広域ソフトウェア分散共有メモリ機構を支援する最適化手法

丹 羽 純 平†

近年，SuperSINET をはじめとする超高速ネットワークが急速に普及しつつあり，超高速 WAN で接続された計算機を並列計算機資源として使用する広域計算が次世代計算プラットフォームとして注目を集めつつある．広域計算のモデルとして，プログラミングの容易性から，共有メモリモデルを選択した場合に，超高速 WAN 上で共有メモリ型並列プログラムを高速に実行できるかどうかが鍵となる．本稿では，超高速 WAN 上のソフトウェア分散共有メモリ機構（広域ソフトウェア分散共有メモリ機構）を効率良く実現する最適化方式を提案する．最適化コンパイラによる静的解析と効率的なランタイムライブラリを組み合わせることで，遠隔メモリアクセスのレイテンシを削減し，スケーラブルな共有メモリを提供する．コンパイラとランタイムライブラリのプロトタイプを作成し，Comet delay/drop を用いた擬似広域環境上の実験により，提案した方式の有効性を検証する．

Techniques for Optimizing Software DSM System on WAN

Jumpei Niwa†

Recent progress in high-speed and high-bandwidth backbone networks such as SuperSINET has led to computer clusters over wide-area network (WAN), called *global computing systems*. On global computing systems, the shared-memory programming model makes it much easier for programmers to write parallel applications than the message-passing model. The question is whether or not shared-memory parallel programs can run on global computing systems efficiently. This paper proposes techniques for optimizing software distributed shared memory (S-DSM) on global computing systems. Both the compiler optimization and the run-time optimization make the latency of remote-memory access small and provide scalable shared memory. These techniques for optimizing S-DSM system have been implemented, and they are evaluated through the experiments under pseudo WAN environment using Comet delay/drop.

1. はじめに

近年急増している新世代 E-Science アプリケーションは，大量の計算機パワーを必要としており，従来のスーパコンピュータを多少増強した程度では扱いが困難であり，コストパフォーマンスの点からも非現実的である．コストパフォーマンスの面では，研究機関内にある計算機資源をネットワークで接続した LAN クラスタが優れているものの，計算機パワーそのものが不足している．

一方，ネットワーク技術の進歩やスイッチング技術の向上により，SuperSINET をはじめとした国内超高速バックボーンネットワークは急速に整備されてきた．そして，国際間の超高速バックボーンネットワークも整備されつつある．

その結果，研究機関の計算資源を超高速ネットワークで接続し，計算資源を広域的に利用する次世代実験科学のための計算プラットフォームがコストパフォーマンスの面からも現実味を帯びたものになりつつある．

広域環境にある計算機資源を十分に活用するには，ユーザ自ら並列プログラムを記述することが求められる．共有メモリモデルは逐次計算のモデルの自然な拡張であり，メッセージパッシングモデルに比べて，並列プログラムを記述しやすいという利点を持つ．もちろん，最適化コンパイラが，共有メモリモデルに従って記述された並列プログラムを，inspector-executor 機構を用いて直接メッセージパッシングコード（分散計算機上のコード）に変換することは可能である．しかし，コンパイラの解析は複雑であり[1)]，packing/unpacking にともなう不要なメモリコピーやアドレス変換のオーバヘッドが問題になる[4)]．実行時にシステム全体で仮想的に共有メモリ機構を提供すれば，アプリケーションが共有アドレスを直接扱うことができるので上記の問題を回避できる．

そこで，分散環境で実行時に共有メモリを提供する機

† 科学技術振興機構さきがけ研究 21「機能と構成」領域
PRESTO, Japan Science and Technology Agency

構：ソフトウェア分散共有メモリ機構（S-DSM）[12),13)]が必要となってくる．もちろん，効率の良い S-DSM を構築できるかどうかがが鍵となるわけだが，LAN 上の分散環境では，オペレーティングシステムの支援と最適化コンパイラの支援とランタイムの支援があれば，高性能な S-DSM を構築することが可能であることが示されてきた[16),23)]．

この結果から，超高速 WAN 上であっても，最適化コンパイラとランタイムの支援があれば，効率の良い S-DSM を構築し，共有メモリモデル型並列プログラムを効率良く実行できると考察した．もちろん，WAN は LAN と比較して様々な問題をかかえている．WAN のレイテンシは LAN のそれより非常に高く，WAN 上で動作させる必要のあるアプリケーションは大規模なものであり，大量の共有メモリを必要とする．そこで，遠隔メモリアクセスのレイテンシを削減し，スケーラブルな共有メモリを可能にするユーザレベル最適化技術（コンパイラ，ランタイム）が必要になる．

本稿の構成に関して以下に述べる．2 章において，広域分散共有メモリ機構について概観し，それを効率良く動作させる最適化技法を提案する．3 章において，各最適化技法の実現方式に関して詳細に記述する．4 章において実験環境ならびに実験結果を述べ，提案した方式の有効性を評価する．5 章において関連研究を紹介し，6 章においてまとめを行う．

2. 広域ソフトウェア分散共有メモリ

本機構は，LAN 上のコンパイラが支援する S-DSM：ADSM[20)] と UDSM[14)] を，WAN 上で効率良く動作するように拡張した機構：WDSM である．まず，基盤となっている ADSM と UDSM について概観し，次に WDSM のデザインに関して述べ，それを実現する最適化技法を提案する．

2.1 基盤となる機構——コンパイラが支援する S-DSM

最適化コンパイラが明示的に並列に書かれたプログラムのソースを直接解析して，共有データアクセスを検出する．次に，検出されたデータアクセスに対し，緩和されたメモリモデルのもとで，コヒーレンス管理操作を行うコード（コヒーレンス管理コードと呼ぶ）をできるだけ大きな粒度で明示的に埋め込むコード生成を行う．

緩和されたメモリモデルを低オーバヘッドで実現するプロトコルの下で，ユーザレベルのランタイムがコヒーレンス管理コードを効率良く実行する．通信が必要な場合には，ブロック転送や細粒度通信のコンバイニングを活用する．

- **Asymmetric DSM（ADSM）[20)]**
  - 既存の OS ベースのシステムと同様に（ソフトウェア）キャッシュミス/ヒット判定にページ管理機構を使用
  - 書き込み時のコヒーレンス管理操作は，store 命令とは分離して，ユーザレベルのコードとして最適化コンパイラが自動的に挿入し，緩和されたメモリモデルを活用して最適化を適用
  - コヒーレンス単位（ソフトウェアキャッシュのブロック）はページであるために，false sharing と不必要なデータ通信が問題
- **User-level DSM（UDSM）[14)]**
  - ノード間の通信オーバヘッドを最小にするために，コヒーレンス単位はユーザ定義のセグメント
  - キャッシュミス/ヒット判定ならびに書き込み時のコヒーレンス管理操作はすべて，ユーザレベルのコードで実現される．それらは対応する load/store 命令とは分離され，最適化コンパイラが自動的に挿入し，ADSM の場合と同様に最適化を適用
  - コヒーレンス管理コードのオーバヘッドをいかにおさえるかが問題

2.2 デザイン

WDSM は，ADSM/UDSM と同様にコンパイラが支援する S-DSM であることに変わりはない．また，コヒーレンス管理プロトコルも ADSM/UDSM と同様に，マルチライタで，AURC[9)] を明示的な通信コードによってソフトウェアエミュレーションするプロトコル[18),20)] を採用する．

ユーザは実行環境（LAN もしくは WAN）を意識せずに，緩和されたモデル（LRC モデル[12)]）に基づいて共有メモリ型並列プログラムを記述する．それをコンパイラが直接解析して，共有メモリアクセスを検知して，それにともなう WAN 用のコヒーレンス管理コード（2.4 節）を明示的にソースコードに埋め込む．最適化コンパイラが静的に，LAN 用だけでなく，後述する WAN 用の最適化を施すところが ADSM/UDSM と違う点である．

最適化されたコヒーレンス管理コードが挿入されたプログラムを，プラットフォーム上の逐次コンパイラ（gcc 等）でコンパイルして，ランタイムライブラリをリンクして実行コードを生成する（**図 1**）．ランタイムが動的に，LAN 用だけでなく，後述する WAN 用の最適化を適用するところが，ADSM/UDSM と違う点である．

最適化コンパイラとランタイムが協調することで，

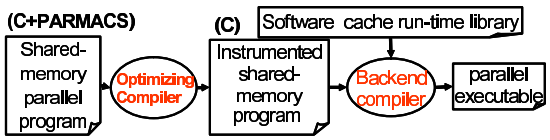

図 1 コード生成プロセス
Fig. 1 Overall compilation process.

種々の最適化が可能になる．以下において，本稿で提案する WAN 用の最適化を概観する．

2.3 ユーザレベル最適化方式

2.3.1 WAN の高レイテンシの隠蔽

- **クラスタキャッシュ**

S-DSM では，遠隔メモリアクセスのレイテンシを削減するために，ソフトウェアキャッシュを導入している．WAN クラスタを LAN クラスタの “クラスタ” ととらえることで，他の LAN クラスタにあるデータを自分のいる LAN クラスタにキャッシュ（**クラスタキャッシュ**）することで，WAN の通信回数（LAN クラスタ間の通信回数）を削減する．

同様の概念が，文献 3), 21) に見受けられる．本機構は，コンパイラが支援する機構であるために twin/diff [12] の操作が不要であるので，diff が分散していることを仮定する文献 3) とは実装方式が大きく異なる．ホームを複数用意する文献 21) と違って，本機構においてホームはただ 1 つであり，不必要になる可能性のあるライトバックは発生しない．

- **階層化バリア**

バリア同期をとる際に，LAN クラスタ内でバリア同期をとってから，LAN クラスタ間でバリア同期をとるという階層的な手法を用いることで，WAN の通信回数を削減する．

- **プリフェッチ**

UDSM では，キャッシュのミス/ヒット判定を実アクセスの前に，大きな粒度で行う方針を採用していた．WDSM では WAN の高レイテンシを考慮して，キャッシュのミス/ヒット判定をできるだけ早く行うコンパイル技術を採用する．さらに，ランタイムがキャッシュミスのリクエストを非同期式に発行することで，計算と通信をオーバラップさせる．

“実際にキャッシュブロックが最新なものになったかどうかを確認する” 必要が生じるが，上記操作は，ADSM のようにページ管理機構を流用することで，命令オーバヘッドを増大させることなく実行される．

- **選択的更新型プロトコル**

科学技術計算では，バリアの含まれたループで反復計算するコードが多いものの，静的に通信集合を求めることが不可能である場合が多い．しかしながら，そのほとんどの場合，実際には参照パターンは変化しない．よって，コンパイラが参照パターンが変化しないと解析した場合には，最初のイテレーションだけ無効化型のプロトコルを適用する．最初のイテレーションの実行の情報を元に，ランタイムは以降のイテレーションを更新型のプロトコルで管理する．

2.3.2 スケーラブルな共有メモリの提供

近年の 64 ビットアーキテクチャの低価格高性能化（IA-64 や AMD64）を考慮して，ノードに 64 ビットアーキテクチャを採用する．さらに，共有メモリサイズを仮想メモリサイズまで拡張することで，大量のメモリを使用する WAN 上のアプリケーションを動作させることが可能になる．ただし，物理メモリサイズを超えて共有メモリを確保する場合，適宜，遠隔メモリのキャッシュをリプレイスする必要がある．

- **ロバストなキャッシュリプレイス手法**

閾値を設定して，その値を超えてキャッシュをマップしようとするときには，その時点でマップしてある有効なキャッシュをリプレイスする方針をとる（ADSM と同様に，無効なキャッシュは同期操作の後で，すでにアンマップされているものとする）．

WDSM では，実際の書き込みと書き込み後のコヒーレンス管理コード（書き込みの発行）が分離されているため，その間にキャッシュのリプレイスが発生すると，実際書き込んだのに，それを見落としてしまう危険がある．そこで，WDSM では，最適化コンパイラが，“プログラム中の以降のコードで書き込みを行う” という書き込み前のコヒーレンス管理コード（書き込みの検知）を挿入する必要がある．

キャッシュをリプレイスする際に，読み出ししかされていないキャッシュに関しては，アンマップする．書き込みの検知は行われたが，書き込みの発行が行われていないキャッシュに関しては，アンマップする前にホームに書き戻す必要がある．ただし，マルチライタなので，自分が書き込む領域以外は，書き戻してはいけない．

リプレイス後に，アンマップされたキャッシュに再度アクセスする場合にはページフォールトが発生する．ページフォールトハンドラが，ホームから有効なキャッシュブロックを入手する．

2.4 コヒーレンス管理コード

WDSM ではコヒーレンス管理操作を行うコードは 3 種類ある．

- 非同期式読み出しの発行（asynchronous read commitment）– $\mathtt{R_a(a,s)}$

プログラム中の以降のコードで，共有メモリの番地 a から s バイト読み出すことを意味する．ランタイムは，実際のアクセスの前に当該キャッシュのミス/ヒット判定を行い，ミスしていた場合には，ホームノードに最新のブロック要求の非同期式メッセージを発行する．リプレイスされたキャッシュに再アクセスする場合には R_a は実行されない可能性がある．その場合には，同様の操作がページフォールトハンドラで実行される

- 書き込みの検知（write detection）– D(a, s)

プログラム中の以降のコードで，共有メモリの番地 a から s バイト書き込みを行うことを意味し，キャッシュをリプレイスする際に必要となる．ランタイムは (a, s) の組をリストに記録しておく．

- 書き込みの発行（write commitment）– W(a, s)

プログラム中の以前のコードで，共有メモリの番地 a から s バイト書き込みが行われたことを意味する．すなわち，静的に diff 情報を算出していることになる．ランタイムはコヒーレンス管理プロトコルに従って，コヒーレンス管理情報を更新する．番地 a から s バイト分を当該キャッシュのホームノードに転送し，write notice [12] にこのページを追加する．D が記録した組のリストの中から (a, s) と等しいものを削除する

3. ユーザレベル最適化方式の実現

3.1 WAN の高レイテンシの隠蔽

3.1.1 クラスタキャッシュ

本最適化は実行時最適化の 1 つである．LAN 上の S-DSM では，ネットワークのレイテンシは局所メモリアクセスのレイテンシよりも大きいので，各ノードは遠隔ノードのデータを自ノードの局所メモリにキャッシュ（ソフトウェアキャッシュ）することで，遠隔メモリアクセスのレイテンシを削減する．

以降の表現では特に明記しない限り，クラスタとは LAN クラスタを意味することとする．WAN 上（クラスタ間）の通信のレイテンシは，LAN 上（クラスタ内）の通信のレイテンシよりも大きいので，本機構はクラスタキャッシュを導入する．つまり，**図 2** にあるように，各クラスタは遠隔クラスタにあるデータを自クラスタにキャッシュすることで，できるだけ WAN 上（クラスタ間）の通信を起こさないようにする．

本機構では，各データブロックに対して，ホームノードを指定するのと同様に，自クラスタのどのノードにキャッシュするか（以降，キャッシュノードと呼ぶ）をユーザが指定できるようにしている．文献 3) にあるように，クラスタキャッシュ管理用のプロセスを作成するのではない．あるブロックに関して，クラスタキャッシュ管理を割り当てられたノードは，通常の処理に加えて，そのブロックに関するクラスタキャッシュ管理の処理を行う．

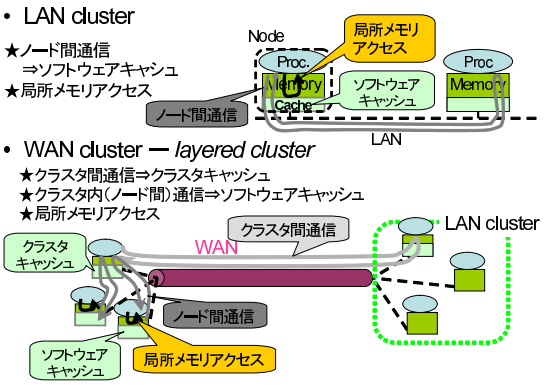

図 2 クラスタキャッシュ
Fig. 2 Cluster cache.

あるクラスタ C において，C 内のノード n が，あるブロック b（b のホームは C 内のノードではない）に関して，クラスタキャッシュの管理を行う場合を考察してみる．クラスタ C 内のすべてのノードはブロック b に関するキャッシュミスリクエストを必ず n に対して発行する．n が有効なキャッシュを持っている場合，n がブロック b を転送する．n が有効なキャッシュを持っていない場合，n が代表して，他のクラスタにある b のホームノードに対してキャッシュミスリクエストを発行して，ブロック b を転送してもらう．この後に，同一クラスタ C 内の別なノードが同一ブロック b に関してキャッシュミスを起こしても，WAN 上の通信は発生しなくなる．

ユーザの指定がない場合には，ブロック番号からラウンドロビンでキャッシュノードを一意に定めて，負荷を分散するようにする．

バリアで囲まれた区間内の計算に関しては上記の手法で問題はない．ただし，ロックで囲まれた区間においてキャッシュミスを起こした場合には，クラスタキャッシュが最新のものであると保証されない．よって，キャッシュノードをバイパスして，ホームから直接取り寄せる必要がある．

3.1.2 階層化バリア

本最適化は実行時最適化の 1 つである．バリア同期に関しても，LAN と WAN の階層化を活用して，長距離通信の数を削減する．本機構では，実装の容易性から，WAN 上（クラスタ間）の遅延は同一であると仮定している．

各クラスタごとに，クラスタ内のバリアのメッセージを管理するクラスタマスタノードを設定する．さら

に，クラスタ間のバリアのメッセージを管理するバリアマスタノードを WAN 上で 1 台設定する．

(1)　各クラスタのノードはクラスタマスタに自分の wtite notice 情報を（DirtyBitTable [16),23)] というビットベクトルの形で）転送する．クラスタマスタはそれらをマージする．

(2)　クラスタマスタが，バリアマスタに自分がマージした write notice 情報を転送し，バリアマスタはそれらをマージする．

(3)　バリアマスタは各クラスタマスタに全体の write notice 情報を配り，各クラスタマスタはそれらを自クラスタ内の各ノードに転送する．

3.1.3　プリフェッチ

本最適化は最適化コンパイラとランタイムが協調して可能になる最適化の 1 つである．まず，コンパイル時最適化について記述し，次に実行時最適化について記述する．

- **コンパイル時最適化**

本コンパイラは，遠隔アクセスのレイテンシ削減のために，asynchronous read commitment: R_a（2.4 節），すなわち，読み出し前のコヒーレンス管理コード（以降，チェックコードと呼ぶ）をできるだけ早く実行することを目標にする．

その際の障害となるものが 2 つある．1 つ目は同期である．チェックコードを同期を超えて遡って発行することは，並列プログラムの意味を変化させてしまう．さらに，適用可能なケースは通信集合が完全に解析できる場合のみと大きく制限されるので，本コンパイラは同期を遡ってコードを発行する方針はとらない．

2 つ目は条件分岐（制御の交わるところ）である．ただ条件節を遡ってコードを発行するだけでは無駄な通信やミス/ヒット判定を誘発する危険がある．そこで，条件を複製し，条件付きのチェックコードを挿入することで，この危険を回避する．

図 3 に示してある例を用いて具体的に説明する．最適化コンパイラ RCOP [16),23)] は UDSM 用のコードを生成する際に，最終的に一括化されたチェックコード $R_s(a, 8*n)$（synchronous read commitment）を条件分岐で分かれた直後に挿入する（図 3 左下）．それより上に移動しない理由は，条件分岐の前に持ってくると，if 節の中を通らない場合には，無駄なミス/ヒット判定ならびに通信を誘発する危険があるからである．

本コンパイラは，if 文の条件式（例の場合には，(MyNum == owner)）が副作用を持たない場合には，条件式を複製し，チェックコードに加えたコード，すなわち，**条件付きチェックコード** “if (MyNum == owner) $R_a(a, 8*n)$” を挿入して，さらに上へと移動する．その結果，図 3 右上のようなコードが生成される．

紙面の都合上，データフロー方程式をはじめとする詳細は割愛するが，文献 22) を参照されたい．

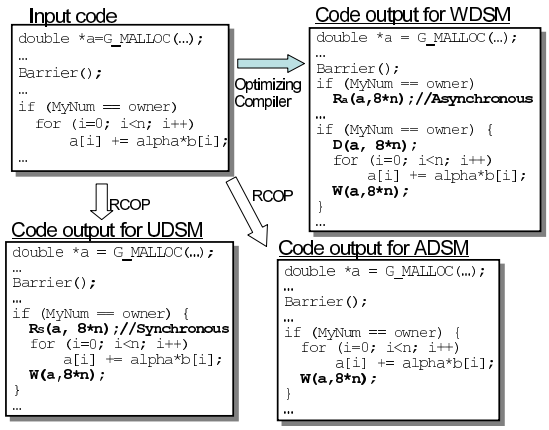

図 3　コード生成例
Fig. 3　Example of generated code.

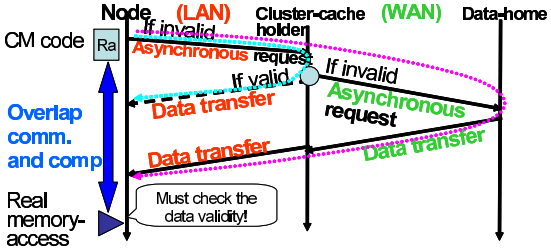

図 4　クラスタキャッシュ使用時の共有読み出しのランタイムの動作
Fig. 4　Behavior of shared-read operations when using cluster cache.

- **実行時最適化**

図 4 はクラスタキャッシュ使用時のプリフェッチを行うランタイムの動作である．3.1.1 項で説明したように，キャッシュミスのときは，各ノードは当該ブロックのキャッシュノードにリクエストを発行するが，発行されるリクエストは**非同期式**である．

キャッシュノードは最新のデータブロックを所有していればそれを転送する．もし所有していなければ，（遠隔クラスタにある）ホームノードに非同期式リクエストを発行する．

ホームノードは必ず最新のデータブロックを有しているので，それをキャッシュノードに転送する．キャッシュノードは，自ノードのデータブロックを最新のものに更新してから，キャッシュミスリクエストを発行したノードに，最新のデータブロックを転送する．この間，図 4 に示してあるように，キャッシュミスリク

エストを発行したノードは計算を実行できるので，通信と計算をオーバラップさせることが可能になる．
ただし，最新のデータブロックが届く前に当該メモリアクセスにたどりついた場合には，データブロックが届くまで待つ．データブロックが届いたかどうかのチェックは，ページ管理機構を流用することで，データブロックが間に合って届いた場合の命令オーバヘッドを増大させずに実行される．

3.1.4 選択的更新型プロトコル

本最適化は最適化コンパイラとランタイムが協調して可能になる最適化の 1 つである．まず，コンパイル時最適化について記述し，次に実行時最適化について記述する．

- **コンパイル時最適化**

バリアを含んだループで反復計算するプログラムの場合には，1 番外側のループに対して，すべての共有メモリアクセスがループ不変であるかどうかを調査する．すなわち，対応する共有アクセス集合の各要素：(アドレス，サイズ，制約（ループ，条件分岐）を表す不等式集合)[16] が，ループ不変であるかどうかを調査する．すべての共有メモリアクセスがループ不変である場合，

– 1 番外側のループの，2 番目のイテレーション以降の最初のバリアの直前に，更新可能のフラグをセットするコードを挿入する．

– 着目した 1 番外側のループの前後にバリアを挿入する．

- **実行時最適化**

– **ディレクトリ情報の生成**

各ノードはループ前のバリアを実行後，キャッシュミスリクエストハンドラにおいてリクエストを発行したノードの番号を記録するようにする．
更新可能のフラグがセットされた最初のバリアにおいて，最初のイテレーションの記録，つまり，自分がホームとなるブロックのコピーを有するノードのリスト（つまり，ディレクトリ情報）を，コピーを有するすべてのノードに対して転送する．これ以降，ディレクトリ情報が変化しないことは，コンパイラが保証している．

– **コピーの有効化**

更新可能のフラグがたった最初のバリアを出る前に，さらに以下の処理を行う．
自ノードがこれまでにアクセスしたブロックの中で，現在無効なブロックに関して，ホームにリクエストを発行して，最新の状態にする．
上記操作が必要な理由は，バリア後に自分のキャッシュミスと他ノードの書き込みの発行が同時に発生すると，タイミングによっては自分に他ノードの書き込みの結果が反映されない危険が発生するからである．

– **更新型書き込み**

以降の書き込みは，書き込まれたデータをホームだけでなく，ディレクトリ情報のリストにあるすべてのノードに対して転送する．また，write notice は生成しない．したがって，キャッシュミスは発生しない．
release 時（ロックを解放するとき，バリアに到着するとき）には，自ノードがデータ転送を行ったすべてのノードに対してデータが届いたことを確認する．したがって，release 操作は無効化型に比べてコストが増大する危険がある．
更新型プロトコルは着目したループ終了後のバリア同期まで続けられる．

3.2 スケーラブルな共有メモリの提供

各ノードにおいて同量の物理メモリ P（GB）を搭載していると仮定する．各ノードは共有領域用に P（GB）を確保する．その内訳は，物理メモリの 3/4，すなわち，$\frac{3}{4}P$（GB）を自分がホームとなる共有メモリとして確保する．残り $\frac{1}{4}P$（GB）をキャッシュ領域に割り当てる．よって，n 台で使用できる仮想共有メモリのサイズは，$\frac{3}{4}Pn$（GB）となる．

キャッシュをリプレイスする閾値を θ とする．すなわち，$\theta\times$ キャッシュ領域 $= \theta \times \frac{1}{4}P$（GB）を超えてキャッシュをマップしようとするときは，キャッシュをリプレイスする．最適な θ の値はアプリケーションに依存するものの，キャッシュのリプレイスはオーバヘッドが大きいので，できるだけ発生しない方が望ましい．よって，θ の値が大きい方が高性能につながると推測される．

3.2.1 ロバストなキャッシュリプレイス

本最適化は最適化コンパイラとランタイムが協調して可能になる最適化の 1 つである．まず，コンパイル時最適化について記述し，次に実行時最適化について記述する．

- **コンパイル時最適化**

本コンパイラは，書き込みの検知 `D(a,s)`（2.4 節），すなわち，書き込み前のコヒーレンス管理コードをできるだけ大きな粒度で，重複なく発行することを目標にする．本最適化は，最適化コンパイラ RCOP における読み出し前のコヒーレンス管理コードの最適化と同様に扱うことが可能である．詳細は文献 16)，23) を参照されたい．

- **実行時最適化**

書き込みの検知 `D` は書き込まれるブロックのアドレスとサイズの組をリストに登録し，書き込みの発行 `W`

(2.4 節) は書き込まれたデータをホームに転送する．キャッシュミス時に，すでに W が発行された書き込みに関しては，あらためてデータを書き戻す必要はない．したがって，W は書き込まれたブロックのアドレスとサイズの組をリストから削除する．

キャッシュのリプレイスが発生した場合，

(1) このリストに残っているアドレスとサイズの各組 (a_i, s_i) に関して，必要な部分だけ，すなわち，連続した領域 $\{x_i \mid \mathtt{a_i} \le x < \mathtt{a_i} + \mathtt{s_i}\}$ を該当ホームに転送する．

(2) 次に，すべてのキャッシュブロックを実際に書き込みを行った/行わないにかかわらずアンマップする．

4. 実　　験

本稿で提案した，最適化コンパイラならびにランタイムについて，プロトタイプを作成し，実験を行い，その有効性について検証する．基本となる実験環境を以下にあげる．

- バックエンドコンパイラ：gcc3.3 (最適化オプションは “-O3”)
- ノード：Dell PowerEdge 1650 × 8 (1.26 GHz PentiumIII，512 KB キャッシュ，2 GB メモリ)
- OS：FreeBSD 5.1-RELEASE
- ネットワーク：NIC は Intel 1000XT で，Summit5i でギガビットイーサ接続
- ベンチマーク：SPLASH-2 [17] から以下の 7 個のアプリケーションを選択

LU-Contig (n = 4 K)，Radix (4 M keys)，FFT (m = 20)，Water-NSquared (n = 32 K)，Water-Spatial (n = 32 K)，Barnes-Spatial (n = 256 K)，Raytrace (balls4)．

4.1 擬似広域環境

図 5 に示すように，LAN 上に WAN 環境を構築する．Summit5i を利用して VLAN を作成し，VLAN どうしの間で通信するときは Comet delay/drop [2] を経由するように設定する．この Comet delay/drop が，ユーザが指定した間隔でパケットを遅延させたり，パケットを落としたり，バンド幅を制限したりする．

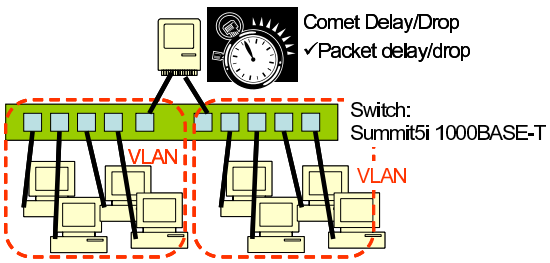

図 5 擬似 WAN 環境
Fig. 5 Pseudo WAN environment.

本実験では，WAN クラスタ (8 ノード) は 2 個の LAN クラスタ (4 ノード) から構成される．また，本実験で挿入する通信遅延は，実行させるアプリケーションのサイズを考慮して，国際間の遅延 (数百ミリ秒) ではなく，メトロポリタン地域で想定される遅延 (0〜10 ミリ秒) である．

4.2 最適化の効果

まず，Comet delay/drop を用いて，LAN クラスタ間に通信遅延を入れて実験を行い，本稿で提案したコンパイラ/ランタイムの最適化の効果 (レイテンシの削減) を調査する．次に，AMD64 を使用して共有メモリのスケーラビリティに関して実験を行う．

4.2.1 レイテンシの削減

4.2.1.1 LU-Contig

図 6 が遅延〔0〜10 (ms)〕を入れたときの WDSM と ADSM における LU-Contig の実行時間である．図 6 の枠内の判例は最適化とその効果を表現している．“プリフェッチ ··· ○” という記述は，LU-Contig というプログラムに対して，プリフェッチ最適化 (3.1.3 項) が適用できかつ性能への効果が高いということを表現している．効果の記述には○のほかに△と×がある．△はその最適化が適用できたが性能への効果が少ないということを表現しており，×はその最適化を適用することができなかったということを表現している．また，“逐次実行時間” は，1 ノードで逐次プログラムを実行したときの時間を意味しており，“1 クラスタの実行時間” は，1 クラスタ (4 ノード) の ADSM 上で並列プログラムを実行したときの時間を意味している．

両機構とも実行時間は遅延時間に比例するが，本最適化により WDSM の方が遅延に対して非常にロバストであると分かる．LU-Contig が，プリフェッチとクラスタキャッシュが効果的に作用する参照パターンを持つことがその理由の 1 つである．

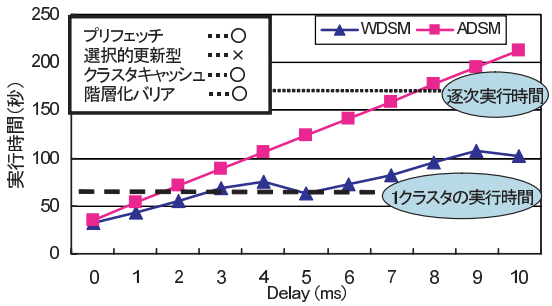

図 6 遅延と LU-Contig の実行時間の関係
Fig. 6 Execution time of LU-Contig.

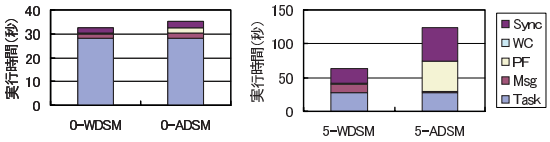

図 7 LU-Contig の実行時間のブレイクダウン〔左：遅延 0（ms），右：遅延 5（ms）〕
Fig. 7 Execution time breakdowns in LU-Contig [left: delay = 0 (ms), right: delay = 5 (ms)].

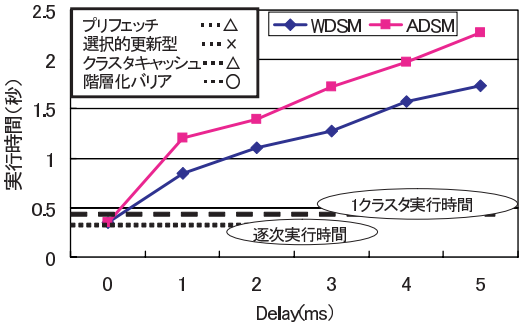

図 8 遅延と Radix の実行時間の関係
Fig. 8 Execution time of Radix.

WDSM では，0〜3，5（ms）の遅延だと，1 クラスタで実行するより 2 クラスタで実行した方が速くなる．遅延が 5（ms）と 10（ms）のときは，WDSM の実行時間が，その前後の遅延のときの結果と比較して減少していることが分かる．現在，原因を究明中である．

図 7 の左側のグラフは遅延 0（ms）のときの WDSM と ADSM の実行時間のブレイクダウンであり，右側のグラフは遅延 5（ms）のときのそれぞれの実行時間のブレイクダウンである．“Sync” は同期プリミティブの実行，ならびに待ち時間，“WC” は `W` の実行時間，“PF” はキャッシュミスの待ち時間，“Msg” はリクエストメッセージを処理する時間，“Task” はプログラム本来の計算時間を示す．WDSM の R_a の実行時間は，Task に含有される．

クラスタ間の遅延がないときでも，プリフェッチの効果により，“PF” の時間が減少しているのが分かる．遅延が 5（ms）のときには，プリフェッチとクラスタキャッシュにより，メッセージ処理の時間が増大しているものの，キャッシュミスの待ち時間が減少していることが分かる．両者の和は最適化により，3 分の 1 以下になっている．その結果，ロードバランスがとれ，階層化バリアが効果的に作用して “Sync” の時間も 2 分の 1 以下になっている．

4.2.1.2 Radix

図 8 が遅延〔0〜5（ms）〕を入れたときの WDSM と ADSM における Radix の実行時間である．枠内の判例の意味は図 6 と同様である．Radix は計算に対する通信の比が大きく，LAN の環境においてですら高速化を達成するのが困難なアプリケーションの 1 つである[6),10)]．TCP/IP を使用しているため通信のオーバヘッドが大きく，クラスタ間の遅延がないときに，逐次より少し高速になるという結果が得られた．

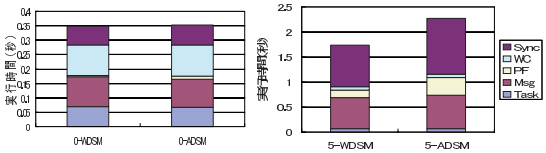

図 9 Radix の実行時間のブレイクダウン〔左：遅延 0（ms），右：遅延 5（ms）〕
Fig. 9 Execution time breakdowns in Radix [left: delay = 0 (ms), right: delay = 5 (ms)].

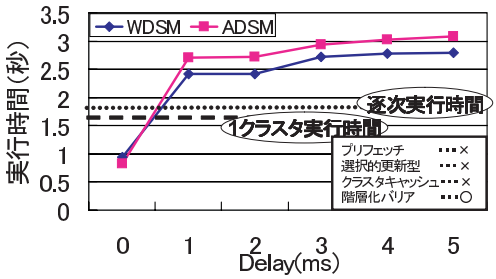

図 10 遅延と FFT の実行時間の関係
Fig. 10 Execution time of FFT.

両機構とも通信遅延に比例して実行時間が増大するが，WDSM の方が ADSM よりロバストである．コンパイラの最適化（Fetch On Write の除去[16),23)]）により，キャッシュミスの数そのものが大幅に削減されていることから，本稿で提案した最適化の中で Radix に最も効果的に作用しているのは，階層化バリアであると考察される．

図 9 の左側のグラフは遅延 0（ms）のときの WDSM と ADSM の実行時間のブレイクダウンであり，右側のグラフは遅延 5（ms）のときのそれぞれの実行時間のブレイクダウンである．階層化バリアの効果で右側のグラフにおいて WDSM の “Sync” の時間が，ADSM のそれより大幅に削減されていることが分かる．

4.2.1.3 FFT

図 10 が遅延〔0〜5（ms）〕を入れたときの WDSM と ADSM における FFT の実行時間である．枠内の判例の意味は図 6 と同様である．FFT は Radix と同様に，計算に対する通信の比が大きく，LAN の環境においてですら高速化を達成するのが困難なアプリケーションの 1 つである[6),10)]．遅延がないときの 8 台実

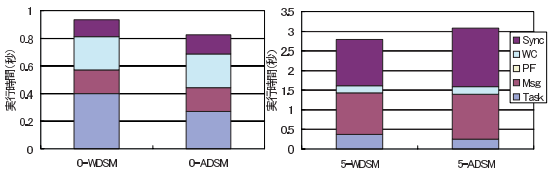

図 11 FFT の実行時間のブレイクダウン〔左：遅延 0（ms），右：遅延 5（ms）〕
Fig. 11 Execution time breakdowns in FFT [left: delay = 0 (ms), right: delay = 5 (ms)].

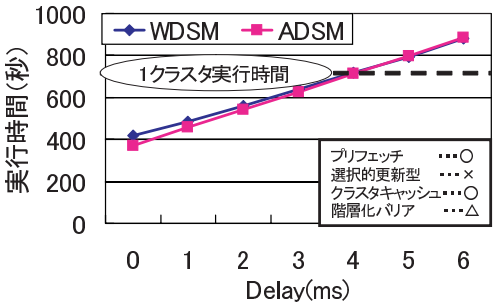

図 12 遅延と Barnes-Spatial の実行時間の関係
Fig. 12 Execution time of Barnes-Spatial.

行の高速化率はおおよそ 2 である．

FFT では通信遅延と実行時間の関係が他のアプリケーションとは異なっているのが分かる．両者とも，通信遅延が 1（ms）のときに実行時間が急激に増大して，以降は遅延を増大させても実行時間はそれほど増大していない．FFT はレイテンシよりバンド幅がボトルネックになっていることに関係していると考察される．

図 11 の左側のグラフは遅延 0（ms）のときの WDSM と ADSM の実行時間のブレイクダウンであり，右側のグラフは遅延 5（ms）のときのそれぞれの実行時間のブレイクダウンである．左側のグラフにおいて（遅延がないときに），WDSM の方が ADSM より遅いのは，R_a の命令オーバヘッドが原因である．コンパイラの最適化（Fetch On Write の除去）により，キャッシュミスが発生しないので，本稿で提案した最適化の中で FFT に作用しているのは，階層化バリアのみであると考察される．右側のグラフにおいて WDSM と ADSM は “Sync” の以外の時間はほとんど変わらないことが分かる．

4.2.1.4 Barnes-Spatial

図 12 が遅延〔0〜6（ms）〕を入れたときの WDSM と ADSM における Barnes-Spatial の実行時間である．枠内の判例の意味は図 6 と同様である．遅延が 4（ms）以下だと，1 クラスタで実行するより 2 クラスタで実行する方が高速となる．このような高速化が得られる理由は，Barnes-Spatial は N 体問題をツリー法で効率良く解くアプリケーションであり，メモリアクセス（すなわち，通信）に対する計算の比が非常に高いからである．

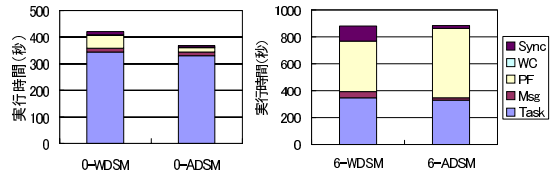

図 13 BarnesSpatial の実行時間のブレイクダウン〔左：遅延 0（ms），右：遅延 6（ms）〕
Fig. 13 Execution time breakdowns in Barnes-Spatial [left: delay = 0 (ms), right: delay = 6 (ms)].

両機構とも通信遅延に比例して実行時間が増大しているのが分かる．また，遅延が小さいときには ADSM の方が高速だが，遅延が増大するにつれて，両機構の差が小さくなり，遅延が 5（ms）以上のところでは，WDSM の方が高速になる．

図 13 の左側のグラフは遅延 0（ms）のときの WDSM と ADSM の実行時間のブレイクダウンであり，右側のグラフは遅延 6（ms）のときのそれぞれのブレイクダウンである．遅延が 0（ms）のときは，クラスタキャッシュを経由する分，メッセージのレイテンシが長くなり，ADSM の方が高速となる．遅延が 6（ms）のときには，クラスタキャッシュやプリフェッチの効果により遠距離通信の数が削減され，“PF” の時間が削減される．ただし，本アプリケーションでは，クラスタキャッシュのリクエストがある特定のノードに固まってしまうので，ロードバランスがくずれ，“Sync” の時間が増大している．それでも，“PF” の時間の削減の効果の方が大きいので，実行時間全体としては WDSM の方が高速になる．

4.2.1.5 Water-NSquared

図 14 が遅延〔0〜5（ms）〕を入れたときの WDSM と ADSM における Water-NSquared の実行時間である．枠内の判例の意味は図 6 と同様である．Water-NSquared の同期プリミティブは，ほとんどがロックである（8 ノードの実行時における，時間計測中の各ノードのバリアの実行回数が 10 回であるのに対し，ロックの実行回数は 40,970 回である）ために，階層化バリアの効果はほとんどない．両機構において，遅延が 1（ms）以下のとき，1 クラスタで実行するより 2 クラスタで実行した方が高速になる．WDSM の方が，ADSM よりも若干良い結果を出している．また，両機構の実行時間の差は，微量ではあるが，遅延に比例して拡大していることが分かる．

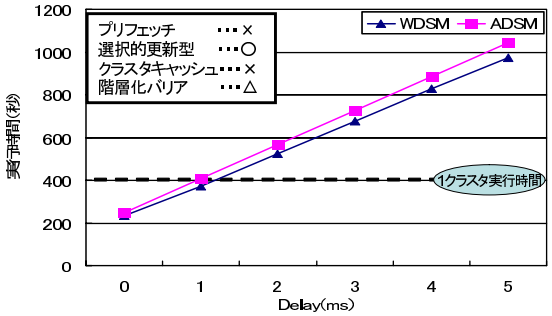

図 14 遅延と Water-NSquared の実行時間の関係
Fig. 14 Execution time of Water-NSquared.

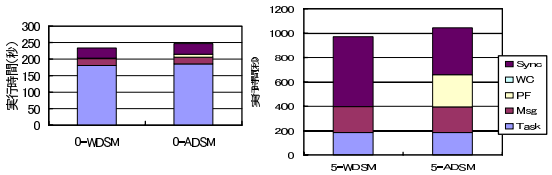

図 15 Water-NSquared の実行時間のブレイクダウン〔左：遅延 0（ms），右：遅延 5（ms）〕
Fig. 15 Execution time breakdowns in Water-NSquared [left: delay = 0 (ms), right: delay = 5 (ms)].

図 15 の左側のグラフは遅延 0（ms）のときの WDSM と ADSM の実行時間のブレイクダウンであり，右側のグラフは遅延 5（ms）のときのそれぞれの実行時間のブレイクダウンである．Water-Nsqared は選択的更新型プロトコルが可能になる一例である．時間を計測し始めてからはキャッシュミスは発生していない．したがって，プリフェッチとクラスタキャッシュの効果はない．

ただし，WDSM では ADSM より “Sync” の時間が増大する．3.1.4 項で述べたように，更新型プロトコルの導入にともない release の操作時間が増大するからである．

4.2.1.6 Water-Spatial

図 16 が遅延〔0〜5（ms)〕を入れたときの WDSM と ADSM における Water-Spatial の実行時間である．枠内の判例の意味は図 6 と同様である．Water-Spatial の同期はバリアのみだが，回数そのものが少ないために階層化バリアの効果はほとんどない．両機構において遅延が 1（ms）より小さいときにのみ，1 クラスタで実行するより 2 クラスタで実行した方が速くなる．クラスタキャッシュの効果で，WDSM の方が ADSM よりも良い結果を出している．また，両機構の実行時間の差は，遅延に比例して拡大している．

図 17 の左側のグラフは遅延 0（ms）のときの WDSM と ADSM の実行時間のブレイクダウンであり，右側のグラフは遅延 5（ms）のときのそれぞれの実行時間のブレイクダウンである．WDSM の “PF” の時間が ADSM のそれに比べて減少しているのは，プリフェッチとクラスタキャッシュの効果である．

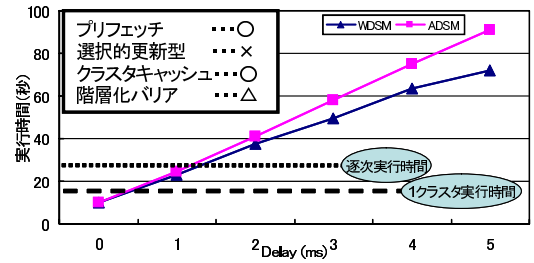

図 16 遅延と Water-Spatial の実行時間の関係
Fig. 16 Execution time of Water-Spatial.

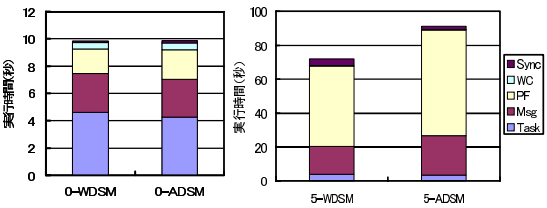

図 17 Water-Spatial の実行時間のブレイクダウン〔左：遅延 0（ms），右：遅延 5（ms）〕
Fig. 17 Execution time breakdowns in Water-Spatial [left: delay = 0 (ms), right: delay = 5 (ms)].

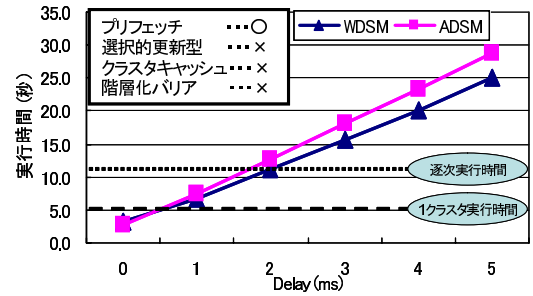

図 18 遅延と Raytrace の実行時間の関係
Fig. 18 Execution time of Raytrace.

4.2.1.7 Raytrace

図 18 が遅延〔0〜5（ms)〕を入れたときの，WDSM と ADSM における Raytrace の実行時間である．枠内の判例の意味は図 6 と同様である．Raytrace の同期はロックのみである．よって，階層化バリアとクラスタキャッシュの効果はない．

両機構において，遅延が 1（ms）より小さいときのみ，1 クラスタで実行するより 2 クラスタで実行した方が高速になる．両機構の実行時間の差は遅延に比例して拡大しており，プリフェッチの効果である．

図 19 の左側のグラフは遅延 0（ms）のときの Ray-

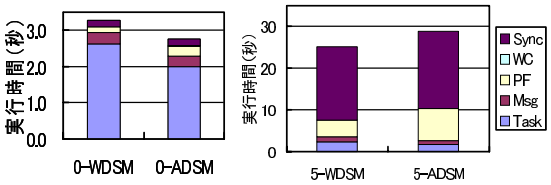

図 19 Raytrace の実行時間のブレイクダウン〔左：遅延 0（ms），右：遅延 5（ms）〕
Fig. 19 Execution time breakdowns in Raytrace [left: delay = 0 (ms), right: delay = 5 (ms)].

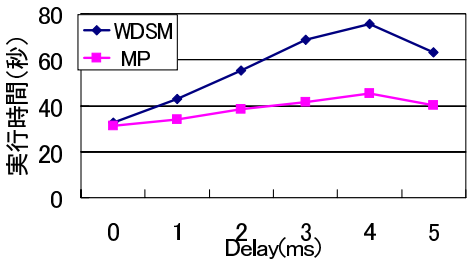

図 20 LU-Contig における共有メモリ版（WDSM）とメッセージパッシング版（MP）の比較
Fig. 20 Comparison between shared-memory version and message-passing version of LU-Contig.

trace の WDSM と ADSM における実行時間のブレイクダウンであり，右側のグラフは遅延 5（ms）のときのそれぞれの実行時間のブレイクダウンである．遅延がないとき，WDSM では R_a の実行による命令オーバヘッドがプリフェッチ効果を上回って，ADSM より実行時間は遅くなる．遅延が大きいとき，WDSM ではプリフェッチ効果により “PF” の時間が減少し，WDSM の方が ADSM より高速になる．

4.2.2 メッセージパッシングとの比較

WAN の環境において，同一のアプリケーションをメッセージパッシングモデル（ただし，アルゴリズムは同一）で記述したときに，どの程度性能差がでるのか比較することは重要である．使用したアプリケーションの中で，LU-Contig に関してのみ，人手で Use-Def 連鎖を完全に解き，メッセージパッシングのコードを生成することが可能であったので，LU-Contig を用いて比較を行った．その結果が**図 20** である．

遅延がないときには両機構の性能は同一である．遅延が増大するにつれてメッセージパッシングの方がより高速になる．原因の 1 つは図 7 が示しているように，WDSM のキャッシュミスのオーバヘッドではなく，バリア同期のオーバヘッドである．いかに階層化してバリア同期のオーバヘッドを削減したとしても，データフロー情報を完全に解析し，バリアそのものを削減したメッセージパッシングの方が WDSM よりも高速である．この比較の結果は，WAN 環境における「静的なデータフロー解析の重要性」を示唆している．

4.2.3 考　　察

本実験では，擬似 WAN 環境において，メトロポリタン地域で想定される，一定の遅延〔0〜10（ms）〕を入れて実験を行った．実際の WAN 環境では，遅延はネットワーク状況により，不安定であると推測される．このような場合における最適化の効果について考察を加える．

まず，キャッシュミスに関してだが，できるだけ早くリクエストを発行する方針はつねに正しい．キャッシュノードが有効なクラスタキャッシュを所有していれば，WAN の通信は発生しないので問題ない．問題はクラスタの外にあるホームノードにリクエストを発行するときである．ホームまでの通信遅延がとても大きくなるような場合であっても，効率良く動作させるために以下のことを行う．

- クラスタ間の遅延を定期的に計測

プログラム実行中に自クラスタから，他クラスタの遅延を一定間隔で，ICMP 等のパケットを飛ばして，定期的に計測する．

- キャッシュミスリクエストの部分的ブロードキャスト

自クラスタからホームノードがあるクラスタまでの遅延を m（ms）とする．キャッシュミスリクエストを受信したキャッシュノードが，もし有効なクラスタキャッシュを持っていない場合には，ホームノードだけにリクエストを発行するのではなく，自クラスタからの遅延が m（ms）以下のすべてのクラスタにリクエストを発行する．もし近いクラスタが有効なキャッシュを持っていれば，レイテンシは削減される．

バリア同期に関しては，LAN と WAN で階層化するだけでは十分ではない．定期的に計測した通信遅延の情報から，遅い WAN 回線を避けるような形で通信を行うことで，不安定な通信遅延に対応することが可能になると推測される．

計算に対する通信の比率が大きいアプリケーション（Radix，FFT）は広域計算には不向きであるということが確認できた．逆に，計算に対する通信の比が特に小さいようなアプリケーション（computation intensive application）である Barnes-Spatial は，小さい問題サイズであっても，通信遅延が 4（ms）以下であれば，台数効果が得られることが分かった．それ以外のアプリケーションであれば，台数効果を得るためには，おおむね 1（ms）以下におさえる必要がある．

この値が実現可能かどうかについて考察する．文

表 1 LU-Contig（$n = 20000$）の平均実行時間とそのブレイクダウン（秒）
Table 1 Average execution time of LU-Contig ($n = 20000$) and its breakdown (sec).

TOTAL	Task	Msg	PF	WC	Sync
8405.15	7780.01	9.18	189.44	21.14	405.33

献 19) では，Cisco MGX WAN スイッチの遅延値は，E1/T1 トランクが使用された場合，スイッチごとに 1（ms）以下であると記載されている．一般に伝播遅延は 10（μs/mile）といわれており，遅延全体を 1（ms）におさえるためには，WAN のスイッチは 1 段で，距離は数マイル以下にする必要があると考察される．現実的な解としては，キャンパスグリッド程度の規模があげられる．

4.2.4 共有メモリのスケーラビリティ

64 bit プロセッサ，AMD の Opteron240 を搭載した PC を 2 台使用して，共有メモリのスケーラビリティに関して予備実験を行った．各 PC とも 2 GB のメモリ（DDR）を搭載しており，NIC は BraodCom BCM5704 で，LAN で接続されている．OS は TurboLinux である．本環境では，3.2 節より，2 台の場合の共有メモリのサイズは 3 GB となる．本実験では，キャッシュをリプレイスする閾値 θ を仮に 0.9 と設定した．

実際に，約 3 GB の共有メモリを必要とする，$n = 20000$ のときの LU-Contig を WDSM 上で走らせたときの結果が，**表 1** である．“Task” 等の意味は 4.2.1 項と同様である．Task が実行時間（TOTAL）の 92%程度になる．

$n = 8192$ のときの LU-Contig は 512 MB の共有メモリを必要とし，アプリケーションの実行に必要なすべてのデータが 1 台の物理メモリに収まる．同一環境における $n = 8192$ のときの ADSM の実行時間は 577.23（s）であった．LU-Contig のアルゴリズムは $O(n^3)$ であることから，$n = 20000$ のときの ADSM の実行時間は 8399.98（s）と見積もることができる．したがって，ADSM から WDSM に移行しても，ほとんどオーバヘッドが増大していないと分かる．

その理由の 1 つに，LU-Contig を行うコアの部分ではキャッシュのリプレイスが実行されていないことがあげられる．LU-Contig では 1 台のノードが代表して，行列の初期化と得られた解の検算を行っているが，その際に 2 回ずつ，計 4 回，キャッシュのリプレイスが行われる．

今後は，台数を増大させて擬似広域環境に接続し，並列計算実行中にキャッシュリプレイスが起こるようなアプリケーションで，種々の実験を行っていく予定である．また，キャッシュをリプレイスする最適な閾値を求めていく予定である．

5. 関 連 研 究

広域計算の研究そのものは多数存在する．代表的なものに Grid [7),15)] がある．Grid は，高性能並列計算というよりは，むしろ，ミドルウェアによる SSI（Single System Image）の実現を目指し，API の作成や，計算資源のサービスの公平な提供（負荷分散やスケジューリング）といった方向に研究が向かっている．それに対して，本研究は実際に広域計算環境のうえで，通信が頻繁に発生する共有メモリ型並列プログラムを効率良く実現する方式の研究である．

WAN 上で実際に共有メモリ機構を実現するものとして Rochester 大学の InterWeave [5)] がある．InterWeave は専用のプログラミングモデルを仮定していて，ユーザが WAN を意識したプログラミングをしなければいけない．さらに，InterWeave は OS ベースのシステムであり，ネットワークのバンド幅を活用するためには，ランタイムシステムがリモートメッセージをバッファリングしなければならない（twin/diff 方式）．

クラスタの階層化を意識して本稿で提案したクラスタキャッシュのようなものを導入した Arantes らの研究[3)] がある．上記機構は twin/diff 方式の S-DSM: TreadMarks [11)] を以下のように改良している．

(1) Inherent cache
キャッシュミスの際に，diff の要求をページを更新した最新のプロセッサに転送するのではない．タイムスタンプから判断して，diff の要求をクラスタの内と外に分けて，クラスタ内で済むものは，クラスタ内で済ませてしまう．クラスタの外には必要最低限の diff 要求メッセージしか発行しない．

(2) Extended cache
キャッシュ管理用のプロセスを 1 個作成して（実際には，クラスタ内に専用のノードを 1 個用意して），クラスタの外への要求はすべてキャッシュ管理プロセスを経由するようにする．これにより，クラスタの外へ同一の diff 要求のメッセージを発行しないようにする．

本機構ではホーム方式を採用しているため，キャッシュミス時にはブロック（ページ）全体を取り寄せることになるので，Inherent cache に関しては単純に比較できない．確かに，Inherent cache の方が，本方式よりクラスタ間の通信量は少なくなる．しかし．それを維持するためのコストが非常に大きい．すなわち，twin/diff に必要なメモリ量，ならびに，diff の GC の

オーバヘッドは，大規模アプリケーションになるほど顕在化してくると思われる．

Extended cache に関しては，本稿で提案したクラスタキャッシュとの違いは，キャッシュ管理用のプロセス（ノード）を必要としないという点にある．彼らがキャッシュ管理用のプロセスを導入した理由は，twin/diff のオーバヘッドが非常に大きいからである．しかし，本稿で提案した方式は，コンパイラの支援により書き込みが静的に検知されるので，twin/diff をまったく必要としないし，そのオーバヘッドも存在しない．

城田ら[21] は JIAJIA[8] を改良し，階層化されたクラスタ上でホームを多重化することで，クラスタ間通信やキャッシュ読み出しのレイテンシを削減する方式を提案している．どれだけホームノードを多重化するかが性能向上の鍵となる．ホームノードの多重化をプロファイリング等で最適化したとしても，アプリケーションによっては，複数のホームへの書き込みの反映は，逆に深刻なオーバヘッドを誘発する場合もある（文献 21) の LU）．一方，本機構では，ホームは 1 つしかなく，クラスタキャッシュはあくまでキャッシュであり，ホームではない．書き込みの結果はホームにしか反映されず，無駄になる可能性のあるトラフィックは発生しない．

物理メモリサイズを超えて共有メモリを確保する S-DSM に JIAJIA[8] がある．JIAJIA では区間の最初において，キャッシュはすべてアンマップする方針を採用している．本機構では，参照の局所性を活用するためにも，無駄なアンマップは行わない．JIAJIA では，使用中にメモリを使い切った場合に，どういう方針でキャッシュをリプレイスするかは明記されていない．

6. ま と め

本稿では，広域分散環境で共有メモリプログラムを高速に動作させる枠組みを提案し，そのうえで，コンパイラとランタイムが行う最適化を提案した．プロトタイプを作成し，擬似広域環境上でメトロポリタン地域で想定される遅延をいれた実験により，提案した方式の有効性を確認した．

SPLASH-2 を用いた擬似広域環境における実験によれば，実行時間の面で台数効果を出すためには，アプリケーションによってばらつきはあるものの，WAN のレイテンシを 1（ms）以下におさえる必要があると分かった．したがって，キャンパスグリッドレベルでは性能を出すことは可能である．今後は，さらにバンド幅を変化させたときの振舞いを調査する予定である．また，N 体問題はメモリアクセスに対する計算の比率が高いので，大規模な問題を動作させると，広域環境でも高性能を達成することが可能であると推測される．

広域共有メモリ機構の意義は，スケーラブルな共有メモリの提供に求められていくものと思われる．そのためにも，今後は，64 ビットアーキテクチャを使用して，長時間の実験に耐えうる効率的な耐故障機能を実装して，種々の実験を行う予定である．

謝辞 Comet delay/drop を貸してくださった東京大学・平木敬教授に感謝します．有意義なコメントをしてくださった査読者の方々に感謝いたします．

参 考 文 献

1) Agrawal, G., Saltz, J. and Das, R.: Interprocedural Partial Redundancy Elimination and its Application to Distributed Memory Compilation, *Proc. '95 Conf. on PLDI* (June 1995).
2) 中村 誠，平木 敬：高レイテンシ環境下におけるデータレゼボワールの性能評価，pp.37–42, HOKKE2003 (Mar. 2003).
3) Arantes, L., Sens, P. and Folliot, B.: The Impact of Caching in a Loosely-coupled Clustered Software DSM System, pp.27–34 (2000).
4) Chang, C., Sussman, A. and Saltz, J.: Object-Oriented Runtime Support for Complex Distributed Data Structures, Technical Report CS-TR-3428, University of Maryland (Mar. 1995).
5) Chen, D., Dwarkadas, S., Parthasarathy, S., Pinheiro, E. and Scott, M.L.: InterWeave: A Middleware System for Distributed Shared State, *Languages, Compilers, and Run-Time Systems for Scalable Computers* (2000).
6) Erlichson, A., Nuckolls, N., Chesson, G. and Hennessy, J.: SoftFLASH: Analyzing the Performance of Clusterd Distributed Virtual Shared Memory, *Proc. ASPLOS-VII* (Oct. 1996).
7) Globus Project. http://www.globus.org
8) Hu, W., Shi, W. and Tang, Z.: Reducing System Overheads in Home-based Software DSMs, *Proc. 1999 IPPS/SPDP*, pp.167–173 (1999).
9) Iftode, L., Dubnicki, C., Felten, E.W. and Li, K.: Improving Release-Consistent Shared Virtual Memory using Automatic Update, *Proc. 2nd HPCA* (Feb. 1996).
10) Jiang, D., Shan, H. and Singh, J.P.: Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors, *Proc. 6th ACM SIGPLAN Symp. on PPOPP* (June 1997).

11) Keleher, P., Cox, A.L., Dwarkadas, S. and Zwaenepoel, W.: Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *Proc. Winter 1994 USENIX Conf.* (Jan. 1994).
12) Keleher, P., Cox, A.L. and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *Proc. 19th ISCA* (May 1992).
13) Li, K.: IVY: A Shared Virtual Memory System for Parallel Computing, *Proc. 1988 ICPP* (Aug. 1988).
14) Matsumoto, T. and Hiraki, K.: Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory, *Proc. 1997 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Los Alamitos, CA, IEEE Computer Society (1998).
15) Ninf Project. http://ninf.apgrid.org
16) Niwa, J.: Study on Optimizing Compilers to Support Software Distributed Shared Memory Systems, Ph.D. thesis, Department of Information Science, The University of Tokyo (2000).
17) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd ISCA* (June 1995).
18) Zhou, Y., Iftode, L. and Li, K.: Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems, *Proc. 2nd Symp. on OSDI* (1996).
19) http://www.cisco.com/japanese/warp/public/3/jp/service/tac/788/voip/delay-details-j.html.
20) 松本 尚, 駒嵐丈人, 渦原 茂, 平木 敬：メモリベース通信による非対称分散共有メモリ，コンピュータシステムシンポジウム論文集 (Nov.1996).
21) 城田祐介，吉川克哉，本多弘樹，弓場敏嗣：マルチホーム方式を用いたマルチクラスタ向けソフトウェア分散共有メモリ，pp.315–322 (2003).
22) 丹羽純平：コンパイラが支援するソフトウェアDSMにおけるプリフェッチ機構，2004-ARC-148, pp.7–13 (2004).
23) 丹羽純平，松本 尚，平木 敬：ソフトウェア分散共有メモリ機構を支援する最適化コンパイラ，情報処理学会論文誌，Vol.42, No.4, pp.879–897 (2001).

(平成 16 年 1 月 30 日受付)
(平成 16 年 6 月 17 日採録)

丹羽 純平

1972 年生．2000 年東京大学大学院理学系研究科情報科学専攻博士課程修了．博士（理学）．現在，科学技術振興機構さきがけ研究 21 研究員．並列化/最適化コンパイラに関する研究に従事．ほかに並列計算機アーキテクチャ，並列分散オペレーティングシステムに興味を持つ．