計算機アーキテクチャ 132−19
システムソフトウェアとオペレーティング・システム 80−19
ハイパフォーマンスコンピューティング 75−19
（1999. 3. 5）

高可搬な自己反映計算によるソフトウエア分散共有メモリの性能評価

八木澤 直哉† 小川 宏高†
早田 恭彦† 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自己反映計算を用いることである．我々は，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，分散共有メモリ機能を付加する並列言語処理系OMPC++を実装した．本稿ではSPLASH2を用いて本システムの評価を行い，自己反映計算によるプログラム変換を用いたソフトウェアDSMが可搬性・保守性に優れており，かつ高性能DSMに匹敵する性能を達成し得ることを示す．

Evaluation of Portable Software DSM employing Reflection

Naoya Yagisawa,† Hirotaka Ogawa,† Yukihiko Sohda† 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. To characterize the effect of our system, we perform SPLASH2 on a PC cluster linked by the Myrinet gigabit network, and resulted in resonable performance compared to a high-performance SMP. We also indicate that it can achieve comparable performance to low-overhead DSMs, such as Shasta.

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だが，ソフトウェアの構成技術はその発展に追従できていない．特にBeowulfシステムに代表される安価な分散メモリ並列計算機は，基本的にコモディティ技術であるハードウェア，OS，プログラミング言語等を用いて構成されるため，これらの変更の都度必要になるプラットフォームポータビリティ(保守性・可搬性・高性能)の維持が問題になる．

我々は，このプラットフォームポータビリティの実現のために，自己反映計算を用いる手法を重視している．自己反映計算では，自己の計算系を表すメタレベルコードを記述できるため，実行時のベースレベルコードの挙動を適切に変更できる．特にOpen Compilerの持つCompile-time MOPの機能を用いると，ユーザが記述したメタクラスによるコンパイラの挙動の変更がコンパイル時に反映されるため，プログラム変換が静的に行われ，実行時のオーバーヘッドを伴わない．また，プログラム変換時のコストが比較的低いため，JITコンパイラへの組み込みの可能性もある．

我々は，SPMDスタイルで書かれたC++プログラムに対して，ソフトウェアDSM機能を付加する並列言語処理系OMPC++[1]を実装した．OMPC++はC++用Open CompilerのOpenC++[2]を用いて，静的なプログラム変換によってプログラム中の必要な箇所に自動的にDSM用コードを挿入するため，実行時のオーバーヘッドが小さく，可搬性，保守性も高いと期待される．

本稿は，並列言語処理系OMPC++をSPLASH2ベ

† 東京工業大学 理学部 情報科学科 / 数理・計算科学専攻
Tokyo Institute of Technology

ンチマークを用いて評価する．さらに，現状での実装の問題点を検討するとともに，オーバーヘッドとなる部分の改良によって，高性能ソフトウェアDSMとして知られるShastaに匹敵する可能性について述べる．

本論文の構成は以下の通りである．第2章ではOMPC++システムの概要の説明を行い，3章ではSPLASH2のFFT，LUを用いて本システムの評価・考察を行う．4章では関連研究を紹介し，5章でまとめと今後の課題について述べる．

2. OMPC++ システム

分散共有メモリ (以下DSMと呼ぶ) は，Kai Li[3]以来，様々な手法で実現されてきた．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・拡張せず，かつ高性能を実現する必要がある．

この要求を満たすため，次の手法を用いてソフトウェアDSMを実現するシステムを構築した[1]．

(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の自己反映計算機能を用いて，ソースプログラムのコンパイル時のプログラム変換により，高性能の達成を図る．

(4) プログラム変換では，共有領域へのアクセスチェックなどの処理を挿入すべき箇所を静的に判断して，必要なところにのみ挿入する．

以下では，OMPC++システムの概要を述べる．

2.1 システムの概要

このシステムは既存のC，C++で記述された共有メモリ型マルチスレッドプログラムに対して，プログラム変換を行い，DSMの機能を付加した実行可能なプログラムを生成する．

具体的には次の手順で行われる (図1)．

- DSM機能を付加するプログラム変換を定義したOpenC++メタクラスからプログラム変換器を生成する (1)．
- 生成されたプログラム変換器でソースプログラムを変換する (2)．
- 変換されたソースプログラムをコンパイルし，DSMテンプレートクラス (分散共有配列クラス) ライブラリ，MPC++実行時ライブラリとリンクして実行プログラムを得る (3)．

ベースレベルの実行環境としては，RWCPで開発された並列プログラミング言語MPC++[4]，クラスタ型並列計算機向け並列実効環境SCore，さらにOpen CompilerとしてOpenC++を用いている．特に我々は，MPC++処理系の汎用のMPIライブラリへの移植も完成させており，OMPC++自身も異なるプラットフォーム上に実現することは容易である．

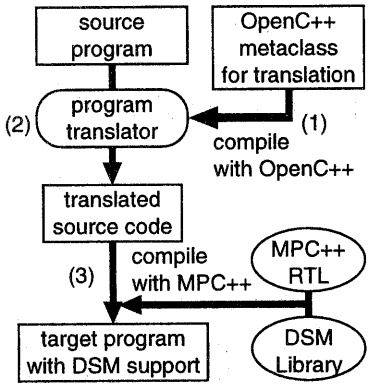

図1 OMPC++システムの概要

2.2 プログラム変換

生成されたプログラム変換器では以下の変換を行う．

- 各プロセッサ (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．
- 共有領域の初期化部 (管理テーブルの作成，メモリの確保) の追加．
- 代入文の両辺に共有領域へのアクセスがある場合，一時変数を利用するように変換 (簡易SSA変換)．
- プログラム中の共有領域へのwriteアクセスに対するロック (write lock) の追加．具体的には代入文の前後にWriteStart()，WriteEnd()を挿入．

プログラムは変換によりDSMの機能を持つMPC++言語で記述されたプログラムとなる．プログラム変換ではOpenC++のCompile-time MOPを利用するため，実行時のメタ計算によるオーバーヘッドはない．

2.3 分散共有配列クラス

DSMを実現する分散共有配列クラスは，C++のテンプレート機能を用い，MPC++で記述されている☆．このため，柔軟性・記述性の面で有利であり，可搬性も満たすことができる．メモリの一貫性プロトコルとして，Write-Invalidateスタイルのsequentialconsistencyを採用している．

配列はメモリ転送の単位であるブロックに分割して管理され，そのブロック毎に管理テーブルが用意される．この管理テーブルには，そのブロックの所有者や実メモリアドレス，メモリ書き込み時のロックの情報などが格納される．read,writeアクセス時にこのテーブルを参照し，メモリへアクセスする．なお，要素へのアクセスはオーバーロードされた配列演算子[]を用いる．

次に，read,writeアクセス時の処理を示す．

readアクセス

- 管理テーブルをチェックしその領域を持っていれば，そのアドレスを返す．
- その領域を持っていない場合，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．

☆ 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十分ではないので，実際にはコンパイル前に手でテンプレートを展開する必要がある

- 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．
- コピーを獲得できたら，そのアドレスを返す．

write アクセス

- WriteStart() では管理テーブルの書き込み中フラグを立てる．
- 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．
- そのブロックのデータを持っていなければ，所有者へコピーを要求する．
- 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．
- 演算子 [] では管理テーブルをチェックしてそのアドレスを返す．
- WriteEnd() では書き込み中フラグを降ろす．

write アクセスの操作のうち，WriteStart(), WriteEnd() はマルチスレッドプログラムで同時書き込みを防ぐために必要となるが，SPMD のプログラムにおいては不要である．以降の評価では，WriteStart(), WriteEnd() を省略したものを前提とする．

3. 評　　価

SPLASH2 の FFT 及び LU カーネルを PC クラスタおよび SMP マシン上に移植・実装し，OMPC++ システムの評価を行った．

3.1 評価環境

評価に用いた計算機環境は以下の通りである．

PC クラスタ　Pentium 166MHz，Cache 512KB，Memory 64MB，NetBSD 1.2.1 の 16 台構成．クラスタ間は Myrinet(転送スピード最大約 160MB/sec) で接続されている．また並列実行環境として SCore，バックエンドの並列言語として MPC++，MPC++ のバックエンドとして gcc 2.7.2 を用いる．

評価のためのプログラムは OMPC++ で実装したものを用いる．最適化のオプションは '-O6 -funroll-loops -fomit-frame-pointer -fstrength-reduce -ffast-math -fexpensive-optimizations' である．バリア同期には，Shuffle Exchange アルゴリズムを採用したものを用いている．

SMP マシン　Ultra Enterprise 10000，Ultra SPARC 250MHz×64，Memory 8GB，SunOS 5.5.1．

評価のためのプログラムの実装には SunCC 4.2，Solaris thread を用いた．最適化のオプションは '-mt -xO5 -fast -xcg92' とした．バリア同期には，バイトフラグを用いたソフトウェアコンバイニングツリーの拡張方式を用いた．

3.2 評価アプリケーション

評価に用いたアプリケーションは以下の通りである．

FFT　SPLASH2 の FFT．6-steps アルゴリズムの 1 次元 FFT，問題サイズ 64k points．

OMPC++ で動作させるためのソースコードへの変更は 1030 行中 70 行程度．

LU　SPLASH2 の LU(Contiguous)．問題サイズ 512 × 512，BlockSize(部分行列 1 個の大きさ)16 × 16．

OMPC++ で動作させるためのソースコードへの変更は 1030 行中 80 行程度．

アプリケーション	PC クラスタ (sec)	SMP (sec)
FFT	0.55	0.213
LU	4.30	1.941

表 1　各アプリケーションの逐次実行時間

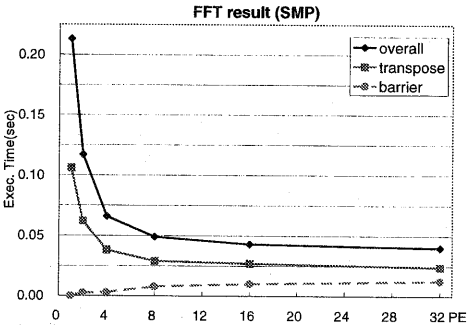

図 2　FFT(SMP) の実行時間 (単位:sec)

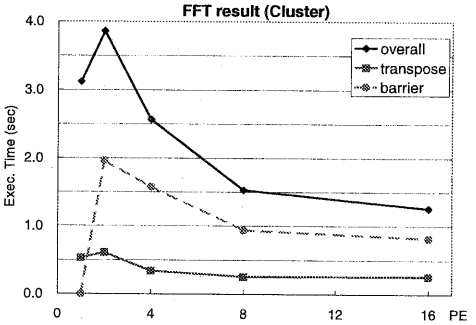

図 3　FFT(クラスタ) の実行時間 (単位:sec)

3.3 評価結果

3.1 で示した 2 つの環境で 3.2 で示した FFT および LU を評価した．

表 1 に FFT，LU の逐次実行時間を示す．この結果は共有メモリのオーバーヘッドを含まない．

FFT に関して，総実行時間 (overall)，その行列の転置に要した時間 (transpose)，バリアに要した時間 (barrier) を**図 2**(SMP)，**図 3**(クラスタ) にそれぞれ示す．LU に関して，総実行時間 (overall)，ブロック分割された部分行列 A_{kk} の計算結果を行列全体に反映させる時間 (interior)，バリアに要した時間 (barrier) を**図 4**(SMP)，**図 5**(クラスタ) にそれぞれ示す．また，1 ノードあたり複数スレッドを許すために，write lock を挿入するようにした場合の FFT および LU の結果を**図 6**, **図 7** に示す．

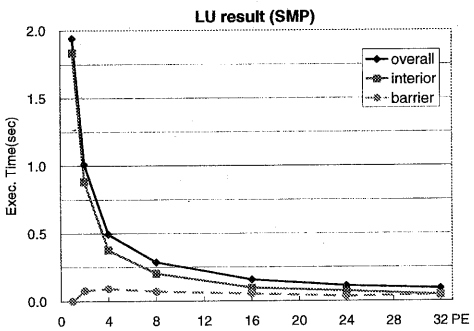

図4 LU(SMP) の実行時間 (単位:sec)

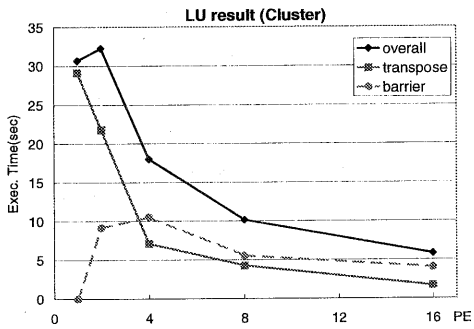

図5 LU(クラスタ) の実行時間 (単位:sec)

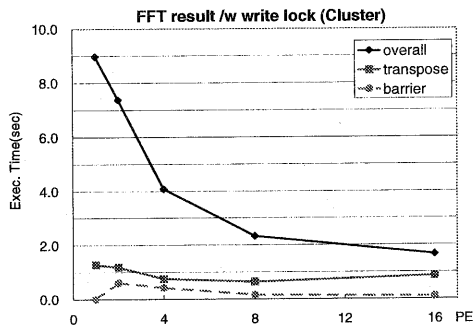

図6 write lock を行った FFT(クラスタ) の実行時間 (単位:sec)

3.4 考　　察

3.4.1 PC クラスタ

PC クラスタでは，FFT，LU とも 1PE から 2PE にかけては性能低下が見られるが，それ以降は性能向上が見られる (図 3, 図 5)．これは 1PE では転送や write 時の invalidate が行われないため、例外的に速く、2PE 以降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．LU では，FFT の場合と異なり，ほぼ台数効果と言ってよい速度向上を得ているが，

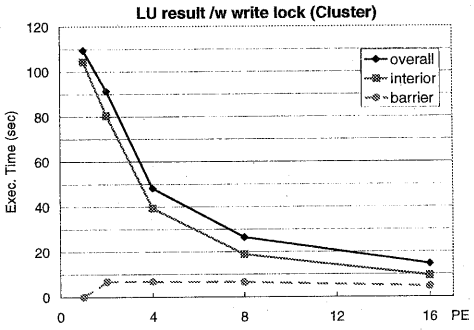

図7 write lock を行った LU(クラスタ) の実行時間 (単位:sec)

これは FFT に比べて LU の計算量が大きいために 16PE 程度ではまだ並列化の効果が顕著であるためである．

また，PE 数の増加とともに，バリアに要する時間が全体に占める割合が増加し，16PE では 65% (FFT), 69% (LU) に達してボトルネックとなっている．これに関しては 3.5 で述べる．

表 1 より，逐次と 1PE の場合を比較すると，FFT では 0.55 秒に対して 3.1 秒，FFT では 4.3 秒に対して 30 秒と，それぞれ 5.6 倍 (FFT)，7.1 倍 (LU) に速度が低下する．これは，最適化を施していない Shasta[5] の同条件での 2.2 倍 (0.43 秒 /0.20 秒) (FFT)，2.3 倍 (8.5 秒 /3.6 秒) (LU) と同等あるいは若干劣る結果である．

また，マルチスレッドに対応するために write lock を挿入した場合 (図 6, 図 7) には，FFT で 1.3 倍から 2.8 倍，LU で 2.5 倍から 3.5 倍と大きな性能低下が見られる．この性能低下を抑えるには，OMPC++ のプログラム変換時に静的解析を行って，安全に除去できる write lock を除去するなどの処理が必要になる．しかし，現状の OpenC++ にはそのような静的解析を支援する機能がないため，write lock 除去の実装は容易ではない．

3.4.2 SMP マシン

FFT では 8PE までは台数効果を示すが，それ以降の速度向上は小さい (図 2)．一方，LU では PE 数が増えてもほぼ台数効果と言ってよい速度向上が見られる (図 4)．これは，PC クラスタで述べたのと同様，FFT と LU の計算量の差に依るものである．

また，バリアに関しては PC クラスタと比較して，FFT，LU とも実行時間に占める割合が小さい．バリアの実現に関しては 3.5 で述べる．

本稿で評価に用いた PC クラスタの PE 性能は SMP マシンの PE 性能と比較して，浮動少数演算性能で 2 ～ 3 倍低い．PE 性能が高く，かつ PE 数が多い PC クラスタと比較すれば，PC クラスタでより高い性能を示す可能性がある．

3.5 バリア同期に関する補足評価

並列アプリケーションのボトルネックの要因としてバリア同期が挙げられるが，上記の結果では PC クラスタでの実行結果においてそれが特に顕著に現れた．ここでは，複数の手法でバリア同期を実現し，それらの比較を

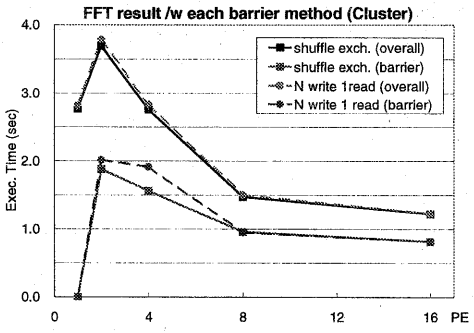

図 8　バリアの方式による FFT(クラスタ) の比較 (単位:sec)

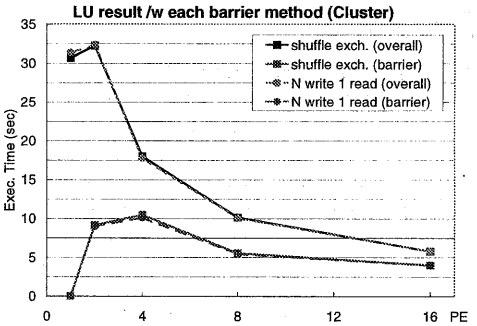

図 9　バリアの方式による LU(クラスタ) の比較 (単位:sec)

行うことで改善の余地を検討する.

3.5.1　分散メモリシステムのバリア同期

特殊な同期ハードウェアやブロードキャストネットワークを持たない分散メモリシステムのバリア同期は基本的にリダクション演算と同様であり，以下の 3 つの手法が広く知られている.

(1)　Master-worker 方式
(2)　Cascade 方式
(3)　Shuffle exchange 方式 (3.1 ~ 3.4 の評価で採用)

このうち，(1) は 1 つのノードが他の全てのノードから同期メッセージを受信し，全ての同期メッセージを確認したら，他の全てのノードにバリア解除メッセージを知らせる方式である．(2)(3) は 2 進木のパターンを重ね合わせたものであり，各ノードは下位の 2 ノードからの同期メッセージを受信すると上位のノードに同期メッセージを送信することで同期が取られる．(2) と (3) では通信相手の組合せが異なる.

(2)(3) では各ノードが $O(\log_2 n)$ 回の通信で済むのに対し，(1) では master ノードで $O(n)$ 回の通信が必要になるが，一概に (1) が不利とはならない．むしろ (2)(3) は $O(\log_2 n)$ 段のメッセージ通信が必要になってレイテンシが大きくなり，(1) は非同期メッセージ通信により効率的に行える場合が多い.

(1) と (3) の方式のバリアを用いて，FFT および LU を行った結果を図 **8**, 図 **9** に示す．この結果から，2 つの方式間の性能の差異はほとんどないことが分かる．従って，3.4.1 で見られた「PE 数増加に伴う，バリアに要する時間の全体に占める割合の増大」傾向はバリアのチューニングによって大きな改善を望めない.

3.5.2　共有メモリシステムのバリア同期

共有メモリシステムのバリア同期は，分散メモリシステムで挙げた 3 つの方式に，atomic increment の有無やバス競合の対策などを組み合わせた様々な方式が考案されてきたが，ここでは以下の 3 つの方式のみ挙げる.

(1)　Mutex lock を用いた集中カウンタ方式
(2)　N-write, 1-read の極性バリア方式
(3)　バイトフラグを用いたソフトウェアコンバイニングツリー方式[6] (3.1 ~ 3.4 の評価で採用)

(1) は，OS のスケジューリングを受けられる利点があるが低速であるのに対し，(2)(3) は OS のスケジューリングを受けられないが，バストラフィックが理論下限の $2n-2$ 付近になり，高速である.

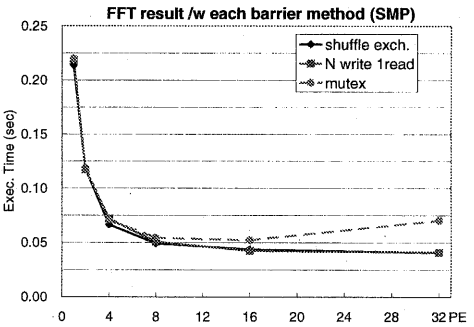

図 10　バリアの方式による FFT(SMP) の比較 (単位:sec)

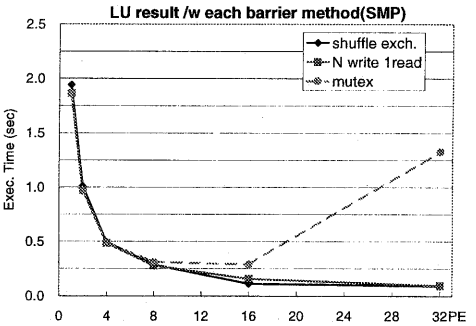

図 11　バリアの方式による LU(SMP) の比較 (単位:sec)

(1) ~ (3) の方式のバリアを用いて，FFT および LU を行った結果を図 **10**, 図 **11** に示す．この結果から，Mutex lock を除く 2 つの方式間の性能の差異はほとんどないことが分かる．SMP では，PC クラスタに比べてバリアがボトルネックになり始める PE 数は大きいが，一旦ボトルネックになってしまうと，やはりバリアのチューニングによる性能の改善は難しい.

4. 関連研究

OMPC++ はDSM機能を自己反映計算を用いたソースレベルのプログラム変換によって実現しており，最適化もまた，ソースレベルのチューニングによって実現している．

OMPC++ では read，write によるアクセスの時のアドレス変換と検査によるオーバーヘッドを軽減させるための工夫として，テーブル検索・アドレス変換においてシフト演算を用いることができるように，テーブルサイズ，配列1要素の大きさを 2^n にしている．また，配列演算子 [] および，write アクセス時に呼び出される手続きはすべて inline 化することで高速化を図っている．また，SPMD プログラムに関しては，ベースプログラムの共有領域への write アクセスで管理テーブルの操作を排他的に行う必要がないため，その処理を省略する．

OMPC++ と同様にプログラム変換を行う高性能DSM として Shasta[5] がある．両者の最大の違いは，OMPC++ がソースレベルの変換を行うのに対し，Shasta がバイナリレベルの変換を行う点である．Shasta では，SMP 用実行プログラムを Shasta コンパイラで変換して，共有領域へのアクセスに対する miss check を含む Alpha クラスタ用実行プログラムを生成する．従って，実行環境 (Alpha のメモリチャネルなど) に大きく依存し，可搬性に乏しい．

Shasta ではさらに，アドレス空間の配置を工夫したり，miss check の命令数を2命令程度に抑えたり，連続する read,write のための check をまとめて行うなどの最適化を施して，DSM のオーバーヘッドを低減している．この最適化を施さない場合には，逐次実行時間の増大が 2.2 ～ 2.3 倍程度であるが，最適化によって 1.05 ～ 1.35 倍程度に抑えることに成功している．

OMPC++ でも，Shasta の最適化技術をソースレベルでの変換において実現することで，汎用性を維持しつつ，現状の 5.6 ～ 7.1 倍程度のオーバーヘッドを大幅に削減できると期待される．

その他の DSM の実装として，オブジェクトベースの Midway[7], CRL[8] 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るため，ユーザが適切な形で挿入できず，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．OMPC++ ではコード生成は自動的に行われるが，ユーザ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．

5. まとめ

本稿では，SPMD スタイルで書かれた C++ プログラムに対して，ソフトウェア DSM 機能を付加する並列言語処理系 OMPC++ の SPLASH2 ベンチマークによる評価を行った．OMPC++ は C++ 用 Open Compiler の OpenC++ を用いて，静的なプログラム変換によってプログラム中の必要な箇所に自動的に DSM 用コードを挿入するため，実行時のオーバーヘッドが小さく，可搬性，保守性も高い．

評価の結果，現状の OMPC++ の実装で，PE 数増加に伴う速度向上は十分に得られた．逐次実行性能は最適化を行っていない Shasta と比較して comparable な結果を得たが，DSM のオーバーヘッドをさらに抑えるには Shasta で用いているような最適化技術を適用する必要がある．

また，PE 数が増加した場合のバリアに要する時間の全体に占める割合の増大が見られた．このため，PC クラスタ，SMP とも複数のバリア方式を実現して比較を行ったが，方式による性能の大きな差は見られず，バリアの改善による性能の向上は基本的に困難であることが明らかになった．

今後の課題は，SPLASH2 の他のアプリケーションを用いて計測を行い，システムの特性をより詳細に明らかにするとともに，ソースレベルのプログラム変換に Shasta で用いている最適化技術を組み込むことである．

謝辞　本研究を行うにあたり，お世話になりました RWCP 石川裕氏，東京大学米澤研 田浦健次朗氏，そして色々とご意見を下さった TEA ミーティング，松岡研の皆様に感謝を申し上げます．

参考文献

1) Sohda, Y., Ogawa, H. and Matsuoka, S.: OMPC++ —A Portable High-Performance Implementation of DSM using OpenC++ Reflection, *submitted to Reflection '99*.
2) Chiba, S.: A Metaobject Protocol for C++, *Proceedings of OOPSLA '95*, ACM SIGPLAN Notices, Vol. 30, No. 10, pp. 285–299 (1995).
3) 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).
4) Ishikawa, Y.: MPC++ MUlti-Threaded Template Library.
5) Scales, D. J. et al.: Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *Proc. of ASPLOS VII*, pp. 174–185 (1996).
6) Mellor-Crummey, J. M. and Scott, M. L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Trans. on Computer Systems*, Vol. 9, No. 1, pp. 21–65 (1991).
7) Bershad, B.N. et al.: The Midway Distributed Shared Memory System, *Proc. of IEEE CompCon Conference* (1993).
8) Johnson, K. L. et al.: CRL: High-Performance All-Software Distributed Shared Memory, *Proc. of SOSP '95* (1995).