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

2003－HPC－96　(16)
2003／10／16

タスク移動による通信最適化を目指した ソフトウェア分散共有メモリの設計と実装

立川　純[†]　小出　洋[†]

本稿では，動的負荷変動の生じる分散メモリシステム上での効率的な並列プログラミングシステムの実現を目的として現在開発中の，OpenMP 向けランタイムシステム (T-SDSM) の設計と実装手法について述べる．T-SDSM では，タスクとして動作する各 OpenMP スレッドが参照するデータをリモートからキャッシュして共有メモリ空間を実現する機能と共に，各タスクの参照アドレスを実行時に解析して，その結果に基づきタスクを配置する機能が実装されている．laplace プログラムによる評価では，実装上のオーバヘッドが大きくメモリ参照パターンにも基づいたタスク配置の効果を確認するに至らなかったが，クリティカルセクション等の逐次処理の実行に対してはタスク配置による有効性を確認した．

Design and Implementation of Software Distributed Shared Memory system that Realize the Communication Optimization by Task Moving

Jun Tachikawa [†] and Hiroshi Koide[†]

This paper describes that the design and implementation detail of runtime system for OpenMP 'T-SDSM'. T-SDSM is the software DSM system for heterogeneous computing environments such as PC-Cluster and Grid, and for distributed computing systems that have dynamic load change. T-SDSM has two functions. One is the data caching, and the other is the task moving. The data caching function loads the memory block accessed by the task from remote node, and many other SDSM system has this function. The task moving function move the task to home node that is owner of the memory block accessed by the task. The evaluation result of T-SDSM with laplace program don't show the effectiveness of communication optimization by the task moving function, because T-SDSM has a large overhead on task execution. However, the evaluation result with critical section show that the effectiveness by task moving.

1. はじめに

近年，計算機クラスタやグリッドなどの分散メモリシステムを用いた並列計算環境に関する研究が盛んに行われている．分散メモリシステム上での並列プログラミングには MPI に代表されるメッセージパッシングが用いられることが多いが，そのプログラミングコストの高さが指摘されており，よりプログラミングの容易な分散メモリシステム上で共有メモリ型のプログラムの実行を可能とするソフトウェア分散共有メモリ (SDSM) システムに関する研究が行われている．また，SDSM システムが提供する API を直接用いたプログラムだけでなく，文献 1)4) 等のように，コンパイラを実装して，共有メモリ型のプログラミングインターフェイスである OpenMP をサポートした研究にも注目が集まっている．

OpenMP では，C や Fortran 等の逐次の言語に対するディレクティブの挿入による段階的な並列化を念頭に置いており，メモリコンシステンシモデルとして緩い一貫性を採用している．そのため，メッセージパッシングのようにアプリケーションに強く依存した並列アルゴリズムの記述には向かない反面，コンパイラやランタイムシステムによる最適化の余地が高く，単一の OpenMP プログラムを様々なプラットフォーム上で効率よく実行できる性能可搬なプログラミングシステムとして期待できる．

そこで我々は，プロセッサやネットワークなどの各資源性能が不均質であるヘテロなクラスタ環境やグリッド環境，あるいは動的な負荷変動の発生し得る分散計算環境であっても，OpenMP 等の共有メモリ型インターフェイスを使用してプログラムを効率よく実行できるようにするための負荷分散機能を組み込んだランタイムシステムに関する研究を行っている．このランタイムシステムを実現する上で，まず負荷分散の単位をシステムに認識させ，負荷分散機能を組み込むフレームワークとしてタスク並列処理に着目し，そのフレームワーク上で動作する SDSM システムである T-SDSM(Task-based Software Distributed Shared Memory) を提案した[6]．T-SDSM では，タスクが参照するデータについて，そのデータを含むメモリブロックをホームノードからキャッシュして共有メモリ空間を実現する一般的な機能に加え，タスクが参照する共有メモリアドレスを，各タスクに対応する

† 九州工業大学 情報工学部 知能情報工学科
Department of Artificial Intelligence, Kyushu Institute of Technology

アドレス計算コードによって求め，その結果に基づきタスクを配置する機能が実装されている．

これまでの実装では T-SDSM システム独自の記述を行う必要があったため，今回 OpenMP コンパイラを考慮したランタイムライブラリを実装し，OpenMP で記述された laplace 変換プログラム，および EPCC OpenMP Microbenchmark[3] (以下 microbench) を用いて初期評価を行った．評価には OpenMP プログラムを今回実装したライブラリを含んだコードに手動で変換することで行った．評価の結果，laplace では実装上のオーバヘッドが大きく十分な性能を確認することはできなかったが，microbench での critical の評価ではタスク移動による効果を確認できた．

2. 特　　徴

T-SDSM システムの主な特徴を以下にまとめる．

- 設計方針
  まず負荷分散の単位としてのタスクと，そのスケジューラを実現するためのフレームワーク (タスク並列処理フレームワーク) を実装し，そのフレームワークの機能を用いてメモリ管理機能，タスク管理機能等を実装し，OpenMP 向けのランタイムシステムとして構築する．
- MPI のみのユーザレベル実装
  タスク並列処理フレームワークは，シグナルやマルチスレッド等を用いない MPI プログラムとして実装している．このためフレームワーク自身を MPI 以外の通信関数を用いて再実装することでシステム全体の移植が可能である．なお，MPICH-G2，Stampi[2] 等のグリッド環境向けの MPI を用いることでグリッド環境での評価等を視野に入れている．
- コンパイラ支援 SDSM
  コンパイラによって，OpenMP プログラム中のディレクティブで指示された領域をスケジューリングの対象となるコード (タスクコード) として取り出すと共に，ランタイムシステムが実行時の最適化に用いる「アドレス計算コード」を各タスクコードを走査して生成する．現在，コンパイラは未実装である．
- アドレス計算コードを用いた共有メモリ参照検知
  タスク実行の前処理として各タスクに対応するアドレス計算コードを実行し，タスクコードが参照する共有メモリアドレスを検知する．これによって OS のページ管理機構を用いずに共有メモリへの参照を検知できる．また，タスクの実行前に参照アドレスが分かるため，一つのタスク内で広い領域への参照がなされる場合にプリキャッシュ等のメモリの最適化が可能となる．
- メモリ参照パターンに応じたタスク移動による通信最適化
  アドレス計算コードからの情報に基づき，メモリブロックを管理するホームノードに対してタスクを移動させることで，メモリ一貫性維持のための通信が削減され，全体の性能向上に貢献する場合が考えられる．ただし，タスク移動によって良い結果を得られるか否かは，実行時の移動先のキャッシュの状況やタスクのメモリ参照パターン等実行時の情報に基づき適切に決定する必要がある．

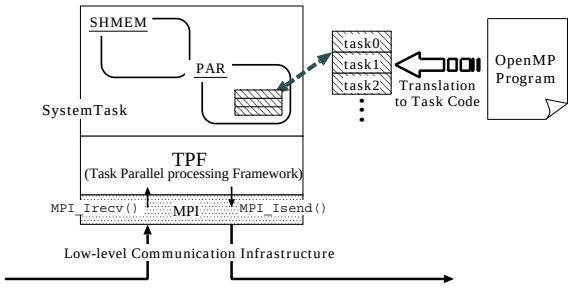

図 1　T-SDSM システムの概観
Fig. 1　Overview of Task-based Distributed Shared Memory System

3. T-SDSM システムの設計と実装

T-SDSM システムは，図 1 に示すように，大きくタスク並列処理フレームワークとその上に実装したシステムタスク群 (PAR/SHMEM) から構成されている．入力 OpenMP プログラムは今回実装したライブラリの呼び出しを含んだタスクの集合に変換され，PAR システムタスクによって実行管理される．

以降では，3.1 節でタスク並列処理フレームワークの構成と MPI による実装方法について述べる．続く 3.2 節で，動作概要を説明するための準備としてコンパイラによるコード変換について述べ，3.3 節で T-SDSM のタスク管理，メモリ管理を中心に動作の詳細について説明する．

なお，本節以降では PAR，SHMEM システムタスク共に PAR，SHMEM と表記する．

3.1 タスク並列処理フレームワーク

タスク並列処理フレームワークは，任意の処理のタスクとしての登録や，登録されたタスクの起動等の実行制御を行う API 群であり，この API を用いたスケジューラを実装するための環境を提供する．また，本フレームワークは MPI をラップしたライブラリとして現在実装されており，登録されるタスクコードはすべてのノードで共有され，システム全体が MPI プロセスとして実行される．タスクは，タスクを示す ID をインデックスとした関数ポインタ配列として実装されており，タスクの起動はタスク ID の送信と，送信先での ID に対応する関数の実行で実現されている．

タスク起動にはメッセージを付属することができ，タスクはメッセージの受信ルーチンとして見なすこともできる．本フレームワークは受信した起動要求タスク ID を内部のキューに蓄え，現在実行中のタスクが終了してか

```
double u[XSIZE+2][YSIZE+2],uu[XSIZE+2][YSIZE+2];
...
int main() {
    int x,y,k;
...
#pragma omp parallel private(k,x,y)
    {
        for(k = 0; k < NITER; k++){
#pragma omp for
          for(x = 1; x <= XSIZE; x++)
            for(y = 1; y <= YSIZE; y++)
              uu[x][y] = u[x][y];
#pragma omp for
          for(x = 1; x <= XSIZE; x++)
            for(y = 1; y <= YSIZE; y++)
              u[x][y] = (uu[x-1][y]
                       + uu[x+1][y]
                       + uu[x][y-1]
                       + uu[x][y+1])/4.0;
        }
    }
}
```

図 2 入力 OpenMP プログラムの例
Fig. 2 OpenMP program example.

ら次のタスクの実行を開始する．但し，内部のキューへアクセスする API を提供しており，既に受信したタスク起動順序を制御してタスク間の優先度を考慮することも可能である．

実装は MPI の 1 対 1 通信関数 (MPI_Send, MPI_Isend, MPI_Recv, MPI_Irecv) のみを用いており，シグナルやマルチスレッド等を用いていない．従って，タスク起動要求の受信処理についてはノンブロッキング受信関数 MPI_Irecv を実行しておき，タスク終了時の次タスクの起動のタイミング (タスクの切れ目) で MPI_Test/MPI_Wait を行うことで完了する．タスク起動要求を受信していないノードでは MPI_Test によるポーリングが行われている．

3.2 コンパイラによるコード変換

T-SDSM での OpenMP プログラムの動作を説明するために，図 2 の OpenMP プログラムと，コンパイラによる変換後のランタイムライブラリ呼び出しを含んだコードを図 3 に示す．

ここで，parallel 指示文や (C 言語での)for 指示文等のワークシェアリング構文で示された領域内のコードで構成されるタスクをユーザタスクと呼ぶ．コンパイラは，ディレクティブで指示された領域をユーザタスクとして切り出し，ディレクティブで指示されていた箇所をランタイムライブラリの呼び出しで置き換える．

このランタイムライブラリには，parallel 指示文に対応する_tsdsm_parallel 関数や，for 指示文に対応する_tsdsm_workshare_for 関数などがある．図 3 中の_task0 は，図 2 で parallel 指示文で示された領域に相当するユーザタスクであり，さらにその parallel 領域中の二つの for 指示文で示された領域が図 3 中 _task1,_task2 としてそれぞれ切りだされている．

OpenMP プログラム中の大域変数は，システムの起動時に SHMEM が管理する共有メモリに領域が確保され，大域変数への参照は確保された共有メモリへの間接参照に変換される．図 2 での u,uu 配列への参照は，図 3 で初期化関数 (_tsdsm_init_shared_variable) によって共有メモリアドレスを示すポインタ変数__SV_u,__SV_uu

```
double u[XSIZE+2][YSIZE+2],uu[XSIZE+2][YSIZE+2];
double (*__SV_u)[YSIZE+2], (*__SV_uu)[YSIZE+2];
...
int _tsdsm_main() {
    _tsdsm_parallel(0);
}
...
int _task0(void *param) {
    int x,y,k;
    for(k = 0; k < NITER; k++){
       _tsdsm_workshare_for(1,1,XSIZE,1,LessEqual);
       _tsdsm_workshare_for(2,1,XSIZE,1,LessEqual);
    }
}
int _task1(void *param) {
    int x,y;
    _for_elem_t *f=(_for_elem_t*)param;
    for(x=f->lb; x<=f->ub; x+=f->step) {
        for(y = 1; y <= YSIZE; y++) {
            __SV_uu[x][y] = __SV_u[x][y];
        }
    }
}
int _task2(void *param) {
    int x,y;
    _for_elem_t *f=(_for_elem_t*)param;
    for(x=f->lb; x<=f->ub; x+=f->step) {
        for(y = 1; y <= YSIZE; y++)
            __SV_u[x][y] = (__SV_uu[x-1][y]
                          + __SV_uu[x+1][y]
                          + __SV_uu[x][y-1]
                          + __SV_uu[x][y+1])/4.0;
        }
    }
}
void _user_shmem_init(void *info) {
    _tsdsm_init_shared_variable(info,
          &__SV_u,
          sizeof(double)*(XSIZE+2)*(YSIZE+2),
          u, "u");
    _tsdsm_init_shared_variable(info,
          &__SV_uu,
          sizeof(double)*(XSIZE+2)*(YSIZE+2),
          uu,"uu");
}
void _ac0(void *info, void *param) {
    ...
}
void _ac1(void *info, void *param) {
    _for_elem_t *f=(_for_elem_t*)param;
    _tsdsm_ac_2d_R( info,__SV_u,
                    f->lb, f->ub, f->step,
                       1, YSIZE, 1);
    _tsdsm_ac_2d_W( info,__SV_uu,
                    f->lb,  f->ub, f->step,
                       1,  YSIZE, 1);
}
void _ac2(void *info, void *param) {
    _for_elem_t *f=(_for_elem_t*)param;
    _tsdsm_ac_2d_R( info,__SV_u,
                     f->lb-1, f->ub-1, f->step,
                           1,   YSIZE, 1);
    _tsdsm_ac_2d_R( info,__SV_u,
                     f->lb+1, f->ub+1, f->step,
                           1,   YSIZE, 1);
    _tsdsm_ac_2d_R( info,__SV_u,
                     f->lb, f->ub, f->step,
                       1-1, YSIZE-1, 1);
    _tsdsm_ac_2d_R( info,__SV_u,
                     f->lb, f->ub, f->step,
                       1+1, YSIZE+1, 1);
    _tsdsm_ac_2d_W( info,__SV_uu,
                     f->lb,  f->ub, f->step,
                        1,  YSIZE, 1);
}
```

図 3 ユーザタスクコードの例
Fig. 3 User task code example.

への間接参照となっている．コンパイラの実装には Omni OpenMP コンパイラの利用を検討しており，このような大域変数や auto 変数の共有化に関しては佐藤ら[4] の手法を参考にしている．

また，コンパイル時には各ユーザタスクに対応するアドレス計算コード (図 3 中_ac0,_ac1,_ac2) を生成する．これはユーザタスクの起動に際して，タスクが参照するメモリアドレスの検知に利用される．

3.3 動作詳細

3.3.1 ユーザタスクの起動とバリア同期

システムの起動時 rank 番号 0 のノードで_tsdsm_main 関数の実行が開始され，_tsdsm_parallel 関数によって第 1 引数で示されたユーザタスクが OpenMP スレッドとして各ノードで実行される．

_tsdsm_workshare_for では，第 2〜5 引数で示されたイタレーション領域全体から各ノードが実行するイタレーション領域を計算し，自ノードに対して計算したイタレーション領域を付属してユーザタスクを起動する．_tsdsm_workshre_for で起動されるユーザタスクは，図 3 中の_task1,_task2 のように，引数 param から渡されるイタレーション領域を用いて計算を行う．

この時起動されるタスクの粒度の調整は，起動に付属するイタレーション領域の値で制御され，ランタイムライブラリが実行時に決定する．現在の実装では，1 タスクの粒度を 1 イタレーションに固定しており，そのノードで実行するイタレーション数に相当するタスクが複数起動される．

3.3.2 タスク管理機構

図 1 で示したように，ユーザタスクは PAR によって管理されている．従って，ランタイムライブラリ_tsdsm_parallel や_tsdsm_workshare_for が行うユーザタスクの起動とは，引数で示されたユーザタスク ID を付属メッセージとした PAR の起動によって実装されている．PAR は内部でユーザタスクの関数ポインタ配列を保持し，受信した ID に対応するユーザタスクコードを実行する．

PAR の役割は，ユーザタスクの実行に際して，後述する SHMEM と協調してメモリのキャッシュ等のユーザタスクの前準備を行うことである．PAR の動作概要を以下にまとめる．

- a) 起動通知をうけたユーザタスクの参照する共有メモリアドレスを求める．
- b) タスクの起動ノードを決定する．
- c) a) で求めたアドレスのホームノードにメモリデータのキャッシュを要求する．
- d) b) で求めたノード先の PAR にユーザタスク起動をエントリする．

a) については 3.3.3 節で詳細を説明する．b) は，これまでの説明で述べたランタイムライブラリではそれぞれの動作に必要なノードがそのまま決定されるが，4 節で述べるようなタスクの移動においては実行時の情報に基づき決定される．

c) はホームノードに対して，要求するメモリブロック番号等をメッセージとして付属し SHMEM を起動することで行われる．d) でのユーザタスク起動のエントリでは，タスクが必要とするメモリブロックのリストが付属されており，c) で発行したすべてのメモリブロックがキャッシュされるまでの間，PAR 内部にタスク起動エントリが保持され，すべてのブロックがキャッシュされ次第 (実行条件の成立)，ユーザタスクの実行が開始される．これらの動作については文献 6) にて報告している．

3.3.3 アドレス計算コードによる共有メモリ参照検知

アドレス計算コードは，タスクコード中の共有変数への参照毎に，_tsdsm_ac_2d_R 等の対応するアドレス計算関数の呼び出しを列挙したコードで構成され，コンパイラによってタスクコードが切り出された後の，共有変数への参照を間接参照に変換する段階で同時に生成される．ここでは図 3 での_task1 に対応するアドレス計算コード_ac1 について説明する．

図 2 中の最初の for 領域では，2 次元共有配列 u への読み込み，2 次元共有配列 uu への書き込みが行われているので，2 次元配列参照チェック関数 _tsdsm_ac_2d_R/_tsdsm_ac_2d_W を共有メモリへのポインタ__SV_u,__SV_uu を引数として実行する．

また，アドレス計算コードの呼び出しに際して，_task1 の引数 param と同じデータが_ac1 の第 2 引数 param に対して渡される．インデックス x の変動領域はこれを利用して f->lb〜f->ub(ステップ数 f->step) として判断され，y の変動領域は入力プログラムから静的に 1〜YSIZE(ステップ数が 1) に固定である．

参照チェック関数は，入力引数の値を元に，共有メモリアドレス領域を求め，書き込みか読み込みかの情報を付加したデータを_ac1 の第 1 引数 info に出力する．参照チェック関数は，info に確保されたデータ領域の大きさを考慮しつつ，先に呼ばれたアドレス領域と重複する領域が必要である場合にはそれらの情報をマージして出力する．

3.3.4 メモリ管理機構

共有メモリは SHMEM によって管理されている．現在の SHMEM の実装では，共有メモリ空間全体を 1024Byte のメモリブロックに分割し，各ブロックのホームノードをサイクリックに割り当てている．ホームノードは自らが管理するブロック毎に全ノード数分のキャッシュしたか否かの情報から構成されるフルマップディレクトリを持つ．またキャッシュしたブロックに対するタグ情報として Invalid/Shared/Dirty の状態を持つ．

SHMEM の持つ主な機能を以下にまとめる．

- キャッシュ要求への対応
  ディレクトリを調べて，要求されたメモリブロックが指定されたノードからキャッシュされたことがなければ，メモリブロックの転送と共にディレクトリ情報をキャッシュ済みに更新する．キャッシュ済みノードへの要求は無視する．
- キャッシュブロック受信への対応
  受信したキャッシュブロックを共有メモリ領域にコピーし，状態を Shared に変更する．同時に PAR が内部で管理しているユーザタスクエントリを調べて，条件が成立したユーザタスクの実行を開始する．受

信したメモリブロックへの書き込みがある場合には，メモリブロックのコピー (Twin) をとり状態を Dirty へと変更する．

- フラッシュ要求への対応
  ホームノードが管理するブロック以外の Shared または Dirty なキャッシュブロックをすべて Invalid に変更し，Shared ブロックはブロックの無効通知を，Dirty ブロックは Twin コピーのと差分 (Diff) をホームノードにそれぞれ送信する．
- 無効化通知/差分データ受信への対応
  無効化通知/差分データを受信したブロックの受信元ノードに対するディレクトリの情報を未キャッシュに変更し，差分データでホームのブロックを更新する．ホームノードとなっている全ブロックが未キャッシュ状態となったら，バリア同期をとり全ノードのフラッシュ操作の終了を待つ．

4. タスク移動による通信最適化

3.3.3 節で述べたように，T-SDSM では共有メモリ領域への参照検知手法としてアドレス計算コードを用いている．アドレス計算コードの出力する情報を用いることで，タスクが参照するアドレスがタスクの実行前に検知できるため，タスクのメモリ参照パターンを考慮した最適化が可能であると考えられる．本節では，現在の T-SDSM に実装した最適化手法について説明する．

4.1 Owner Computing Rule

ここで言う Owner Computing Rule とは，各タスクの参照アドレスの中で，書き込みデータを含むメモリブロック (書き込みブロック) に着目し，各書き込みブロックのホームノードの中で最も多いノードにタスクを配置するというものである．例えば，あるタスクのアドレス計算コードの出力結果から，書き込みブロックが (0,1,5,6) で，各々のホームノードが (0,1,1,2) であった場合，ノード 1 上でこのタスクの実行を行う．

現在，上記のルールに従い，該当するイタレーションに対応するタスクのみを起動するように，`_tsdsm_workshare_for` を実装している．この機能により，`for` 指示文の出口での暗黙のバリア時のメモリフラッシュにおいて更新処理のオーバヘッドの軽減が見込める．

4.2 クリティカルセクション

一般的な SDSM システムが提供する lock/unlock や，OpenMP での `critical` 指示文を用いたクリティカルセクション (CS) は，排他的なデータに対する逐次処理であり，そもそも並列度を阻害する部分である．従って，クリティカルセクション間の切り替え時のオーバヘッド (unlock から次の lock の獲得までのオーバヘッド) を小さくすることが重要となる．

従来の SDSM が提供する lock/unlock を用いた CS では，どのようなメモリコンシステンシモデルであっても unlock 時に次の lock 獲得者への通知のための通信が発生してしまう．これに対して T-SDSM では，`critical` や `atomic` 指示文を用いて記述された CS について，この CS をタスク (CS タスク) として切り出し，実行時に排他的データのホームノードに起動する．

こうすることで他ノードからの critical 領域の実行開始や lock 獲得要求に相当する，CS タスク要求が同一ノード上に集中してくるので，CS 切り替え時には通信を発行する必要がなくなる．このためのランタイムライブラリとして`_tsdsm_critical` を実装した．

表 1 評価環境
Table 1 Evaluation environment.

MPI	MPICH-SCORE(SCore5.4)
CPU	Pentium4-2.4GHz(×8)
Network	1000BASE-T(ns83820)

表 2 laplace の実行結果
Table 2 Execution result of laplace.

ノード数	1	2	4	8
Omni/SCASH(sec)	5.25	7.72	4.94	4.33
T-SDSM(sec)	7.25	20.12	10.83	10.02

表 3 critical の実行結果
Table 3 Execution result of critical.

ノード数	1	2	4	8
Omni/SCASH(usec)	0.63	368.57	594.11	505.73
T-SDSM(usec)	91.86	185.19	295.03	226.83

5. 性能評価

今回実装したランタイムライブラリの PC クラスタでの初期評価を行う．評価対象プログラムはラプラス方程式の解を求めるもので，図 2 はそのプログラムの一部である．問題の大きさは 1000×1000 で，100 反復とした．

評価に用いた PC クラスタ環境を表 1 に，実行結果を表 2 に示す．また，比較対象としてクラスタ上の評価環境に利用したシステムソフトウェアである SCore5.4 に含まれるクラスタ上の OpenMP 処理系である Omni/SCASH を用いた．

結果から判るように，2 ノード以上の性能ははいずれも 1 ノードでの結果を下回り並列化の効果を得られなかった．この主な原因としてタスクの粒度が考えられる．現在の実装ではタスク粒度を 1 イタレーションとしており，各タスク毎にアドレス計算関数が呼び出されるためアプリケーション全体で大きなオーバヘッドとなっていると考えられる．

次に，T-SDSM での CS 実行時のタスク移動による性能向上への効果を確認するために，microbench に含まれる `critical` 指示文のテストを行った．図 3 と同様，`critical` 指示文を`_tsdsm_critical` で置き換えたコードを手動で生成したものを用いて評価する．この評価で

は CS が rank 番号 0 のノードで実行されている．

表 3 から T-SDSM では `critical` 時のオーバヘッドがノード数の増加に対して低く押さえられていることが確認できた．また，ノード数 1 での実行結果には，ユーザタスク実行におけるオーバヘッドの大きさが顕著に現れている．

6. 関連研究

コンパイラによって共有メモリへの参照の直前に，参照するアドレスのキャッシュ状態をチェックするコードを挿入することで，ページ管理機構等を用いずに共有変数への参照を検知する手法が提案[5)8)] されている．これらの手法では，さらに同期区間等で無駄なチェックコードを削除するなどのコンパイラによる最適化を行う．

T-SDSM でのアドレス計算コードによる参照検知手法は，コンパイラによるコード生成を前提としている点でコンパイラによる支援を受けているが，タスクを移動することによる負荷分散の実現を目的としており，タスク実行前にまとめてメモリへの参照チェックを行う点で異なる．

また，栄ら[7)] は，動的負荷分散機能を組み込んだ OpenMP ランタイムシステムの実現手法として，並列ループの初期の数回の実行で各ノードの性能測定を行い，測定した負荷情報に基づき残りのループを再分割する機能を Omni/SCASH へ実装した．また，ループ再分割によってローカリティが低下する問題について言及しており，それに対しては各ページのリモートからの参照をカウントし，頻繁にリモートから参照されるページを最も参照するノードに移送するページマイグレーション機能との併用を検討している．

7. まとめと今後の課題

本稿では，負荷分散機能を有する向け OpenMP ランタイムシステムの実現を目的として，タスク並列処理に基づく SDSM システム T-SDSM の設計と実装方法について述べた．T-SDSM ではアドレス計算コードを用いて，参照される共有メモリアドレスの検知をタスク実行前に行い，それに基づいたタスク配置によってローカリティの向上を目指している．

現在の実装では，`for` 指示文に対して，各イタレーション毎に最も多い書き込みブロックに対してタスクを配置している．laplace による評価では，タスク粒度が 1 イタレーションと固定であったため，アドレス計算コードの実行がオーバヘッドとなり，タスク移動を用いたローカリティ向上による性能への貢献を確認することはできなかった．タスク粒度と性能との影響についての調査や，適切な粒度を求めて本手法の有効性を実証することを今後の課題とする．

謝辞 本研究の一部は，文部科学省科学研究費補助金 (若手研究 (B) 課題番号 14780231，及び特定領域研究 課題番号 14019074) により行われた．

参考文献

1) Hu, Y., Lu, H., Cox, A. L. and Zwaenepoel, W.: OpenMP for Networks of SMPs, *Proc. 13th Int'l Parallel Processing Symp. and 10th Symp. on Parallel and Distributed Processing* (1999).
2) Imamura, T., Tsujita, Y., Koide, H. and Takemiya, H.: An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers, *Proc. of EuroPVM/MPI 2000, LNCS 1908*, pp. 200–207 (2000).
3) J.M.Bull: Measuring Synchronisation and Scheduling Overheads in OpenMP, *Proceedings of the First European Workshop on OpenMP*, pp. 99–105 (1999).
4) 佐藤三久, 原田浩, 長谷川篤志, 石川裕: 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, 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. SIG9(HPS3), pp. 158–169 (2001).
5) 佐藤茂久, 草野和寛, 佐藤三久: OpenMP 向けコンパイラ支援ソフトウェア DSM, 情報処理学会論文誌, Vol. 42, No. 4, pp. 788–801 (2001).
6) 立川純, 福田健一郎, 平孝則, 大西淑雅, 佐藤寿倫, 小出洋: タスク並列処理を用いたソフトウェア分散共有メモリの提案, 情報処理学会研究報告, HPC-2003-94, 情報処理学会 (2003).
7) 栄純明, 松岡聡, 佐藤三久, 原田浩: Omni/SCASH のループ再分割を用いた動的負荷分散拡張の実装と評価, 先進的計算基盤システムシンポジウム SACSIS2003, pp. 307–314 (2003).
8) 丹羽純平, 松本尚, 平木敬: ソフトウェア DSM 機構を支援する最適化コンパイラ, 情報処理学会論文誌, Vol. 42, No. 4, pp. 879–897 (2001).