社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004−ARC−157 (11)
2004−HPC− 97 (11)
2004／3／2

IA64クラスタ上のソフトウェア分散共有メモリシステム

松葉浩也† 石川 裕†

本論文ではIA64アーキテクチャ用に，FDSM64と呼ばれる新しい分散共有メモリシステムを提案する．FDSM64はループの1回目でアクセスパターンを取得し，共有メモリ中で通信と必要とする領域を求める．2回目以降の実行時には，この領域のみを一貫性維持の対象とすることでオーバーヘッドを取り除き高速化を達成する．通信の必要な領域を正確に計算するためには，アプリケーションの共有メモリへの書き込みアクセスパターンを，1バイトの粒度で取得する必要がある．FDSM64はIA64アーキテクチャの持つ，広範囲にわたる領域を監視可能なデバッグレジスタを使用し，これを実現する．

The Software Distributed Shared Memory System on the IA64 Cluster

Hiroya Matsuba† and Yutaka Ishikawa†

This paper proposes a new software distributed shared system called FDSM64 for the IA64 architecture. FDSM64 analyzes the access pattern of an application at the first iteration of a loop and obtains the area called the communication set, which is the area in shared memory where the communication is required to maintain memory consistency. 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. To achieve this granularity, FDSM64 uses debug registers of the IA64 architecture, which may watch the large area of memory.

1. はじめに

分散共有メモリは，PCクラスタなどの分散メモリ型の並列計算機上で，共有メモリ型のプログラミングモデルを提供する方法の一つであり，長い研究の歴史がある[5]．しかし，一貫性維持にかかるオーバーヘッドが大きく，その性能は高くない．

我々は，分散共有メモリの一貫性維持にかかるオーバーヘッドを削減することを目標としたFDSMと呼ばれる分散共有メモリシステムを開発している．多くのアプリケーションではループを使ったアルゴリズムが使用され，ループ内での共有メモリへのアクセスパターンが一定であるという性質を持つ．FDSMはこの性質を利用し次のように一貫性維持を行う．まず，1回目のループ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する．このアクセスパターン情報を使用すれば，共有メモリ内で通信の必要な領域を計算することが可能であるため，ループの2回目以降では，1回目で求められた通信領域のみを一貫性維持の対象とする．

この手法はすでにIntel x86アーキテクチャ上では実装，評価されている[6),12)]．本論文ではIA64アーキテクチャにおける実装について述べる．IA64におけるFDSMをFDSM64と名付ける．

2. 設 計

2.1 IA64について

まず，FDSM64のターゲットとなるIA64[2)]について，その特徴をFDSM64で重要となる部分を中心に紹介する．

IA64はHP社とIntel社が共同で開発した64bitアーキテクチャであり，x86とはまったく異なったアーキテクチャである．その実装として，現在Itanium2プロセッサが販売されている．IA64は以下のようなプロセッサである．

- 基本的設計
  IA64はRISCに近いアーキテクチャであり，固定長の命令セットを持つ．
- 命令フォーマット
  IA64は128bit固定長の命令フォーマットを持つ．この128bitは5bitのテンプレートと呼ばれる部分と41bit長の命令3個から成る．テンプ

† 東京大学
The University of Tokyo

レートは3個の命令の種類を示し，その種類には，算術論理整数演算，整数非算術論理演算，メモリ処理，浮動小数点演算，分岐がある．また後述する命令グループの区切りを示すためにも使用される．41bit の命令には他の多くの RISC プロセッサと同様に，命令コード，オペランドレジスタなどが設定される．プログラムカウンタは3個の命令を含むこの 128bit 単位で進む．

- 明示的並列計算
  IA64 ではスーパースカラのように命令レベル並列性をハードウェアによって抽出するのではなく，ソフトウェアによって明示的に指定する．これは命令レベルの並列性の発見を，ハードウェアよりも大域的な情報を持つコンパイラに任せることができるというメリットを持つ．並列実行可能な命令群は命令グループと呼ばれ，アセンブリ上では ";;" によって区切られ，命令フォーマット上では，上述したテンプレートに命令グループの区切りを入れる仕組みがある．
- レジスタファイル
  IA64 は 128 本の 64bit 汎用レジスタ，128 本の 82bit 浮動小数点レジスタ，比較命令の結果を格納し分岐の際に使用する 64 本の 1bit プレディケートレジスタを持つ．また，プログラムカウンタなど用途の決まっている 64bit のアプリケーションレジスタが 128 本ある．
- レジスタスタック
  上述した 128 本の汎用レジスタは 32 本のスタティック部分（rg0 - rg31 と書く）と 96 本のリネーミング部分（rg32 - rg127 と書く）に分けられる．スタティック部分は常にアクセス可能なレジスタであり，アセンブリのニーモニックとしては r0 - r31 でアクセスされる（以下物理的なレジスタ番号を rg, アプリケーションから使用する際のレジスタ番号を r で書く）．一方リネーミング部分は各関数で alloc 命令により，使用を宣言した場合のみ使用可能となる．例えば最初の関数が3個のレジスタを alloc した場合，その関数内では rg32, rg33, rg34 がそれぞれ r32, r33, r34 としてアクセス可能となる．次にその関数がさらに別の関数を呼び出し，呼び出された関数が2個のレジスタを alloc すると，rg35, rg36 がそれぞれ r32, r33 としてアクセス可能となる．このようにリネーミング部分は物理的なレジスタ番号ではなく，alloc 命令により割り当てられたリネーミングされた番号でアクセスされる．従って単に r32 と書いても，それが物理的にどのレジスタであるかは一意に定まらない．
- メモリアクセス
  メモリアクセスは基本的にロード，ストアで行い，アドレッシングはレジスタ間接のみである．ロード，ストア以外でメモリアクセスを伴う命令は同期用の命令がある．

尚，これ以外に IA64 のデバッグ機能は FDSM64 にとって重要となる．これについては後述する．

2.2 FDSM64 の概要

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 FDSM64 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 回目以降のアクセス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 回目以降においてもこの領域は 1 回目と同じであるので，ループの 2 回目以降では 1 回目で求められた領域を通信するのみで一貫性維持操作は完了し，通信以外にかかるオーバーヘッドをなくすことが可能となる．

2.3 アクセスパターン解析

FDSM64 では得られたアクセスパターンを用いて通信の必要な領域を計算する．このためには共有メモリへの各領域に書き込みを行ったプロセスを正確に把握する必要があり，書き込みを行ったプロセスが一意に特定できない領域はいかに小さな領域であろうと存在してはならない．

つまり FDSM64 ではアクセスパターンを 1 バイトの粒度で解析する必要がある．以下では IA64 においてこの粒度での解析を実現する方法について述べる．

2.3.1 ページ保護機能による解析の問題点

アクセスパターン解析では，共有メモリに対するアクセスで何らかの例外を発生させ，例外ハンドラでアクセス場所を記録することになる．例外を発生させる方法としては，共有メモリ空間へのアクセスを禁止し，page fault を発生させる方法が考えられるが，この手法では 1 バイトの粒度を実現するに当たって以下の困難な問題が生じる．

Page fault の場合，例外ハンドラは例外の原因を取り除かなくてはならない．さもないと例外ハンドラか

らの復帰直後に，再び同じインストラクションが同じ原因で例外を起こし，実行が進まなくなるためである．アクセスパターン解析においては，page fault の原因を取り除くとはアクセス禁止属性を解除することである．しかし，アクセスパターン解析中にアクセス禁止属性を解除してしまうと，同じページに対する以後のアクセスでは page fault は発生しなくなるため，1 バイトの粒度で解析を行うことは不可能となる．

このようにアクセス保護違反での例外ではアクセスパターン解析には不都合な矛盾が生じるため，FDSM64 では別の例外を使用する．

2.3.2 デバッグ例外

メモリアクセスによって例外を起こす方法として，アクセス保護違反の他にデバッグ例外がある．これはアクセスされたメモリのアドレスがデバッグレジスタと呼ばれる特殊レジスタの値と一致した際に発生する例外であり，多くのプロセッサでデバッグ支援のために提供されている．

アクセス保護違反とは違い，デバッグ例外の例外ハンドラは，例外の原因を取り除く必要はない．デバッグ例外のハンドラを終了する際（多くの場合，デバッガでの操作を終え実行再開のコマンドを発行する際），引き続き同じアドレスを監視したまま実行が再開できるよう，復帰後の最初のインストラクションに限りデバッグ例外を発生させない機能がプロセッサに備わっているからである．

このデバッグレジスタに共有メモリ領域のアドレスを指定して監視させれば，共有メモリへのアクセスを捕らえることができ，例外ハンドラからの復帰も問題なく行える．FDSM64 はこの方式でアクセスパターン解析を行う．IA64 では単一の変数だけでなく，共有メモリ領域のような広い領域を監視対象とできる．次節ではこの機能を紹介する．

2.3.3 IA64 のデバッグレジスタ

IA64 のデバッグレジスタは，アドレスレジスタとマスクレジスタの 2 本 1 組で構成され，最低で 4 組持つよう定められている．アプリケーションレジスタ psr の db ビットが立っているとき，プロセッサは全メモリアクセスについて，アクセスのあったアドレスとマスクレジスタの論理積とアドレスレジスタの値を比較し，一致した場合に例外を送出する．

マスクレジスタは 56bit の長さを持つため，最大 2^{56} バイトもの広大な範囲を監視対象とすることができる．FDSM64 は 1 回目のループ時に共有メモリの全空間を監視対象にするよう 1 組のデバッグレジスタを設定する．これにより共有メモリへのアクセスで例外が発生するため，そのハンドラでアクセスアドレスを記録する．

また，例外ハンドラからの復帰の際にはアプリケーションレジスタ psr の dd ビットを on にすることにより，1 命令のみデバッグ例外を発生させないようにできる．これにより，復帰直後，先ほど例外を起こしたインストラクションが再びデバッグ例外を起こすことが防がれる．

2.3.4 アクセスアドレスの取得

IA64 ではデバッグレジスタに設定された条件と一致するアドレスにアクセスがあった際，例外を起こすことのみを求めており，アクセスされたアドレスは報告されない．従ってアクセスパターン解析には，アクセスのあったアドレスとその長さの解析を行う必要がある．

例外の際，オペレーティングシステムは例外を起こしたプログラムの状態を保存するため，プロセッサ内のレジスタの値をカーネル内の決められたメモリ空間に退避する．IA64 において，この退避はアプリケーションレジスタと汎用レジスタのスタティック部分が対象となる．以後「退避されたレジスタ」などと書くものは，このようにメモリに退避された実行中のアプリケーションのレジスタを指すこととする．

例外の際，例外を起こした命令のアドレスはプログラムカウンタであるアプリケーションレジスタ ip の値としてメモリに退避されている．またこのプログラムカウンタが指す 3 個の命令のうち，例外の原因となった命令は，退避されたアプリケーションレジスタ psr の ri ビットの値を調べることにより判別可能である．以上の情報を用いれば，例外を起こしたインストラクションを一意に特定することが可能であり，その種類により書き込みの長さ（1，2，4，8 バイト），そのオペランドにより書き込み先アドレスを調べることが可能である．

IA64 のメモリに対するアドレッシングは，レジスタ間接のみであるので，アクセスのあったメモリアドレスはオペランドレジスタの値を調べることとなる．オペランドが汎用レジスタのスタティック部分である場合，このレジスタの値は退避されたレジスタに含まれている．従ってこれを読み出すことによりメモリアクセスの番地が取得可能である．

一方，オペランドが汎用レジスタのリネーミング部分であった場合は値の取得方法が異なる．汎用レジスタのリネーミング部分は例外の際，Register stack engine と呼ばれるハードウェアによってプロセス空間の決まった場所にコピーされる．このコピーは物理的

なレジスタの並び順でコピーされるが，メモリアクセスのオペランドレジスタは，リネーミングされたレジスタ番号で指定されている．従ってリネーミングされたレジスタ番号を，物理的なレジスタ番号に変換する必要があるが，これは退避されたアプリケーションレジスタ cfm の値を参照することによって可能となる．

以上により，メモリアクセスの行われた番地とその長さが判明するため，それを記録することによりアクセスパターンの解析が可能となる．

2.4 FDSM64 の実行手順

以下ではアクセスパターンを取得するタイミング，またその使用方法を含め，FDSM64 が共有メモリ空間を提供する方法について述べる．

2.4.1 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ー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

2.4.2 同期のタイミング

FDSM64 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．いずれの場合においても，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．

2.4.3 初回実行モード

各ブロックの初回実行時，FDSM64 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前節で述べた方法を使用する．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，ホームノードは一貫性の取れた状態のページを準備する．また，各プロセスは全ページをアクセス禁止にし，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．

一貫性の取れたページをホームに準備するために FDSM64 は前回にホームの一貫性を取って以来，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の「最新書き込みの解析」を行う．この解析は各プロセスの各ブロックにおけるアクセスパターンをマスタープロセスと呼ばれる 1 つのプロセスに集めた上で，マスタープロセスが実行する．

この解析を行うことにより，マスタープロセスは共有メモリ内の各領域について，どのプロセスが最新の値を持っているかを知る．マスタープロセスはこの結果を各プロセスに通知し，受け取った各プロセスは，マスタープロセスから受け取った情報を基に，自身が持つ共有メモリ領域の値をホームノードに送る．

2.4.4 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用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．

まず，ブロックが初めて高速モードで実行されるとき，FDSM64 は一貫性維持のために必要な通信，つまり通信集合を求める．この通信集合は以下の条件をすべて満たす共有メモリ内の領域である．

(1) 当該ブロックで読み込みアクセスが発生する
(2) 当該ブロックの前回の実行から現在までに書き込みが発生している
(3) (1) の領域と (2) の領域が重複する
(4) (1) を行うプロセスと (2) を行うプロセスが異なる

マスターノードは以上の条件を満たす領域を解析することで，通信集合を求め，それを全プロセスに通知する．これを受け取った各プロセスは，通信集合のうち，自身が書き込みを行った領域について，その値を読み込みを行うプロセスに直接転送する．

この通信集合は一度求めておけば繰り返し使用可能である．従って高速モードの 2 回目以降，つまりループの 3 回目以降では再び同じ解析を行う必要はない．

3. OpenMP との融合

FDSM64 はユーザーが直接 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[13] コンパイラが生成した C 言語のコードをコンパイルし，FDSM64 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 FDSM64 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上で図 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() 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，FDSM64 ではこの本来の作業

```
#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

```
{
  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();
```

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

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 FDSM64 で動作させることを実現している．

4. 実　　装

アクセスパターン解析にはデバッグ例外を使用するが，この例外を捕らえる方法は 2 種類存在する．カーネル内で捕らえる方法と ptrace システムコールにより監視を行う方法である．

カーネル内で捕らえる場合，カーネル内に存在するデバッグ例外のハンドラを変更し，FDSM64 によるデバッグ例外時にはそのハンドラがアクセスアドレスを記録するよう書き換える．

一方で ptrace による監視では，親プロセスが ptrace システムコールにより子プロセスの監視を行う．子プロセスがデバッグ例外を発生させた際，Linux カーネルは子プロセスを停止させ，監視を行っている親プロセスに例外の発生を伝える．これを受け取った親プロセスが ptrace システムコールにより，停止している子プロセスの退避されたレジスタやメモリ領域を読み込めば，子プロセスがアクセスを行ったアドレスの取得が可能である．

カーネル内で例外を捕らえる場合，ptrace と比較してコンテキストスイッチのオーバーヘッドは少なくなるものの，この手法はカーネルソースの修正を必要とする．従って FDSM64 はカーネルに修正の必要がない，ptrace による方法でアクセスパターン解析を行う．

実際の FDSM64 の実装においては，SCore Cluster System Software[8] を用い，ネットワークに Myrinet-XP[7]，通信ライブラリに PM/Myrinet[10] を使用する．

表 1 に提供される予定の API を示す．

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

5. 関 連 研 究

分散共有メモリシステムの実装レベルにはハードウェアによるサポート，オペレーティングシステムのカーネルによるサポート，ユーザーレベルライブラリのみでの実装など多くの種類がある．このうちユーザーレベルライブラリのみによる実装では，アーキテクチャには強く依存しない実装となる可能性が高く，IA64 向きに開発されたものでなくとも，大きな変更なく IA64 での動作が可能であることが推測される．このようにユーザーレベルライブラリのみで実現されている分散共有メモリシステムには，SCASH[11] や TreadMarks[4] がある．

共有メモリ空間を提供する手法に関しては，FDSM64 と似たアプローチとして Inspector/Executor 法[9] が挙げられる．この手法では，コンパイラにより Inspector と呼ばれるアクセスパターン解析専用のコードを生成し，それによりアプリケーションのアクセスパターンを取得する．そして Executor と呼ばれるコードが

Inspector で収集された情報を利用して実際の計算を行う．この手法の利点は，いかなるアクセスパターンを持つアプリケーションでも実行可能であることだが，反面，Inspector の大きなオーバーヘッドが問題となる．また，この手法では特殊なコンパイラを必要とする．FDSM64 は Inspector/Executor 法を動的に行うこととほぼ同じである．ただし Inspector とは異なりアクセスパターン解析中も実際の計算を行う．また，FDSM64 では特殊なコンパイラは必要ない．

実行中に収集した情報を使用するという点では，producer-push 法 [3] も FDSM64 と共通点がある．producer-push 法は過去に必要であった通信を利用して，次に同じ場所を実行した際に必要となる通信を予測する．この手法はソフトウェア分散共有メモリで広く使用されている Twin & Diff による手法 [1] の拡張であるが，FDSM64 では Twin & Diff は用いていない．

また，OpenMP をサポートするソフトウェア分散共有メモリとしては前述の SCASH[11] や TreadMarks[4] が挙げられる．

6. おわりに

本論文では FDSM64 と呼ばれる IA64 上での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．FDSM64 は広範囲を監視することのできる IA64 のデバッグレジスタを使用することにより，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 1 バイトの粒度で取得し，通信の必要な領域を求める．

ループを使用したアルゴリズムの多くは，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という性質を持つため，ループの 1 回目でアクセスパターンを取得し，通信の必要な領域を求めておくことにより，ループの 2 回目以降では 1 回目で求めた通信の必要な領域のみを一貫性維持の対象とすれば良く，一貫性維持にかかるオーバーヘッドが削減される．

現在，本手法を Linux 上で実装中である．

参考文献

1) J.B. Carter, J.K. Bennett, and W.Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOS P'91)*, pp. 152–164, October 1991.
2) Intel Itanium Architecture Software Developer's manual. http://www.intel.com/design/itanium2/.
3) Sven Karlsson and Mats Brorsson. Producer-push - a protocol enhancement to page- based software distributed shared memory systems. In *ICPP 1999*, 1999.
4) P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pp. 115–131, January 1994.
5) K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, Vol. II, pp. 94–101, August 1988.
6) Hiroya Matsuba and Yutaka Ishikawa. OpenMP on the FDSM software distributed shared memory. In *Proceedings of the Fifth European Workshop on OpenMP (EWOMP '03)*, pp. 71 – 78, September 2003.
7) http://www.myri.com.
8) SCore. http://www.pccluster.org.
9) Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel H. Saltz. Run-time and compile-time support for adaptive irregular problems. In *Supercomputing*, pp. 97–106, 1994.
10) Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiros hi Harada, and Yutaka Ishikawa. PM2: High performance communication middleware for heterogeneous network environments. In *SC '00, IEEE*, 2000.
11) 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、情報処理学会, pp. 237 – 244, 1999.
12) 松葉浩也, 石川裕. 動的アクセスパターン解析による分散共有メモリ．ＡＣＳ論文誌 第 7 号, 先端的計算基盤システムシンポジウム SACSIS 2004 投稿中.
13) 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、情報処理学会, pp. 15–22, 2001.