Cluster-enabled OpenMP：ソフトウェア分散共有メモリシステム SCASH 上の OpenMP コンパイラ

佐藤三久[†,☆] 原田浩[†]
長谷川篤史[††] 石川裕[†]

クラスタにおいて，透過的に OpenMP プログラムをコンパイル，実行できるようにする “cluster-enabled” OpenMP コンパイラを開発した．クラスタ上で，共有メモリ機能をユーザレベルライブラリとして提供する，ページ保護機能を用いたソフトウェア分散共有メモリシステム SCASH に対して OpenMP プログラムをコンパイルする．コンパイラは，共有されるデータを実行時に割り当てるようにプログラムを変換する．OpenMP の API を拡張して，クラスタの各ノードへのデータのマッピングとデータマッピングに対応してループのイタレーションをスケジューリングする指示文を加えた．いくつかのベンチマークで性能評価を行い，SCASH においてはデータマッピングは性能に大きく影響することが分かった．さらに，NAS parallel benchmark の SP，BT について，OpenMP により並列化し，拡張した指示文によって性能を改善することができた．

Cluster-enabled OpenMP: An OpenMP Compiler for Software Distributed Shared Memory System SCASH

Mitsuhisa Sato,[†,☆] Hiroshi Harada,[†] Atsushi Hasegawa[††] and Yutaka Ishikawa[†]

In this paper, we present an implementation of a “cluster-enabled” OpenMP compiler for a page-based software distributed shared memory system, SCASH on a cluster of PCs, which allows OpenMP programs to run transparently in a distributed memory environment. The compiler transforms OpenMP programs into parallel programs using SCASH so that shared global variables are allocated at run time in a shared address space of SCASH. A set of directives is added to specify data mapping and a loop scheduling method which schedules iterations onto threads associated with the data mapping. Our experimental results show that the data mapping gives a great impact on the performance of OpenMP programs in the software distributed memory. The performance of BT and SP, in NAS parallel benchmark program suite parallelized by OpenMP was improved by our extended directives.

1. はじめに

本稿では，ソフトウェア分散共有メモリシステム SCASH 向けの OpenMP コンパイラを開発したので，その実装と性能について報告する．

マイクロプロセッサが高性能化するにつれて，ワークステーションや PC などをネットワークで結合するクラスタシステムが並列処理のプラットフォームの一般的な形態となってきている．このような分散メモリ型のマルチプロセッサでは，並列プログラミングに MPI や PVM などのメッセージ通信ライブラリを用いるのが一般的である．分散メモリ型の並列システムは，構築が容易かつ安価で，スケーラブルなシステムを構成できるのが利点であるが，このシステムを使うためにはメッセージ通信でプログラムを構成しなくてはならないため，プログラミングが複雑になり，プログラミングのコストが高いことが指摘されている．

最近，共有メモリマルチプロセッサ上で逐次プログラムを並列化するプログラミングインタフェースとして，OpenMP [1)]が提案され，注目を集めている．OpenMP では，C や Fortran など従来のプログラミング言語に指示文（C では pragma）を用いて並列記述を行う．OpenMP は，これまで各社独自仕様であった，コンパイラに対する並列化指示を共通化し，可搬

† 新情報処理開発機構
Real World Computing Partnership
†† NEC 情報システムズ
NEC Informatec Systems, Ltd.
☆ 現在，筑波大学
Presently with University of Tsukuba

性の高い並列プログラム開発を可能とすることを目的としている．並列実行ループ，同期と排他制御などの並列実行制御の指示文として指定し，並列プログラムと逐次プログラムを同一ソースで管理することができる．共有メモリマルチプロセッサにおいては，従来は POSIX Thread などのスレッドライブラリで並列プログラミングを行わなくてはならなかったが，これを簡便に行うことができ，並列化のコストを大幅に低減できる．我々は，すでに様々な共有メモリマルチプロセッサシステムに移植性の高い OpenMP コンパイラ Omni OpenMP コンパイラシステム[2]を開発している．

本研究の目的は，OpenMP を共有メモリ型マルチプロセッサだけではなく，分散メモリ型マルチプロセッサのプログラミングに用いようというものである．OpenMP を分散メモリ型マルチプロセッサに用いるための 1 つの方法は，分散メモリシステム上でソフトウェアにより仮想的な共有メモリを提供する，ソフトウェア分散共有メモリシステム（Software Distributed Shared Memory，SDSM）を OpenMP のターゲットのレイヤとして用いることである．SCASH[6),11)]は，並列オペレーティングシステム SCore 上で提供されている，ページ管理機構を利用した SDSM システムである．

ほとんどの SDSM システムでは，プロセッサ間でアドレス空間の一部分のみ共有される．SCASH では，実行時に SCASH のライブラリ関数で確保されたアドレス空間を複数のプロセッサで共有することができる．プログラム内であらかじめ静的に宣言された変数は個々のプロセッサでのみアクセスされる．共有空間を動的にしか確保できない，このようなモデルを “shmem モデル” と呼ぶことにする．たとえば，unix の shmem システムコール（mmap システムコールを用いて実装されることもある）を用いて記述した共有メモリプログラムもこのモデルの一例である．この shmem モデルでは，すべての共有変数は並列実行の前に，実行時に割り当てなくてはならない．我々は，Omni OpenMP コンパイラシステムを用いて，OpenMP プログラムを shmem モデルの並列プログラムに変換する OpenMP コンパイラを開発した．コンパイラは，共有される変数への参照を検出し，その変数に対して実行時に割り当てられる領域へのポインタによる間接参照に置き換える．これにより，ユーザは共有メモリ向けに記述した OpenMP プログラムを変更なしに，SCASH で実行できる．

次章では本研究の背景として，Omni OpenMP コンパイラとソフトウェア分散共有メモリシステム SCASH の概略について述べる．3 章において，SCASH 向けの OpenMP プログラムの変換と実行時システム，拡張した OpenMP 機能について述べ，4 章において，性能評価の実験結果について報告する．5 章において，関連研究について触れ，6 章において，結論・課題について述べる．

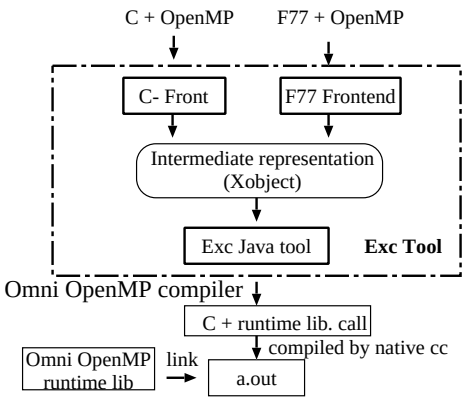

図 1 Omni OpenMP コンパイラシステム
Fig. 1 Overview of Omni OpenMP compiler system.

2. 背　　景

2.1 Omni OpenMP コンパイラシステム

我々は，OpenMP を並列化インタフェースとした SMP マシン上の並列処理環境，Omni OpenMP コンパイラとその実行時ライブラリ[2),9)]を開発した．**図 1** にその構成を示す．

OpenMP コンパイラは，OpenMP プログラムを入力としてライブラリ呼び出しを含む並列 C プログラムに変換するトランスレータである．システムは，入力言語に対するフロントエンドと OpenMP に対する変換を行う Omni Exc ツールキットからなる．フロントエンド部が入力プログラムを Omni の中間表現である Xobject へと変換する．入力言語は C と FORTRAN77 に対応しており，それぞれ C-front と F-front がフロントエンドである．この中間表現（Xobject）は，変数の型情報，グローバルな宣言情報，および実行文を保持する構文木の 3 つの情報を含んでいる．Exc ツールキットは，java で記述されたクラスライブラリであり，OpenMP の構文を含むプログラムの解析や変換，C への変換を行うことができる．構文木のノードが java のオブジェクトとして表現されているなど，高レベルな変換を簡単に実装できるように設計されている．OpenMP の変換についても，このツールキットの一部分として実装されており，プログラムの解析情

報を用いて，OpenMP の指示に基づいて入力プログラムを実行時ライブラリ呼び出しを含む並列プログラムに変換し，C のプログラムとして出力する．

以上，述べた手順により変換されたプログラムは，Omni の実行時ライブラリとリンクすることで，並列実行を行う．この際に，様々な計算機に容易に移植することができることを考慮して設計，開発を行った．SMP 向けには SUN や Linux をはじめ，POSIX スレッドを持つプラットフォーム対応している．

2.2 ソフトウェア分散共有メモリシステム SCASH

ここでは，SCASH システムの概要について述べる．詳細については，文献 6)，11) を参照していただきたい．

SCASH は，Myrinet 上に低通信遅延かつ高通信帯域幅を提供する高速通信ライブラリ PM[10]を用いたソフトウェア分散共有メモリシステムである．オペレーティングシステムのメモリ管理機能を利用し，ユーザレベルのライブラリとして実現されている．

ソフトウェア分散共有メモリシステムでは，共有メモリ空間を実現するために，同じ領域を各プロセッサに持ち，共有メモリ空間の同じアドレスでは各プロセッサにおいて同じデータを参照する．SCASH では参照するデータのコピーを各プロセッサで持ち，同期点において一貫性の制御を行う．

共有メモリ領域の一貫性維持は，オペレーティングシステムが提供するページ単位で行われる．一貫性モデルとして ERC（Eager Release Consistency）[5]を採用し，その実装にはマルチプルライタプロトコル[4]を用いている．マルチプルライタプロトコルをサポートするために，参照するページのほかに，更新前のページのコピー（twin）を持ち，更新時にはその差分を計算し，更新された個所のみを転送する．さらに，ページ単位の一貫性維持プロトコルとして，ページの無効化を通知する無効化プロトコルと，ページデータを送信し，ページの更新を通知する更新プロトコルの双方を実装し，実行時に選択できる．本稿の OpenMP の実装には，無効化プロトコルのみを用いている．

以下に SCASH が提供している機能と API を示す．

- 共有メモリの初期化，割当て，開放
  共有メモリを全ノード上で，割当て，開放を行う．SCASH は共有メモリを全ノード上で等しいメモリ空間に割り当てる．共有メモリを割り当てるメモリアドレスは，初期化時に指定可能である．
- 同期機構
  SCASH はメモリバリアとロックの 2 つの同期機構を提供している．メモリバリアは，全ノードでバリア同期をとり，共有メモリの全内容が，全ノード上で同一の最新内容に更新されることを保証する同期機構である．ロックはブロッキングロックが提供される．ロックは分散ロックキューによって実現されている．
- 一貫性制御機構
  SCASH では，書き込み共有メモリデータのホームノードへの書き戻し，ホームノードからの読み込みなどの，一貫性維持機構の一部をライブラリとしてユーザに開放している．
- その他
  SCASH は，グローバルメモリのデータをブロードキャスト，ページ単位でのホームノードの指定などの機能を提供している．特に，局所性が高い共有メモリ領域に関しては，ホームノードを指定することによって通信量を削減し，実行性能を向上させることができる．

PM では，従来のメッセージパッシングによる通信のほかに，送信元のユーザ空間から，送信先のユーザ空間へ直接メモリ転送を行う，ゼロコピー通信（遠隔メモリ読み込み，遠隔メモリ書き込み）をサポートしている．遠隔メモリ読み込みによるページ転送では，ページ転送元ノードの計算を中断せずに，ページデータを得ることができる．また，PM のゼロコピー通信では，送信元から送信先へ直接メモリ転送を行うので，ページコピーのオーバヘッドは発生しない．PM のゼロコピー通信機能を用いたページ転送によって，転送元ノードの計算中断，および，ページデータをコピーするオーバヘッドの削減により，Myrinet などの高性能ネットワークの高帯域幅を活かした効率的なメモリバリアを実現している．なお，現在，PM ライブラリは Myrinet だけでなく，ギガビットイーサネットなどにも対応している．

3. SCASH 向け OpenMP プログラムの変換

3.1 shmem モデルへの共有メモリプログラムの変換

OpenMP プログラムでは，静的に宣言された大域変数は特別な指定がない限り共有される．しかし，SCASH では共有されるメモリ領域は実行時にしか割り当てることができない．

冒頭に述べたとおり，このような共有メモリ領域が動的に割り当てなくてはならない共有メモリモデルを shmem モデルと呼ぶことにする．shmem モデルに

対し OpenMP プログラムを変換するために，以下のようにすべての共有変数を実行時に割り当てるようにコードを変換する．

- すべての静的な大域変数の宣言を実行時に割り当てられる共有メモリ領域へのポインタ変数の宣言に変換する．
- 大域変数への参照を，このポインタ変数を用いた間接参照に変換する．
- コンパイル単位（ファイル）ごとに，そのコンパイル単位で宣言された変数を実行時に割り当てるための初期化関数を生成する．

たとえば，以下のコード：

```
double x; /* global variable declaration */
double a[100];/* global array declaration */
   ...
a[10] = x;
```

は，次のように変換される．

```
double *_G_x;/* indirect pointer to ‘‘x’’ */
double *_G_a;/* indirect pointer to ‘‘a’’ */
   ...
(_G_a)[10] = (*_G_x);
     /* reference through the pointers */

/* initialize function */
static __G_DATA_INIT(){
  _shm_data_init(&(_G_x),8,0);
  _shm_data_init(&(_G_a),80,0);
}
```

この概略を**図 2** に示す．

コンパイラにコンパイル単位ごとに生成された初期化関数は，ctor セクション（constructor section：C++において，クラス初期化関数が置かれるセクション）に置かれ，リンカによって集約され，main 関数が実行される前に実行されるようにしている．

SCASH においては，実行開始時に各プロセッサにおいて，この初期化が実行されるが，実際には共有変数に関するテーブルを生成，初期化するだけである．ユーザの main プログラム実行前に，マスタになるプロセッサ 0 において，このテーブルを使って，共有メモリが実際に割り当てられた後，そのアドレスをすべてのプロセッサに broadcast する．各プロセッサでは，それを受け取って，個々のプロセッサ内の共有変数へのポインタ変数を初期化する．

本方式は，動的にとらなくてはならない共有メモリシステムにおいて，コンパイラのコード変換により静的な共有変数を実現する方法であり，straightfoward な方法であるが，同様な制限を持つ共有メモリシステムにおいても有効な方法である．OpenMP の実装に必要な共有メモリを提供するためにオペレーティングシステムや標準ライブラリも変更する必要がない．他の応用例として，並列計算機 Cenju-4 においては分散共有メモリ機構はハードウェアによって提供されているが，実行時の API によって用いるようになっており，本方式を用いた OpenMP の実現が報告されている[13]．ユーザレベルのライブラリとして提供されることが多いソフトウェア共有メモリシステムでは，直接ライブラリを使うよりも共有データの割当てなどの繁雑なプログラミングを省くことができる利点がある．

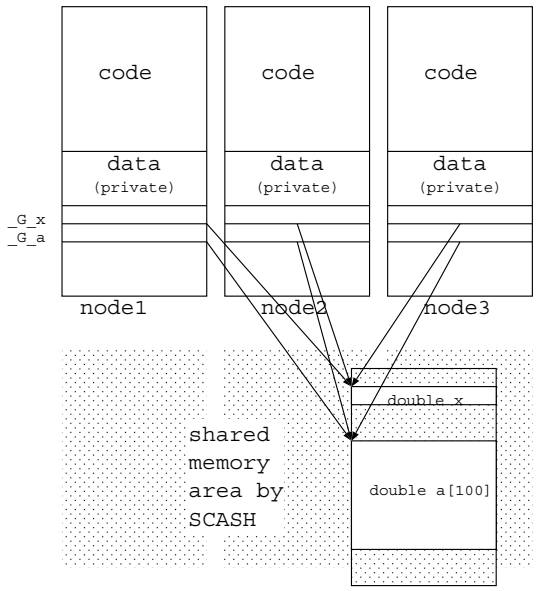

図 2 共有変数への間接ポインタ参照
Fig. 2 Indirect pointer reference to shared variables.

3.2 OpenMP 指示文の変換と実行時ライブラリ

OpenMP プログラムは，fork-join 型の並列プログラムである．SCASH においては，OpenMP スレッドは各プロセッサに 1 対 1 に対応している．ネストされた並列性はサポートしていない．

Omni OpenMP コンパイラでは ‘parallel’ 指示文によって指定された並列実行される並列リージョンを 1 つの関数にし，この関数を並列実行するコードを生成する．マスタプロセッサがその関数を引数として，実行時ライブラリ関数を呼び出し，マスタ以外のスレーブプロセッサで実行させる．スレーブプロセッサは，プログラム実行開始時に起動されており，この実行時ライブラリ関数による並列実行を待機している状態になっている．そして，スレッドで実行していた関数の

実行が終了した時点で，再びマスタ以外のスレーブスレッドは待機状態となる．

並列実行において，マスタの auto 変数が共有する必要がある場合には，並列実行する前に，共有メモリ領域にコピーされ，この領域が各プロセッサにおいて共有される．並列実行が終了すると，この領域が元の auto 変数にコピーされる．

OpenMP の指示文は，コンパイラによって，その指示文に応じた実行時ライブラリを含むコードに変換される．生成されるコードは，基本的には共有メモリに対するコードとほとんど変わらないが，実行時ルーチンでは SCASH のライブラリ関数を使って，効率的にプロセッサ間の同期，通信を行うように実装されている．

バリア同期は，SCASH のメモリバリア同期関数にコンパイルされる．この関数では，プロセッサの同期だけでなく，メモリ内容の一貫性操作を行う flush 操作も含まれる．ちなみに，メモリの一貫性操作がない場合のバリア同期の実行時間は，述べる実験に用いた PCC4 では，2 ノードで，70 μ秒，32 ノードは 360 μ 秒程度である．

SCASH では，OpenMP の実行環境を構築するにあたり共有変数の同期に対応した API を提供している．たとえば，OpenMP の flush 指示文に対しては，“指定された領域が更新されていれば，ホームノードに更新内容を反映させ，更新されていれば，ホームノードから最新の値を読み込む”API を提供しており，これを用いて flush 指示文の実装を行っている．

3.3 データマッピングとループスケジューリングのための OpenMP 拡張

SDSM システムでは，各ページのホームノードの割当てがプログラムの性能に大きく影響する．この影響は，ハードウェアで一貫性制御を行う NUMA システムよりも大きい．SCASH では，他のノードがホームノードになっているページをアクセスするとページフォルトが起き，そのページはホームノードから転送される．また，バリアポイントにおいて，ホームノードになっている他のノードにあるページを更新するために，変更されたメモリの内容が転送される．したがって，SCASH 上で性能を得るためには，できるだけ，各スレッドがアクセスするデータをスレッドの実行されるプロセッサのホームノードに一致させ，転送量を低減させる必要がある．OpenMP で提供されている指示文ではプログラマは各計算をスレッドに分割することはできるが，データを特定のメモリ領域にマッピングする機能はない．また，ループスケジューリングの指示句にも，イタレーションをデータのマッピングにあわせてスケジューリングする指示句はない．

そこで我々は，OpenMP の API を拡張し，データの配置とスレッドの割当てを行うための指示文を拡張した．データマッピング指示文は配列データのマッピングを指示するものである．この拡張は，HPF のものを参考にした．次に，Fortran と C でのデータマッピング指示文の例を示す：

Fortran:

```
      dimension A(100,200)
!$omn mapping(A(*,block))
```

C:

```
double A[200][100];
#pragma omni mapping(A[block][*])
```

この例では，2 次元配列の 2 次元目をブロック分割するマッピングを指示するものである．1 次元目のアスタリスク (*) は，それぞれの行の配列要素は同じノードに割り当てることを示す．`block` キーワードは，指定された次元をブロック分割することを示す．したがって，上の例では配列を連続したブロックに等分割し，同じノードにマッピングされることになる．また，サイクリックマッピングのために，`cyclic`(n) も用いることができる．

ただし，SCASH では，メモリの一貫性制御がページ単位でしか行われないため，ページ単位の粒度でしか，マッピングが行われない．もし，マッピングの粒度がページ単位よりも細かい場合には，マッピングは有効にならない．また，HPF のデータ分散指示とは異なり，各ノードでは同じアドレスにデータのコピーを持つ．ページに対し，ホームノードの “マッピング” を指定するだけであり，データをノードに分割する HPF の “分散” の指示とは異なる．

データのマッピングに加えて，配列のマッピングに合わせてループのイタレーションの割当てを行う指示句 “affinity” をループスケジューリングのための指示句に加えた．例を示す：

```
#pragma omp for schedule(affinity,a[i][*])
  for(i = 1; i < 99; i++)
   for(j = 0; j < 200; j++)
     a[i][j] = ...;
```

この例では，各イタレーションを `a[i][*]` を持つプロセッサに割り当てることを示す．

なお，OpenMP では単一レベルの並列性しかサポートしていないため，現在の実装では，マッピングやイタレーションのスケジューリングに対しては，1 つの次元のみに限っている．

表 1 RWP PC Cluster II（Pentium Pro 200 MHz，256 MB memory，Myrinet network，Linux）での実行時間（秒）と対逐次性能向上率
Table 1 Exectuion time (sec) and speedup of RWP PC Cluster II (Pentium Pro 200 MHz, 256 MB memory, Myrinet network, Linux).

No. of nodes	seq	2	4	8	16	32
lap/BLK	17.74(1)	14.30(1.24)	7.84(2.26)	4.69(3.78)	2.88(6.16)	1.79(9.91)
lap/RR	17.74(1)	49.39(—)	33.88(—)	20.15(—)	12.87(1.38)	10.15(1.75)
cg/BLK	83.79(1)	48.90(1.73)	29.49(2.84)	20.86(4.02)	18.08(4.63)	19.65(4.26)
cg/RR	83.79(1)	55.20(1.52)	33.88(2.47)	23.33(3.59)	18.83(4.44)	19.73(4.24)

3.4 SMP 上でのプログラムとの互換性

プログラマに対しては，我々の SDSM 上の OpenMP 環境は，以下のいくつかの例外を除いて，SMP での環境と互換性のある環境を提供する．

- クラスタなど分散メモリ環境では，入出力操作はそれぞれのノードで独立に行われる．あるノードで open されたファイルデスクリプタは他のノードでは使えない．
- C OpenMP プログラムでは，標準入出力ライブラリなど外部の関数ライブラリなどがリンクされる．通常のコンパイラでコンパイルされた外部関数で定義された変数を参照する場合には，OpenMP のプログラムでは threadprivate にしておかなくてはならない．
- 標準のライブラリ関数 malloc で割り当てられた動的なヒープ領域は，共有されない．共有するヒープ領域を確保するには，SCASH が提供する関数 `ompsm_galloc` を使わなくてはならない．
- スレッド数は，環境変数ではなく，SCASH プログラムを実行するための SCore のコマンド `scrun` で与えられる．

4. 性能評価

4.1 Laplace と NBP CG による予備評価

まず，RWC PC cluster II（pcc2）において，データマッピングの効果とスケーラビリティについて，予備的な評価を行った．pcc2 は，CPU は Pentium Pro 200 MHz，メモリ 256 MB のノードが，128 ノード，Myrinet によって結合されたシステムである．このうち，32 プロセッサを用いた．

ベンチマークとして，2 次元 5 点ステンシルの Jacobi 法による Laplace 方程式の解法 lap（サイズ 1024×1024，繰返し 50 回），C で記述された NPB1 の CG 法（クラス A）を用いた．前者のベンチマークは定型的なデータ参照を持つベンチマークとして，後者は不規則なデータ参照を持つベンチマークとして取り上げた．Omni/SCASH の backend コンパイラ，および，逐次版のコンパイラには egcs-1.1.2 を使用し，最適化オプションには，“-O3 -malign-double -funroll-loops” を指定した．結果を**表 1** に示す．なお，逐次版は OpenMP の指示文をすべて無視をしてコンパイルしたものであり，OpenMP によるオーバヘッドはいっさい含まれていない．

SCASH では，特別なホームの指定がない場合には，ページを round-robin にプロセッサに割り当てる．これを RR で示す．BLK は，配列に関して，プロセッサに等分割するブロック分割を指定した場合の実行時間である．ループに関しては，特別な指示は行っていないため，ブロックスケジューリングが行われている．

lap においては，ホームの割当てが性能に大きな影響を及ぼすことが分かる．これは，配列の参照・更新に規則性があるにもかかわらず，RR においてはこれが考慮されず，更新のたびに大きなトラフィックが生じ，性能低下するためである．BLK では，データのブロック分割のマッピングがデフォルトのループスケジューリングであるブロックスケジューリングとほぼマッチしているため，性能が良くなっている．スケーラビリティに関しては，この問題サイズでは 16 プロセッサ程度まで，スケールしているのが分かる．

一方，cg においては，配列の参照が基本的にランダムであり，lap に比べて差は小さい．cg では主要な計算は疎行列の行列ベクトル積である．行列は read-only であり，一度呼び込まれると再利用されるが，ベクトルの方はイタレーションごとに更新される．このベンチマークではベクトルの参照パターンはランダムであり，基本的に全対全通信を必要とする．この通信がスケーラビリティを制限していると思われる．

この評価結果をふまえて，現在の実装では，配列に関して特別の指定がない場合には，配列の最大の次元に関してブロック分割によるマッピングをしている．我々の OpenMP コンパイラでは並列ループに対して，特別の指定がない場合には，ブロックスケジューリングとしており，lap の結果で見たとおり，特別な指示がない場合，適当な性能が得られる場合が多い．

また，SDSM 向けのコンパイルでは，大域変数を実行時に割り当てて，間接ポインタで参照しているが，

これらのベンチマークで確認したところ，そのオーバヘッドは，数%と小さいことを付け加えておく．

4.2 NPB BT，SP による性能評価

我々は，NAS parallel benchmark（NPB）のバージョン 2.3 の逐次版を元に，この中のいくつかのベンチマークを OpenMP により並列化した．並列化においては，逐次コードに OpenMP の指示文を加えるのみにとどめ，元のコードはほとんど変更していない．この OpenMP コードを元に，実際にアプリケーションに近い BT と SP を用いて，拡張した指示文を使った性能改善の効果について評価した．なお，サイズはクラス A を用いた．

4.2.1 OpenMP による並列化の概要

SP，BT の各処理は 3 重の do ループから構成されている．OpenMP が，ネスト並列をサポートしていないため，配列の最大次元に対応したループで並列化した．デフォルトでは配列データは全体をブロック分割し，ホームノードを割り当てるため，ループ内でのデータのアクセスパターンが，ホームノード割当てと一致することが多くなり，自ノード上にあるデータが使用されることが多くなる．ただし，データに依存関係が存在する箇所に関しては，その次の次元に対応したループで並列化している．ループ間で依存がない場合には，明示的に nowait を指定し，バリアによる同期のデータのオーバヘッドを削減するようにしている．これは，バリアにおいて前のループで書き込んだデータが次のループで同じプロセッサで使われるとしても，ホームノードに書き戻されるが，それを抑制する．

SCASH では共有されるメモリ領域に実メモリが割り当てなくてはならないため☆，使用できる共有メモリのサイズに制限があり，実験に使用した計算機では，クラス A の BT では，すべての変数を共有メモリ上に置くことができなかった．また，不必要な変数を共有メモリ上に配置することは，同期オーバヘッドが増大することから，データの同期が必要な変数のみ共有メモリ上に置き，初期化後，書き換えられない変数など，同期を必要としない変数は threadprivate 変数とした．これらの変数は，参照される前に，copyin 指示節を用いて，各スレッドの分散メモリ上に値をコピーするように指示文を追加している．この結果，SP で使用する共有変数は 80 MB のデータ領域のうち 38 MB，BT で使用する共有変数は 304 MB データ領域のうち 34 MB となっている．

☆ SCASH では，pin-down により共有領域に割り当てるメモリ量を制限している．

4.2.2 拡張指示文による最適化

データはデフォルトではデータ全体をブロック分割して割り当てられるが，SP においては，配列データの最大次元に対応した do ループが存在しないものがある．このような配列データに対しては mapping を指定して，並列化された do ループに合わせたホームノードの割当てが行われるように最適化した．たとえば，SP の主要な配列である u は，以下のような 4 次元配列であり，

```
dimension u(0:IMAX/2*2, 0:JMAX/2*2,
            0:KMAX/2*2,  5)
```

ループは，内側の 3 次元をアクセスする．この場合には，

```
!$omn mapping(u(*,*,block,*))
```

と，宣言し，最外側のループでアクセスする最大次元（3 番目）の次元で分割した．BT では，ほとんどの配列は 3 次元配列であり，最外側の次元の分割でループ分割に対応できる．

また，これらのベンチマークには，do ループの対象が配列全体でなく，配列の一部がループ範囲外に存在するループがある．このようなループに対しては，affinity スケジューリングを指定し，do ループの配列変数のホームノードの割当てに合ったループスケジューリングを行うようにした．たとえば，SP の一部である以下のコードでは，インデックスが 0 から始まる u に対して，インデックス 1 から配列がアクセスされているが，これを affinity スケジューリングを行うことにより，配列の mapping に適合したループのスケジューリングを行う．

```
  do m = 1, 5
!$omp do schedule(affinity, rhs(*,*,k,*))
   do  k = 1, grid_points(3)-2
    do  j = 1, grid_points(2)-2
     do  i = 1, grid_points(1)-2
      u(i,j,k,m)=u(i,j,k,m)+rhs(i,j,k,m)
     end do
    end do
   end do
!$omp end do nowait
  end do
```

さらに，affinity スケジューリングにより，各スレッドがアクセスするデータが決まるため，以下のように，スレッドがアクセスするデータが同じである場合，ループ範囲が異なる do ループでも，データの同期なしに連続して実行できる．

表 2 SCore クラスタ III のノードとシステム構成
Table 2 Specification of SCore cluster III.

CPU	:	Dual Pentium III 800 MHz
Chip Set	:	ServerWorks III LE
Memory	:	Registered SD-RAM 512 MB
NIC	:	Myrinet M2M-PCI64A-2, Intel Ether Express 100 Pro
OS	:	RedHat 6.2 Linux kernel-2.2.17

```
!$omp do schedule(affinity, array(i))
       do i=0, 100
         array(i) = array(i) + 1
       end do
!$omp end do nowait
!$omp do schedule(affinity, array(i))
       do i=1, 99
         array(i) = array(i) + 2
       end do
!$omp end do
```

これを利用して，SP，BT では，バリア同期の回数を削減し，同期によるオーバヘッドを削減した．

4.2.3 実験結果

性能評価には，最新のマイクロプロセッサの性能を反映するため，予備評価とは異なるクラスタを使用した．ノードとして，**表 2** に示した NEC Express5800/Rc-2 32 ノードを Myrinet で結合したクラスタ SCore クラスタ III を用いた．なお，各ノードは 2 プロセッサの SMP であるが，本実験では，各ノード，1 プロセッサを用いた．

図 3，**図 4** に，SP，BT の実行時間を示す．なお，Omni/SCASH の backend コンパイラ，および，逐次版のコンパイラには予備評価と同じく，egcs-1.1.2 を使用し，最適化オプションには，“-O3 -malign-double-funroll-loops” を指定した．

“serial” は，逐次の実行時間，“none” は OpenMP のみによって並列化したものの実行時間を示す．“mapping” は，mapping の指示文を加えたもの，“affinity” はさらに，ループを affinity スケジューリングにより最適化したものの実行時間である．比較のために，“mpi” にベンチマークとして標準的に用いられている MPI 版 NPB2.3 の実行時間を示す．

なお，1 プロセッサの実行時間は 1 プロセッサの SCASH 版のものであるが，バリア同期，コード変換のオーバヘッドを含んでいる．ただし，1 プロセッサで実行するときには複数プロセッサで実行する場合に必要となる一貫性管理のためのページフォルトなどは起きないようになっている．この結果，32 ノードで最大，SP では約 9.1 倍，BT では約 16.6 倍の性能向上が得られた．

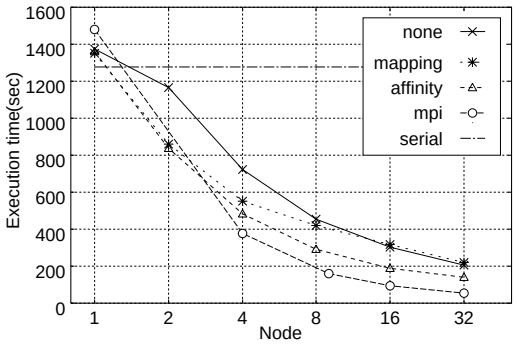

図 3 NPB SP の実行時間
Fig. 3 Execution time of NPB SP.

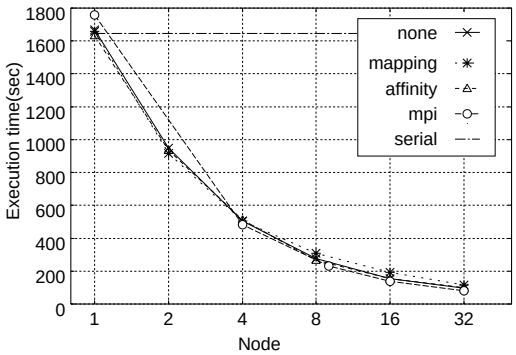

図 4 NPB BT の実行時間
Fig. 4 Execution time of NPB BT.

4.2.4 考察

SP では，affinity と mapping により，性能が改善されていることが分かる．**図 5**，**図 6** に，SP と BT におけるページの転送回数を示す．SP では，デフォルトのホームノードの配置がループスケジューリングと異なる配列変数が存在したため，mapping 指示文を指定することで，ページの転送回数が削減され，性能が改善されている．BT では，実行時間に対し相対的にページの転送回数が小さいため，どのケースでもそれほど差が出ていない．

16 ノード以上になるとスケーラビリティが制限されている原因の 1 つは，各ノード分割して割り当てられる次元のサイズが 65 であり，並列化されているループ長のほとんどが，同様に 65 以下と小さいためである．mapping において block mapping を指定している場合，現在の実装では “要素数/ノード数” を切り上げたサイズを用いている．そのため，たとえば，65 を 16 ノードの場合に分割する場合，サイズは 3 になっており，1/3 のノードに対してはホームノードが

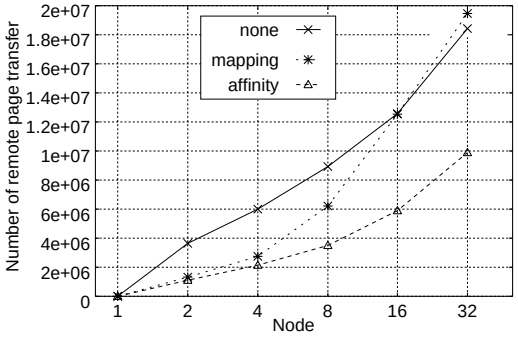

図 5 NPB SP のページ転送回数
Fig. 5 The number of remote page transfer in NPB SP.

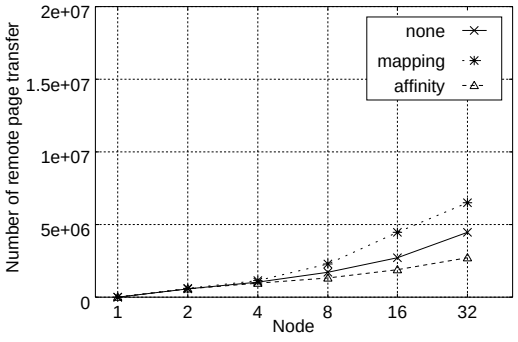

図 6 NPB BT のページ転送回数
Fig. 6 The number of remote page transfer in NPB BT.

割り当てられないことになってしまう．その結果，この配列を基準に affinity スケジューリングをしてしまうと，ホームノードが割り当てられないノードはループを実行しないため，性能が低下する．また，このプログラムではループ変数が 1 から 62 になるループが多くあるが，affinity スケジューリングしない場合には，各スレッドに割り当てられるループサイズは 2 になるものの，ホームノードが一致しないため，これも性能低下を招いてしまうことになる．このように，分割される次元サイズやループ長が小さい場合には，陽にブロックサイズを指定するなどのチューニングが必要であることが分かった．

16 ノード以下では，BT の方が SP よりもスケーラビリティが良いことが分かるが，これは BT の方が通信量に対しての計算量が多いためである．BT と SP は類似点が多いベンチマークである．どちらのコードも，3 次元空間における各ポイントにおける独立な方程式を解くものであるが，これらのデータは多次元配列に格納されている．3 次元空間に対応し，x，y，z 方向に対応した部分配列がそれぞれ，`x_solve`，`y_solve`，`z_solve` の 3 つのフェーズでアクセスされている．これらのプログラムを並列化する場合には，

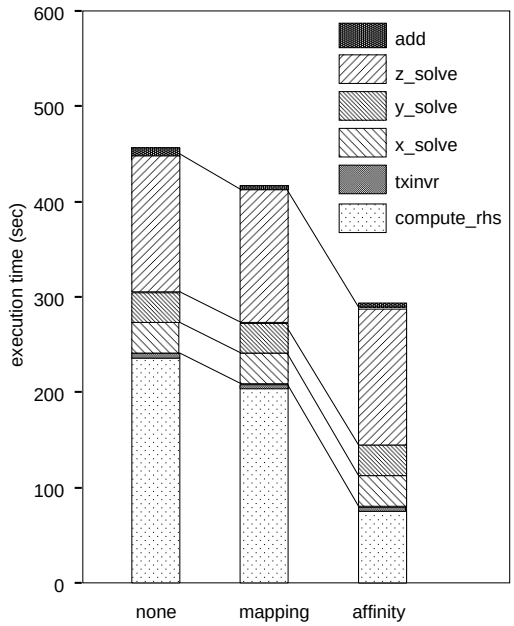

図 7 NPB SP の実行時間内訳（8 nodes）
Fig. 7 Break-down of execution time in NPB (8 nodes).

この配列を z に対応する次元（配列で外側の次元にあたる）でマッピングしている．このマッピングにより，`x_solve` と `y_solve` は，効率的に実行されるようになるが，`z_solve` では，スレッドがアクセスする範囲がノード全体にわたるようになってしまうため，大量の転送が必要になり，性能を改善することができない．これを解決するためには，データの再マッピングや転置などにより，`z_solve` においても，スレッドのアクセスパターンとデータマッピングがなるべく一致するようにしなくてはならないであろう．

図 7 に，SP を 8 ノードで実行した場合の主な関数別実行時間の内訳を示す．4.2.2 項に述べたとおり，まずは主要な配列を単なる最大次元で分割するのではなく，ループのアクセスに合わせた mapping をすることで，`compute_rhs` の実行時間が減少する．さらに，正確にループを mapping に合わせることにより，さらに実行時間が短くなることが分かる．x，y 方向のアクセスを行う `x_solve` と `y_solve` はほぼ効率的に実行されているが，やはり，次元のまたがる z 方向をアクセスする `z_solve` では時間がかかってしまうことが分かる．

4.2.5 NPB2.3 MPI 版との比較

参考のために，図 3，図 4 に NPB2.3 の MPI の実行時間を示した．NPB2.3 の MPI 版のプログラムでは，各プロセッサに 2 次元分割を行って分散配置している．本稿で用いたものは逐次版を OpenMP を用い

て並列化したもので，1次元で分割したものに相当し，MPI版の性能と一概に比較することができない．

もちろん，逐次版プログラムを2次元分割に対応するように書き換えて，メッセージ通信に近付けるようにして，OpenMPプログラムの評価することも可能であるが，人手により，このような書き換えを行い最適化することは逐次プログラムを変更を最小にすることができるOpenMPの利点も失うことになる．

全項に述べたとおり，BTの場合は通信に比べて計算量が多く，通信の影響が少ないため，本方式でもMPIとそれほど違わない性能が得られている．SPの場合は，計算量に比べて，通信の比率が大きく，さらに，z方向のアクセスが1次元分割ではとびとびになり非効率になってしまうため，プロセッサが多くになるにつれ，その差は大きくなってしまう．

科学技術計算にはアプリケーションには本評価プログラムに見られるような3次元の配列の計算が多くあり，いまのところ，ネストしないループレベルの並列化では1次元のみの配列の分割にしか対応できず，2次元以上の分割に対応できないのはOpenMPの欠点の1つといえる．

4.3 問 題 点

前節のNPBのSPやBTで指摘したとおり，プログラム中で異なったアクセスパターンを持つ場合には，単一のデータマッピングではカバーできないケースがある．解決方法としては，次のような方法が考えられる．

- 再マッピング．データのホームノードをデータのアクセスパターンに適合するように変える．
- 再配置．マッピングを変えずに，データ自体を入れ換える．

SPとBTに関しては，`z_solve`のアクセスパターンに合わせてマッピングを指定しても，ページ単位の粒度以下になってしまうため，マッピングが無効になってしまう．再配置する場合には，インデックスの入れ換えなどの書き換えをコンパイラ，あるいは，人手で行わなくてはならない．この他の方法として，リモートのページのデータをアクセスする場合にはそのページを計算と並行にプリフェッチする方法があるが，データ転送量は変わらないため，ネットワーク容量を必要とする．

今回の評価では，32ノードまでのスケーラビリティが得られなかった．この根本的な要因は，SCASHなど共有メモリシステムでは，一貫性の粒度がオペレーティングシステムでサポートされているページサイズに限られていることである．大規模なクラスタシステムになり，ノード数を増やすに従って，1つのノードあたりのデータ量が少なくなり，最適にノードに割り当てたとしても，ページ境界をまたがるデータが増えてくる．ページが共有される割合も多くなり，更新のコストも高くなる．

SCASHは，ページの一貫性制御にERC（Eager Release Consistency）を用いている．TreadMarks[3]など，他のSDSMでは，LRC（Lazy Release Consistency）を用いているものもあるが，本稿で取り上げた評価プログラムでは主に，ループとそれに引き続くバリア同期がほどんどであり，両者のバリアの実装の仕方に差がなければ，その差は少ないとみられる．LRCでは，実装によっては，管理の複雑さからオーバヘッドが大きくなる可能性がある．ただし，スレッドを個別に制御し，ロックなどで同期するデータを明示するプログラムではLRCのほうが優位な場合もあるであろう．しかし，OpenMPでのデータの同期のAPIとして提供されているflush指示文は，指定されたデータが更新されていた場合には共有メモリに反映し，他のプロセッサによって更新されていた場合にはその最新のデータを反映するというものである．この指示文では明示的なlock/unlockの操作を行わない．また，同期する変数を指定しない場合は共有メモリ全体を対象とする．そのために，LRCを用いたときにもOpenMPのflush操作に適したインタフェースを提供する必要がある．

大規模なクラスタシステムにおいて，十分な効果を出すためには問題を大きくする必要があるが，分散共有メモリにおいては同じアドレス空間にマップするため，32ビットアドレスでは不十分になってくる．また，SCASHではリモートメモリ通信のために共有するデータに対して，ノードごとに実メモリが必要であり，結果として大量のメモリを必要とすることになる．さらに，SCASHでは参照するためのページメモリのほか，一貫性プロトコルの効率化のために，更新する前のページ（twin）を持っており，さらに多くのメモリを必要とすることになっている．共有メモリ領域とその複製をメモリアクセスに応じてon demmandで確保するようにすべきであるが，それは高速化との兼ね合いであり，これからの課題である．

これらの問題の1つの解決方法としては，複数のCPUを持つSMPをノードとすることである．これにより，ノード数をおさえ，プロセッサを増し，相対的に必要なメモリ容量を増やすことが可能になる．

大きなデータを扱うためのもう1つのアプローチとして，HPFなどに見られるようように配列を分散化さ

せて管理することが考えられるが，これはOpenMPの仕様の大きな変更，あるいはメッセージ通信コードに変換するコンパイラが必要になる．これについては共有メモリシステムを前提とする本稿の範囲を越えるものである．

大規模のクラスタに適用する場合は上に述べたように問題があるものの，評価結果で示したベンチマーク規模は十分実用的なものであり，16ノード程度の中規模のクラスタには，本システムで十分に実用に適用できるといえる．また，OpenMPプログラムが対象としている共有メモリシステムの多くは，この規模であり，比較的コストが低いクラスタシステムで，OpenMPプログラムが実行できるメリットは大きいと思われる．将来，64ビットのアドレス空間が用いられるようになり，メモリのコスト低下，大容量化により，メモリ容量にかかわる以上の問題点は緩和されるとものと期待したい．

5. 関連研究

これまで，OpenMP のソフトウェア分散共有メモリシステムへの実装が発表されている．Lu ら[8]は，TreadMarks[3]ソフトウェア分散共有メモリ向けのOpenMPコンパイラについて発表している．その実装の詳細については述べられていないが，SDSMを効率的に用いるためにOpenMPの仕様を変更・拡張を提案し，評価している．この変更は性能を得るためには良い点があるが，OpenMPの目的の1つである並列ソフトウェアの可搬性を著しく阻害する可能性がある．本稿で提案したOpenMPの実装においてはfull setのOpenMPの仕様がサポートされており，共有メモリマルチプロセッサシステムでのOpenMPのプログラムを変更せずに実行できる．

また，Hu ら[7]は，同様に TreadMarks を用いたSMPクラスタ向けのOpenMPコンパイラについて述べている．現在，本システムはSMPクラスタについては対応していないが，これから対応する予定である．

我々は，別プロジェクトとしてページベースのソフトウェア分散共有メモリを用いずに，コンパイラにより一貫性制御と通信のコードをコード中に挿入，そのコードを最適化するコンパイラ[12]の開発を行っている．

6. 結論・課題

本稿では，ソフトウェア分散共有メモリシステムSCASH向けのOpenMPコンパイラを設計，開発した．ユーザレベルでサポートされるSDSMシステムに対して，コンパイラはすべての大域データを実行時に割り当て，参照できるようにコードを変換する．我々のシステムを用いることにより，共有メモリマルチプロセッサにおいて実行されるOpenMPプログラムを変更なしにそのまま，分散メモリシステム上で実行できる．これにより，並列化，並列プログラミングのコストを大幅に軽減できる．NAS Parallel benchmarkをはじめとするいくつかのベンチマークを用いて評価し，ある程度の性能向上を得ることができた．

SCASHにおいては，各データのホームの割当てが大きく性能に影響を及ぼす．各ページのホームノードを決定するデータマッピングとループのイタレーションのスレッドの割当てをプログラマから制御するために，OpenMPを拡張し，その効果を評価した．プロセッサ数が多くなるにつれて，性能向上を得るためには正確にデータマッピングとイタレーションを制御してやることが必要である．

BTやSPのベンチマークプログラムにおいては，ループ間でデータのアクセスパターンが異なり，単一のデータマッピングでは性能向上を得られない部分があることも分かった．この場合には，データの再マッピングや再配置が必要である．このためのコンパイラのサポートは，これからの課題である．

また，現在，クラスタは，dualプロセッサシステムなど，複数のCPUを持つノードを用いるのが一般的になりつつある．本稿では1CPUノードのクラスタでの評価に限ったが，クラスタにおいて，SMPノードでSCASHを用いる実装については，ほぼ，設計・実装が終了しており，評価中である．

謝辞 本研究を遂行するにあたり，議論，協力いただいた新情報処理開発機構SCore開発チームならびにOmniコンパイラ開発チームの皆様に感謝いたします．

参考文献

1) OpenMP ARB Home Page, http://www.openmp.org/.
2) Omni OpenMP Project, http://www.rwcp.or.jp/lab/pdperf/Omni.
3) Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W. and Zwaenepoel, W.: Treadmarks: Shared memory computing on networks of workstation, *IEEE Comput.*, Vol.29, No.2, pp.18–28 (1996).
4) Amza, C., Cox, A., Dwarkadas, S. and Zwaenepoel, W.: Software DSM Protocols that Adopt between Single Writer and Multiple Writer, *Proc. 3rd IEEE Sympo. on High-Performance Computer Architecture* (*HPCA-*

3), pp.261–271 (1997).
5) Gharachoroloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Henessy, J.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessor, *Proc.17th ISCA*, pp.15–26 (1990).
6) Harada, H., Ishikawa, Y., Hori, A., Tezuka, H., Sumimoto, S. and Takahashi, T.: Dynamic Home Node Reallocation on Software Distributed Shared Memory, *Proc.HPC Asia 2000*, Beijing, China, pp.158–163 (2000).
7) Hu, Y.C., Lu, H., Cox, A.L. and Zwaenepel, W.: OpenMP on Networks of SMPs, *Proc. 13th IPPS*, pp.302–310 (1999).
8) Lu, H., Hu, Y.C. and Zwaenepel, W.: OpenMP on Network of Workstations, *Proc. Supercomputing'98* (1998).
9) Sato, M., Satoh, S., Kusano, K. and Tanaka, Y.: Design of OpenMP Compiler for an SMP Cluster, *Proc. 1st European Workshop on OpenMP* (*EWOMP'99*), Lund, Sweden, pp.32–39 (1999).
10) Tezuka, H., Hori, A., Ishikawa, Y. and Sato, M.: PM: An Operating System Coordinated High Performance Communication Library, Lecture Note in Computer Science, *High-Performance Computing and Networking*, pp.708–717 (1997).
11) 原田，手塚，堀，住元，高橋，石川：Mryrinet 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 (1999).
12) 佐藤，草野，佐藤：OpenMP コンパイラにおけるメモリ一貫性制御の最適化，並列処理シンポジウム JSPP2000，pp.221–228 (2000).
13) 草野和寛，佐藤茂久，佐藤三久：Cenju-4 の分散共有メモリ機構を用いた Omni OpenMP コンパイラ，情報処理学会 HPC 研究会 (Oct. 2000).

(平成 13 年 2 月 8 日受付)
(平成 13 年 5 月 16 日採録)

佐藤　三久（正会員）

1959 年生．1982 年東京大学理学部情報科学科卒業．1986 年同大学院理学系研究科博士課程退学．同年新技術事業団後藤磁束量子情報プロジェクトに参加．1991 年通産省電子技術総合研究所入所．1996 年より，新情報処理開発機構つくば研究センターに出向．同機構並列分散システムパフォーマンス研究室室長．2001 年より筑波大学電子・情報工学系教授．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術等の研究に従事．日本応用数理学会会員．

原田　　浩（正会員）

1988 年東京理科大学理学部物理学科卒業．同年（株）ソフトウェア・リサーチ・アソシエイツ入社．1997 年より技術研究組合新情報処理開発機構研究員．現在に至る．オペレーティングシステム，並列・分散システム等に興味を持つ．ACM 会員．

長谷川篤史（正会員）

1994 年大阪電気通信大学大学院工学研究科卒業．同年，現，株式会社 NEC 情報システムズ入社．現在に至る．並列分散システムの業務に従事．

石川　　裕（正会員）

1987 年慶応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 年より技術研究組合新情報処理開発機構に出向中．並列分散システムソフトウェアつくば研究室室長．日本ソフトウェア科学会，ACM，IEEE 各会員．