タスク並列処理を用いたソフトウエア分散共有メモリの提案

立川 純[†]　福田 健一郎[†]　平 孝則[†]
大西 淑雅[††]　佐藤 寿倫[†,†††]　小出 洋[†]

計算機クラスタやグリッドに代表されるコモディティ技術を用いた計算環境では，CPU やネットワーク等の資源に不均質を伴うことが多く，これらの資源を有効活用してアプリケーションの性能向上に反映させることが重要となる．そのためには，環境に依存しないプログラミングインターフェイスを提供すると共に，システムによる環境への適応が必要であると考えられる．そこで，コンパイル時に入力される並列プログラムから，タスクへの分割と各タスクが参照するアドレスを計算するためのコードを生成し，それを利用して実行時に共有メモリを実現するタスクベースソフトウェア分散共有メモリ (T-SDSM) を提案する．T-SDSM はタスクの実行に必要なデータの所有ノードやプロセッサの負荷情報を考慮に入れてタスクの実行ノードの決定を行うことで負荷分散による環境への適応を目指している．本稿では T-SDSM の構成，及び現在検討している通信の最適化手法について報告する．

T-SDSM: Task-based Software Distributed Shared Memory based on Task-Parallel processing Framework

Jun Tachikawa ,[†] Kenichiro Fukuda ,[†] Takanori Hira ,[†]
Yoshimasa Ohnishi ,[††] Toshinori SATO [†,†††] and Hiroshi Koide[†]

In heterogeneous computing environments such as PC-Cluster and Grid, it is difficult to execute parallel programs with a high average resource utilzation rate. And also, it is necessary to conceal the heterogeneity from the programmers. Therefore, we propose a new method for the parallel programs based on the shared memory model such as OpenMP. In our method, at the compile time, a parallel source program is translated to a set of tasks and is scanned to generate the function for calculating the memory address accessed by these tasks. The system, T-SDSM, which we propose in this paper realize a shared memory space by calling the function described above. And T-SDSM provides task allocation with a high average resource utilization rate. This paper describes the mechanism of T-SDSM, and reports our ideas about communication optimization technique by T-SDSM.

1. はじめに

近年，計算機クラスタやグリッドに代表される，ヘテロジニアスな特性を持った大規模並列分散計算環境に関する研究が盛んに行われている．ヘテロな環境下では，コモディティパーツで構成されることによるプロセッサやネットワークの静的な性能差や，さらにはマルチジョブ/マルチユーザによる動的な負荷変動等によって計算資源の有効活用がなされないことが問題となる．

従って，プログラマが環境毎にこれらの問題点を考慮することは困難であるため，処理系による環境への適応が必要となる．そこで我々は，仮想的な共有メモリ空間を実現することで抽象度の高いインターフェイスを提供するソフトウェア分散共有メモリ (SDSM) に注目し，効率的な資源活用を伴った SDSM に関する研究を行っている．

通常，プログラマによる性能チューニングを前提とする MPI/PVM 等のメッセージパッシングと比較して，SDSM では最新データの所有ノードの管理や，データの更新処理に必要となる通信や処理をシステムがプログラマの代わりに行うことで共有メモリを提供している．従って，そのオーバヘッドのために，原理的に SDSM がメッセージパッシングに性能で勝ることは難しい．

一方，コンパイル時に SDSM コードを最適化されたメッセージパッシングコードに変換することで，性能を得ようとする研究[1)2)] もある．しかし，それらの手法は，動的な負荷変動を含んだ場合を想定していないため，本研究の対象とするヘテロ環境での有効性は不明である．

そこで我々は，コンパイル時に SDSM コードに対して一定の変換と情報の抽出を施し，それをヘテロ環境における実行時の最適化方策に利用する手法について検討している．この手法では，コンパイル時に，SDSM コードから同期を含まない処理単位をタスクとして抽出すると共に，タスクコード中で参照される共有メモリアドレスを計算するためのコードを生成する．それを用いて実行時にタスクが参照する全ての共有メモリアドレスを計算し，タスクの実行ノードに対してデータを集めることで共有メモリ空間を実現する．このときのタスクの実行ノードの決定の際に，プロセッサやネットワーク等の負荷を考慮した負荷分散を行うことで，動的な負荷変動を伴う

† 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
†† 九州工業大学 情報科学センター
Information Science Center, Kyushu Institute of Technology
††† 九州工業大学 マイクロ化総合技術センター
Center for Microelectronic Systems, Kyushu Institute of Technology

環境への適応を行う．

この研究基盤としてタスク並列処理フレームワークを提案[3]した．このタスク並列処理フレームワークでは，OpenMP プログラムを対象としたタスクコードへの変換を検討しており，タスクの実行ノードの割り当てポリシーを記述するためのフレームワークを提供する．現在，上記の SDSM の実現を目指し，このフレームワーク上で共有メモリを実現するタスクベースソフトウェア分散共有メモリ (以降，T-SDSM) の実装を行っている．

本稿では T-SDSM の動作の概要と現在までの実装状況について報告する．また従来の SDSM と比較して，不規則なメモリアクセスパターンを持ったアプリケーションに対してプリキャッシュによってメモリアクセスの遅延隠蔽が可能可能であること，クリティカルセクション等の排他制御における同期オーバヘッドの軽減が期待できること，さらには Inspector-Executor 法を導入することによって通信の最適化が可能であることについて示す．

2. タスク並列処理フレームワーク

我々は，ヘテロな計算環境を効率的に利用するためには，実行する環境が有する資源を有効活用して最適な性能を得ると共に，プログラマに対して環境を透過にすることが重要であると考えている．これらを同時に解決する手法として，ユーザの記述したプログラムをタスクに分割し，最適なタスク割り当てをシステムが行うタスク並列処理が挙げられる．

そこで，ユーザプログラムからタスクに分割されたプログラム (タスクコード) への変換を行うコンパイラや，最適なタスクスケジューラを研究するための基盤としてタスク並列処理フレームワークを提案している[3]．タスク並列処理フレームワークは最初の段階として，プロセッサやメモリ搭載量，及びネットワークについて，アーキテクチャは均質であるが各資源の性能が異なる，性能ヘテロな計算機クラスタを対象に MPI を用いて C 言語により実装している．タスクコードは次節で述べる起動プリミティブを提供する実行時ライブラリと gcc 等のバックエンドコンパイラによってリンクされ，MPI プロセスとして性能ヘテロクラスタ上で実行される．

なお，ヘテロ環境の隠蔽とプログラムの可搬性，及びコンパイラの実装コストを考慮して共有メモリ型のインターフェイスである OpenMP を採用している．

2.1 タスクコードの構成

タスクコードの例を図 1 に示す．タスクコードは次タスクの起動を示すプリミティブ呼び出しを含んでおり，タスクコード自身がタスク間の実行順序関係を示している．また起動プリミティブは，分散メモリであるクラスタ上で実行することを考慮して，任意のメッセージを付属し，起動されたタスクコードの引数でそのメッセージを受け取ることでデータの受け渡しを実現する．タスクは別のタスクによる起動によってのみ生成され，ノンプリエンプティブに実行される．

```
int task_0( void *msg ) {
    int next_param[2]={1,2};
    _texec_invoke(1, _task(1),
                  sizeof(int)*2, next_param );
    return RETURN_EXTASK;
}
int task_1( void *msg ) {
    int *a = ((int*)msg);
    int *b = ((int*)msg+1);
    return RETURN_EXTASK;
}
```

図 1 タスクコードの例
Fig. 1 Task code example.

2.2 タスク実行メカニズム

タスク実行メカニズムは，2.1 節で述べた起動プリミティブを提供すると共に，このプリミティブの振る舞いを記述するためのフレームワークを提供している．具体的には，タスクコードが起動プリミティブを用いて指示したタスクやその実行ノード (Task Invocation Policy) や，さらには到着したタスクの実行順序 (Local Schedule Policy) をこのフレームワークを用いて制御することができる (図 2)．これによって，入力プログラムから変換されたタスクコードを書き換えることなく，タスク割り当てポリシーを変更することができる．また，ユーザプログラムからコンパイルによって生成されたタスクをユーザタスクと呼び，それとは別にこのフレームワーク上に何らかの処理を実現するための任意のタスクを定義することができる．これをシステムタスクと呼ぶ．実行メカニズムはこれらのタスクを区別せずに同じ条件で実行する．

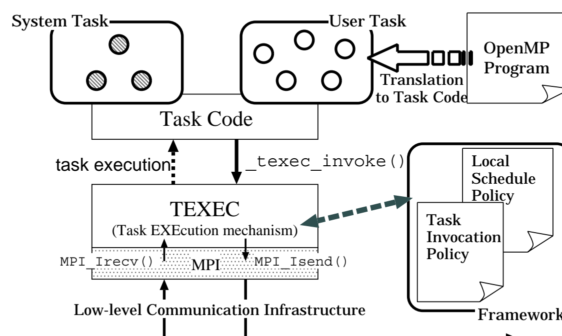

図 2 タスク並列処理フレームワークの外観
Fig. 2 Overview of Task Parallel processing Framework.

2.3 タスクコード変換に関する検討

図 2 で示したタスクコードへの変換では，ディレクティブによって明示される並列領域や，flush/barrier などの明示的なメモリコンシステンシ維持を契機としてタスク分割を行い，各タスクコード中には同期や通信を必要とする処理を含まないようにする．例えば，OpenMP でよく用いられるループの並列処理は，図 3 及び以下に示すようにディレクティブに従ったシンプルな変換を想定している．

- (1) 分岐タスク/(2) 並列計算タスク/(3) 同期タスクの 3 つのタスクへの変換
- (1) からの付属メッセージを用いて (2) の計算領域を指示
- (3) の起動ノードを固定
- (3) 上のローカル変数のカウンタとして用い，(1) での分岐数を閾値として同期成立を判定

ただし，タスク実行メカニズムは実行ノード間に跨っ

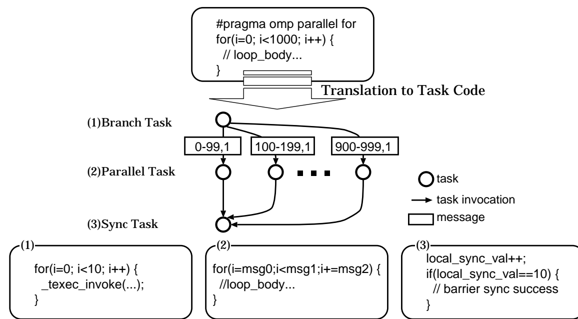

図 3 タスクコードによる fork-join 型並列処理
Fig. 3 Fork-join parallel processing implemented by task code.

た共有メモリ空間を提供していないため，図 3 のように単純にディレクティブに従って分割したままでは，タスクコードは正しく動作しない．そこで次節で述べるタスク並列処理フレームワーク上で実現する SDSM を提案する．

3. タスク並列処理を用いたソフトウェア分散共有メモリ

2 節で述べたタスクの実行モデルと OpenMP は，特に共有メモリの有無という点で大きく異なっている．これに対応する手段として，タスクコードへの変換時に詳細なデータ参照解析を行い，タスク起動に必要となるデータを付属するようなコードを出力する方法が考えられる．しかし，この方法では複雑なタスクコードを生成することとなり，2.3 節で述べたようなシンプルな変換では対応ができず，コンパイラの実装が困難である．

そこで，タスク実行メカニズムの枠組み内で共有メモリを実現する実行時システムとして，タスクベースソフトウェア分散共有メモリ (T-SDSM) を提案する．T-SDSM では，タスクコード中で参照される全ての共有変数のアドレス計算をタスク実行開始前に行い，求めたアドレスの管理ノードからユーザタスクの実行ノードに対してデータを事前に転送 (プリキャッシュ) することで共有メモリ空間を実現する．T-SDSM は，タスクコードに対する解析/ 変換を行うコンパイラ (3.1 節) 及び，2.2 節で述べたフレームワークとシステムタスクによって実装される実行時システム (3.2 節) から構成される．

3.1 T-SDSM におけるタスクコードに対する解析/変換

2.2 節，2.3 節で述べたように，ユーザタスクはプログラマの記述した OpenMP コードからディレクティブに従って分割したタスクコードである．この段階のコンパイル (第 1 フェイズと呼ぶ) は，共有メモリ空間が提供されていることを前提としており，タスクコードは共有変数参照を含んでいる．

その例として図 4 に OpenMP コードから分割されたタスクコードを示す．図 4 中の配列 A は元の OpenMP コード中で大域宣言された共有変数であり，タスク X が配列 A への参照を含んでいる．従って，このままでは配列 A はローカルな変数であり，配列 A を共有メモリ空間上のデータ領域として実行時システムに管理させる必要がある．

```
int A[100];
int task_X( void *msg ) {
    int *_lb = ((int*)msg);
    int *_ub = ((int*)msg+1);
    int *_st = ((int*)msg+2);
    int i;
    for(i=*_lb;i<=*_ub;i+=*_st)
       A[i] = ...;
}
```

図 4 入力されるタスクコード例
Fig. 4 User task code example(input)

```
int task_X( void *msg ) {
       ......
   for(i=*_lb;i<=*_ub;i+=*_st)
      *((int*)(_shmem+_ofA) + i) = ...;
       ......
}
void AC_task_X( void *msg, void *adr ) {
       ......
 _tsdsm_ac_1d_W(&adr,_ofA,4,100,*_lb,*_ub,*_st);
}
```

図 5 変換後のタスクコード例
Fig. 5 Translated user task code(output)

従って，後述する実行時システムが共有メモリ空間を実現するために，すべてのタスクコードを走査して，各タスクコードが参照する共有メモリアドレスを計算するためのコード (以降，アドレス計算コード) の生成を行い，各タスクとの対応付けを行う．実行時システムはこのアドレス計算コードを実行時に呼び出すことでタスクによって参照される共有メモリアドレスを求めることができる．具体的に本節が述べているコンパイル第 2 フェイズでは，ユーザタスクコードを走査して以下を同時に行う．

- 共有変数の共有メモリ空間へのマッピング
- 共有変数参照から共有メモリアドレス参照へのコード変換
- アドレス計算コードの生成とタスクへの対応付け

以下では，入力の例として図 4 を用い，タスクコードに対する変換後のコードを図 5 に示す．

3.1.1 共有変数の共有メモリ空間へのマッピング

コンパイル時に宣言されたすべての共有変数のそれぞれのサイズを計算して，各変数毎の共有メモリ空間内にマップし，そのオフセットを求める．図 5 の場合では，A のオフセットが`_ofA` になったとする．

3.1.2 共有メモリアドレス参照コードへの変換

共有変数への参照を T-SDSM 実行時システムが提供する共有メモリ空間への参照に変換する．具体的には，配列 A を，共有メモリ空間の先頭ポインタ (`_shmem`) + `_ofA` で置き換え，必要なキャストを行い，配列 A の宣言を削除する．

3.1.3 アドレス計算コードの生成

各タスク毎に参照する共有メモリアドレス (共有メモリ空間内のオフセット) を実行時に計算するためのコードを生成する．具体的には，参照する共有変数の形式 (1 次元配列/多次元配列/間接参照) と参照パターン (読み出し/書き込み) 毎のアドレス計算関数に対して，タスクに渡さ

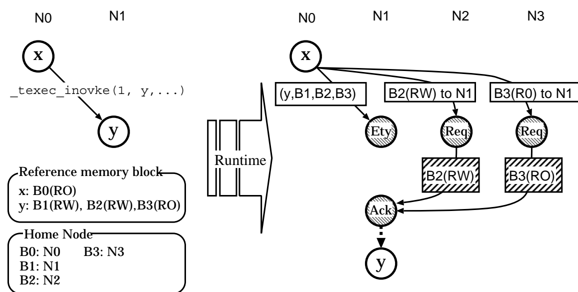

図 6 キャッシュとタスクエントリ
Fig. 6 Memory cache and user task entry.

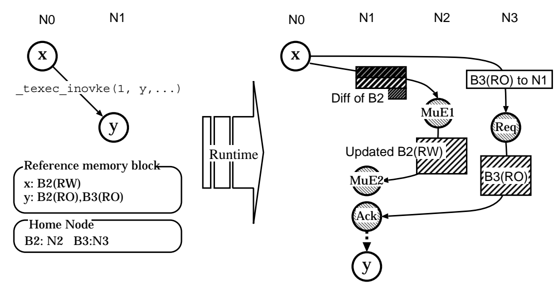

図 7 データ依存を伴ったタスク起動
Fig. 7 Task invocation with data dependency.

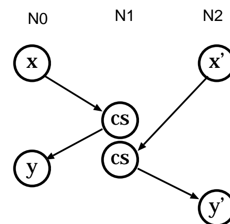

図 8 クリティカルセクションの実行
Fig. 8 Execution of critical section.

れるパラメータを引数として与え，それをタスクコード中に現れる全ての共有変数について列挙したものがそのタスクのアドレス計算コードとなる．図 5 では，アドレス計算関数`_tsdsm_ac_1d_W()` に対して，計算結果の出力バッファポインタ，配列 A のオフセット，要素サイズ，配列サイズ，配列添字の変動領域 (タスクパラメータ) を与えている．

3.2 T-SDSM 実行時システム

3.2.1 メモリキャッシュとユーザタスクの実行

まず，初期化時にメモリブロック毎にホームノードを割り当て，各ホームノードが割り当てられたメモリブロック毎の状態をフルマップディレクトリによって管理する．実行時には，ユーザタスクによる次タスクの起動プリミティブ呼び出しをフレームワークによって認識し，指定されたユーザタスクに対応するアドレス計算コードを呼び出しそのタスクが必要とする共有メモリアドレスを求める．次に，指示されたタスクの実行ノードに対するタスク起動エントリ (図 6 中 Ety) リクエストと，求めた共有メモリアドレスを管理するすべてのホームノードに対するメモリブロックのリクエスト (図 6 中 Req) を送信する．

この時の Req リクエストは,「指定するノードへのメモリブロックの転送を指示する Req システムタスクの起動」として実現される (図 6 の場合はノード N1 への転送指示)．Req タスクは，自らの管理するディレクトリをチェックし，以前にメモリブロックを送信したことがないノードに対してのみ転送する．このメモリブロックの送信も「付属するメモリブロックのローカルへのキャッシュを指示する Ack システムタスクの起動」として実現される (図 6 中 Ack)．

Ack タスクは，付属したメッセージを共有メモリ領域へコピーし，ローカルに管理するキャッシュタグを更新する．また，付属したメッセージが書き込みであるか否かの情報 (図 6 中の RW/RO) を，最初の Req タスクからこの段階まで付属しておき，書き込みである場合には後にメモリブロックの更新差分 (Diff) を求めるためのコピー (Twin) をローカルに別に残しておく．

Ety タスクはそのノード上でタスク実行が決定されたことを示しており，図 6 中の (y,B1,B2,B3) が示すようにメッセージとして必要なメモリブロックのリストが付属されている．Ety タスクは，必要なメモリブロックがすべてキャッシュされているかどうかをチェックし，キャッシュされていればタスク実行を開始し，そうでなければエントリされたタスクの情報を記録する．同様に，Ack タスクでも毎回最後に，記録されているユーザタスクの実行開始条件のチェックを行い必要な共有メモリデータがそろったタスクを実行する (図 6 点線)．

なお，図 6 中ノード N2,N3 からの合計 2 回起動される Ack タスク，およびノード N0 から起動される Ety タスクの実際の実行順序は不定である．

3.2.2 一般的なタスク起動に伴う差分情報の送信

図 6 はタスク x とタスク y に，データ依存関係がない特殊な状況を示している．これはタスク x で書き込んだメモリブロックへの参照が，タスク y には含まれない場合を意味する．

しかし，一般的なタスク起動では，タスク起動プリミティブで順序づけられるタスク間にはデータ依存がある場合が想定される．その様子を示した図 7 では，タスク x とタスク y にメモリブロック B2 を介したデータ依存関係があることを示している．

この場合は，3.2.1 節と異なり，タスク起動プリミティブが呼び出された時点で Diff を求め，起動するタスクが必要とするメモリブロックの Diff を付属したメモリ更新付きエントリ (MuE) タスクを，メモリブロックのホームノードで起動する (図 7 中 MuE1)．

ホームノードでは，ディレクトリのチェックを行い，目的の実行ノードが当該メモリブロックをキャッシュしていなければ，図 7 で示しているように更新したメモリブロックを付属して実行ノードに MuE タスクを起動する (図 7 中 MuE2)．このとき，キャッシュしていれば Diff だけを付属する．こうして実行順序，及びデータ依存関係にあるタスク間でのメモリコンシステンシは常に満たされる．

このメカニズムによって，通常ロック/アンロックで実現されるクリティカルセクション (以下，CS) が T-SDSM においても実現可能となる．この場合，コンパイル第 1 フェイズでは，図 8 に示すような，CS 部分の計算を行

うCSタスクを固定ノードに起動するようなタスクコードを生成しておく．タスク実行メカニズムはノンプリエンプティブに各タスクの実行を進めるので，排他的な処理がそもそも実現可能であり，上記のとおりCSを呼び出すタスク，CSタスク，CSタスクから呼ばれるタスク間でのコンシステンシも保たれている．

3.2.3 バリア同期時の更新処理

OpenMPでは，バリア同期の成立時にメモリの一貫性が保証される必要がある．そこで図3で示したバリア同期の成立時に無効化型の一貫性維持操作を行う．その様子を図9に示す．

具体的には全ノードに対して無効化(Inv)タスクを起動し，Invタスクはキャッシュされた全てのメモリブロックを無効化し，書き込み属性のメモリブロックについてはTwinとの差分を求め，Diffデータを付属してホームノードにライトバック(Wb)タスクを起動する(図9は通信対象が最も多い場合)．Wbタスクは Diffをブロックに更新すると共にディレクトリをチェックしてすべての書き込み属性のメモリブロックのDiffが送信されたら，マスターノードに対してメモリバリア(Mb)タスクを起動する．Mbタスクは全ノード数を閾値としてバリア同期の成立を判定し，メモリバリア成立後ユーザタスクの同期成立とする．

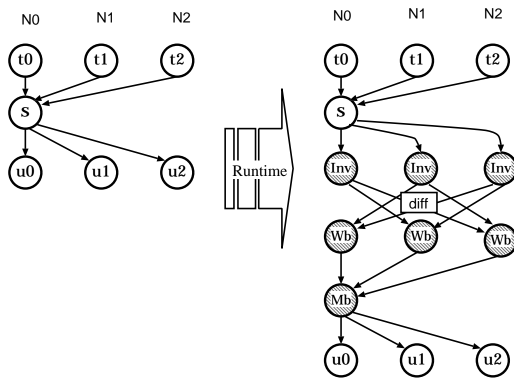

図9 メモリバリア
Fig. 9 Memory barrier.

3.2.4 ユーザタスク実行ノードの決定

これまでの議論で述べたユーザタスクの実行ノードは，ユーザタスクコードが示したノードである．ユーザタスクコードは一定の負荷バランスを考慮したタスクの分散を指示しているのことを想定しているで，ホモジニアスな環境でかつ動的な負荷変動のない条件下ではこれまでの方針で問題はない．ただし，本研究の対象であるヘテロジニアスかつ動的負荷変動を仮定する環境では，この実行ノードの決定にプロセッサやネットワークの様々な負荷情報を考慮することが必要である．

具体的な実行ノードの決定ポリシーの検討は今後の課題であるが，タスク起動プリミティブで必要なメモリブロックが特定された後，Reqタスク等にメモリブロックの転送先情報を渡す必要があるため，この時点で負荷を予測して最終的な実行ノードを決定することになる．

また，別の視点からでは，最も多くの書き込みを行うブロックのホームノードにタスクを起動させる方法についても検討している．ただしその場合には一つのノードに対してタスクが集中し，並列度の低下をまねく可能性もあるため，負荷予測やタスクコードの指示に従う方針とをうまく組み合わせる必要がある．

4. T-SDSMに関する考察

前節までの議論でT-SDSMに動作の詳細について述べた．T-SDSMではユーザの記述するプログラムがタスクに分割されていること，およびタスクの実行開始前に必要なデータが解析できることを利用して，実行時の最適なタスク割り当てによって通信量の軽減が可能である．本節では，従来のSDSMシステムの一つとしてT-SDSMを捉えた場合の特徴について考察する．

4.1 プリキャッシュによるメモリアクセス遅延隠蔽

T-SDSMでは，各タスクの実行開始前に必要なメモリブロックがプリキャッシュされるのに対し，OSのページ管理機構を用いた従来のSDSMではオンデマンドにしか必要なデータが特定されない．従って，不規則で広範囲に渡ってアクセスするようなアプリケーションを対象とした場合には，T-SDSMはタスクの実行開始時に必要なデータリクエストを集中的に起動することでメモリアクセス遅延時間の隠蔽が期待できる．

ただし，これはアドレス計算コードによる参照メモリブロックの特定の精度に影響する．例えば，任意の関数の返し値をインデックスとして示すような配列参照を含むアプリケーションでは実行時のアドレス計算が困難であり，本手法が適用できるか否かは不明である．そうした場合には，従来のページ管理機構を用いた仮想共有メモリを併用する必要がある．

4.2 クリティカルセクション実行に伴う同期オーバヘッド軽減

従来のSDSMにおけるコンシステンシモデルの一つにLRC[4]が挙げられるが，LRCではロック獲得者にのみ，ロック解放者の更新情報を送信することで通信量の低減を行い性能向上を達成した．

それに対してT-SDSMでは，3.2.2節で述べたように，CSが排他的なデータアクセスを主目的した逐次処理であることに着目し，固定のノード(理想的には，排他データのホームノード)上にCSタスクを起動する．この場合，同CSタスクの起動に伴うコンシステンシ維持のためのReqやMuE等のシステムタスクの起動(通信)が，先に実行開始されたCSタスクの実行と並列に行われる．さらにCSタスクが実行終了するまでに，他のCSタスクの実行準備が完了しておけば，CSタスク終了時に次のCSタスクへの切り替えに待ちを伴うことがない．

LRCを含め従来のコンシステンシモデルでは，ロックの解放時(CS部分の終了時)にロックマネージャ等への通信が発生することに対して，T-SDSMでは他のCSに伴う同期オーバヘッドが小さいことが期待できる．

4.3 Inspector-Executor法を導入したタスクスケジューリングによる通信最適化

4.1節で述べているように，一つのタスクの実行開始までに注目すると，プリフェッチによってメモリアクセス遅延の隠蔽が可能である．これを応用することで，例えば図3における(1)分岐タスクのような複数のタスクを

起動する場合に，これから起動する全てのタスクが必要とするメモリブロックを先行的に求めて，通信量を最小化するような通信スケジュールが可能となる．具体的には，Inspector-Executor 法[5] を応用して，ユーザタスク実行開始までの通信量の最小化と，ユーザタスク実行後のデータ更新処理に伴う通信量の最小化が可能となる．

そのために，分岐タスクで起動するすべてのタスクの参照メモリブロックを特定した後，各タスクが参照するメモリブロックの内，最も参照頻度の高いメモリブロックのホームノードを，そのタスクの実行ノードとなるように実行ノードを決定する (Inspector に相当)．次に，この実行ノードの決定に従って，これまでの議論どおりユーザタスクを起動する (Executor に相当)．

こうすることで，ユーザタスクを起動する際に，メモリキャッシュのための Req タスクの起動が最小になることが期待できる．また，Inspector 時の実行ノードの決定方法で，メモリブロックの参照のうち書き込みが最も多いメモリブロックのホームノードを実行ノードとする場合には，後のバリア同期時の通信量の最小化が期待できる．

5. 実　　装

現在，MPI を用いてタスク実行メカニズム，及び T-SDSM での解析部分 (第 2 フェイズ) の一部，実行時システムの一部について実装が完了している．C 言語で記述されたタスクコードへの解析には，PC クラスタコンソーシアムが公開する Omni OpenMP コンパイラ[6] の提供する C-Front/Exc Java tools を用いている．OpenMP をタスクコードへ変換するコンパイラ (第 1 フェイズ) については未実装である．

6. 関 連 研 究

逐次プログラムに対する自動並列コンパイラに関する研究において，文献 7) のように入力プログラムをタスクに分割してタスク間の制御依存/データ依存に基づき，スケジューラによって必要なデータの通信が指示されることで分散されたタスク間で共有変数を提供する方法について研究がなされている．本提案手法では，タスクの実行ノードが従来の SDSM の管理手法を応用してデータ依存を解決することで，スケジューラにまかせず各ノードが自立的にタスクの割り当てや実行を行う．

7. まとめと今後の課題

本稿では，入力並列プログラムをコンパイルによってタスクに分割し，さらにタスクコードを解析して参照される共有メモリアドレスを実行時に計算するためのコードを生成することで，実行時システムがそのコードを呼び出して共有メモリ空間を実現する T-SDSM の基本メカニズムについて述べた．また，従来の SDSM と比較した場合に，事前にメモリ参照領域を特定可能であるという点を利用した最適化や，ユーザプログラムの並列処理部分がタスクとして抽出されていることを利用して，CS 処理等においてタスクを移動させることによる最適化で得られる効果について述べた．

このように本研究の方向性として，OpenMP をインターフェイスとした SDSM システムに関する研究として現在位置づけている．今後はまずこの方針に則り，SDSM システムとしての実装を行い実アプリケーションによる評価を行う予定である．

その後段階的に静的なヘテロ環境から動的負荷変動を含んだ環境へと性能低下の要因となる条件を追加し，それぞれの条件下での有効な適応方法，特に負荷予測のメカニズムやそれを用いたタスク割り当てポリシーについて検討する．

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) 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).
2) Scacles, D. J., Gharachorloo, K. and Thekkath, C. A.: Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *ASPLOS-VII* (1996).
3) 立川純, 福田健一郎, 平孝則, 大西淑雅, 佐藤寿倫, 有田五次郎: 性能ヘテロクラスタにおけるタスク並列処理フレームワークの提案, 先進的計算基盤システムシンポジウム SACSIS2003 論文集 (2003).
4) Keleher, P., Cox, A. L., Dwarkadas, S. and Zwaenepoel, W.: TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *Proc. Winter USENIX Conference*, pp. 115–132 (1994).
5) Koelbel, C. and Mehrotra, P.: Compiling Global Name-Space Parallel Loops for Distributed Execution, *IEEE Trans. of Parallel and Distributed Systems*, pp. 440–451 (1991).
6) 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 SCACH 上の OpenMP コンパイラ, 並列処理シンポジウム JSPP2001 論文集, pp. 15–22 (2001).
7) 上田哲平, 本多弘樹, 吉瀬謙二, 弓場敏嗣: 分散メモリシステム上でのマクロデータフロー処理の実現, 情報処理学会研究報告, HPC-2002-89, 情報処理学会, pp. 203–208 (2002).