共有メモリマルチプロセッサの分散シミュレータ Shaman の設計と実装

松尾治幸[†,☆] 今福茂[†,☆☆]
大野和彦[†] 中島浩[†]

本論文は，我々が開発した共有メモリマルチプロセッサのための実行駆動型分散シミュレータ *Shaman* について述べたものである．Shaman はフロントエンドとバックエンドから構成され，PC クラスタで実行される．フロントエンドは対象システムの命令レベルの動作を複数ノードを用いて並列にシミュレートし，単一ノード上のバックエンドでは対象システムのメモリ系のシミュレーションを行う．フロントエンドでのシミュレーション対象には共有メモリの論理的な挙動も含まれ，ソフトウェア分散共有メモリ（DSM）の技法により行うメモリ参照の履歴がバックエンドに送られて，メモリ系のシミュレーションに用いられる．Shaman の重要な特徴は，この参照履歴を DSM の技法とフロントエンドでのキャッシュの部分的シミュレーションを組み合わせた**参照フィルタ操作**により削減することにある．この手法と本論文で述べるシミュレータ特有の DSM 実装技法により，Shaman はきわめて高い性能を達成している．すなわち，16-way の対象マルチプロセッサにおける SPLASH-2 カーネルの実行を 16 台のフロントエンドノードを用いてシミュレートしたとき，1 秒間にシミュレートしたクロック数が LU 分解では 335×10^6，FFT では 392×10^6 となり，我々のシミュレーション手法の有効性が実証された．

Design and Implementation of the Shaman Distributed Simulator of Shared Memory Multiprocessors

Haruyuki Matsuo,[†,☆] Shigeru Imafuku,[†,☆☆] Kazuhiko Ohno[†] and Hiroshi Nakashima[†]

This paper describes our distributed architectural simulator of shared memory multiprocessors named *Shaman*. The simulator runs on a PC cluster that consists of multiple front-end nodes to simulate the instruction level behavior of the target multiprocessor in parallel and a back-end node to simulate the target memory system. The front-end also simulates the logical behavior of the shared memory using software DSM technique and passes the memory references to drive the back-end. A remarkable feature of our simulator is the *reference filtering* to reduce the amount of the references utilizing the DSM mechanism and coherent cache simulation on the front-end. This technique and our sophisticated DSM implementation discussed in this paper give an extraordinary performance to the Shaman simulator. We achieved 335 million and 392 million simulation clock per second for LU decomposition and FFT in SPLASH-2 kernel benchmarks respectively, when we used 16 front-end nodes to simulate a 16-way target SMP.

1. はじめに

マルチプロセッサのアーキテクチャに関する研究開発には，高速なシミュレータが必要不可欠である．しかし単一のプロセッサで複数のプロセッサをシミュレートすると，少なくとも対象システムのプロセッサ数に比例した時間を要することとなる．また並列離散イベントシミュレーション（PDES）の技法は，共有メモリマルチプロセッサのアーキテクチャシミュレーションに必ずしも適合しない．すなわち，並列分散シミュレーションの単位であるプロセッサ・キャッシュ対は，複雑で巨大な状態を持ち，相互の結合が密であるため，状態の保存・復元が必要な楽観的分散時刻管理法や，

† 豊橋技術科学大学情報工学系
Department of Computer and Information Sciences, Toyohashi University of Technology
☆ 現在，株式会社富士通プライムソフトテクノロジ
Presently with Fujitsu Prime Software Technologies Ltd.
☆☆ 現在，セイコーエプソン株式会社
Presently with Seiko Epson Corporation

大きなプロセス間通信遅延を前提とした手法は適用困難である．

そこで我々は分散シミュレータ *Shaman* の設計にあたり，上記のような PDES 技法とはまったく違った方法で並列・分散処理を行うこととした．すなわち Shaman は PC クラスタなどの環境で動作するが，イベント順序や時刻の管理は対象システムのメモリ系（コヒーレントキャッシュを含む）の**物理的挙動**をシミュレートする「単一の」バックエンドノードだけが行う．一方，他のフロントエンドノードは対象システムの命令実行をシミュレートし，共有メモリへのアクセス履歴をバックエンドへ送付する．その際，当然共有メモリが必要となるが，実際にシミュレートしなければならないのはロードやストアの実行結果などの**論理的挙動**である．そこで，PC クラスタなどで一定の効率が得られることが明らかになっているソフトウェア分散共有メモリ（DSM）を用いて，フロントエンドでの共有メモリを実現している．

この方法の問題は，フロントエンドで行うすべての参照をバックエンドに送付してしまうと，明らかな性能ボトルネックが生じてしまうという点である．しかしメモリ系の物理的挙動をシミュレートするために必要な参照は，コヒーレントキャッシュをミスする（共有ブロックへの書き込みを含む）ようなもののみである．そこで我々は，フロントエンドでコヒーレントキャッシュの部分的なシミュレーションを行い，そこでヒットした参照をバックエンドへ送付する履歴から除去する**参照フィルタ**を文献 7)，8) で提案した．またこれらの文献ではフィルタ操作の正当性を証明するとともに，予備評価結果として全体の 1.4%以下の参照しかバックエンドへ送付されないことも示した．この結果は Shaman の設計方針の妥当性を支持し，本論文で述べる実装を進める強い根拠を与えるものであった．

以下本論文では，分散シミュレータ Shaman について次のような順序で議論する．まず 2 章では本研究の背景となる関連研究を概観し，次に 3 章で Shaman の概要と参照フィルタについて述べる．続く 2 つの章が本論文の核心であり，まず 4 章で実装上の諸問題とその解決策について述べ，5 章では性能評価結果とそれについての考察を示す．最後に 6 章では結論と今後の課題を述べる．

2. 関 連 研 究

共有メモリマルチプロセッサには明らかに並列性が内在しているにもかかわらず，シミュレータの並列化や分散化を行った成功例はきわめて少ない．この主な理由は，対象プロセッサの論理構造がきわめて複雑であるため，有力な PDES 技法である Chandy-Misra 法[4]や TimeWarp 法[9]などの非同期的な手法の適用が困難なことである．たとえば実機との実行時間比が 100 であるような単一プロセッサシミュレータを動作周波数 1 GHz の PC からなるクラスタの各ノードに実装し，ノード間を結合する LAN のブロードキャスト遅延がソフトウェアのオーバヘッドも含めて 100 μ 秒であるとする．この場合，ノード間のイベント伝達遅延は少なくとも 1,000 クロックとなるため，イベントの生起時刻が 1,000 クロック程度逆転することを許容する必要があるが，プロセッサやキャッシュの論理構造の複雑さを考えると，この遅延は禁止的に大きな値である．

したがって，並列化されたシミュレータである MPTrace[11]や SimOS[17]は，ホストマシンの共有メモリ機構を用いて対象システムの共有メモリをシミュレートしている．この方法はワークロードの実行という観点からは適切なものであるが，精密な時刻管理という面では問題がある．たとえば MPTrace ではメモリに関するイベントの生起時刻を正確に再現するために，すべてのメモリ参照をいったん履歴ファイルに出力する必要がある．SimOS のバイナリ変換モードは on-the-fly のシミュレーションを行うが，バスやメモリのアクセス競合による影響や，false sharing のような事象を正しくシミュレートできない．

高速性と正確性の両面で成功した数少ない並列・分散シミュレータの例としては，Wisconsin Wind Tunnel（WWT）[16]があげられる．WWT での時刻管理手法は保守的 PDES 手法の一種であり，ある小さい時間区間 Q ごとにシミュレータの全ノードの同期をとるというものである．この Q の値はノードに割り当てられる対象システムの構成要素間（すなわちキャッシュ間）の最小通信遅延以下であることが求められるが，文献 16) での性能評価は $Q = 100$ サイクルというかなり大きな値に基づいている．また WWT の良好な性能は，ホストである CM-5 の高速バリア同期機構に負う部分が大きいが，PC クラスタなどの安価な環境ではこのような機構を望むことは困難である．WWT の後継である WWT-II[14]は，NOW 型のシステム上で良好な性能を示しているが，Q の値はやはり大きく，また 67 MHz の SuperSPARC と 1 Gbit/sec の Myrinet という組み合わせは，現在の技術動向（たとえば 2 GHz の Pentium IV と Gigabit Ethernet）からは大きく乖離したネットワーク偏重の構成といわざるをえない．

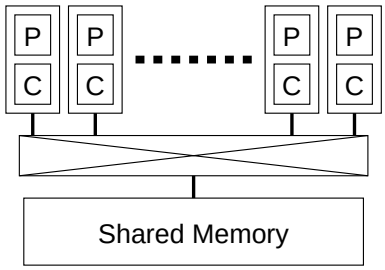


図 1 対象システム
Fig. 1 Target system.

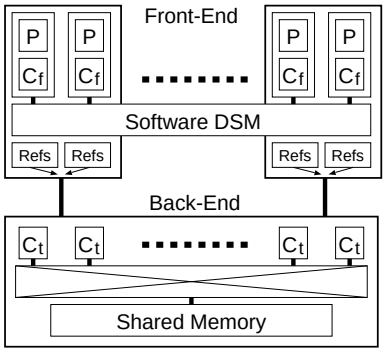


図 2 Shaman の構成
Fig. 2 Configuration of Shaman.

3. Shaman の概要

3.1 対象システムとワークロード

Shaman が対象とするシステムは，一般に**図 1** のような構成を持つ集中型あるいは分散型の共有メモリマルチプロセッサである．対象システムの各プロセッサ（P）はコヒーレントキャッシュ（C）を持ち，プロセッサ/キャッシュとメモリは任意の構成を持つネットワークで結合される．Shaman でシミュレート可能なプロセッサは，現時点では単一パイプライン構成で SPARC の命令セットアーキテクチャに基づくものに限定されるが，原理的にはエミュレーションのモジュールを置換することにより他の実行方式や命令セットにも対応可能である．

キャッシュは，ライトバック型の**拡張 MSI プロトコル**[8]と呼ぶ無効化型コヒーレンスプロトコル[19]に基づくものであることが望ましい．ここで拡張 MSI プロトコルとは，少なくとも **M**，**S**，**I**（Modified，Shared，Invalid）の 3 状態を持ち，他の付加的な状態（たとえば **E** = Exclusive）がこの 3 状態のいずれかに縮退写像可能なものである．他の構成パラメータ，たとえば容量，連想度，ブロック（ライン）サイズ，統合型か命令/データ分離型か，単一であるか階層型であるか，などは対象システムに応じて設定することができる．

Shaman のフロントエンドは 3.3 節で述べるように，lazy release consistency（LRC）[10]の機構により共有メモリをシミュレートする．したがってワークロードのプログラムは，Adve と Hill の定義[1),2)]に基づく *data-race-free*（DRF）の性質を有していなければならない．またロックなどの**獲得**（acquire，S_A）と**解放**（release，S_R）の操作は，Shaman が検出可能な同期プリミティブ（あるいはその一部）でなければならない．

最後に，意味のあるシミュレーションを行うために，ワークロードは以下の定義により決定的でなければならない．

定義 1 あるプログラムの特定の数のプロセッサを用いた実行について，同期プリミティブに含まれないすべてのメモリ参照が実行によらず同じ結果をもたらすとき，またそのときに限り，そのプログラムは決定的であるという．

なおスピンロックのような同期プリミティブ中のメモリ参照については，非決定性が許容されることに注意されたい．

3.2 シミュレータの構成

対象システムの Shaman へのマッピングは，**図 2** に示すように行われる．Shaman のフロントエンドは複数のノードからなり，ワークロードを並列実行しながらメモリ参照履歴を生成する．個々のフロントエンドノード，たとえばクラスタ内の 1 個の PC は，対象システムのプロセッサを 1 つないし複数担当する．なお今後，対象システムのプロセッサおよび対応するフロントエンドの実行モジュールを，単にプロセッサと呼ぶ．

フロントエンドでは，共有メモリへのアクセスを 3.3 節で概説する LRC に基づくソフトウェア分散共有メモリ（DSM）によりシミュレートする．フロントエンドが生成した参照履歴とその局所時刻はバックエンドに送られるが，すべての参照履歴を送信するときわめて大きな通信オーバヘッドが生じる．そこで個々のフロントエンドノードは，プロセッサごとにフィルタキャッシュ（C_f）を持ち，このキャッシュにミスした参照のみをバックエンドに送ることによって履歴送信量を削減する．対象システムのキャッシュのブロックサイズが l，連想度が w（すなわち w-way セット連想），容量が $w \times \gamma$ であるとき，C_f はブロックサイズが l で容量が γ のダイレクトマップ型となる．また C_f のコヒーレンスは MSI プロトコルにより管理される．

バックエンドでは対象システムのメモリやネットワークだけでなく，対象システムのキャッシュと同じ構成の対象キャッシュ（C_t）のシミュレーションも行う．すなわちフロントエンドから送られた参照に対し，対象キャッシュにより再度ヒット/ミスの判定を行い，真のミスだけが抽出されてメモリやネットワークのシミュレーションに用いられる．3.4 節で述べるように，対象キャッシュのミス（共有ブロックの書き込みを含む）を引き起こす可能性があるすべての参照は，必ずフィルタキャッシュを通過することが保証されるので，バックエンドはシミュレーションに必要なすべての参照を入手することができる．対象キャッシュをミスした参照は，メモリやネットワークの挙動に応じて再スケジュールされ，局所時刻が大域時刻に変換される．

3.3 LRC によるソフトウェア分散共有メモリ

前述のように Shaman のソフトウェア DSM の管理機構は，Keleher らによって提案された LRC をベースとしている．以下，LRC の機構について今後の議論に必要な事項を概説するが，DRF プログラムに対する動作保証，実装，性能などについては文献 10) を参照されたい．

ページ コヒーレンス制御の単位であり，あるプロセッサによる参照 m によりページの一部が更新された場合，同じページに対して $m \overset{\text{hb1}}{\rightarrow} m'$ なる参照 m' を行うすべてのプロセッサに，ページの更新情報が伝えられる．ここで $m \overset{\text{hb1}}{\rightarrow} m'$ は m の m' に対する論理的な先行関係 happens-before-1 であり，プログラム順とロック移送などの S_R/S_A の対の非反射的推移閉包として定義される[2]．

インターバル（σ_p^i） プロセッサ p が実行する S_A または S_R に挟まれた i 番目の時間領域に対応する構造．σ_p^i で実行された任意の参照 m と σ_q^j で実行された任意の参照 m' について $m \overset{\text{hb1}}{\rightarrow} m'$ であるとき，またそのときに限り $\sigma_p^i \overset{\text{hb1}}{\rightarrow} \sigma_q^j$ であるとする．

書き込み通知（$w_p^i(\pi)$） プロセッサ p が σ_p^i においてページ π を更新したときに生成される構造．$\sigma_p^i \overset{\text{hb1}}{\rightarrow} \sigma_q^j$ なるインターバルにあるプロセッサ q には S_R/S_A 対の実行時に必ず $w_p^i(\pi)$ が伝達されており，伝達の時点で q が保持する π のコピーは無効化される．

diff（$\Delta_p^i(\pi)$） σ_p^i において更新された π 中のロケーションとその値をエンコードした情報☆．$w_p^i(\pi)$ により無効化された π の参照時に，$\Delta_p^i(\pi)$ を取得して π の正しい値を得る．

vector time-stamp（v_p^i） p が σ_p^i にあるとき，$p \neq q$ なるプロセッサ q について，$v_p^i[q] = j$ は σ_p^i に $\overset{\text{hb1}}{\rightarrow}$ 関係で先行する q の直近のインターバルが σ_q^j であることを示す．また $v_p^i[p]$ はつねに i に等しい．S_A を実行するプロセッサ a は，対応する S_R を実行するプロセッサ r から v_r^j を受け取り；

$$v_a^i[p] \leftarrow \max(v_a^{i-1}[p], v_r^j[p])$$

として上記の関係が満足されるようにする．

3.4 参照フィルタ

3.2 節で述べたフィルタキャッシュ C_f は，Puzak が証明したように[15]対象キャッシュ C_t において不可避性，容量性，あるいは競合性ミスを生じる参照に対し，必ずいずれかのミスを生じる．またプロセッサ p に付随する C_f にキャッシュされたメモリブロック b に関する，他のプロセッサのアクセスの影響による状態遷移を以下のように定義すると，**DRF ブロック**については C_t のコヒーレンスミスが C_f の何らかのミスを生じることが証明できる[8]．

(1) p が b の更新を含むような diff を受け取ると，b の状態は無条件で **I** に遷移する．
(2) p 自身が行った b の更新を含むような diff を，p が他のプロセッサの取得要求に対して初めて渡したとき，b の状態が **M** であれば **S** に遷移する．

ただし DRF ブロックは以下の定義による．

定義 2 ある実行におけるメモリブロック b に対する少なくとも一方が書き込みであるような任意の参照の対 m と m' が，$m \overset{\text{hb1}}{\rightarrow} m'$ または $m' \overset{\text{hb1}}{\rightarrow} m$ と順序付けられるとき，またそのときに限り，b はその実行において DRF であるという．また DRF ではないブロックを非 DRF ブロックという．

さて，ブロックは変数の論理的な集合ではなく単に隣接アドレスに割り付けられたものの集合であるので，プログラムが DRF であっても非 DRF ブロックが存在することがある．そこで以下のアルゴリズムにより非 DRF ブロックを検出し，ブロックに対するフィルタ操作を行わないようにして全参照をバックエンドに渡せば，対象キャッシュの動作を正しくシミュレートすることができる．

(1) DSM の機構によりプロセッサ p にコピーされている各メモリブロック b に対して，**最終参照インターバル** $\tau_p(b)$ と呼ぶフィールドを付加する．ある時点において b への最後の参照がインターバル σ_p^i で行われたとき，またそのときに限り $\tau_p(b) = i$ である．

☆ 元々の LRC での diff $\Delta_p^i(\pi)$ は，σ_p^i の先頭と末尾における π の内容の差分情報であるが，Shaman では 4.1 節で述べるようにわずかに異なった（より厳格な）定義を用いている．

(2) 各々の diff $\Delta_p^i(\pi)$ に対し，vector time-stamp v_p^i を付加する．プロセッサ q が diff $\Delta_p^i(\pi)$ を取得した際，ブロック b の更新が $\Delta_p^i(\pi)$ に含まれていて，かつ $\tau_q(b) > v_p^i[q]$ であれば，b を非 DRF ブロックとしてマークする．

(3) ワークロードの実行完了時点で，非 DRF ブロックとしてマークされていないブロックは，DRF ブロックである．

なおアルゴリズムの正当性については，文献 8) を参照されたい．Shaman ではこのアルゴリズムに基づき，全ブロックが DRF であると仮定しつつワークロード実行/参照履歴生成と非 DRF ブロック検出を行うフェーズ 1 と，非 DRF ブロックが検出された場合にのみ改めてワークロード実行/参照履歴生成を行うフェーズ 2 に分けてシミュレーションを行う．

4. 実装に関する議論

ソフトウェア DSM の性能は，ページ単位のコヒーレンス管理のオーバヘッド，特にノード間での更新データ移送のための通信オーバヘッドに大きく影響される．Shaman では diff ベースの DSM を用いているので，diff の生成と移送がコヒーレンス管理オーバヘッドの主因であることが予想され，実際 5 章で示すように初期の実装では大量のメモリ消費と通信が生じていた．また Shaman には，diff をフィルタキャッシュの状態遷移と非 DRF ブロック検出のためにも用いることと，1 つのフロントエンドノードが複数のプロセッサを担当するという，一般の DSM にはない性質があり，これらを正しくかつ効率的に実装するための工夫が必要である．

そこで本章では，Shaman の DSM の特徴である diff の実装に関する以下の点について議論する．

diff の生成 diff をフィルタキャッシュの状態遷移と非 DRF ブロック検出に用いるため，一般の DSM とは異なりページの各ロケーションの更新の有無を示す**更新ビットベクタ**を用いて diff を生成する方法と，そのオーバヘッドについて議論する（4.1 節）．

diff の併合 diff 移送オーバヘッドを削減する方法として，1 つのページに関する複数の diff を併合する方法と，併合された diff をキャッシュしておく方法，およびそれぞれの効果について議論する（4.2 節）．

diff の共有 diff 移送オーバヘッドを削減する別の方法として，1 つのフロントエンドノードが担当する複数のプロセッサの間で diff を共有する方法と，その効果について議論する（4.3 節）．

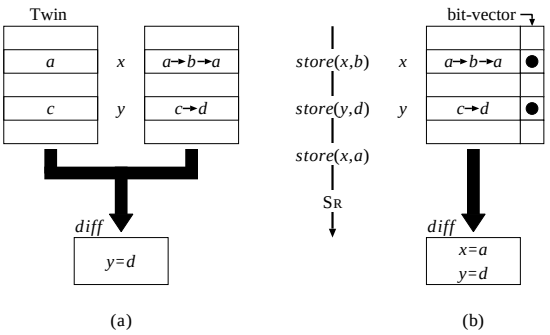


図 3 diff の生成
Fig. 3 Creating a *diff*.

4.1 diff の生成

3.3 節では，プロセッサ p の i 番目のインターバルでのページ π の更新に関する diff $\Delta_p^i(\pi)$ は，π の中で p により更新されたロケーションとその値をエンコードした情報であると述べた．一方，元々の LRC や類似する他の DSM では，diff はその名が示すとおりインターバルの先頭と末尾での π の内容の差分情報である．したがって通常の DSM では diff を以下のように生成する．

(1) あるページに対してあるインターバルの最初のストアを行う際に，そのページのコピーである *Twin* を生成する．ページに対するインターバル内の後続するストアは，単に元のページを更新する形で行う．

(2) インターバルの末尾でページの内容を Twin と比較する．あるロケーションの値がページと Twin で異なっていれば，そのロケーションとページの値が diff の要素となる．

通常の DSM では，diff を取得するプロセッサが必要な情報がページの「値」のみであるので，この方法で正しい diff を作ることができる．また最初のストアをメモリアクセス例外で検出できることと，後続のストアに特別な操作が付随しないことは，DSM の性能の観点から重要な性質である．

一方 Shaman では，あるロケーションが更新されたという「事実」が必要であるため，上記の方法を用いることができない．たとえば**図 3** (a) に示すように，あるロケーション x の値が a から b に更新され，その後さらに a に戻された場合，diff には x の更新に関する情報は含まれない．したがって diff を取得したプロセッサはフィルタキャッシュの x に対応するブロックを無効化しないが，対象システムでは最初のストアにより無効化されるので，正しいフィルタ操作を行うことができない．

そこで Shaman では図 3 (b) に示すように，各ページに**更新ビットベクタ**を付加し，あるロケーションが更新されたか否かを示すようにしている．すなわちインターバルの先頭ではビットベクタはクリアされており，ストアのたびに対象ロケーションに対応するビットがオンになる．インターバルの末尾では，ビットベクタと更新されたロケーションの値，および非 DRF ブロック検出のための vector time-stamp をまとめて diff を生成し，ビットベクタをクリアする．

この方法ではストアのたびに更新ビットをオンにする操作が必要となるが，その時間的オーバヘッドは十分許容可能な程度に小さい．すなわち，一般の DSM ではこのような操作は禁止的に大きなオーバヘッドであるが，Shaman はシミュレータであるのでストア命令の実行にはそもそも数十命令を要するため，性能に及ぼす影響はごく小さいものとなる．また仮に更新ビット操作のオーバヘッドが無視できないとしても，そのかなりの部分は Twin の生成（コピー）や比較を除去できることにより相殺される．

空間的オーバヘッドについては，32 ビットワード単位のビットベクタとすることにより，ページの大きさの 1/32 という十分小さな値とすることができる．Shaman ではワード単位ベクタをデフォルトとし，比較的頻度が少ないバイトあるいは 16 ビットハーフワードを対象とするストアが実行されたときにバイト単位のベクタに置換する方法を用いている．Shaman のページサイズは 4 KB であるので，科学技術計算応用のようにワードあるいはダブルワード単位の書き込みが支配的である場合，ビットベクタによるページあたりの空間的オーバヘッドは，$4\,\mathrm{KB} \times 1/32 = 128\,\mathrm{B}$ となる．

またこの縮小法をさらに拡張し，インターバルの末尾における更新されたワード列と更新されなかったワード列の長さの最小値が 2^n であれば 2^n ワード単位とすることもできる☆．ページの全ワードが更新された場合，この縮小を最大限まで適用してページあたり 1 ビットのビットベクタとすることができるが☆☆，科学技術計算などではそのような例も少なくないものと考えられる．

4.2 diff の併合

diff を用いる LRC の問題点の 1 つに，特定のページの更新と同期を繰り返すようなプログラムでは，大量の diff が生成されてしまうというものがある．たとえば 5 章の評価に用いる LU 分解では，行列の要素 $a_{i,j}$ の更新があるプロセッサで $\min(i,j)$ 回だけ行われるが，他のプロセッサからはその最終的な更新結果のみが参照される．このような場合，diff のために更新を行うプロセッサのメモリが大量に消費されるだけでなく，更新されたページが参照されたときに大量の diff が転送されてネットワークバンド幅も消費される．

この問題の解決法の 1 つとして，ページが固定的なホームノードを持つホームベースの LRC が提案されている[23]．この方法ではあるノードがページを更新すると，作成した diff を内部に保存せずにホームノードに送り，ページのマスターコピーの更新を依頼する．一方，書き込み通知によりページを無効化されたノードでは，diff を取得するのではなくホームノードからページ全体を取得する．この方法は，ページの更新を行うのがホームノードのみであるときに高い効果を発揮するので，プログラマ，コンパイラ，あるいは DSM の実行時システムが，データのページへの割り付けや，ページのノードへの割り付けを注意深くチューニングすることが多い．

しかし Shaman では，ページの参照者が diff を取得することがフィルタキャッシュの状態管理や非 DRF ブロックの検出のために本質的に必要であるので，この方法を適用することができない．また通常の DSM との重要な相違点として，ワークロードは共有メモリマルチプロセッサでの実行を目的として設計されているため，DSM のためのチューニングを期待できないということがある．したがって Shaman では，かなり多くのページが複数の更新者を持つ可能性が高いことに対処しなければならない．

そこで我々は，diff の取得を要求されたときに，送付すべき複数の diff の「値」に関する情報を併合するという方法を用いて，送付するデータ量を削減している．Shaman のページには前節で述べたように更新ビットベクタが付加されているので，これを diff の一部として保存すれば個々のロケーションが更新されたタイミングを記憶することができる．また要求されたすべての diff に関してビットベクタの論理和をとれば，対応する全インターバルで更新されたロケーションが明らかになり，それに基づき最新の値を求めることができる．実際，インターバルごとに生成する diff には「値」に関する情報を持たせる必要はなく，値情報は上記の論理和の結果とページの最新イメージから求めることができる．そこで以後，取得要求への返答として送信される diff を diff_T，インターバルごとに生成される diff を diff_E と表記し，両者を明確に区別する

☆ この縮小法は，得られる効果とコストのトレードオフを詳細に検討する必要があるため未実装である．

☆☆ もちろんどこまで縮小したかを示す情報は別途必要である．

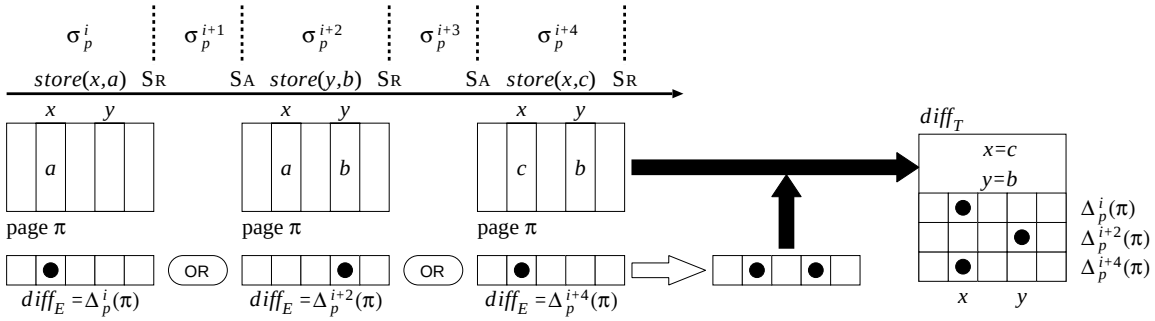


図 4 diff の併合
Fig. 4 Merging *diffs*.

こととする．

たとえば**図 4** に示すように，プロセッサ p がインターバル σ_p^i においてページ π のロケーション x に値 a を書き込み，引き続き σ_p^{i+2} で π のロケーション y に値 b を，σ_p^{i+4} で再び x に c を書き込んだとする．このとき，個々のインターバルごとに生成される diff_E である $\Delta_p^i(\pi)$，Δ_p^{i+2} および Δ_p^{i+4} は，図に示す更新ビットベクタと（図では省略している）vector time-stamp からなり，書き込まれた値の情報は持たない．一方 p に対して他のプロセッサからこれら 3 つの diff の取得要求が行われると，各ビットベクタの論理和に基づいて x と y の値がそれぞれ c と b であることを示す情報と，3 つの diff_E からなる diff_T が生成され，要求元のプロセッサに送られる．

この方法により，diff_T を小さくしてネットワークのバンド幅消費を抑制できるだけでなく，diff_E も小さくなるのでメモリ消費量も抑制できる．この効果を見積もるために，バリア同期する M 個のプロセッサによって実行される SPMD 型のプログラムを想定し，4 KB のページ π が $m \leq M$ なる m 個のプロセッサ $p_1, \ldots, p_m$ により共有されているものとする．またプロセッサ p_1 が π 中の w ワードの連続領域を，別々のバリア同期区間において k 回だけ更新し，その後に他のプロセッサ $p_2, \ldots p_m$ が π を参照するものとする．Shaman ではページ中の更新された連続領域をその先頭のページ内オフセット（2 バイト）と領域サイズ（2 バイト）で表現し，vector time-stamp をプロセッサあたり 4 バイトの配列で表現する．したがって，diff の併合を行わない場合，個々の diff の大きさは $4w + 4 + 4M$ バイトとなり，diff に関する全体でのメモリ消費量 S_{BL} と転送量 T_{BL} は；

$$S_{\mathrm{BL}} = 4km(w + 1 + M)$$
$$T_{\mathrm{BL}} = 4k(m-1)(w + 1 + M) \qquad (1)$$

となる．一方 diff の併合を行う場合は，個々の diff_E は w にかかわらず 128 バイトのビットベクタと $4M$ バイトの vector time-stamp からなり，diff_T は k 個の diff_E と $4w+4$ バイトの値情報からなる．したがって全体のメモリ消費量 S_{MR} と転送量 T_{MR} は；

$$\begin{aligned} S_{\mathrm{MR}} &= 4km(M + 32) \\ &= S_{\mathrm{BL}} \times \frac{M + 32}{w + 1 + M} \\ T_{\mathrm{MR}} &= 4(m-1)((w+1) + (M+32)k) \\ &= T_{\mathrm{BL}} \times \frac{w + 1 + (M+32)k}{k(w + 1 + M)} \end{aligned} \qquad (2)$$

となる．この両者を比較すると，$w > 31$ であれば $S_{\mathrm{MR}} < S_{\mathrm{BL}}$ となり，多くの科学技術計算応用のように大きな配列データが共有されて全体が更新されるページが支配的である場合（すなわち $w = 1024$），diff によるメモリ消費量を 3%程度まで削減できることが分かる．また同じ前提 $31 \ll w \approx 1024$ を用いると，$k > 1$ であれば $T_{\mathrm{MR}} < T_{\mathrm{BL}}$ となり，diff 転送量をほぼ $1/k$ に削減できることが分かる．なお $k = 1$ であるときには $T_{\mathrm{MR}} = T_{\mathrm{BL}} + 128(m-1)$，すなわち 1 回あたりの diff 転送量が 128 バイトだけ増加する．しかしこの場合は，転送される個々の diff の大きさがページサイズを若干上回る程度であるため，小さいメッセージ転送では定数オーバヘッドが支配的な PC クラスタでは，悪影響は軽微であると考えられる．

なおビットベクタや vector time-stamp の併合は困難であり，安易に併合するとこれらを用いた非 DRF ブロックの検出が不正あるいは不正確となる．たとえば複数の diff_E の vector time-stamp を併合して diff_T には最新の値だけを付加するようにすると，diff_T を取得したプロセッサが行う最終参照インターバルとの比較対象である vector time-stamp の値が不当に大きくなり，非 DRF ブロックを見過ごしてしまうおそれがある．逆に最古の vector time-stamp を用いるとその値は不当に小さくなり，DRF ブロックを誤って非 DRF と判定してしまうおそれがある．なお時間的に隣接する 2 つの diff_E のビットベクタの各ビット b_{old}

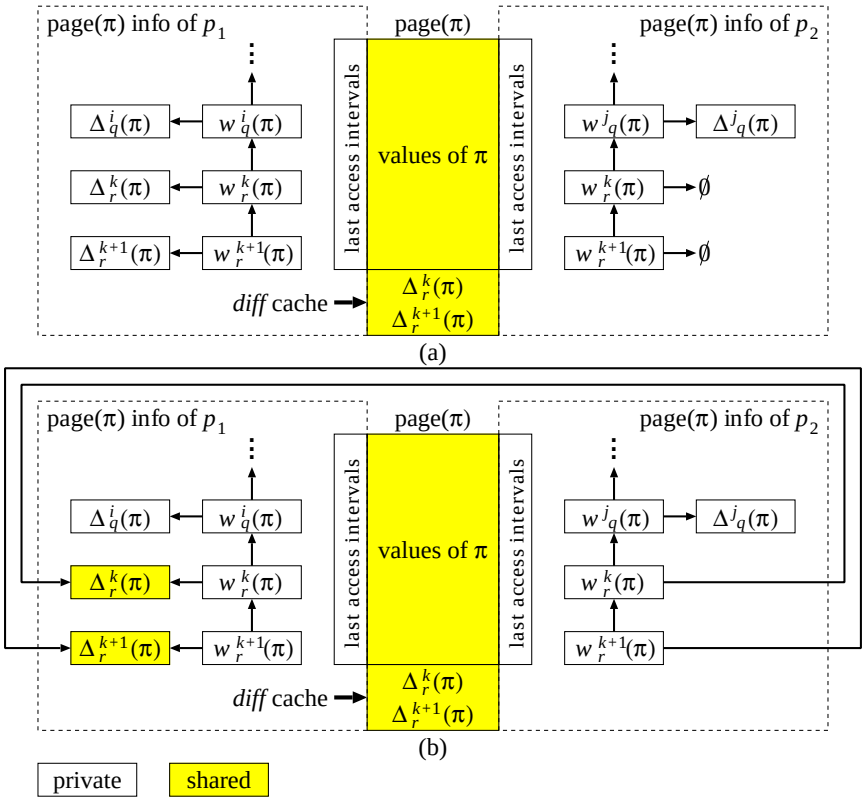


図 5 diff の共有
Fig. 5 Sharing *diffs*.

と b_{new} について，$\neg b_{new} \vee b_{old}$ が成り立つ場合に新しい diff_E を古い diff_E に併合するのは安全でかつ正確性を損なわないが，この最適化は実装していない．

べつの観点からの最適化としては，ある diff_T への取得要求が複数のプロセッサからなされたときに，2 番目以降の要求に対する diff_T の生成コストを削減することが考えられる．我々の実装では diff_T の簡単なキャッシュを用いて，このコスト削減を行っている．すなわち各ページには最後に生成あるいは取得した diff_T のキャッシュがあり，diff 取得要求が指定する書き込み通知の集合とキャッシュされた diff_T の中の diff_E の集合が完全に 1 対 1 対応すれば，キャッシュの内容を取得要求に対して返答する．この最適化は，ページを共有するプロセッサが多いほど効果的であることが明らかであり，前述の m 個のプロセッサによる共有の場合，ヒット率は $(m-2)/(m-1)$ となる．一方キャッシュも含めた diff に関する全体のメモリ消費量 S_{DC} は；

$$S_{\mathrm{DC}} = 4km(M+32) + 4m(w+1) = S_{\mathrm{MR}} + 4m(w+1) \quad (3)$$

となり，特に k が小さいときには無視できない空間的オーバヘッドが生じる．なお diff_T の大きさや転送回数に変化はないので，diff の転送量 T_{DC} は $T_{\mathrm{DC}} = T_{\mathrm{MR}}$ となる．

4.3 diff の共有

Shaman のフロントエンドノードは複数のプロセッサを担当することがあるが，通常の DSM ではこのような実装を行う意味はないので☆，この実装に関する議論は Shaman に固有のテーマである．

1 つのノードに割り付けられた複数のプロセッサはメモリを論理的に共有しているので，ノードが保有するページの値情報をプロセッサが共有するのは自然であり，かつ安全でもある．一方，ページに関する他の情報の共有は単純ではなく，本質的に共有できないものもある．実際，各ブロックの最終参照インターバルが各プロセッサに固有の情報であることは明らかであるし，書き込み通知も各プロセッサが固有の $\overset{\mathrm{hb1}}{\rightarrow}$ 関係を持っていることからやはり固有である．

ページ情報の残りの部分，すなわち diff_E と diff_T

☆ 1 つのノードで複数スレッドを実行することは考えられるが，それは別の問題である．

のキャッシュは，状況により共有することができる．そこでまず diff_E については**図 5** に示すように，直接ページに付加するのではなく対応する書き込み通知からリンクする形で実装している．同図 (a) では，ノード ν にプロセッサ p_1 と p_2 が割り付けられており，p_1 は m_1 番目のインターバル $\sigma_{p_1}^{m_1}$ に，p_2 は m_2 番目のインターバル $\sigma_{p_2}^{m_2}$ にあるものとする．また p_1 は $\sigma_q^i \overset{\mathrm{hb1}}{\rightarrow} \sigma_r^k \overset{\mathrm{hb1}}{\rightarrow} \sigma_r^{k+1} \overset{\mathrm{hb1}}{\rightarrow} \sigma_{p_1}^{m_1}$ なる 3 つの先行インターバルについて，ページ π の書き込み通知 $w_q^i(\pi)$, $w_r^k(\pi)$, $w_r^{k+1}(\pi)$ と，各々に対応する diff_E を得ている．したがって書き込み通知は新しい順にリンクされ，また書き込み通知から対応する diff_E へのリンクも存在する．さらに最新の 2 つの diff_E である $\Delta_r^k(\pi)$ と $\Delta_r^{k+1}(\pi)$ については，それらからなる diff_T をキャッシュしている．

一方 p_2 も同様に $\sigma_q^j \overset{\mathrm{hb1}}{\rightarrow} \sigma_r^k \overset{\mathrm{hb1}}{\rightarrow} \sigma_r^{k+1} \overset{\mathrm{hb1}}{\rightarrow} \sigma_{p_1}^{m_2}$ なる 3 つの先行インターバルについて，π の書き込み通知 $w_q^j(\pi)$, $w_r^k(\pi)$, $w_r^{k+1}(\pi)$ を得ている．ただし取得・適用が完了している diff_E は $w_q^j(\pi)$ に対応する $\Delta_q^j(\pi)$ のみであり，p_1 と共通の先行インターバルに関する書き込み通知である $w_r^k(\pi)$ と $w_r^{k+1}(\pi)$ には，対応する diff_E が存在しない．

この状態で p_2 がページ π をアクセスすると，π に関する最新の書き込み通知 $w_r^{k+1}(\pi)$ には対応する diff_E がないこと，すなわち p_2 にとっては π は無効状態にあることが分かる．そこで p_2 は本来なら書き込み通知を生成したプロセッサ r に diff の取得を要求するが，その前に p_1 が持つ π の書き込み通知を「スヌープ」する．その結果，p_2 が処理しようとしている 2 個の書き込み通知はいずれも p_1 にもあり，かつそれらには diff_E が付加されていることが判明する．すなわちノード ν 上の π の値が p_2 にとっても有効であり，かつ p_2 が必要とする diff_E がすでにノード内にあることが判明するので，それらを用いてフィルタキャッシュの無効化と非 DRF ブロックの検出操作を行う．またこれらの diff_E へは，図 5 (b) に示すように p_2 の書き込み通知からもリンクを張り，p_1 と共有する．なおキャッシュされている diff_T は変更しない．

なお，あるページに関してプロセッサ q が保有する書き込み通知を新しい順に $w_1(q), w_2(q), \ldots$ としたとき，上記の書き込み通知と diff_E のスヌープ操作は，スヌープするプロセッサ p とスヌープされるプロセッサ p' について $w_1(p)=w_1(p') \wedge \ldots \wedge w_m(p)=w_m(p')$ であるときにのみ成功したと判断する（ただし m は p が必要とする diff_E の個数）．この判断は $w_i(p) = w_j(p)$ $(i < j)$ なる書き込み通知が存在する可能性を無視しているが，このような可能性は低く，かつ探索の限界を設定するのも困難であるので，妥当な判断基準であると考えられる．また同様の理由で 1 ノードが 3 個以上のプロセッサを担当する場合，スヌープするプロセッサ p は他のプロセッサの最新の書き込み通知のみを巡回探索し，$w_1(p)$ と一致しかつ diff_E がリンクされているものが見つかった時点で，そのプロセッサ p' だけを対象に $w_2(p)$ 以降の一致判定を行うこととしている．

プロセッサ p によるスヌープ操作が書き込み通知 $w_i(p)$ で失敗したとき，すなわちスヌープされるプロセッサ p' について $w_i(p) \neq w_i(p')$ であるか，あるいは $w_i(p')$ に diff_E が付加されいない場合には，$w_i(p)$ を作成したプロセッサ p'' に対して $w_i(p)$ 以降の書き込み通知に関する diff の取得要求を行う．またそれらの diff を取得すると，その結果に基づき diff_T のキャッシュを更新する．

この diff 共有の効果を，前節で述べた M 個のプロセッサによる SPMD 型プログラム実行での m 個のプロセッサ $p_1, \ldots, p_m$ によるページ共有を例に見積もる．ただしここでは，ページ π を共有する m 個のプロセッサが，s 個ずつ $n = m/s$ 個のノードに均等に割当てられているものとする．この場合，diff 共有を行わないときの全体のメモリ消費量 S'_{DC} と転送量 T'_{DC} は；

$$S'_{\mathrm{DC}} = S_{\mathrm{DC}} = 4km(M+1) + 4m(w+1)$$
$$T'_{\mathrm{DC}} = 4(m-s)((w+1) + (M+32)k) \quad (4)$$

となる．一方 diff 共有を行った場合の値 S_{SH} と T_{SH} は；

$$\begin{aligned} S_{\mathrm{SH}} &= 4kn(M+1) + 4n(w+1) \\ &= S'_{\mathrm{DC}} \times (m/n) = S'_{\mathrm{DC}}/s \\ T_{\mathrm{SH}} &= 4k(n-1)((w+1) + (M+32)k) \\ &= T'_{\mathrm{DC}} \times (n-1)/(m-s) = T'_{\mathrm{DC}}/s \quad (5) \end{aligned}$$

となり，どちらも $1/s$ に削減されることが分かる．

5. 性能評価

5.1 評価環境とワークロード

Pentium III ベースの PC（DELL OptiPlex GX-200, 1 GHz, 256 MB, Vine Linux 2.1.5）を Gigabit Ethernet により結合した PC クラスタに Shaman を実装し，性能評価を行った．対象システムは 16-way のバス結合共有メモリマルチプロセッサであり，シミュレーションはフロントエンドを N ノード（$N = 1, 2, 4, 8, 16$），バックエンドを 1 ノードとして実行した．すなわち使用したノード数は $N+1$ であり，プロセッサ p_m $(0 \leq m < 16)$ は $n = m \bmod N$ なるノード

ν_n にサイクリックに割り当てられる．また DSM のページサイズは 4 KB である．

対象システムの要素プロセッサは，SPARC 命令セット（version 8）を単一パイプラインで実行するものとし，ロード/ストア以外の命令遅延はすべて 1 とした．対象キャッシュは 1 レベルの命令/データ分離型のダイレクトマップとし，容量は各々64 KB，ブロックサイズは 16 B，コヒーレンス管理は MESI プロトコルとした．またロード/ストアの遅延はキャッシュヒット時が 1，キャッシュミス時はキャッシュ間転送でブロックが得られる場合には 10，メモリからのブロックロードが必要な場合には 30 とし，バスやメモリで競合すると待ち合せ時間が加算される．

ワークロードには，SPLASH-2 カーネル[22]の中の LU 分解と FFT を用いた．LU 分解の行列サイズは 256×256 であり，プロセッサへの割り当て単位である小行列は 8×8 である．行列の要素は倍精度浮動小数点数であるので，行列の大きさは $256 \times 256 \times 8 = 512\,\mathrm{KB} = 128$ ページであり，各プロセッサは連続空間に配置された 64 個の小行列（$64 \times 8 \times 8 \times 8 = 32\,\mathrm{KB} = 8$ ページ）の分解を担当する．またプロセッサ p_m が分解する小行列は $m' = m \bmod 4$ および $\lfloor m''/4 \rfloor = \lfloor m/4 \rfloor$ なるプロセッサ $p_{m'}$ と $p_{m''}$ から参照される．したがって p_m が書き込むページは他の 6 個のプロセッサから参照される．このほか，各小行列の先頭へのポインタ配列（$64 \times 4 = 256\,\mathrm{B}$）など総計 1 ページ以内の大域（共有）変数が使用される．

FFT の問題サイズは 2^{16} であり，$256 \times 256 = 2^{16}$ 要素の倍精度複素数 2 次元配列 3 個（$2^{16} \times 8 \times 2 \times 3 = 3\,\mathrm{MB} = 768$ ページ）が使用される．これらはいずれも行方向にブロック分割され，初期化を除く書き込みはブロックを担当するプロセッサのみが行う．このうち 2 個の配列に対しては一方から他方への転置操作が合計 3 回行われ，各プロセッサは転置前の配列の各行を重複しない 1/16 の断片に分割した形で参照する．しかし配列の 1 行は $256 \times 8 \times 2 = 4\,\mathrm{KB} = 1$ ページであるので，この分割アクセスはページ単位で見ると完全に重複し，結果的にすべてのページがすべてのプロセッサに共有される．残りの 1 個の配列は係数行列であり，特定のプロセッサが初期化した後，各プロセッサが担当ブロックを排他的に参照する．このほか，256 要素の倍精度複素数の係数配列 1 個（$256 \times 8 \times 2 = 4\,\mathrm{KB} = 1$ ページ）が使用され，全プロセッサから共有参照される．

なおいずれのワークロードもキャッシュを意識したコーディングがなされているのでヒット率が良く，フィルタキャッシュを通過する参照は LU 分解では 0.5%，FFT では 1.6%ときわめて僅かなものとなっている．またブロックに対する write-write false sharing などは生じず，したがって非 DRF ブロックは発見されなかった．ただしワークロードが PC クラスタなどの DSM 環境で実行されることは想定されておらず，FFT の行列転置に見られるようにページ単位のコヒーレンス処理に対する配慮はなされていない．

表 1 LU 分解の実行時間
Table 1 Execution time of LU decomposition.

N	1	2	4	8	16
BL	29.05	16.68	10.84	7.28	4.96 [sec]
MR	27.74	15.76	10.28	6.68	4.49
MR+DC	27.69	16.21	10.26	6.64	4.43
MR+DC+SH	25.34	14.52	9.65	6.44	—

表 2 FFT の実行時間
Table 2 Execution time of FFT.

N	1	2	4	8	16
BL	30.50	21.30	11.71	8.44	6.71 [sec]
MR	34.22	23.62	12.82	7.99	6.02
MR+DC	32.64	24.41	12.86	7.68	5.74
MR+DC+SH	23.51	13.00	7.49	7.23	—

5.2 評価結果

表 1 と**表 2** は LU 分解と FFT の実行時間を，フロントエンドのノード数 $N = 1, 2, 4, 8, 16$ について示したものである．また行 BL は 4.2〜4.3 節で述べた DSM 実装効率化を行わないベースライン実装での性能であり，それ以下の各行は以下に示す 3 つの技法を組み合わせた場合の性能である．

MR diff の併合（4.2 節）
DC diff_T のキャッシュ（4.2 節）
SH ノード内での diff の共有（4.3 節）

なお SH は $N = 16$ では適用できないが，$N = 1$ には適用可能でありノード内での diff_E のコピーを省く効果がある．**図 6** と**図 7** は，以下の定義式に基づく実行速度 V_N [14]を示したものである．

$$V_N = (\sum_{i=1}^{16} c_i) / \max_{1 \le j \le N} (t_j)$$

ただし c_i はプロセッサ p_i に関してシミュレートしたクロック数であり，t_j はノード n_j におけるシミュレーション実行時間である．

表 3 は，$N = 16$ でのノードあたりの平均メモリ消費量と平均通信量，およびそれらの diff に関する部分を，BL，MR，MR+DC の 3 つの実装に関して示したものである．また**表 4** は，MR+DC と MR+DC+SH

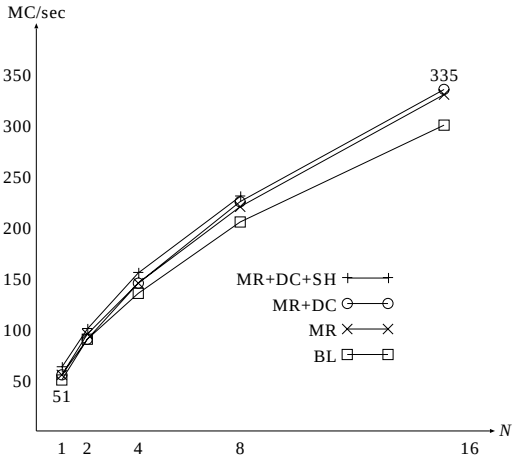


図 6 LU 分解の実行速度
Fig. 6 Execution speed of LU decomposition.

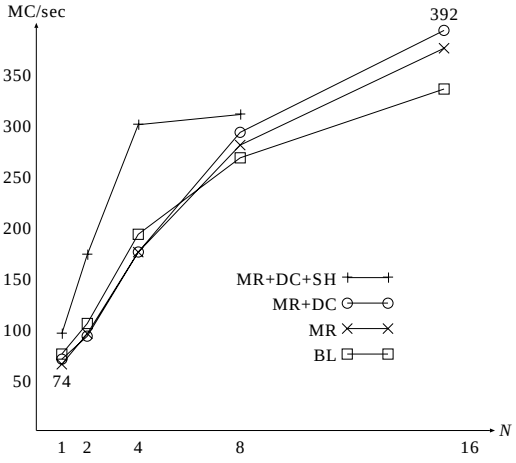


図 7 FFT の実行速度
Fig. 7 Execution speed of FFT.

での同様の測定値の diff に関する部分のみを，$N = 1$〜8 の各々について示したものである．また**図 8** は，$N = 16$ での実行時間に占める diff 取得の待ち時間を，BL，MR，MR+DC の 3 つの実装について示したものである．

ベースライン実装での 1 ノードの性能は，LU 分解で 51 MC/sec（単位は 100 万クロック毎秒），FFT で 74 MC/sec であり，対象システムが単純な構成であることを考慮に入れても，非常に高い性能であるといえる．FFT に関しては MR と MR+DC の 1 ノードの性能がベースライン性能を下回っているが，これらの技法が通信負荷を計算負荷に移すことを主眼としたものであるのでやむをえない．逆に SH はきわめて効果的であり，特に FFT では diff のノード内コピーを省

表 3 メモリ消費量と通信量（$N = 16$）
Table 3 Amount of memory consumption and message transmission ($N = 16$).

		メモリ消費量		通信量	
		総計	diff	総計	diff
LU	BL	2600	2335	2474	1967 [KB]
	MR	454	189	929	522
	MR+DC	667	402	929	522
FFT	BL	8055	5335	5351	4944
	MR	2966	246	4542	4135
	MR+DC	5144	2424	4542	4135

表 4 diff によるメモリ消費量と通信量（$N = 1$〜8）
Table 4 Amount of memory consumption and message transmission by *diff* ($N = 1$ to 8).

		N	1	2	4	8
LU	MR+DC	S	3032	3218	1609	804 [KB]
		T	—	1840	1380	926
	MR+DC+SH	S	485	958	937	583
		T	—	921	1380	812
FFT	MR+DC	S	3936	19393	9696	4848
		T	—	17645	13233	7719
	MR+DC+SH	S	288	3099	2764	2546
		T	—	2440	3459	3918

S: メモリ消費量，T：通信量

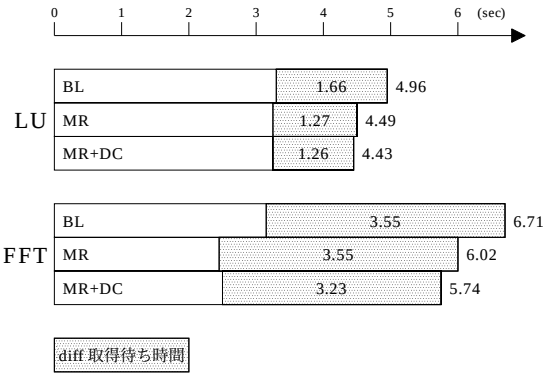


図 8 実行時間に占める diff 取得待ち時間（$N = 16$）
Fig. 8 Execution time and its portion for *diff* retrieval ($N = 16$).

いたことによって，表 4 に示すように diff によるメモリ消費量が 4.3 節の式 (5) の見積値に近い 1/14 程度となり☆，おそらく参照局所性の向上により性能が 40%程度向上している．

並列性能については，16 ノード使用時の MR+DC の性能が LU 分解で 335 MC/sec，FFT で 392 MC/sec という，絶対値としてきわめて高い値が得られた．なお $N = 1$ に対する $N = 16$ の台数効果は，LU 分解

☆ ノード上でページ共有するプロセッサ数 s が 1 や 2 のページがあるので，1/16 とはならない．

が 6.6，FFT が 5.3 とさほど大きくないが，実際に使用しているノードがバックエンドを加えて $N+1=2$ と $N+1=17$ であることを勘案すれば，容認できる程度の値であるともいえる☆．

DSM 実装技法の中では，MR が LU 分解の並列性能向上に最も効果的であり，8 ノードでの性能を 9%，16 ノードでの性能を 10%，それぞれ向上させている．4.2 節で述べたように，LU 分解は共有データである行列要素を何度も更新するため diff が大量に生成され，ベースライン実装ではその最終結果だけを必要としているプロセッサにすべてが送られてしまう．実際，4.2 節の式 (1) の更新回数 k の平均値は 6.3 と大きく，また更新ワード数 w の平均値も 904.6（$=3618.4\ \mathrm{B}=0.88$ ページ）と大きい．

一方 MR の技法を用いると不要な diff の値情報の通信がほぼ完全に除去されるので，16 ノードの場合 diff に関する通信量が 73.5%，全体の通信量も 62.5%削減され，それにより性能が向上している（表 3）．なお，この diff 通信量削減率は 4.2 節の式 (2) から導かれる値 79.2% よりも若干小さいが，これはページ共有の関係が 4.2 節で仮定したモデルよりも複雑であるためである．また図 8 に示すように，通信量の削減率に比べて diff 取得の待ち時間の削減率は小さい．これは MR が diff 転送回数を削減するものではなく，通信量に依存しない定数オーバヘッドが除去できないためであると考えられる．技法 DC の効果が小さいのは，diff_T キャッシュのヒット率が 66%と低いためであると考えられる．

MR の FFT に対する効果は，8 ノードおよび 16 ノードで 6%および 11%の性能向上として現れるが，1〜4 ノードでは逆効果となっている．MR は，(a) diff を保存するためのメモリ領域を削減し，(b) 1 つのページに関する複数の diff が一度に送信される場合はネットワークのトラフィックも削減するが，(c) 送信する diff_T を作成するための手間が増える．FFT では共有配列の要素がある同期区間で更新されると，初期化の直後を除いて次の同期区間で他のプロセッサから参照されるので，図 8 に示すように（b）の効果はほとんどなく，ノード数が少ない場合には（c）の影響で性能が劣化している．

一方 8 および 16 ノードの場合は，(a) によりワーキングセットが縮小して（おそらく）キャッシュのヒット率が向上するため，(c) の悪影響を上回る効果が現れているものと思われる．実際 FFT では，ほとんどの場合ページ全体の更新が行われるので，表 3 に示すように 16 ノードの場合は diff のメモリ消費量が 1/22 程度となり，4.2 節の式 (2) の見積りに合致する．

また技法 DC は（c）の diff_T 作成コストを削減するためのものであり，FFT では diff_T キャッシュのヒット率が 93%と高いため，8 および 16 ノードでは期待どおり 4%および 5%の性能向上効果が得られている．しかし 2 ノードおよび 4 ノードでは効果がなく悪影響さえ見られる．この原因は diff_T キャッシュによるメモリ消費量増加にあると考えられるが（4.2 節の式 (3)），より詳細な解析が必要である．

一方，技法 SH は FFT の並列性能に劇的な改善効果をもたらしており，2，4 および 8 ノードの性能がそれぞれ 88%，72%および 6%向上している．5.1 節で述べたように FFT の 2 次元配列転置操作は，各々 1/16 の断片しかアクセスしないページを全プロセッサが共有するという悲劇的状況をもたらすが，SH はこれを大幅に改善する効果を持っている．すなわち SH を用いなければ，1 つのノード内の個々のプロセッサがまったく同じ diff の集合を何度も他ノードから取得するが，SH はこれを 1 回だけにする効果がある．実際表 4 に示すように，diff の通信量は 4.3 節の式 (5) の見積りどおりほぼ $N/16$（$N=2,4,8$）に削減され，通信回数も同様に削減される．したがってノード内のプロセッサ数が多い場合，つまりノード数が小さい場合に SH が大きな効果を発揮することが，評価によって裏付けられている．

6. おわりに

本論文では，共有メモリマルチプロセッサを対象とする分散シミュレータ Shaman について，その実装と性能評価を中心に述べた．Shaman は複数のフロントエンドノードと単一のバックエンドノードからなる PC クラスタ上で稼働し，フロントエンドではソフトウェア DSM の技法を用いて対象システムでの命令実行を，バックエンドでは対象システムのメモリ系の挙動を，それぞれシミュレートする．DSM の実装は Shaman の重要な特徴である参照フィルタ操作に深く関係しているため，効率的なシミュレーションのために 4 章で議論した種々の技法を考案した．その結果，16 台のフロントエンドノードを用いた 16-way の対象マルチプロセッサのシミュレーションにおいて，LU 分解では 335 MC/sec，FFT では 392 MC/sec というきわめて高い性能を達成することができた．この優れた結果は我々独自の並列・分散シミュレーション方式，

☆ 現在の実装では，1 つのノードがフロントエンドとバックエンドの双方を兼ねることができないので，真の 1 ノード性能をベースとした台数効果を測定することはできない．

参照フィルタ操作，および DSM 実装の有効性を実証するものである．また個々の DSM 実装技法の効果についても評価し，それぞれが効率の面で重要な役割を果たしていることを確認した．

本研究に関する今後の主要な課題は，Shaman の性能をさまざまな角度から評価することである．対象システムについては，高度なプロセッサアーキテクチャ，ハードウェアによる分散共有メモリ，大規模なシステム構成などに対しても，Shaman が効率的に機能することを示す必要がある．また，より大規模かつ複雑なワークロードを用いた評価も計画している．さらに Shaman の実装の改良，たとえば通常の DSM で用いられている技法の採用やバックエンドの並列化なども，今後の課題として残されている．

参 考 文 献

1) Adve, S.V. and Hill, M.D.: Weak Ordering—A New Definition, *Proc. 17th Intl. Symp. Computer Architecture*, pp.2–14 (1990).
2) Adve, S.V. and Hill, M.D.: A Unified Formalization of Four Shared-Memory Models, *IEEE Trans. Parallel and Distributed Systems*, Vol.4, No.6, pp.613–624 (1993).
3) Brewer, E.A., Dellarocas, C.N., Colbrook, A. and Weihl, W.E.: PROTEUS: A High-Performance Parallel-Architecture Simulator, Technical Report MIT/LCS/TR-516, MIT (1991).
4) Chandy, K.M. and Misra, J.: Asynchronous Distributed Simulation via a Sequence of Parallel Computations, *Comm. ACM*, Vol.24, No.11, pp.198–206 (1981).
5) Gabbay, F. and Mendelson, A.: Smart: An Advanced Shared-Memory Simulator—Towards a System-Level Simulation Environment, *Proc. MASCOTS'97* (1997).
6) Goldschmidt, S.R. and Hennesy, J.: The Accuracy of Trace-Driven Simulation of Multiprocessors, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp.146–157 (1993).
7) Imafuku, S., Ohno, K. and Nakashima, H.: Reference Filtering for Distributed Simulation of Shared Memory Multiprocessors, *Proc. 34th Annual Simulation Symp.*, pp.219–226 (2001).
8) 今福　茂，大野和彦，中島　浩：共有メモリ・マルチプロセッサの分散シミュレーションのための参照フィルタ方式，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム，Vol.42, No.SIG 9(HPS3), pp.93–105 (2001).
9) Jefferson, D.R.: Virtual Time, *ACM Trans. Programming Languages and Systems*, Vol.7, No.3, pp.404–425 (1985).
10) Keleher, P.: Lazy Release Consistency for Distributed Shared Memory, Ph.D. Thesis, Dept. Computer Science, Rice Univ. (1994).
11) Keppel, D.K., Koldinger, E.J., Eggers, S.J. and Levy, H.M.: Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems* (1990).
12) Magnusson, P. and Werner, B.: Efficient Memory Simulation in SimICS, *Proc. 28th Annual Simulation Symp.* (1995).
13) 松尾治幸，大野和彦，中島　浩：共有メモリ型並列計算機の分散シミュレータ Shaman の実装と評価，並列処理シンポジウム JSPP'02，pp.111–118 (2002).
14) Mukherjee, S.S., Reinhardt, S.K., Falsafi, B., Litzkow, M., Huss-Lederman, S., Hill, M.D., Larus, J.R. and Wood, D.A.: Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator, *Proc. Workshop on Performance Analysis and Its Impact on Design* (1997).
15) Puzak, T.: Analysis of Cache Replacement Algorithms, Ph.D. Thesis, Univ. Massachusetts (1985).
16) Reinhardt, S.K., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C. and Wood, D.A.: The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp.48–60 (1993).
17) Rosenblum, M., Herrod, S.A., Witchel, E. and Gupta, A.: Complete Computer System Simulation: The SimOS Approach, *IEEE Parallel & Distributed Technology*, Vol.3, No.4, pp.34–43 (1995).
18) Rothman, J.B. and Smith, A.J.: Multiprocessor Memory Reference Generator Using Cerberus, *Proc. MASCOTS'99*, pp.278–287 (1999).
19) Stenstrom, P.: A Survey of Cache Coherence Schemes for Multiprocessors, *Computer*, Vol.23, No.6, pp.14–24 (1990).
20) Uhlig, R.A. and Mudge, T.N.: Trace-Driven Memory Simulation: A Survey, *ACM Computing Surveys*, Vol.29, No.2, pp.128–170 (1997).
21) Veenstra, J.E. and Fowler, R.J.: MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors, *Proc. MASCOTS'94*, pp.201–207 (1994).
22) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Consid-

erations, *Proc. 22nd Intl. Symp. Computer Architecture*, pp.24–36 (1995).
23) Zhou, Y., Iftode, L. and Li, K.: Performance Evaluation of Two Home-Based Lazy Consistency Protocols for Shared Virtual Memory Systems, *Proc. 2nd Symp. Operating Systems Design and Implementation*, pp.75–88 (1996).

(平成 14 年 6 月 6 日受付)
(平成 14 年 9 月 16 日採録)

松尾 治幸

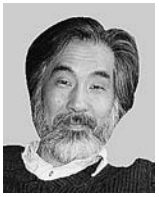
2002 年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年富士通プライムソフトテクノロジ入社．在学中は共有メモリ型並列計算機のアーキテクチャとシミュレーションに関する研究に従事．

今福 茂

2000 年豊橋技術科学大学大学院工学研究科情報工学専攻修士課程修了．同年セイコーエプソン（株）入社．在学中は並列マシンの分散シミュレーションに関する研究に従事．

大野 和彦（正会員）

1998 年京都大学大学院工学研究科情報工学専攻博士後期課程修了．同年豊橋技術科学大学助手．並列プログラミング言語の設計と最適化に関する研究に従事．博士（工学）．



中島 浩（正会員）

1981 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機（株）入社．推論マシンの研究開発に従事．1992 年京都大学工学部助教授．1997 年豊橋技術科学大学教授．並列計算機のアーキテクチャなど並列処理に関する研究に従事．工学博士．1988 年元岡賞，1993 年坂井記念特別賞受賞．IEEE-CS，ACM，ALP，TUG 各会員．