyrinetで接続された8台構成のPentium Proクラスタ上で性能評価を行い，リフレクションを用いた言語拡張，特に分散共有メモリ実現の有効性を確認した．

☆ 本稿ではPentiumProクラスタでの実現に留まっているが，WSクラスタでの実現も現在試みている．

2. 可搬かつ高効率なソフトウェアDSMの実現

分散共有メモリは，Kai Li[4)]以来様々なアプローチで実現されてきた．しかし，我々が目指すプラットフォームポータビリティを満たすためには，既にコモディティとして利用されているハードウェア，OS，プログラミング言語等の構成要素を大きく変更・拡張しないことが求められる．また，実用的なシステムとして機能するためには，このような前提と高性能を両立する必要がある．

従って，本稿で提案するソフトウェアDSMは以下の点に留意して設計された:

(1) システムの構成要素としてコモディティを考慮したハードウェア，ソフトウェアを用いて移植性・可搬性を高める．

(2) 既存のC++言語で書かれた共有メモリ型マルチスレッドプログラムを対象とし，言語の文法・意味を拡張しない．

(3) Open Compilerのリフレクション機能を用いて高性能な実装，特に実行時のオーバーヘッドを抑える．

以下では，本システムの概要を説明するとともに，用いたポータブルな構成要素について紹介する．

2.1 システムの概要

本システムは，既存のC++言語で書かれた共有メモリ型マルチスレッドスタイルのプログラムに対して，プログラム変換を行い，DSM機能をサポートした実行可能なプログラムを生成する（**図1**）．

具体的には，DSM機能を提供するテンプレートクラス（分散共有配列クラス），およびソースプログラムに対するプログラム変換（共有領域へのアクセスのチェック，DSMの初期化など）を行うOpenC++のメタクラスを用意する．さらに，これらを用いて，OpenC++メタクラスから生成されたプログラム変換器によって，ソースプログラムを変換し，DSMテンプレートライブラリ，およびMPC++実行時ライブラリのリンクを行うことで実行プログラムを得る．

本システムの方法は単にクラスライブラリとして実現するのに比較して効率の点で優れる．本システムでは，共有領域へのアクセスのチェック等，実行時のオーバーヘッドとなる処理を挿入するプログラム中の箇所が静的に判断できるため，必要なところにだけこの種の処理を挿入できる．クラスライブラリのみでの実現では，プログラマの判断でこれらの処理を挿入するか，あるいは実行時に冗長な検査を行う必要があるが，メ

タ計算が必要ないため，本システムに比べてコンパイル時間の短縮は期待できる．

2.2 ポータブルな構成要素

本システムを構成する要素である OpenC++, MPC++, SCore はいずれも汎用性・可搬性に優れており，異なるプラットフォームにおいても比較的容易に実現可能であると考えられる．

2.2.1 OpenC++

OpenC++は C++言語のための Compile-time MOP を提供する．適切なメタクラスを記述し，それを OpenC++コンパイラによってプログラム変換を含む適切な処理を行うコンパイラを生成する．生成されたコンパイラを用いてベースレベルのプログラムの変換，コンパイルを行う．Compile-time MOP によりメタ計算の実行時オーバーヘッドがなくすことができる．本システムでは OpenC++ 2.5.3 を用いた．

2.2.2 MPC++

MPC++ 2.0 Level 0 は C++のテンプレートと継承の機能を使って高レベルな通信モデルのプログラミングをサポートしている，並列プログラミング言語である．本システムでは MPC++ 2.0 Level 0 の MTTL の機能であるローカル/リモートでのスレッド生成，グローバルポインタ，リダクションなどを用いる．また，リフレクション機能をサポートした Level 1[5] もあるが本システムでは用いない☆．

2.2.3 PC クラスタ

本システムの実装に用いた PC クラスタは RWCP で開発された PC Cluster II のサブセットである PentiumPro 8 台構成クラスタである．各ノードはギガビット級の転送速度をもつ Myrinet で接続されている．このクラスタ上では同じく RWCP で開発された Myrinet 用通信ライブラリ PM，クラスタ型並列計算機向け並列実行環境 SCore が動いている．SCore は既存の OS にデーモンとして動作するスケジューラの追加のみによって実現された並列 OS である．

3. Open ソフトウェア DSM の実装

Open ソフトウェア DSM でのコンパイルの処理の流れを以下に示す（図 1）．

- メタクラスよりプログラム変換を行うプリプロセッサが生成される (1).
- 生成されたコンパイラでソースプログラムを変換する (2). 変換・生成されたプログラムは MPC++ のプログラムとなる.
- MPC++で SCore ライブラリ等のコンパイル変数が付加され，バックエンドの g++へ渡されて，実行可能プログラムが生成される (3).

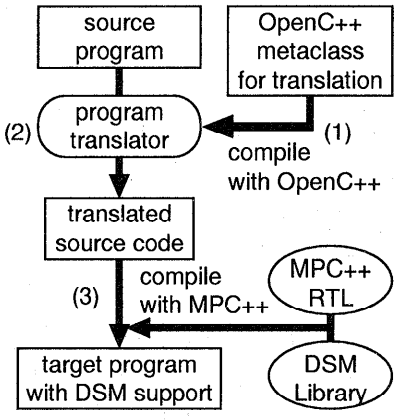


図 1 プログラム変換・コンパイルの流れ

```
tcsh> make cg
occ -- -g -o SharedClass SharedClass.mc |(1)
./SharedClass -v -- -g                  *
                  -o cg cg.cc problem.o |
[Preprocess... mpc++                    |(2)
  -D__opencxx -E -o cg.occ -x c++ cg.cc]|
compiling cg.cc                         |
[Translate... cg.occ into: cg.ii]       *
[Compile... mpc++                       *
              -g -o cg problem.o cg.ii]|(3)
compiling cg.ii                         *
[done.]
[GC happened 12 times.]
```

図 2 コンパイル経過

実際のコンパイルの実行の様子を図 2 に示す（数字は図 1 の番号に対応する）．

以下では，プログラム変換の手続きおよびそれを実現するメタクラス，ソフトウェア DSM クラスライブラリの実装について述べる．

3.1 プログラム変換

DSM の機能を付加するプログラム変換の過程には 3 つある．

(1) 各 PE 上に生成されるメモリ管理オブジェクト等の初期化，終了プロセスの追加.

(2) 共有領域の初期化（管理テーブルの作成，メモリの確保）.

(3) プログラム中の共有領域（変数）に対するロック等の挿入.

☆ Level 1 のメタ機能を今回用いなかったのはコンパイルが遅いなどの理由に因る.

```
Shared<double> a(100);     // ..(1)
Shared<double> b;
Shared<double> c;
sdsm_main(){
  c = (double*)malloc(sizeof(double) * 10);
  a[10] = b[1];  ... };  // ^..(3)
sdsm_init(){  a[0] = 10.0;
  b = (double*)malloc(sizeof(double) * 10); };
sdsm_done(){ ... };       // ^..(2)
```

図 3 変換前のプログラム

図3, 図4にプログラム変換の例を示す☆. OpenC++で記述可能なプログラム変換は，ベースクラスの変換，特にクラスのメソッド部分の置換に限られている．そこで，本システムでは次のような API を定めた．

```
sdsm_main(){ .. };   // 計算本体
sdsm_init(){ .. };   // ユーザーによる初期化
sdsm_done(){ .. };   // ユーザーによる終了処理
```

sdsm_init() は DSM 側の初期化処理が終了すると呼び出される．sdsm_main() 関数は並列に計算を行う本体であり，共有領域の初期化が終了すると各 PE 上で呼び出される．sdsm_done() は sdsm_main() が終了すると呼び出され，その後，DSM 側の終了処理が行われ，プログラムは終了する．sdsm_init() と sdsm_done() は 1PE のみで逐次に（通常 PE0 で）行われ，sdsm_main() は全 PE 上で行われる．

プログラム変換によって次の関数が追加される．

```
mpc_main(){ .. };        // メイン関数
InitShare(){ .. };       // 共有領域の初期化1
InitAll(){ .. };         // 共有領域の初期化2
FinalAll(){ .. };        // 共有領域の解放
```

プログラム変換によって生成されるプログラムは MPC++のコードであるので，プログラムは mpc_main() から開始される．このメイン関数 mpc_main() では InitShare() を各 PE 上で呼び出し，ユーザーによる初期化関数である sdsm_init() を呼び，InitAll() を各 PE 上で呼び，計算本体である sdsm_main() を呼ぶ．計算が終了するとユーザーによる終了関数 sdsm_done() を呼び，最後に FinalAll() を各 PE 上で実行し，プログラムを終了する．

3.1.1 初期化，終了プロセス

初期化プロセスではメモリ管理オブジェクト，分散共有オブジェクトの初期化を行う．実際には排他制御を行う時のロック（Sync オブジェクト）の初期化（初期状態として “解放” にする）のみである．

☆ なお，現時点では OpenC++のメタクラスでテンプレートを扱う機能がまだ十分ではないので，実際には OpenC++でコンパイルする前に手でテンプレートの展開を行っておく必要がある．

```
Shared<double> a(100);
Shared<double> b;
Shared<double> c;
sdsm_main(){ c.allocate(10);       // ..(3)
           { double _sym52501_8 = b[1];//(6)
             a.WriteStart(10);     // ..(4)
             a[10] = _sym52501_8;
             a.WriteEnd(10); } }; // ..(4)
sdsm_init(){{a.WriteStart(0);      // ..(5)
             a[0] = 10.0;
             a.WriteEnd(0); }... };// ..(5)
sdsm_done(){ ... };
InitShare(){ a.allocate(100); };  // ..(1)
InitAll(){   b.allocate(10); };   // ..(2)
FinalAll(){  a.free();
             b.free();
             c.free(); };
mpc_main(){
  invoke InitShare() on all PEs.
  sdsm_init();
  invoke InitAll() on all PEs.
  invoke sdsm_main() on all PEs.
  sdsm_done();
  invoke FinalAll() on all PEs. };
```

図 4 変換後のプログラム

終了プロセスではメモリ管理オブジェクト，分散共有オブジェクトの解放を行う．分散共有オブジェクトの管理テーブル，メモリ領域の解放を行う．

3.1.2 共有領域の初期化

共有領域の初期化には 3 つのタイプがある．それは領域のサイズが決まる場所によって分かれる．まず 1 つは共有領域（変数）宣言時にサイズを指定した場合の初期化は InitShare() で行われる．具体的には図 3-(1) から図 4-(1) へと変換される．次に，宣言時にはサイズを指定せずに，ユーザーによる初期化関数 sdsm_init() 内で領域の割り当て（malloc）が行われた場合は 2 つめの共有領域の初期化関数 InitAll で初期化が行われる．つまり，図 3-(2) から図 4-(2) へと変換される．最後は宣言時にサイズを指定せず，計算本体 sdsm_main() 内で領域の割り当てが行われた場合はその場で図 3-(3) から図 4-(3) のように初期化コードに変換される．

3.1.3 共有領域へのアクセス

共有領域へのアクセスは書き込み（store）時のみ Write lock, unlock を行う文が挿入される（図 4-(4),(5)）．代入文の右辺に共有変数が現れる場合は二重のロックを防ぐため，一時変数に代入する（図 4-(6)）．読み込み（load）アクセスでは不要である．

3.2 プログラム変換用メタクラス

プログラム変換を記述するメタクラスについて述べる．OpenC++ではメタクラスをクラス Class のサブ

クラスとして記述し，そのメソッドをオーバーライドすることにより，プログラムの挙動を変えることができる．本システムでオーバーライドするメソッドは次の3つである．

TranslateInitializer クラスオブジェクトが生成される時に呼び出される．これにより分散共有配列オブジェクトの名前，サイズ等の情報を得ることができる．これらの情報は共有領域の初期化を行う時に使われる．

TranslateAssign クラスオブジェクトへの代入が行われる時に呼び出される．これにより分散共有配列オブジェクトのプログラム中の初期化，書き込みアクセスを変換できる．実際にはメソッドの引数として渡される式を解析し，malloc 文であれば TranslateInitializer と同じように情報を取得し，さらに `sdsm_main()` 内であれば，その場で共有領域の初期化コードに置き換えられる．また，代入文であればその前後に WriteStart() と WriteEnd() のメソッドを挿入する．

FinalizeInstance 変換が終了する時に呼び出される．ここで前に述べた初期化や終了処理，共有領域の初期化のための関数が挿入される．具体的には TranslateInitializer と TranslateAssign によって得られた情報から InitShare(), InitAll(), FinalAll() を作成し，メイン関数 mpc_main() をソースプログラムへ追加する．

`TranslateAssign` のメタプログラムの一部を**図5**に示す．ここでは write アクセスに対するロックをその文の前後に挿入している．メタプログラムは全体で約250行ほどと非常にコンパクトである．

```
Ptree* SharedClass::
 TranslateAssign(Environment* env,
   Ptree* obj, Ptree* op, Ptree* exp){
 Ptree* exp0 = ReadExpressionAnalysis(env, exp);
 Ptree* obj0 = WriteExpressionAnalysis(env,obj);
 return Ptree::List(obj0, op, exp0);  };
Ptree* SharedClass::
 WriteExpressionAnalysis(Environment* env,
                         Ptree* exp){
 Ptree *obj = exp->First();
 Ptree *index = exp->Nth(2);
 if (!index->IsLeaf())
   index = ReadExpressionAnalysis(env, index);
 InsertBeforeStatement(env,
  Ptree::qMake("`obj`.WriteStart(`index`);\n"));
 AppendAfterStatement(env,
  Ptree::qMake("\n`obj`.WriteEnd(`index`);\n"));
 return Ptree::qMake("`obj`[`index`]"); }
```

図5 メタプログラム

表1 共有配列の管理テーブル

addr	実際のメモリへのポインタ (4)
copyowner	コピーを持つ PE のリストへのポインタ (4)
owner	ブロックの所有者 (2)
copyowers	コピーを持つ PE の数 (2)
havedata	データを持っているかどうかの flag(1)
lock	ロック (1)
dummy	テーブルのサイズ調整 (2)
	() 内は各フィールドのバイト数

3.3 分散共有メモリ

本システムでは分散共有メモリの実装の手段として，テンプレート機能を用いた分散共有配列クラスを用いる．従来のページベースの実装に比べてソフトウェアチェックが必要なため性能が多少低下する可能性がある．しかし，Shasta[6] に見られるようにキャッシュブロックサイズを任意に設定できるなどの利点があり，効率の良い実装が可能である．また，柔軟性・記述性の面で有利であり，可搬性を求める上でも有効である．以下で分散共有配列クラスの実装と高速化について述べる．

3.3.1 分散共有配列クラス

分散共有配列クラスは MPC++で書かれたプログラムである．このクラスは配列を分散共有するものであり，このオブジェクトは全ての PE 上に生成される．メモリ一貫性のプロトコルは Write-Invalidate スタイルで sequential consistency である．LRC などの weak consistency 系のアルゴリズムは用いていない．配列はメモリ転送の単位であるブロックに分割されて管理される．このブロックの大きさは BlockSize によって指定でき，現時点では 2^n の大きさで可変である．ブロックの管理テーブルは全てこのクラス内に存在する．管理テーブルの構造は**表1**に示す通りであり，1テーブルあたりのサイズは 16bytes である．これはアクセス時のアドレス計算を高速化するためである．領域確保のメソッド `allocate(size)` が呼ばれると（size/BlockSize）分の管理テーブルが作成され，各 PE に（size/PE 数）要素分のメモリが確保される．

3.3.1.1 Read アクセス処理

要素へのアクセスは配列演算子 [] を用いるので，分散共有配列クラス内で [] 演算子のオーバーロードを行う．read アクセス処理は次のように行われる．

(1) 管理テーブルの `havedata` flag が立っていれば，そのアドレスを返す．

(2) `havedata` flag が立っていない場合，必要なメモリ領域を確保してそのブロックの所有者に確保した領域へのグローバルポインタを渡してコピーを要求する．

(3) 要求を受けたブロックの所有者はその領域をグローバルポインタで指定された領域へ書き込む.
(4) コピーを獲得できたら，そのアドレスを返す.

3.3.1.2 Write アクセス処理

書き込みアクセスもまた配列演算子 [] を用いるが，プログラム中でその前後で Write lock，unlock を行うメソッド WriteStart()，WriteEnd() を呼んでいる．write アクセスの処理は次のように行われる．

(1) WriteStart() では管理テーブルの `lock flag` を立てる.
(2) そのブロックの所有者であり，かつコピーを持つ PE があれば invalidation を発行する.
(3) そのブロックのデータを持っていなければ所有者へ（read 時と同様）コピーを要求する.
(4) ブロックの所有者でなければ，所有者 PE より所有権を得る．この時，前の所有者は他の PE に対して所有者が変わったことを伝える.
(5) 演算子 [] では `havedataflag` をチェックし，そのアドレスを返す.
(6) WriteEnd() では `lock flag` を降ろす.

3.3.2 高 速 化

分散共有メモリの実装で最も問題となるのはオーバーヘッドである．そのオーバーヘッドの原因の多くは invalidation によるものではなく，最も回数の多い read, write によるアドレス変換とチェックである．そこでテーブル検索，アドレス変換において乗算，除算をシフト演算に置き換えることができるように，テーブルサイズ，配列 1 要素のサイズを 2^n へと調整している．また，配列演算子 [] 及び，WriteStart()，WriteEnd() などの手続きはすべて inline 化することにより，高速化が図られている．さらに，前回のアクセスで計算したブロック・アドレス情報をキャッシュする．このため，連続して同一ブロックにアクセスした時にはキャッシュした情報を用いることで，read，write の連続アクセスが高速化される.

4. 性 能 評 価

前章で説明したシステムの有効性を確認するため，PC クラスタ上で性能評価実験を行った．実験では read，write の基本性能の測定の他，アプリケーションとして CG カーネル，SPLASH2[7] の FFT カーネルの評価を行った.

4.1 評 価 環 境

評価環境として，RWCP で開発された PC Cluster II のサブセットである，PentiumPro 200MHz クラスタ 8 台構成を用いた．各ノードのメモリは 64MB

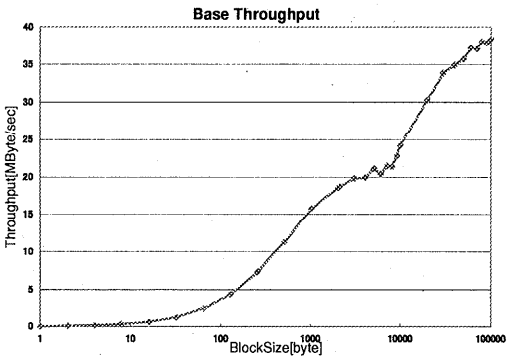


図 6 基本リモートメモリオペレーションのスループット

で Myrinet（LinkSpeed 160MB/s）で接続されている．このシステムは OS が NetBSD 1.2.1，並列実行環境として SCore，並列開発言語として MPC++ Version 2.0 Level 0 によって構成されている．コンパイラは gcc 2.7.2 を用い，最適化のオプションは "-O6 -fomit-frame-pointer -funroll-loops -fstrength-reduce - ffast-math -fexpensive-optimizations" とした．Sun CC4.2 での最適化のコンパイルオプションは "-fast -xcg92 -xO5" である.

4.2 基 本 性 能

本システムの基本性能の評価として read，write のアクセスの時間を計測した．共有配列のアクセスパターンとして連続アクセス，ストライドアクセス，invalidation を伴う write アクセスの 3 パターンを行った.

4.2.1 連続 read，write アクセス

サイズ 1024×1024 の double の共有配列の read，write アクセスにかかる時間を計測した．メモリのアクセス時の状況は read 時，write 時共に全て他の PE が持っており，アクセス時にはまずそのブロックのコピーを取ってくることから始まる．また，write 時にはのメモリのコピーを持つ PE はないため，write のアクセス時間には invalidation の時間等は含まれず，純粋に書き込みの時間となっている．連続したアクセスの 1 回あたりの平均時間を BlockSize を変えて計った時の結果を図 **7** に示す.

read，write とも BlockSize の増加に伴ってアクセス時間が減少する．これはメモリ転送の回数が少なくなり，メッセージハンドリングやロックなどの処理のオーバーヘッドが小さくなるためである．また，BlockSize が一定以上になると，ネットワークの転送速度の限界に近付いて，頭打ちになっている．BlockSize 64kbytes 時では 0.34μsec/1dword であり，これは 22.5Mbytes/sec に相当する．生の MPC++ の RemoteMemoryRead での転送速度は 36Mbytes/sec 弱

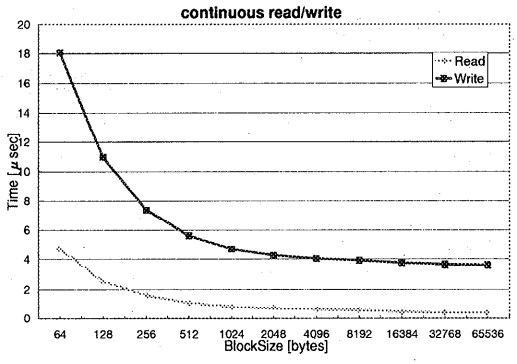


図 7 連続 read, write アクセスのコスト

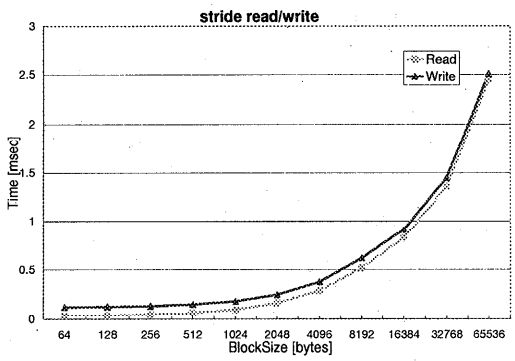


図 8 ストライド付き read, write アクセスのコスト

である（図 6）．この差異は read アクセス時のアドレス変換とチェックに時間を要しているものと考えられる．

4.2.2 ストライド read, write アクセス

ストライドアクセスでは，共有配列をストライド幅 BlockSize で順に read, write アクセスを行う．サイズ及びアクセス時のメモリの状況は連続アクセスの場合と同じであるが，BlockSize に伴ってストライドが変化するので総メモリアクセス量は BlockSize によって異なる．この方法で read, write アクセスを行った時の 1 回あたりの平均時間を同様に BlockSize を変えて計測した結果を図 8 に示す．

read, write とも BlockSize の増加に伴ってアクセス時間が増加する．これは BlockSize が大きいほど 1 回のアクセスに伴うメモリ転送量が大きくなるためである．また，BlockSize の増加に伴って read アクセス時間と write アクセス時間の差が小さくなる．これは read, write アクセス時間のうち，メモリ転送の占める割合が大きくなった結果，アドレス変換やロックなど他の処理が相対的に無視できるほど小さくなったためである．

表 2 invalidation を伴う write アクセスのコスト
（BlockSize=1kbytes,16kbytes）

共有数 (PE 数)	時間（μsec） BlockSize 1kbytes	BlockSize 16kbytes
0	4.80	5.26
1	31.27	33.05
2	46.06	48.27
6	76.68	78.24

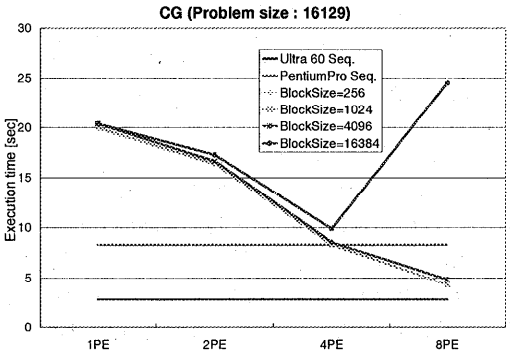


図 9 問題サイズ=16129 での CG カーネルの結果

4.2.3 invalidation を伴う write アクセス

ブロックの所有者が自分であり，さらに複数の PE がそのブロックを持つ場合，書き込み時に他のコピーを持っている PE に対して invalidation が発行される．ブロックのコピーの共有数の増加に伴い，write アクセスに要する時間が増える．そこである PE が全ての領域の所有者である時コピーを持つ PE を 0,1,2,6 個，BlockSize を 1kbytes，16kbytes と変化させて，invalidation のコストを計測したものが表 2 である．

共有数がいくつであっても，invalidation は非同期に発行されるので，共有数が多いほど効率が良い．また，アクセス時間は BlockSize にほとんど依らない．これは，invalidation が起きる場合にはメモリ転送を伴わないので，BlockSize に依存したオーバーヘッドはないためである．

4.3 CG

並列数値アプリケーションの例として前処理なし CG カーネルでの性能評価を示す．ここでは問題サイズ，PE 数を変えて所要時間を測定した．また同等の逐次実行のプログラムを PC クラスタ，Sun Ultra60（UltraSPARC II 300MHz，メモリ 256Mbytes，Sun CC4.2）上で計測した．問題サイズは 16129（反復回数 197 回），261121（反復回数 806 回）の 2 つ，BlockSize は 256，1k，4k，16kbytes，PE 数は 1, 2, 4, 8 で計測した．これらの結果は図 9, 図 10 の通りである．

図 9 の結果より，2PE では 1PE に対して 20%ほど

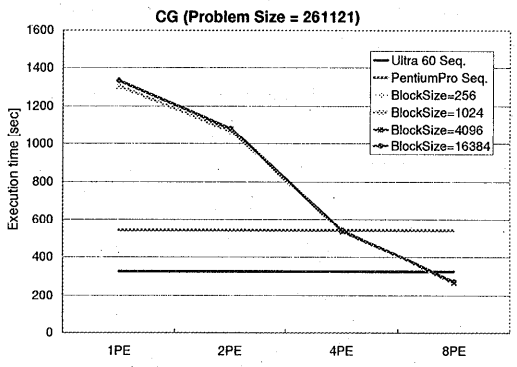


図 10 問題サイズ=261121 での CG カーネルの結果

しか性能が向上していないが，2PE から 4PE，4PE から 8PE ではほぼ倍の性能向上がみられる．これは 1PE だけの場合には転送や Write 時の invalidate が行われないため，例外的に速くなっており，2PE の場合には並列化の効果とこれらの処理のオーバーヘッドが相殺するためである．また，8PE で 4.0〜4.9 倍の台数効果が見られる．8PE 時でも Sun Ultra60 での逐次性能には及ばないが，クラスタ PC（PentiumPro）での逐次実行に比べて約 2 倍の性能が出ている．また，BlockSize が 16kbytes で 8PE で実行した場合に，性能が低下しているのが分かる．これは計算時に実際に必要なデータの量より BlockSize が大きいために無駄なメモリ転送が生じることによるオーバーヘッドである．この現象は問題サイズが大きい場合（図 10）には起きていない．

図 10 で問題サイズが大きくなっても全体的な傾向はほぼ同様と言ってよい．ここでも 8PE で 4.6〜5.0 倍の台数効果が見られる．主な特徴は問題サイズが小さい場合と異なり，BlockSize を変えてもあまり差がないことである．これは計算時に実際に必要なデータの量が BlockSize より十分大きいために，BlockSize を大きくしても無駄な転送が生じず，また計算時間全体からみると，BlockSize が小さい時に顕在化しがちなメッセージハンドリングのオーバーヘッドが無視できるほど小さいためである．

同じプログラムのマルチスレッド版を Enterprise4000（UltraSPARC 10CPU）で実行すると 8 台で 6 倍強と，ほぼ同等の性能が得られた．逐次版に比べて台数効果が低いのはバリア同期などの理由による．

4.4 FFT

最後に，分散共有メモリシステムの性能評価に広く用いられている SPLASH2[7] のうち，FFT カーネルによる性能評価を示す（**図 11**）．問題サイズは 64Kpoints とし，BlockSize を変えて測定を行った．

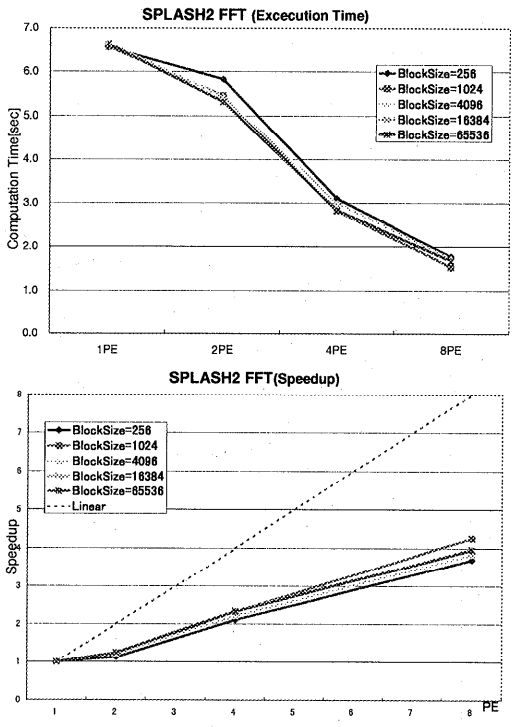


図 11 SPLASH2 FFT（問題サイズ 64K points）

グラフから性能は CG カーネルの場合と同様，BlockSize にはとんど依らず，速度向上率の傾向も似ている．また，ソースコードの変更は 1008 行中 70 行程度であり，比較的小さく抑えられている．

5. 関連研究

プログラム変換をして DSM 機能を付加するものとして Shasta[6] がある．Shasta コンパイラは SPMD スタイルのプログラムをプログラム変換することで DSM 機能を支援する．付加される DSM 機能では miss check のオーバーヘッドを 2 命令程度に抑えており，厳密な比較は難しいが，本システムより高速であると思われる．

その他の DSM の実装として，オブジェクトベースの Midway[8], CRL[9] がある．Midway ではメモリの一貫性プロトコルとして entry consistency を用い，miss check などのコードはコンパイラによって付加される．CRL では miss check のコードなどはユーザーが明示的に書く必要がある．つまり，Midway ではコードの挿入がコンパイラによって自動的に行われてしまい，ユーザーが適切な形で挿入できない．CRL では逆に全てユーザが記述するためプログラムが繁雑になりやすい．本システムではコード生成は自動的に行われるが，ユーザーがカスタマイズしたい場合はメ

タクラスの記述を変更するだけで対応できるため，これらに比べて柔軟性が高いと言える．

また，共有メモリ型並列実行環境で汎用性を持たせたものとしてOpenMP[10]が標準化されつつあり，現在のところFortran bindingのみが決まっている．これはユーザーがannotationを記述することにより，コンパイラが適切なコードを生成する．しかし，処理系自体の高効率の実装や実行時環境に依らない高い性能を達成するコードの記述が難しいなどの問題点がある．

ポータビリティの点からDSMの実装をみると，

- 従来のページベースのようなDSMシステムはOSになんらかの手を加える必要があるため，OS依存となってしまう．
- 上で述べたShastaのようなリンク時にmiss checkなどを入れるシステムではプログラム（binary）・環境依存である．
- クラスライブラリのみを用いてポータブルに実現することも可能であるが，本研究のアプローチのようにメタ計算と組み合わせて適切な箇所に適切なロックを挿入することができず，効率が犠牲となる．
- マクロなどのプリプロセッサのみによる実装はポータビリティの面では良いが実装が難しく，フルセットの言語の実装やサポートが困難である．

本システムで述べたような自己反映的な方法は，プリプロセッサで行うよりはるかに容易であり，ポータビリティも十分確保できる．また，クラスライブラリのみを用いるより効率が良い．さら，DSMなどにおいてプロトコルを変更する場合，メタクラスを書き換えるだけで済むため保守性も高い．

6. まとめと今後の課題

我々は様々な並列実行環境上でプラットフォームポータビリティを実現する並列プログラミング環境で実現を，従来の逐次の最適化コンパイラとOpen Compilerの組合せで試みた．

本稿では，その予備段階として，C++言語のOpen CompilerであるOpenC++2.5のリフレクション機能を用いて，OpenC++のCompile-time MOPを用いて元のプログラムをプログラム変換することにより，SPMDスタイルで書かれたC++の並列プログラムに対して，共有分散メモリ機能を付加する並列言語処理系を実現した．

また，このシステムをMyrinetで接続された8台構成のPentium Proクラスタ上で性能評価を行い，リフレクションを用いた言語拡張，特に分散共有メモリ実現の有効性を確認した．

今後の課題としては，まずは実装の完成度を上げることである．例えば，MPC++のRemoteMemoryReadでのスループットは最大38MByte/secであるが，本システムでは最大28MByte/secである．これはアドレス変換やテーブルチェックのオーバーヘッドによるものであり，さらにチューニングする必要がある．

DSMの実装である分散共有配列クラスでは配列の要素としてクラスオブジェクトなどを扱うことが原理的には可能である．ポインタに代入されているアドレス空間が共有空間だけでなくローカルメモリである場合もあり得るため，そのチェックやローカルアドレスへのポインタであった場合の対処などを実装する必要がある．

また，OpenC++のテンプレートのサポートが十分でないため，システムの手続きの一部が手動になっている．この点についてはOpenC++に然るべき対処を行うことで対応可能であると考える．

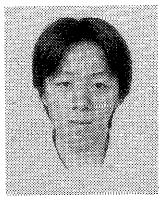
今回は性能評価指標として，CGカーネルとSPLASH2のFFTを用いたが，FFT以外のSPLASH2による性能評価も今後の課題である．

参考文献

1) Chiba, S.: A Metaobject Protocol for C++, *Proceedings of OOPSLA'95*, pp. 285–299 (1995).
2) Chiba, S.: *OpenC++ 2.5 Reference Mannual* (1997).
3) Ishikawa, Y.: Multiple Threads Template Library – MPC++ Version2.0 Level 0 Document –, TR 96012, RWCP (1996).
4) Li, K. and Hudak, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems*, Vol.7, No.4, pp. 321–359 (1989).
5) Ishikawa, Y. and et.al.: Design and Implementation of Metalevel Architecture in C++ – MPC++ Approach –, *Proceedings of Reflection'96* (1996).
6) Scales, D. J., Gharachorloo, K. and Thekkath, C. A.: Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *Proceedings of ASPLOS VII* (1996).
7) Woo, S. C., Ohara, M., Torrie, E., Singh, J. P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24–36 (1995).
8) Bershad, B.N., Zekauskas, M.J. and Sawdown,

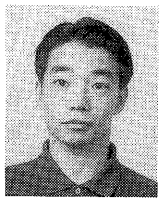
W. A.: The Midway Distributed Shared Memory System, *Proceedings of the IEEE CompCon Conference* (1993).
9) Johnson, K. L., Kaashoek, M. F. and Wallach, D. A.: CRL: High-Performance All-Software Distributed Shared Memory, *Proceedings of the Fifteenth Symosium on Operating Systems Principles* (1995).
10) OpenMP Architecture Review Board: *OpenMP: A Proposed Industry Standard API for Shared Memory Programming.* (1997).

(平成10年 7 月24日受付)
(平成10年10月10日採録)



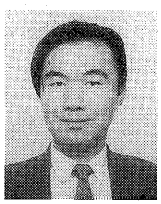
早田 恭彦

昭和49年生．平成10年東京工業大学理学部情報科学科卒業．現在，同大学大学院情報理工学研究科数理・計算科学専攻修士課程在学中．並列・分散システム，オブジェクト指向言語，広域分散システムなどに興味を持つ．



小川 宏高（正会員）

昭和46年生．平成6年東京大学工学部計数工学科卒業．平成8年同大学大学院工学系研究科情報工学専攻修了．平成10年同博士課程中退．現在，東京工業大学大学院情報理工学研究科数理・計算科学専攻助手．プログラミング言語，言語処理系，オブジェクト指向技術，並列計算機アーキテクチャ，広域分散システムに興味を持つ．ACM会員．



松岡 聡（正会員）

昭和38年生．昭和61年東京大学理学部情報科学科卒業，平成元年同大学大学院博士課程中退．同大学情報科学科助手，情報工学専攻講師を経て，平成8年より東京工業大学情報理工学研究科数理・計算科学専攻助教授．理学博士．オブジェクト指向言語，並列システム，リフレクティブ言語，制約言語，ユーザ・インターフェースソフトウェアなどの研究に従事．現在進行中の代表的プロジェクトは，世界規模の高性能計算環境を構築するNinfプロジェクト，計算環境に適合・最適化を目指すJava言語の開放型Just-In-Timeコンパイラ OpenJIT，制約ベースのTRIPユーザインターフェースなど．並列自己反映型オブジェクト指向言語ABCL/R2の研究で1996年度情報処理学会論文賞受賞．1997年はオブジェクト指向の国際学会ECOOP'97のプログラム委員長を努める．ソフトウェア科学会，ACM，IEEE-CS各会員．