動的アクセスパターン解析によるソフトウェア分散共有メモリ

松葉浩也[†] 石川裕[†]

本論文では FDSM と呼ばれる新しい分散共有メモリシステムを提案する．FDSM はループの 1 回目でアクセスパターンを取得し，通信集合を求める．2 回目以降の実行時にはこの通信集合を使用することで一貫性維持にかかるオーバヘッドを削減し，高速化を達成する．通信集合を求めるためには 1 バイト単位でのアクセスパターンが必要であり，FDSM はインストラクションのエミュレーションによってそれを実現する．姫野ベンチマーク，NAS Parallel Benchmarks の CG を使用したベンチマークでは FDSM は SCASH と呼ばれる別の分散共有メモリシステムと比較して，それぞれ 36%，38%の高速化を達成していることを示す．

The Software Distributed Shared Memory System by Using the Access Pattern Analysis of Applications

Hiroya Matsuba[†] and Yutaka Ishikawa[†]

This paper proposes a new software distributed shared memory system called FDSM. FDSM analyzes the access pattern of the application at the first iteration of a loop and obtains the communication set. This communication set is used in the rest of the iterations and this results in the elimination of the overhead to keep the coherency of the shared memory. A single-byte granularity is required to get the precise communication set. The instruction emulation method is introduced to achieve this granularity. FDSM is evaluated by using the benchmark applications: Himeno Benchmarks and CG in the NAS Parallel Benchmarks. This shows that FDSM is 36% and 38% faster than another software distributed shared memory called SCASH, respectively.

1. はじめに

並列プログラミングには共有メモリ型のプログラミングモデルと分散メモリ型のプログラミングモデルが存在するが，共有メモリ型の方がプログラマにとっては使いやすく，OpenMP などのコンパイラによるサポートも進んでいる．一方でスケーラビリティの観点から，分散メモリ型の並列計算機が普及しており，共有メモリ型のプログラミングモデルを，分散メモリ型の計算機上で使用可能とするソフトウェアの研究が多くなされている．

それらの研究成果のうちの 1 つとして，分散共有メモリを仮定せず，メッセージパッシングコードをコンパイラが生成する手法[6] が提案されている．この手法ではコンパイラによる共有メモリアクセスの解析が完全に成功した場合には高い性能を示すが，解析が不可能なアプリケーションも存在し，その場合多量のブロードキャストが発生するため現実的な実行手段とはならない．

Inspector/Executor と呼ばれる手法[9),17)] では，コンパイラが共有メモリへのアクセスを調べる Inspector と呼ばれるコードと，実際の計算を行う Executor と呼ばれるコードを生成する．Inspector は Executor に先立って実行され．Executor で実際に計算が行われる際には Inspector が収集したアクセス情報を使用して通信が行われる．この手法はコンパイラが正確にアクセスパターンを計算できない種類のアプリケーションも実行可能であるが，特殊なコンパイラが必要である．

ソフトウェア分散共有メモリ（SDSM）は，共有メモリ空間をハードウェアのサポートによらずに提供するソフトウェアである．実現方法として，ライブラリのみで実現する方法と，OS のメモリ保護機能を利用する方法の 2 つがある．ライブラリのみによるソフトウェア分散共有メモリの実現では，共有メモリアクセス用のライブラリを用意する[2)]．このためプログラム

† 東京大学情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo

のアクセスパターンが分かっている必要がある．一方，OS のメモリ保護機能を利用した SDSM 実現手法では，コンパイラのサポートを必要とすることなく，いかなるアクセスパターンを持つアプリケーションも実行可能である．しかし動的に一貫性維持を行うためのオーバヘッドは小さくない[19]．この SDSM にコンパイラによるサポートを加え，両者の性能上の問題点を補い合う手法も提案されている[5]．

本論文では新しい分散共有メモリシステム FDSM を提案する．FDSM はソフトウェア分散共有メモリにおける一貫性維持のためのオーバヘッドの削減を実現する．FDSM は反復法と呼ばれるアルゴリズムを使用したアプリケーションに多く見られる，ループ内でのアクセスパターンが一定であるという性質を持つアプリケーションを対象とする．このようなアプリケーションでは，ループ内での同期操作において必要となる通信のパターンが一定となる．FDSM は反復の 1 回目でアプリケーションの共有メモリへのアクセスパターンを取得し，一貫性維持に必要な通信パターンを計算する．2 回目以降の反復における同期においては，1 回目で求めた通信パターンを使用し，通信パターンを求めるオーバヘッドを削減する．

共有メモリ空間内で通信を必要とする領域を正確に得るためには，1 バイトの粒度でアクセスパターンを取得することが求められる．これは通常のメモリ管理機構を使用し，page fault を発生させるのみでは実現不可能である．FDSM ではインストラクションのエミュレーションを行うことにより 1 バイトの粒度を実現する．

FDSM は SCore Cluster System Software [16] 上で動作し，Omni/OpenMP [20] コンパイラでコンパイルされた OpenMP [12] プログラムを実行することが可能である．アプリケーションが，ループ内で一定のアクセスパターンを持つという FDSM の対象とする性質を持つか否かの判断はユーザに任せられる．

性能評価においては姫野ベンチマークのサイズ M と NAS Parallel Benchmarks [1] の OpenMP C 言語版から CG のクラス B を用い，同じく SCore 上で開発された分散共有メモリ SCASH [19] と比較して，それぞれ 36%，38%の高速化を実現することを示す．

2. ソフトウェア分散共有メモリ

本章ではメモリ管理機構によるソフトウェア分散共有メモリ（以下，SDSM）の実現方法とその問題点について述べる．

SDSM では，プログラム開始時に全プロセスにおいて全共有メモリ空間が書き込み禁止に設定される．その後，共有メモリに書き込みが発生すると OS のメモリ保護機能により page fault が発生する．SDSM はその例外をとらえ，書き込みのあったページを書き込み可能とするとともにページの複製（Twin）を作成する．つまり Twin には書き込みのない状態のページが保存されている．

同期時，SDSM は現在のページと Twin を比較することにより書き込みのあった場所とその内容を特定し（Diff の作成），それをホームと呼ばれるノードに送信する．ホームノードは各ページごとに設定されており，つねに一貫性の保たれた状態のページを保持しているノードである．

ここまでの作業が終了すると，SDSM は再び共有メモリ空間をアクセス禁止にし，次の同期区間の実行を開始する．そして page fault 発生の際にはホームノードからページをコピーすることにより，一貫性の保たれた値を得る．

以上が典型的な SDSM であり，これは Twin & Diff による方法と呼ばれる[3]．一貫性モデルの違いによりホームへのデータ転送のタイミングには種類があるが，Twin & Diff 自体は広く使用されている．

Twin & Diff 方式においては Twin の作成の際にページ全体のコピーが必要であり，Diff 作成の際には 2 ページ分の領域の読み込みが必要である．これらによるオーバヘッドは無視できないほど大きい[19]．また，アクセスのあるページに関しては，1 同期区間につき必ず 1 回の page fault が発生し，オーバヘッドとなる．

6 章以降の性能評価では比較対象として，SCASH [19] と JUMP [4] を用いる．これらはともに Twin & Diff による方法を用いた SDSM である．SCASH は SCore Cluster System Software 上で Myrinet の使用が可能であり，また Omni/OpenMP コンパイラでコンパイルされた OpenMP プログラムを実行することができる．JUMP は Scope Consistency [8] を採用し，Home Migration プロトコルを提案した JIAJIA [7] の改良であり，Migrating-home プロトコルを提案している[4]．

3. 設　　計

本章では FDSM が共有メモリ空間を提供する方法について述べる．

3.1 概　　要

反復法と呼ばれる種類のアルゴリズムに代表される数値演算アプリケーションではループが多用され，実行時間の大半がループの実行に費やされる．また，そ

れらのループ中での共有メモリ空間に対するアクセスは一定のパターンを持つことが多い．そこで FDSM ではループの初回の実行時にアクセスパターンを解析し，2 回目以降の反復では初回と同じ領域をアクセスするものと仮定する．

ループの 1 回目の反復では，計算を実行しつつ共有メモリへのアクセスパターンを取得する．このアクセスパターンを使用すれば，同期の際に通信を必要とする共有メモリ空間の領域を求めることが可能となる．上記仮定により，ループの 2 回目以降の反復においてもこの領域は 1 回目と同じであるため，2 回目以降では 1 回目で求められた領域を一貫性維持の対象とすればよく，Twin & Diff や page fault のオーバヘッドなく必要な通信を行うことが可能となる．なお，上記仮定を満たさないアプリケーションについて，FDSM は Twin & Diff を用いた通常の SDSM として動作し，共有メモリ空間を提供することができる．アプリケーションが FDSM の仮定を満たすか否かの判断はユーザに任されており，ユーザがプログラム実行時に選択する．

FDSM の実行方式はアクセスパターンを動的に取得し，通信の必要な領域を求める点では Inspector/Executor 法と同じである．しかし Inspector/Executor 法がコンパイラによって生成された，アクセスパターン解析専用のコードによって情報を取得するのに対し，FDSM は特殊なコンパイラを必要とせず，アクセスパターン解析中も本来の計算を行う．

3.1.1 用 語 定 義

まず，今後の議論のために用語を定義する．

初回実行モード 計算とアクセスパターン解析を同時に行いながら実行を進める方式

高速モード アクセスパターン情報を用いることによりオーバヘッドを削減した実行方式

通信集合 アクセスパターン情報を使用して求められた通信の必要な共有メモリ内の領域

実行ブロック 共有メモリ空間へのアクセスを含んだ同期区間

さらに今後使用される表現の定義も行う．

Region（Area，Process，TimeStamp）
アクセスのある共有メモリ上の連続領域を示す．Area は開始アドレスと長さの tuple であり，Process はアクセスするプロセス番号，TimeStamp はアクセス時刻である．

RegionSet
Region の集合．連続しない領域のアクセスを表現する．

ExBlock（WAR，RAR，TimeStamp）
実行ブロックを示す．WAR は書き込みアクセス領域，RAR は読み込みアクセス領域であり，これらは RegionSet である．また TimeStamp は最新の実行時刻である．WAR，RAR の各要素である Region の TimeStamp は，実行ブロックの TimeStamp と同じものが保持されているとする．

Communication（Area，Src，Dst）
連続領域 Area をプロセス Src から Dst に通信することを示す．Area は開始アドレスと長さの tuple である．

CommunicationSet
Communication の集合で通信集合を表現する．

Region R のエリアを R.Area のように表記する．

3.1.2 同期のタイミング

FDSM はブロックの開始時，それが初めて実行されるブロックか 2 回目以降であるかを判断し，初回実行モードまたは高速モードを選択する．この判断のために FDSM はブロックに一意な番号を付け，ブロックが実行済みかどうかを記録している．この番号はブロック開始を告げる後述の API 関数，fdsm_before_loop が呼び出された際，その呼び出しに使用された call 命令の番地を調べることで一意な値を得る（ただし FDSM を OpenMP で使用する場合は後述の_ompc_default_sched() を呼び出した命令のアドレスを用いる）．

実行モードが初回実行モードの場合においても高速モード場合においても，一貫性維持操作はブロックの開始前に行われ，ブロック終了時には何も行わない．

3.1.3 初回実行モード

各ブロックの初回実行時，FDSM は初回実行モードで動作する．このモードの役割は，当該ブロック内での共有メモリへのアクセスパターンを取得することと，ブロックの初回実行を正しく行うための共有メモリの一貫性維持である．アクセスパターンの解析方法は次節で詳しく述べることとし，ここでは初回モード時の一貫性維持方法を説明する．

初回実行モードでのブロック開始時，FDSM はホームノードに一貫性のとれた状態のページを準備し，全ページをアクセス禁止にする．そして，ブロックの実行中，各ページ最初のアクセスにおいてホームノードからページをコピーする．

一貫性のとれたページをホームに準備するために，FDSM は前回にホームの一貫性をとって以来，現在までの書き込みを調べ，共有メモリ内の各領域について最新の値を書いたプロセスを特定する．この解析は，

各ノードのアクセスパターンを 1 個のノード（マスタノードと呼ぶ）に集めたうえで，マスタノードが以下のように計算する．

まず 2 個の Region に対する演算 $\otimes$ を定義する．この演算は 2 個の Region をマージした RegionSet を求めるが，エリアが重複する部分については，タイムスタンプの新しものを優先する．この演算 $\otimes$ は

$c_1 = (a_1, p_1, t_1)$

$c_2 = (a_2, p_2, t_2)$ $(t_1 \leq t_2)$

とすると以下のように定義される．

(1) 連続領域 a_1 が a_2 に完全に含まれるとき
（$a_1 \subset a_2$ のとき）
$c_1 \otimes c_2 = \{c_2\}$

(2) 連続領域 a_1，a_2 に共通部分がないとき
（$a_1 \cap a_2 = \phi$ のとき）
$c_1 \otimes c_2 = \{c_1, c_2\}$

(3) 連続領域 a_1 の一部と a_2 の一部が重なるとき
（(1) 以外で $a_1 \cap a_2 \neq \phi$ のとき）
$c_1 \otimes c_2 = \{c_2, (a_1', p_1, t_1)\}$

ただし，a_1'，a_1'' は a_1 から a_1，a_2 の共通部分を除いた残りのエリアである．またこの演算は可換なので $t_1 > t_2$ のときは入れ替えて上記演算を行う．

この演算を使用することで，RegionSet R_1，R_2 をマージし，領域の重なっている部分については TimeStamp の新しい方を優先する演算，$R_1 \oplus R_2$ は次のように定義できる．

$$R_1 \oplus R_2 = \bigcup_{r_1 \in R_1, r_2 \in R_2} (r_1 \otimes r_2)$$

ただし $\cup$ は和集合を意味する．以上の演算を用いることにより，ある時刻 $time$ より後に実行されたブロック中での最新の書き込みを示す RegionSet を返す関数，$LatestWrite(time)$ は，$b_i.time \geq time$ であるすべての ExBlock $b_1, b_2, ..., b_n$ に対し

$LatestWrite(time) =$
$b_1.WAR \oplus b_2.WAR \oplus ... \oplus b_n.WAR$

として定義できる．

初回実行モードのブロック実行前に必要となるものは，前回にホームの一貫性をとって以来の最新の書き込みである．そこでホームの一貫性をとった後にはグローバルな変数 $last\_update$ に現在時刻を保存することにすれば，求める RegionSet R は

$R = LatestWrite(last\_update)$

である．

マスタノードは以上の解析を行い，結果である R を各プロセスに通知する．受け取ったプロセスは R の中から自身が書き込みノードである Region を検索し，ホームノードに転送する．

3.1.4 高速モード

ブロックの 2 回目以降の実行は高速モードで実行される．このモードの役割は，ブロック実行前に当該実行ブロックで必要となる共有メモリ領域の一貫性維持操作を行うことである．

一貫性維持のために FDSM は実行ブロックの先頭で通信を行う．このときに使用される通信集合である CommunicationSet S は以下のように解析される．

まず，アクセスパターンをマスタノードに集めたうえで，当実行ブロックの前回の実行から現在までの最新の書き込み LW を解析する．

$LW = LatestWrite(CB.TimeStamp)$

ここに CB は，現在実行中のブロックである．また，TimeStamp は解析終了後に現在時刻に更新するため，ここでは前回実行時の時刻が保存されている．

この RegionSet LW の要素である Region l と，ブロックの読み込みアクセスパターン RAR の要素である Region r から，Communication c を求める演算 $c = l \odot r$ を次のように定義する．

$l = (a_1, p_1, t_1)$

$r = (a_2, p_2, t_2)$

として

$$l \odot r = \begin{cases} (a_1 \cap a_2, p_1, p_2) & (p_1 \neq p_2 \text{ かつ } a_1 \cap a_2 \neq \phi) \\ \phi & (\text{上以外}) \end{cases}$$

この演算を用いて，必要となる CommunicationSet S は

$$S = \bigcup_{l \in LW, r \in TB.RAA} (\{l \odot r\})$$

と求められる．マスタノードはこの CommunicationSet S を各プロセスに通知する．受け取ったプロセスは S 中から自身が Src である Communication を検索し，実際に通信を行う．

CommunicationSet S を求める処理は 2 回目の実行時のみ行われる．なぜなら Communication Set S は 3 回目以降の実行でも同じ集合だからである．

3.2 アクセスパターン解析

アクセスパターンの取得は FDSM において最も重要な操作である．本節ではそのアクセスパターン解析の手法について述べる．

3.2.1 粒　　度

プロセッサが提供するメモリ管理機構を使用すればページ単位でのアクセスパターンは容易に取得可能で

ある．しかしながらページ単位でのアクセスパターン情報では正確な通信集合を求めることはできないのでFDSMは1バイト単位でのアクセスパターンを取得する．ただし，後述するように1バイト単位でのアクセスパターンの取得にはオーバヘッドがともなうので，この粒度での解析は書き込みアクセスのみに対して適用し，読み込みアクセスについてはページ単位での解析とする．

3.2.2 読み込みアクセスパターンの解析

FDSMはループの実行前に全ページをアクセス禁止にする．共有メモリにアクセスがあるとpage faultが発生するため，例外ハンドラで読み込みのあったページを記録すればページ単位での読み込みアクセスパターンが取得できる．

3.2.3 書き込みアクセスパターンの解析

読み込み同様，FDSMはループの実行前に共有メモリを書き込み禁止にすることで，書き込み時にpage faultを発生させる．そして例外ハンドラではアクセスのあったアドレスを記録する．本来ならこの後，例外ハンドラはアクセスのあったページを書き込み可能にしなくてはならない．さもないと，例外ハンドラからの復帰直後に再実行されたインストラクションが再び例外を起こし実行が進まなくなる．しかしながら例外ハンドラがページを書き込み可能にした場合，後に同じページに対して行われる書き込みアクセスをとらえることは不可能となり，1バイト単位でのアクセスパターン解析が不可能となる．

そこでFDSMでは例外ハンドラにインストラクション・エミュレーション機能を持たせ，例外を起こしたインストラクションを実行する．これにより例外ハンドラからの復帰後に例外の原因となったインストラクションを再実行する必要がなくなるため，アクセスのあったページの書き込み禁止属性を保持することが可能となる．

3.2.4 インストラクション・エミュレーション

FDSMにおけるインストラクション・エミュレータの役割は，共有メモリに対する書き込みアクセスを例外を起こしたインストラクションに代わって実行することである．我々はその手法をIntelの386アーキテクチャ上で開発した．インストラクションエミュレーションは以下に示す6ステップで実現される．

Step 1：Fetch 例外が発生した際，UNIX系OSのカーネルではプログラムカウンタの値を決められた場所に保存している．その値は再開時のインストラクションのアドレスであり，page faultの場合は例外を起こしたインストラクションを指している．したがって，このアドレスから命令を読めば例外を起こしたインストラクションを知ることが可能である．

Step 2：Decode Step 1で得たインストラクションを解析し，インストラクションの種類，オペランドレジスタ，命令長などを得る．

Step 3：Read オペランドが整数レジスタの場合，例外発生時にレジスタの値はメモリに保存されているのでその値を読み込む．オペランドが浮動小数点レジスタの場合，値はメモリには退避されず，プロセッサ内のレジスタに残ったままであるが，適切な命令を発行することで読み込みが可能である．

Step 4：Execute インストラクションが値の移動であればこのステップでは何も行わない．メモリの値のインクリメント命令など，何らかの演算が必要な場合はそれを行う．

Step 5：Write このステップで実際に共有メモリへの書き込みを行う．書き込み先のアドレスはオペランドレジスタの値を見ることによって取得可能であるが，より簡単には，例外の原因となった書き込み禁止アドレスを調べればよい．IA32の場合，このアドレスはプロセッサの決められたレジスタに格納されている．

得られたアドレスはユーザ空間のアドレスであるが，FDSMでは共有メモリ空間はカーネルからもマップされているため，ユーザ空間のアドレスをカーネル仮想アドレスに変換し，それを用いて共有メモリに書き込みを行っている．

Step 6：Increment PC 上述したようにメモリに退避されたプログラムカウンタは実行再開のアドレスであるので，このアドレスを書き換えれば再開のポイントをずらすことが可能である．例外を起こしたインストラクションの長さはデコード時に判明しているので，FDSMは退避されているプログラムカウンタの値にそのインストラクション長を加える．これにより例外を起こしたインストラクションは復帰直後に再実行されることはなく，次のインストラクションから実行が再開される．

以上の手順により例外ハンドラが例外を起こしたインストラクションに代わってメモリアクセスを行うことが可能となる．FDSMの実装はIA32を用いているが，この手法自体は他のアーキテクチャでも使用可能であり，たとえばRISCプロセッサにおいてはstore命令をエミュレートすればよい．

4. 実　　装

FDSM はカーネルレベルのドライバとユーザレベルのライブラリから構成される．本章ではそれらの実装について述べる．

4.1 カーネルレベルドライバ

FDSM は Linux 2.4.21 を使用して実装されている．共有メモリ空間は Linux のキャラクタデバイスとして提供されており，ユーザはデバイスファイルを mmap() することにより共有メモリ空間へのアクセスが可能となる．またこのデバイスは実際の計算に使用する共有メモリ空間を提供するほか，アクセスパターンなどの情報も ioctl() システムコールまたは mmap() によるメモリ共有を通してユーザレベルライブラリに提供する．また，アクセスパターン解析においては page fault ハンドラが重要な役割を担う．これらは大部分がカーネルモジュールとして実装されているが，page fault ハンドラの一部はカーネル本体への修正が必要である．

以下，デバイスドライバが行う重要な操作である，ページのアクセス権の管理とインストラクション・エミュレーションを含む page fault の処理について述べる．

4.1.1 アクセス権の管理

ドライバは共有メモリ領域の各ページについて，そのページテーブルへのポインタを持ち，必要に応じてページテーブルを変更できるようにしている．アクセス権を変更するのは，アクセスパターン解析開始および終了時，または page fault 時である．前者はユーザレベルライブラリから，ioctl() システムコールを通じて依頼される．後者はアクセスパターン解析中，ページに対しての初めての読み込みアクセス発生時に，例外ハンドラがページを読み込み可能に変更する．

4.1.2 Page Fault の処理

通常の Linux カーネルで発生する page fault には主に以下のような原因があり，カーネルは原因に応じて適切に対処をしている．

- デマンド・ページング
- コピー・オン・ライト
- アクセス違反

FDSM ドライバではこれらに加え，共有メモリ空間で発生した page fault については以下の原因も調査する．

(1)　アクセスパターン解析中の書き込み操作
(2)　アクセスパターン解析中の読み込み操作

ページに対する最初の page fault のときは，(1) あるいは (2) のいずれであっても FDSM はまず，ホームノードから最新のページをコピーする．そして 2 回目以降も含め，つねに (1) のときはインストラクションのエミュレーションを行いアクセスされたアドレスを記録する．(2) の場合はアクセスされたページを記録し，ページの読み込みを許可する．

表 1　提供される API
Table 1　Provided API's.

API	説明
fdsm_init	初期化
fdsm_finalize	終了処理
fdsm_alloc	共有メモリ領域の確保
fdsm_free	共有メモリ領域の解放
fdsm_before_loop	アクセスパターン解析の開始，通信
fdsm_after_loop	アクセスパターン解析の終了
fdsm_lock	ロックの取得
fdsm_unlock	ロックの解放
fdsm_reduce	OpenMP のリダクションに使用

4.2 ユーザレベルライブラリ

ユーザレベルライブラリは**表 1** に示すような API をアプリケーションに提供するほか，通信集合の計算とバリア同期を行う．

通信集合の計算は各ブロック 2 回目の実行の最初に行われる．このとき，すべてのノードはマスタノードにアクセスパターンを送信し，マスタノードが通信集合を算出する．そしてそれぞれのノードに必要な通信集合を知らせる．

FDSM は SCore Cluster System Software [16] 上で動作し，通信には Myrinet [10] を用いる．Myrinet の通信ライブラリは PM/Myrinet [18] であり，リモートメモリアクセス，システムコールを用いないユーザレベル通信をサポートしている．ループの 2 回目以降に必要な通信では，求められた通信集合に従って，必要な部分を PM/Myrinet のユーザレベル，リモートメモリライトを用いて読み込みノードに転送している．

5. OpenMP との融合

FDSM はユーザが直接 API を使用することによっても使用可能であるが，OpenMP を使用すればより容易に使用が可能となる．この場合は Omni/OpenMP [20] コンパイラが生成した C 言語のコードをコンパイルし，FDSM ライブラリとリンクする．現時点では #pragma omp for で示されたループはすべて FDSM 動作の仮定を満たすものと見なされる．

Omni/OpenMP コンパイラは変数の共有メモリへの配置を行ったうえで**図 1** から**図 2** のような変換を行う．この変換後のコードにおいて_ompc_default_sched()

```
/* 外側のループ内でアクセスパターンは一定 */
for (i = 0; i < IMAX; i++) {
  #pragma omp for
  for (j=1; j<=lastcol-firstcol+1; j++){
    z[j] = z[j] + alpha*p[j];
    r[j] = r[j] - alpha*q[j];
  }
}
```

図 1 OpenMP のソース
Fig. 1 OpenMP source.

```
/* 外側のループ内でアクセスパターンは一定 */
for (i = 0; i < IMAX; i++) {
{
  auto int j_41;
  auto int j_42;
  auto int j_43;
  (j_41)=(1);
  (j_42)=((((*__G_lastcol)-
           (*__G_firstcol))+(1))+(1));
  (j_43)=(1);
  _ompc_default_sched(&j_41,&j_42,&j_43);
  for((j)=(j_41);(j)<(j_42);(j)+=(j_43)){
      (*((z)+(j)))=((*((z)+(j)))+
         ((*__G__L_alpha_4)*(*((p)+(j)))));
      (*((r)+(j)))=((*((r)+(j)))-
         ((*__G__L_alpha_4)*(*((q)+(j)))));
  }
}
_ompc_barrier();
} /* end i loop */
```

図 2 変換後のコード
Fig. 2 Converted OpenMP program.

は仕事分散のためにループ変数の初期値と終了値を変更する関数であるが，FDSM ではこの本来の作業に加え，API 関数 fdsm_before_loop() を呼び出している．また_ompc_barrier() でも本来のバリア同期に加え fdsm_after_loop() を呼び出す．

この OpenMP ライブラリに対する変更により，コンパイラに対する変更を行うことなく OpenMP プログラムを FDSM で動作させることを実現している．

6. 性能評価

Xeon プロセッサ搭載の PC クラスタを使用して，FDSM の評価を行う．評価に使用したクラスタのハードウェア構成を**表 2** に示す．

各ノードの OS は FDSM 用に修正した Linux 2.4.21 であり，すべてのノードが SCore Cluster System Software の計算ノードとなっている．

表 2 クラスタのスペック
Table 2 Specification of the PC cluster.

# of nodes	8
CPU	Intel Xeon Processor 2.8 GHz
Cache	512 KB
Chipset	Intel E7501
Memory	2 GB
Network	Myrinet 1.25 GBits/sec (Lanai 4.3, SAN ケーブル) 1000BASE-T (Intel 82546EB)
Ethernet Switch	Dell Power Connect 5224

6.1 節では姫野ベンチマークのサイズ XS を用いて，アプリケーション実行時に発生するページフォルトなどのオーバヘッドおよびバリア同期にかかるオーバヘッドを測定する．結果は他の DSM システム，SCASH [19] および JUMP [4] と比較する．システム全体の性能測定として，6.2 節では姫野ベンチマークのサイズ M，6.3 節では NAS Parallel Benchmarks 2.3 [1] OpenMP の C 言語バージョン[13] から CG のクラス B を用いて性能測定を行う．これらの結果は SCASH [19] および MPI と比較する．

SCASH は FDSM と同様，Myrinet が使用可能であり，通信性能の面で平等な比較が可能であるため比較対象として選択した．また JUMP が採用する Home-migrating プロトコルは，FDSM が対象とするループ内のアクセスパターンが一定であるアプリケーションに対して効果的に働くことが予想され，JUMP は他のシステム JIAJIA との比較においても，JIAJIA を上回ることが報告されている[4] ので，比較対象として選択した．

6.1 ページフォルトおよび同期によるオーバヘッドの測定

本節ではアプリケーション実行時に発生するページフォルトなどのオーバヘッド，およびバリア同期にかかるオーバヘッドを測定し，これを SCASH および JUMP と比較する．

6.1.1 測定方法

測定には姫野ベンチマークを用いる．JUMP では最小サイズ XS のみが動作し，またノード数についても 2 ノードでしか動作しなかったため，ここでの測定はサイズ XS を 2 ノードで行う．

このベンチマークでは 1 回のループの中にメモリアクセスを含む一連の浮動小数点演算と 1 回の同期操作が入る．ここではその浮動小数点演算などの時間と同期のそれぞれに要した時間を測定し，オーバヘッドを調べる．

表 3 遅延
Table 3 Short message RTT.

PM on Myrinet	UDP on Ethernet
16.1 μs	109.8 μs

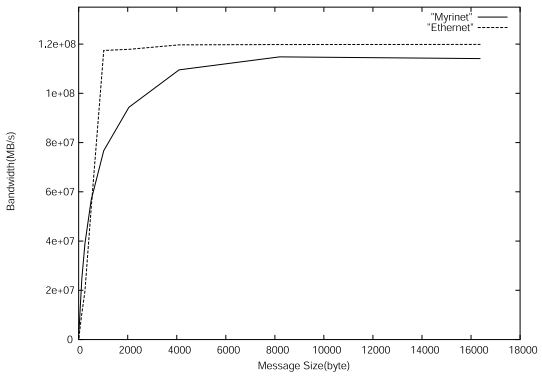

図 3 バンド幅
Fig. 3 Bandwidth.

FDSM と SCASH については Omni/OpenMP コンパイラが生成した C 言語のコードに，プロセッサのクロックレジスタの値を読み込むコードを加えて測定する．JUMP については JUMP API を使用して手動で並列化したものに，クロックレジスタの値を取得するコードを加えて測定する．

FDSM と SCASH ではネットワークに Myrinet を用いる．JUMP は Myrinet を使用するように設計されていないため Gigabit Ethernet を用いる．それぞれの Short message round trip time（RTT）とバンド幅は**表 3** と**図 3** のとおりである．Myrinet の通信ライブラリは PM/Myrinet であり，測定値は PM の性能測定ツール pmtest の結果である．Gigabit Ethernet の測定値は性能測定ツール netperf[11] による測定結果である．JUMP は UDP を使用しているので UDP の性能を測定した．

規格上のバンド幅は Gigabit Ethernet よりも Myrinet の方が広くなっている．しかし測定に使用した Lanai 4.3 を用いた Myrinet は，32 bit 33 MHz の PCI 接続であるためバス速度がネックとなる．このため，実験で使用した Myrinet のバンド幅は PCI-X 接続された Gigabit Ethernet よりも狭くなっている．

6.1.2 結果・検討

ベンチマークの計算部分に要した時間とメモリバリアに要した時間を**表 4** に示す．FDSM，SCASH，JUMP の結果は 2 ノードそれぞれでの測定値の平均値である．Serial の結果は比較のため，1 ノードで実行した測定値を 2 で割った値が示されている．

表 4 同期・計算時間
Table 4 Time for barrier or calculation.

(μsec)

	FDSM 高速モード	SCASH	JUMP	Serial
計算	2,279	3,198	23,740	2,167
同期	1,536	1,219	6,655	

計算部分に要した時間には浮動小数点演算そのものに要した時間と，メモリアクセスに要した時間が含まれ，さらに，一貫性管理のためにページフォルトが発生した場合はその処理のための時間も含まれている．メモリバリアに要した時間は一貫性維持のためのバリア命令に要した時間である．これにはノードをまたがるリダクション演算に要した時間も含まれる．なお，表中の FDSM の値は高速モードにおけるものであり，初回実行モードでの所要時間は計算部分に 113,200 μs，同期に 1,980 μs である．

FDSM の高速モードではページフォルトは発生しない．したがって，一貫性管理のためにページフォルトを使用する SCASH や JUMP と比較して高速であり，計算のみを行っているシリアルバージョンと同等の値となっている．

同期のコストは JUMP よりは低いものの，SCASH よりは若干高い結果となった．これは姫野ベンチマークの性質上，ループの初期段階では 0 の上に 0 を書き込むことが多く，Twin & Diff を用いた SCASH では Diff に反映されないのに対し，FDSM は書き込みがあった領域として通信対象に含まれてしまうためである．また，JUMP との差は通信遅延の差よりも大きくなっている．

6.2 姫野ベンチマーク

本節ではベンチマークアプリケーションによる評価として，姫野ベンチマークのサイズ M を用いて測定を行う．姫野ベンチマークは並列アプリケーションの典型である Jacobi 法を使用しているため，FDSM の評価に用いることとした．性能は SCASH および MPI と比較する．現在入手可能な JUMP にはバグがあり，測定を行うことができなかった．

6.2.1 測 定 方 法

姫野ベンチマークを OpenMP で並列化したプログラムを Omni/OpenMP コンパイラでコンパイルし FDSM，SCASH を用いて実行する．Omni/OpenMP コンパイラが生成する C 言語ソースは gcc 3.2.2 を使用してコンパイルする．この測定では $128 \times 128 \times 256$ の単精度浮動小数点数の配列を 14 個使用するサイズ M を用いる．また，比較のために同サイズの問題を解

表 5 姫野ベンチマークにおけるオーバヘッド
Table 5 Overhead in Himeno Benchmark.

1,000 回	# of PE		1	2	4	8
	Total		310.4 (sec)	179.39 (sec)	98.49 (sec)	68.35 (sec)
	初回実行モード	解析・計算	-	8.53 (4.75%)	4.44 (4.51%)	2.59 (3.79%)
		アップデート	-	1.27 (0.71%)	1.12 (1.14%)	1.3 (1.9%)
	高速モード	計算	-	163.57 (91.18%)	80.84 (82.08%)	51.5 (75.35%)
		通信	-	6.03 (3.36%)	12.1 (12.28%)	12.96 (18.96%)
100 回	Total		30.93 (sec)	26.29 (sec)	15.07 (sec)	10.39 (sec)
	初回実行モード	解析・計算	-	8.58 (32.63%)	4.46 (29.59%)	2.61 (25.1%)
		アップデート	-	1.26 (4.81%)	1.12 (7.41%)	1.29 (12.46%)
	高速モード	計算	-	15.7 (59.72%)	8.22 (54.57%)	5.11 (49.15%)
		通信	-	0.75 (2.84%)	1.27 (8.43%)	1.38 (13.3%)

く MPI 版プログラムの性能も測定する．

オリジナルの姫野ベンチマークは，最初に 3 回ループを実行した結果をもとに，実行時間が約 1 分となる反復回数を選ぶが，FDSM では初回の実行は遅くなるため，本測定では 1,000 回と 100 回の 2 通りに固定して測定を行う．以上の測定を 2，4，8 プロセッサで行う．

6.2.2 結果・検討

表 6 に絶対性能，**図 4** に 1,000 回ループ時の台数効果，**図 5** に 100 回ループ時の台数効果を示す．また**表 5** は要した時間の内訳である．初回実行モードについては，計算およびアクセスパターン解析に要した時間とホームへのアップデートに要した時間が示されており，高速モードについては計算に要した時間と通信に要した時間が示されている．

1,000 回ループでプロセッサ数が 2，4，8 のとき，それぞれ 1 台と比較して 1.73 倍，3.15 倍，4.54 倍の台数効果を得ている．これはそれぞれ SCASH の 1.32 倍，1.36 倍，1.22 倍の性能である．1,000 回ループの場合，アクセスパターン解析を行う初回実行モードでの計算に要した時間は，全体の 5%弱となっている．1,000 回ループのうちの 1 回の実行に 5%ほどの時間を要していることとなり，すべての書き込みで例外を発生する初回実行モードでは，高速モードの 50 倍ほどの時間がかかることが分かる．しかしながら初回実行モードでの計算および解析に要した時間は，4 ノード以上では通信によるオーバヘッドの 3 分の 1 以下まで小さくなっており，全体的な性能も SCASH を上回っている．したがって，このオーバヘッドは，2 回目以降の反復の高速化によって全体のパフォーマンスを向上させるためのものとして，許容できる範囲内である．

一方 100 回ループのときの台数効果は 2，4，8 台で 1.18 倍，2.06 倍，2.99 倍であり，これは SCASH とほぼ同等である．100 回ループでは実行時間の 30%ほどを初回実行モードに費やしており，通信によるオーバヘッドを大きく上回っている．このようにループ回数が少ないアプリケーションは FDSM には不向きである．

なお，MPI に対しては性能が劣る結果となっているが，この理由については 6.4 節で述べる．

表 6 姫野ベンチマークの結果
Table 6 Results of Himeno Benchmark.
(MFLOPS)

	1	2	4	8
FDSM（1,000 回）	441.7	764.3	1392	2005
SCASH（1,000 回）	441.7	577.2	1023	1647
MPI（1,000 回）	441.7	942.2	1760	3040
FDSM（100 回）	441.5	521.5	910.0	1319
SCASH（100 回）	441.5	522.7	848.9	1298
MPI（100 回）	441.5	942.3	1759	3038

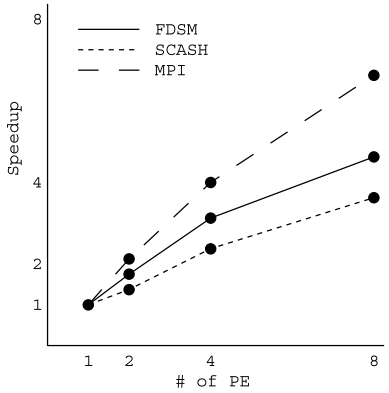

図 4 姫野ベンチマーク台数効果（1,000 回）
Fig. 4 Speedup of Himeno Benchmark (1,000 iterations).

6.3 CG

本節では NAS Parallel Benchmarks 2.3 OpenMP C 言語版から CG のクラス B を用いて性能評価を行う．CG はループ内での一定したアクセスパターンを持つものの，間接参照による不連続なアクセスパターンを持つので，分散共有メモリを含むメモリアクセス

表 7 CG におけるオーバヘッド
Table 7 Overhead in CG.

# of PE		1	2	4	8
Total		798.4 (sec)	425.6 (sec)	233.0 (sec)	140.2 (sec)
初回実行モード	解析・計算	-	3.208 (0.75%)	1.948 (0.84%)	0.937 (0.67%)
	アップデート	-	2.772 (0.65%)	2.652 (1.13%)	3.103 (2.21%)
高速モード	計算	-	405.4 (95.3%)	204.7 (87.8%)	102.8 (73.3%)
	通信	-	14.22 (3.34%)	23.70 (10.2%)	33.36 (23.8%)

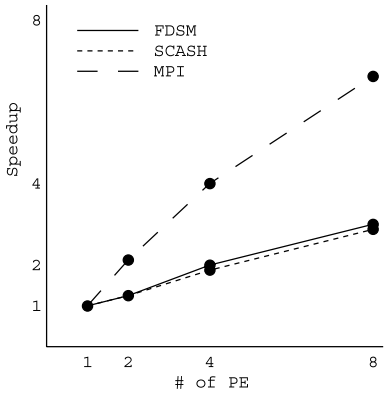

図 5 姫野ベンチマーク台数効果（100 回）
Fig. 5 Speedup of Himeno Benchmark (100 iterations).

に関する性能を測定するのには適したベンチマークである．結果は SCASH および MPI と比較を行う．JUMP については姫野ベンチマークのサイズ M と同様，測定を行うことができなかった．

6.3.1 測定方法

測定は FDSM，SCASH ともに NAS Parallel Benchmarks 2.3 OpenMP C 言語版の CG クラス B を Omni/OpenMP コンパイラでコンパイルし実行する．姫野ベンチマークと同じく Omni/OpenMP コンパイラが生成するソースのコンパイルには gcc 3.2.2 を用いる．MPI の性能は NAS Parallel Benchmarks 2.3 から MPI 版の CG クラス B を mpich-1.2.5 を用いて実行することにより測定する．MPI 版は Fortran を使用しておりコンパイラは gcc 3.2.2 中の g77 である．

オリジナルの CG では性能測定に先だち，時間計測を行わずに 1 回だけループの予備実行を行う．しかし FDSM で予備実行を行うと，オーバヘッドの大きい初回実行モードが時間計測の対象にならなず，公正な値を得ることができない．したがって，本測定ではこの予備実行は行わないようにベンチマークを変更した．この変更は SCASH および MPI の性能測定の際にも行っている．

表 8 CG の結果
Table 8 Results of CG.

(Mop/s)

	1	2	4	8
FDSM	68.38	128.51	234.72	389.86
SCASH	68.38	114.72	195.68	281.72
MPI	73.91	290.25	480.12	948.23

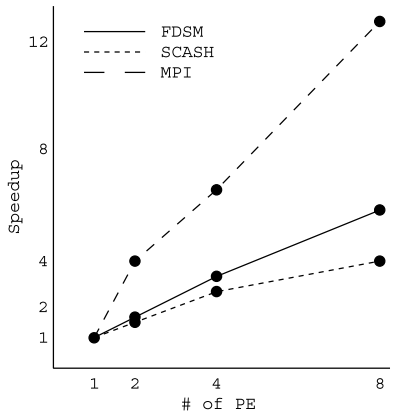

図 6 CG 台数効果
Fig. 6 Speedup of CG.

6.3.2 結果・検討

表 8 に CG（class B）の絶対性能，**図 6** に台数効果を示す．FDSM，SCASH の性能の基準となる 1 ノードの測定には，使用した OpenMP プログラムを OpenMP ディレクティブを無視してコンパイルしたものを用いている．MPI 版は Fortran で記述されているため，1 ノードの値にも Fortran で書かれたシリアルバージョンを用いている．また，姫野ベンチマークの結果と同様，**表 7** に全実行時間に占める各種時間の割合を示す．

ベンチマークの結果は，FDSM は 8 プロセッサでやや線形から離れるものの，おおむね直線的な台数効果を得ている．2，4，8 台での台数効果はそれぞれ，それぞれ 1.88 倍，3.43 倍，5.70 倍である．また SCASH との比較では 2，4，8 プロセッサでそれぞれ 1.12 倍，1.20 倍，1.38 倍の高速化を達成している．

CG のクラス B は 75 回のループの中に 25 回のネ

ストしたループが存在する構造となっており，子ループの中での初回実行モードは 1,875 回の反復中，最初の 1 回だけとなる．このため，初回実行モードでの解析および計算時間は 1%に満たないものとなっており，許容範囲内である．

MPI の性能は FDSM，SCASH よりも高くなっており，スーパリニアの台数効果を示している．この理由については次節で考察する．

6.4 FDSM の性能が MPI に及ばない理由

6.4.1 キャッシュの影響

これまでに示した姫野ベンチマークおよび CG の両方において，MPI は FDSM を上回る性能を出している．特に CG ではその差は大きく，MPI における通信を含めた総実行時間が，FDSM の計算のみの時間を下回っている．つまり，ここにはオーバヘッドの差ではない計算時間そのものの差が存在する．

浮動小数点演算のスピードが FDSM と MPI で異なることは考えられないので，速度差の原因はメモリアクセスのスピード，つまりキャッシュのミス率にある可能性が高い．そこで，プロセッサのパフォーマンスカウンタを用いてキャッシュのミス率を測定した．

キャッシュミス率の測定のためには，プロセッサのパフォーマンスカウンタを取得することを可能とする perctr[15] と呼ばれるカーネルパッチおよび perfctr が提供する機能を，アプリケーションから使用するためのライブラリ PCL[14] を使用した．

表 9 は CG における結果である．CG の MPI 版においては最大で 3%ほどしか 2 次キャッシュのミスがないのに対し，FDSM やシリアル版では 25%ほどの 2 次キャッシュミスがある．この事実は FDSM と MPI で計算時間に差が出ること，また MPI でスーパーリニアな台数効果が出ることとよく合致する．また，**表 10** が姫野ベンチマークにおける結果である．姫野ベンチマークでは L1 キャッシュの影響が大きかったので，L1 キャッシュのミス数が示されている（L1 のヒット数はライブラリが非対応でカウントができなかったのでミス率ではなくミス数を示す）．この結果でも MPI 版の 1 次キャッシュのミス数が FDSM でのミス数と比較して少なくなっていることが確認できる．

表 9 CG の L2 ミス率
Table 9 Cache miss ratio on CG.

実行パターン	ミス率 (%)	実行パターン	ミス率 (%)
Serial（C 言語）	25.6	Serial（Fortran）	25.8
FDSM 2CPU	26.63	MPI 2CPU	2.86
FDSM 4CPU	25.82	MPI 4CPU	3.09
FDSM 8CPU	24.86	MPI 8CPU	1.16

表 10 姫野ベンチマークの L1 ミス数
Table 10 Number of L1 miss on Himeno Benchmark.

実行パターン	ミス数 ($\times 10^9$)	実行パターン	ミス数 ($\times 10^9$)
Serial（C 言語）	41.13		
FDSM 2CPU	31.33	MPI 2CPU	5.109
FDSM 4CPU	16.40	MPI 4CPU	2.534
FDSM 8CPU	7.488	MPI 8CPU	2.093

6.4.2 通信量の影響

FDSM と MPI のオーバヘッドの差として，読み込みアクセスパターンをページ単位でしか取得しないことによる無駄な通信の発生がある．つまり，1 つのページに対し読み込みを行うプロセッサと書き込みを行うプロセッサが存在し，それらが重ならない領域にアクセスする場合である．この場合，読み込みプロセッサが使用しないメモリ領域のデータも送信されてしまう．

7. おわりに

本論文では FDSM と呼ばれる新しいソフトウェア分散共有メモリシステムを提案した．FDSM はループの中での共有メモリへのアクセスパターンが一定であるアプリケーションに対して，高速なソフトウェア分散共有メモリ機構を提供する．FDSM はループの 1 回目にアクセスパターンを解析し通信の必要な領域を求める．2 回目以降のループでは，1 回目で求められた領域のみを一貫性維持の対象とすることで，一貫性維持にかかるオーバヘッドの削減を実現する．この手法は Inspector/Executor 法を，コンパイラによらず，動的に行うことと等価である．

本論文はまた，インストラクションのエミュレーションによって 1 バイト単位でのアクセスパターンを動的に取得する手法，およびこれにより取得されたアクセスパターンを使用して，一貫性維持に必要な通信領域を算出するアルゴリズムを提案した．

姫野ベンチマークのサイズ M を用い，1,000 回のループを実行した性能評価では，2，4，8 プロセスで単一プロセスでの性能と比較し，それぞれ 1.73 倍，3.15 倍，4.54 倍の高速化を達成した．これは分散共有メモリの実現方法として一般的な Twin & Diff による方法を採用した SCASH と比較して，それぞれ 1.32 倍，1.36 倍，1.22 倍の性能である．また，NAS Parallel Benchmarks の CG クラス B を使用したベンチマークでは 2，4，8 プロセスでそれぞれ 1.88 倍，3.43 倍，5.70 倍の台数効果を得た．これは SCASH との比較でそれぞれ 1.12 倍，1.20 倍，1.38 倍の性能である．

性能評価で使用した Myrinet は 1.25 Gbps であるが，最新の Myrinet-2XP は 4 Gbps のバンド幅を持つ．したがって，FDSM を Myrinet-2XP 上で動作させれば，通信のオーバヘッドが少なくなり，性能向上が期待できる．

今後の課題としては姫野ベンチマークの 100 回ループで見たようなループ回数の少ないアプリケーションに対応する方式の検討があげられる．

謝辞 本研究の一部は，科学研究費補助金・基盤研究 A（1）14208026 の援助により行われた．

本論文を査読し多数の有益なコメントをくださいました査読者の方々に感謝いたします．

参 考 文 献

1) Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, D., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V. and Weeratunga, S.K.: The NAS Parallel Benchmarks, *The International Journal of Supercomputer Applications*, Vol.5, No.3, pp.63–73 (1991).
2) Bershad, B.N., Zekauskas, M.J. and Sawdon, W.A.: The Midway Distributed Shared Memory System, *Proc. 38th IEEE Int'l Computer Conf.* (*COMPCON Spring'93*), pp.528–537 (1993).
3) Carter, J.B., Bennett, J.K. and Zwaenepoel, W.: Implementation and Performance of Munin, *Proc. 13th ACM Symp. on Operating Systems Principles* (*SOS P'91*), pp.152–164 (1991).
4) Cheung, B.W.-L.: A Migrating-Home Protocol for Implementing Scope Consistency Model on a Cluster of Workstations.
5) Dwarkadas, S., Cox, A.L. and Zwaenepoel, W.: An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System, *Proc. 7th Symp. on Architectural Support for Programming Languages and Operating Systems* (*ASPLOSVII*), pp.186–197 (1996).
6) Hiranandani, S., Kennedy, K. and Tseng, C.-W.: Compiling Fortran D for MIMD distributed-memory machines, *Comm. ACM*, Vol.35, No.8, pp.66–80 (1992).
7) Hu, W., Shi, W. and Tang, Z.: JIAJIA: A Software DSM System Based on a New Cache Coherence Protocol, *HPCN Europe*, pp.463–472 (1999).
8) Iftode, L., Singh, J.P. and Li, K.: Scope Consistency: A Bridge between Release Consistency and Entry Consistency, *Proc. 8th ACM Annual Symp. on Parallel Algorithms and Architectures* (*SPAA'96*), pp.277–287 (1996).
9) Koelbel, C. and Mehrotra, P.: Compiling global name-space parallel loops for distributed execution, *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.4, pp.440–451 (1991).
10) Myrinet. http://www.myri.com
11) Netperf. http://www.netperf.org/
12) OpenMP. http://www.openmp.org
13) NAS Parallel Benchmarks 2.3 OpenMP C 言語版．http://phase.hpcc.jp/Omni/
14) PCL — The Performance Counter Library. http://www.fz-juelich.de/zam/PCL/
15) PerfCtr. http://user.it.uu.se/%7emikpe/linux/perfctr/
16) SCore. http://www.pccluster.org
17) Sharma, S.D., Ponnusamy, R., Moon, B., Hwang, Y.-S., Das, R. and Saltz, J.H.: Runtime and compile-time support for adaptive irregular problems, *Supercomputing*, pp.97–106 (1994).
18) Tezuka, H., Hori, A., Ishikawa, Y. and Sato, M.: PM: An Operating System Coordinated High Performance Communication Library, *HPCN Europe*, pp.708–717 (1997).
19) 原田 浩，手塚宏史，堀 敦史，住元真司，高橋俊行，石川 裕：Myrinet を用いた分散共有メモリにおけるメモリバリアの実装と評価，並列処理シンポジウム JSPP'99，pp.237–244, 情報処理学会 (1999).
20) 佐藤三久，原田 浩，長谷川篤史，石川 裕：Cluster-enabled OpenMP: ソフトウエア分散共有メモリシステム SCASH 上の OpenMP コンパイラ，並列処理シンポジウム JSPP'01，pp.15–22, 情報処理学会 (2001).

(平成 16 年 1 月 31 日受付)
(平成 16 年 6 月 17 日採録)

松葉 浩也（学生会員）

1980 年生．2003 年東京大学理学部情報科学科卒業．現在東京大学大学院情報理工学系研究科コンピュータ科学専攻修士課程在学中．クラスタコンピューティングに興味を持つ．

石川 裕（正会員）

1987 年慶應義塾大学大学院理工学研究科電気工学専攻博士課程修了．工学博士．同年電子技術総合研究所入所．1993 年技術研究組合新情報処理開発機構出向．2002 年より東京大学大学院情報理工学系研究科コンピュータ科学専攻助教授．クラスタ・グリッドシステムソフトウェア，高信頼システムソフトウェア開発技術，実時間分散システム，次世代高性能コンピュータシステム等に興味を持つ．