ハイパフォーマンス
コンピューティング 77−13
(1999. 8. 3)

SMPクラスタ向けOpenMPコンパイラ

佐藤茂久[†] 草野和寛[†]
田中良夫[†] 佐藤三久[†]

OpenMP APIを用いて記述された共有メモリ向け並列プログラムをSMPクラスタ上で実行するためのコンパイラと実行時ライブラリを設計・開発した。明示的なリモートコピーとソフトウェア分散共有メモリを併用してノード間でのデータ共有を実現する。OpenMPコンパイラは、OpenMPプログラムを実行時ライブラリの呼び出しを含むマルチスレッドプログラムに変換する。コンパイル時に共有データの参照パターンを解析し、実行時のオーバーヘッドが小さい共有方法を選択することで、従来の分散メモリ向け並列化コンパイラやソフトウェアDSMとは異なるコードを生成する。PentiumProプロセッサを用いたSMPクラスタでの実行性能も示す。

An OpenMP Compiler for SMP Clusters

SHIGEHISA SATOH, KAZUHIRO KUSANO, YOSHIO TANAKA,
and MITSUHISA SATO[†]

We designed and developed an compiler and its runtime system to execute OpenMP programs on SMP clusters. Shared data are kept coherent using two methods: explicit remote copies inserted by the compiler and a compiler-directed software DSM system. Our compiler translates OpenMP programs into instrumented multi-threaded programs which contain calls for the runtime routines. Compile-time analysis of references to shared data determines the data sharing mechanism to use. Thus the output programs are diffrent from ones generated by parallelizing compilers for distributed memory systems or other compiler-directed software DSM systems. We also present preliminaly performance evaluation on a Prentium Pro based SMP cluster.

1. はじめに

低価格で高性能なマイクロプロセッサと高速なネットワークの普及により、クラスタシステムを用いた高性能計算が研究・実用化されている。従来、クラスタシステムを含む分散メモリシステム向けの並列プログラムはメッセージパッシングライブラリを、共有メモリシステム向けの並列プログラムはマルチスレッドライブラリを用いて記述するのが一般的であった。

メッセージパッシングプログラムは、共有メモリシステムでも実行可能であり、高性能なライブラリを用いればマルチスレッドプログラムに匹敵する性能を得ることが出来る。しかし、メッセージパッシングプログラムは開発コストが大きく、逐次プログラムとの互換性もない。一方、マルチスレッドプログラムは、開発コストは比較的小さいものの、性能のチューニングが難しく、分散メモリシステム上での利用も一般的ではない。

共有メモリ並列プログラミング言語の標準規格を目指して、OpenMP APIが提案されている[1)2)3)]。OpenMP APIは既存の逐次プログラミング言語(Fortran 77/90/95, C/C++)を拡張する並列化指示文・実行時ライブラリ・環境変数の仕様である。インクリメンタルな並列化が可能で、並列化指示文を無視して逐次プログラムとしてコンパイルすることも出来るため、逐次プログラムからの移行が容易であると言う特徴を持つ。さらに、プログラマはプログラムの並列性の記述を主に行い、ターゲットマシン向けの最適化はコンパイラによって行えるため、ポータブルな並列プログラムの標準としても期待されている。

現在のOpenMPの仕様は、CC-NUMAのような分散共有メモリシステムを含む、共有メモリシステム向けに定められており、それらの並列システム向けのOpenMP処理系が各社から発表されている。さらに、分散メモリシステムでのOpenMPプログラムの利用についても研究が行われている。

H.Luらは、OpenMPプログラムをソフトウェアDSMであるTreadMarksを用いたプログラムに変

† 新情報処理開発機構 つくば研究センタ
Tsukuba Research Center, Real World Computing Partnership

換し、クラスタ上で実行する方法を提案している[4]。C.Brunshen らも TreadMarks を用いたクラスタ上での OpenMP プログラムの実行の研究を行っている[5]。

我々は、既に SMP 向けの OpenMP 処理系を開発しており[8]、本論文では OpenMP プログラムを SMP クラスタ上で実行するための実行環境およびコンパイル手法について述べる。

我々の方法の特徴は、同一プログラム中で、共有データの一貫性制御を可能な限り静的 (コンパイル時) に行い、それが困難な場合には動的 (実行時) に行うことにある。即ち、コンパイル時に共有データのデータフローが正確に解析できる場合にはそのデータを書き込むスレッドと読み込むスレッドの間でリモートコピーを直接行うことにより共有データの一貫性をコンパイル時に保証し、そうでない場合には実行時に一貫性制御を行うようなプログラムに変換する。前者は、HPF のような分散メモリ向け並列化コンパイラの手法であり、後者はソフトウェア DSM の一手法である。

共有メモリ並列プログラムにコンパイラで一貫性制御のためのコードを挿入する方式のソフトウェア DSM としては、Shasta[6] や ADSM[7] がある (後者は書き込み時のみ)。これらの方法では、一貫性制御を効率良く行うためのコンパイル時最適化を行っている。しかし、共有データの参照に対して実行時に一貫性制御のためのコードを実行する必要があるため、実行時オーバーヘッドが生じる。我々は、コンパイル時にデータフローを正確に解析できる共有データに対しては、必要な場合にのみ直接リモートコピーを行うことによって、実行時のオーバーヘッドを削減する。

本論文は、まず本システムの概要を述べた後、実行時ライブラリおよび OpenMP コンパイラの処理内容を述べる。次いで簡単な OpenMP プログラムを例に、PentiumPro プロセッサを用いた SMP クラスタ上での実行性能と本方式の特徴を具体的に示す。最後に、本論文のまとめと今後の課題を述べる。

2. システム概要

我々の目的は、OpenMP プログラムを分散メモリシステム、特に SMP クラスタで実行することにある。そのために、分散メモリ上でのデータ共有・マルチスレッド実行などを実現するための実行時ライブラリと、OpenMP プログラムをその実行時ライブラリを用いたプログラムに変換するための OpenMP コンパイラを新たに設計・開発した。

OpenMP コンパイラの目的コードは C 言語のプログラムであり、既存の C コンパイラでオブジェクトプログラムにコンパイルした後、実行時ライブラリとリンクして実行可能プログラムが生成される。

生成された実行可能プログラムは、SPMD 方式の並列プログラムであり、SMP クラスタの各ノードで一つずつ別のプロセスとして実行される。各ノード内では既存のマルチスレッドライブラリを用いて複数のスレッドを実行できる。Myrinet などのネットワークを用いたリモートデータ転送を用いて、ノード間のデータ共有や同期のための通信を行う。

我々のシステムでは共有メモリ向けの OpenMP プログラムをプログラマが書き換えることなく、透過的に分散メモリ上で実行できる。ユーザーの管理下にあるプログラム中のデータに関しては、ノード間で共有することができ、分散共有メモリシステムを実現していると言える。ただし、OS レベルでノード間のデータ共有を実現してはいないため、共有メモリシステムと全く同じ動作をするわけではない。具体的には、ファイル構造体や、ソケットなどの OS の管理下にあるリソースのノード間での共有はサポートされていない。

3. 実行時ライブラリ

我々の実行時ライブラリは SMP クラスタ上で以下の機能を提供する。

(1) メッセージ通信
(2) スレッド管理
(3) 同期機構 (バリア、ロック)
(4) 共有データ管理

これらの機能を用いて OpenMP API を SMP クラスタ上で実現している。各機能について以下に述べる。

メッセージ通信

ノード間の通信は、受信側ノードのスレッドが介在することのないリモートコピーにより行うが、受信側のスレッドで何らかの処理が必要な場合には、リモートコピーを用いたメッセージ通信を行う。これは汎用的なメッセージ通信機構を提供するものではなく、本ライブラリで必要な処理に特化した機能である。メッセージの受信は、受信側スレッドでポーリングを行うことにより行う。ポーリングは実行時ライブラリ内の同期待ちなどで不定期に行われる。

スレッド管理

SMP クラスタ上でのマルチスレッド実行を、ノード内では POSIX スレッドや Solaris スレッドなどの既存のマルチスレッドライブラリ、ノード間では SPMD 方式を組み合わせることによって実現する。実行可能プログラムは、各ノードで同時に、しかし独立したプロセスとして起動される。各プロセスは、ノード毎の指定されたスレッド数 (通常はプロセッサ数) のスレッドを、既存のマルチスレッドライブラリを用いて起動する (これらを物理スレッドと呼ぶ)。物理スレッドはプログラムの起動から終了まで実行され続ける。OpenMP の並列領域におけるスレッドは仮想スレッドと呼ぶデータ構造で表し、それぞれの物理スレッドに特定の仮想スレッドの実行を指示するメッセージを

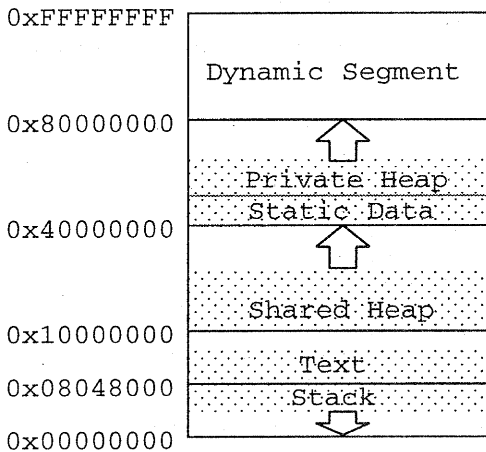

図1　メモリレイアウト

送ることで実行される。例えば、プログラムの逐次実行部ではただ一つの物理スレッドが逐次実行部に相当する仮想スレッドを実行し、他の物理スレッドは仮想スレッドの割当を待っている。OpenMP の並列領域の実行が開始されると、その領域のスレッド数分の仮想スレッドが作成され、それぞれの物理スレッドより実行される。

同期機構

同期機構としては、排他制御のためのロック機構と、同一並列領域内のスレッド間のバリア同期機構を、リモートコピーとノード内のロック機構を組み合わせて実装した。ノード間でのロックの要求や解除のための通信は、リモートコピーを用いたメッセージ通信によって実現した。

共有データ管理

共有メモリ並列プログラムを SMP クラスタ上で実行するために、一種のソフトウェア DSM を実行時ライブラリに組み込んだ。そのために、共有データの割付や解放、データの一貫性制御のための機能を設けた。

まず、共有データの割付について述べる。分散メモリ上に仮想的な共有メモリを設けるため、仮想アドレス空間の一部をノード間で共有することとした。Intel386 アーキテクチャ向け Solaris での実装では、仮想アドレス空間を図 1のように分割した。

プロセスの使用するメモリを、スタック、テキスト、ダイナミックセグメント、静的データ、共有ヒープ、プライベートヒープに分類する。このうち、スタックとテキスト、ダイナミックセグメント (共有ライブラリやスレッドのスタックに利用される) は従来と同様に配置され、ノード間で共有はされない。プライベートヒープは従来のプロセス単位のヒープであり、これも共有されない。

静的データと共有ヒープは共有データであり、ソフトウェアによりノード間の一貫性を維持する。これらの領域の先頭は特定のアドレスに配置され、全てのノードで同じアドレスには同じデータが配置されているものと見なされる。共有ヒープはノード間で共有可能なデータを動的に確保した際に用いられる。従来の動的メモリ管理の関数群 (malloc() や free()) に対応して、共有ヒープを用いた動的メモリ管理のための関数群を用意した。

OpenMP ではユーザーが動的に確保したデータは常に共有データと見なされる。そのため、コンパイル時に OpenMP プログラム中の動的メモリの割付や解放の関数呼出を対応する共有ヒープの割付や解放の関数の呼出に書き換える。

次に、共有データの一貫性制御について述べる。OpenMP では緩いメモリコンシステンシモデルを採用しており、flush 指示文が挿入された点でのみ一貫性が保証される必要がある。そのため、隣接した flush 指示文にはさまれた区間内で、一貫性制御の最適化が可能となる。

前述の共有ヒープおよび静的データからなる共有データは、ページ・ブロック・ラインと言う単位で管理する。ページは固定長 (4KB) で、ページを単位にホームノードを定める。ホームノードとは、その共有データの常に最新の値を持っていると見なされるノードである。ただし、共有データの書き込みは直後の flush 指示文までに完了すれば良いため、真に最新の値を持っているとは限らない。ブロックは、共有ヒープの動的記憶割付により生じる可変長の領域である。ラインは固定長 (64 バイト) の領域で、共有データはラインに分割され、各ノードはライン毎にその最新のデータを持っているかを示す状態フラグを持っている。

実行時の一貫性制御は、チェックコードと呼ぶ次の三種類のコードをプログラム中に挿入することにより行う。

読み込み前チェック　実行しているノードに参照しようとする共有データの最新の値がない場合、そのデータのホームから最新の値をリモートコピーにより取り寄せる。このときの更新は読み込むデータだけでなくそれを含むライン全体に対して行う。

書き込み前チェック　書き込もうとする共有データの含まれるラインの一部しか書き換えない場合、書き換え後にそのライン全体の最新の値を持つように、そのラインを最新の値に更新する。

書き込み後チェック　書き換えた共有データの値をそのホームにコピーし、書き換えたノードとホーム以外の持つそのラインのコピーを無効化または更新する。

ここで、一貫性プロトコルには更新プロトコルと無効化プロトコルの両方をサポートしており、同一プログ

ラム中に混在できる。そのため、参照パターンに応じてコンパイル時に適切なプロトコルを選択してチェックコードを挿入することができる。全て更新プロトコルにした場合、各ノードは常に共有データの最新の値を持っていると考えられるため、読み込み前・書き込み前のチェックを省略することも出来る。

逐次実行部分では、全ての共有データのホームをマスタースレッドの実行されるノードとし、動的な一貫性管理を行わない。そのために、並列領域の入口と出口では、マスタースレッドの実行されるノード (通常はノード 0 番) が最新のコピーを持つように、必要に応じてリモートコピーが行われ、状態フラグが更新される。

以上の動的な一貫性管理の方法は Shasta と類似している。しかし、静的データも共有可能なことや、更新プロトコルと無効化プロトコルが混在できることなどが異なる。また、このソフトウェア DSM 機能は、本システムでは補助的な機能であり、次に述べる静的な一貫性制御と組み合わせて用いる点が、本システムの特徴の一つである。

静的な一貫性制御では、コンパイル時に共有データのデータフローを解析し、共有データの値を書き込むスレッド (ノード) と読み込むスレッド (ノード) の間で直接リモートコピーを行うことにより共有データの一貫性を保証する。データフローが正確に解析できる場合には、この方法が最も効率が良い。しかし、常にそのような解析が出来るわけではないため、全て静的に解決しようとすると、既に最新の値を持っている可能性があってもリモートコピーを行うようなコードになってしまう。そこで前述の動的な一貫性制御を併用する。プログラム中の各共有データの参照の一貫性制御を静的に行うか動的に行うかは、次節で述べるコンパイラにより決定する。

4. OpenMP コンパイラ

SMP クラスタ向けの OpenMP コンパイラは SMP 向けの OpenMP コンパイラ [8] を拡張して開発した。SMP クラスタ向けの実行時ライブラリの API も、SMP 版と共通な機能については互換性があるように設計した。そのため、この節では SMP クラスタ版に特有の機能を中心に説明する。

SMP クラスタ版で新たに必要になったのは、前節で述べた動的記憶割付に関する関数の置換や共有データの一貫性制御のためのコードの挿入である。

簡単な OpenMP プログラムの例として、daxpy ループを図 2に示す。このプログラムで OpenMP の指示文は `i` ループで DOALL 型の並列化をするように指示している。また、変数 `a`, `x[]`, `y[]` は共有変数であり、ループ制御変数 `i` はプライベート変数になる。

OpenMP コンパイラは並列領域から各スレッドが

```
  double a, x[N], y[N];
  int i;
#pragma parallel for shared(a,x,y) private(i)
  for (i = 0; i < N; i++)
    y[i] += a * x[i];
```

図 2 daxpy ループの例

```
 int _dompc_thread(void **args) {
  double *a, *x, *y;
  int i, lb, ub;
  a = (double *)args[0];
  x = (double *)args[1];
  y = (double *)args[2];
  barrier();
  static_schedule_init(0, N, 1);
  get_chunk(&lb, &ub);
  for (i = lb; i < ub; i++) {
    check_before_read(a, sizeof(double));
    check_before_read(x+i, sizeof(double));
    check_before_read(y+i, sizeof(double));
    y[i] += (*a) * x[j];
    check_after_write(y+i, sizeof(double));
  }
  barrier();
}
```

図 3 スレッド本体のコード (最適化前)

実行するための関数を作成する。単純に変換した場合のスレッド本体のコードを図 3に示す。

各スレッドは親スレッドから引き継がれる共有データのアドレスの配列を引数として受け取り、共有データを参照するためのポインタを初期化する。並列領域内での共有データの参照には、前節で述べたチェックコードを挿入する。`check_before_read()` と `check_after_write()` はそれぞれ読み込み前、書き込み後のチェックである。配列 `y[]` の要素は読み込んだ直後に書き込まれるため、書き込み前のチェックは省略される。この例での SMP 向けのコードとの違いは、チェックコードを挿入する点だけである。このコードでは、チェックコードを配列要素の参照毎に挿入しているため、チェックコードがループの繰り返し毎に実行され、個々のチェックのオーバーヘッドが小さいとしても、一貫性制御のためのオーバーヘッドは性能を大きく低下させる。

この例では並列領域内に flush 指示文がないため、共有データの値の一貫性は、並列領域の最初と最後でのみ保証されれば良い。そのため、ループ中のチェックを全てループ外に出して、一括して行うことが出来る。そうして得られたコードを図 4に示す。

このような動的な一貫性制御の最適化は、Shasta や ADSM でも行われていた。我々の方法では、動的な一貫性制御に加えて、コンパイル時に可能な限り静的な一貫性制御を行うように最適化を行う。

データ並列な処理を行う for ループは、静的な一貫性管理が行える場合がある。例えば、ループの各反復

```
check_before_read(a, sizeof(double));
check_before_read(x, (ub-lb)*sizeof(double));
check_before_read(y, (ub-lb)*sizeof(double));
for (i = lb; i < ub; i++) {
  y[i] += (*a) * x[i];
}
check_after_write(y, (ub-lb)*sizeof(double));
```

図4　最適化後のループ部

で同じ配列の異なる要素を参照する場合、各スレッドが配列のどの要素をいつ参照するか、また、各要素がいつどのスレッドによって参照されるかが、コンパイル時に解析できる。そこで、同一ループ内、あるいは同一並列領域内の複数の並列ループにまたがって、次のような最適化を行える。

- あるスレッドで書き換えた配列要素の値が、他のスレッドに読み込まれることがないことがコンパイル時にわかれば、その代入文に対する書き換え後のチェックコードを生成しなくてよい。
- あるスレッドで読み込もうとする配列要素の最新の値が、そのスレッドの実行されるノードにはないことがコンパイル時にわかれば、読み込み前のチェックコードの代わりに最新の値を持つノードから変数の値を直接コピーするコードを生成できる。
- あるスレッドで書き換えた配列要素の値が、どのスレッドによって読み込まれるかがコンパイル時にわかれば、通常の書き込み後のチェックコードの代わりに、読み込むスレッドのあるノードへのみ最新の値をコピーするコードを生成できる。

さらに、SMPクラスタであることから、同一ノード内のスレッドで書き換えられた値を同一ノードの他のスレッドで読み込むことがコンパイル時にわかる場合には、最新の値がそのノードにあることが保証されるので、参照前のチェックコードを省略するという最適化も行うことが出来る。

各スレッドで参照する共有データのスレッドをまたがるデータフローの有無を解析することで、以上のような最適化を行うことができる場合を検出することが出来る。

次節では、このような最適化が可能なOpenMPプログラムを用いて性能評価を行う。

5. 性能評価

5.1 ベンチマークプログラム

OpenMPプログラムの例として、連立一次方程式の反復解法の一種である、緩和係数付のJacobi法のプログラムを用いる。図5に示すプログラムの計算の主要部分は、行列・ベクトル積と残差平方和を求めるリダクション計算を含む並列ループであり、そのループを解が収束するまで繰り返す。

```
double *A, *b, *x, *y, omega, err, epsilon;
int N, i, j;
#pragma parallel private(j)
do {
#pragma single
  err = 0.0;
#pragma for reduction(+:err) schedule(static)
  for (i=0;i<N;i++) {
    y[i]=b[i];
    for (j=0;j<N;j++)
      if (j!=i) y[i]=y[i]-A[i*N+j]*x[j];
    y[i]=x[i]+omega*(y[i]/A[i*N+i]-x[i]);
    err+=(x[i]-y[i])*(x[i]-y[i]);
  }
#pragma for schedule(static) nowait
  for (i=0;i<N;i++) {
    x[i]=y[i];
  }
} while (err<epsilon);
```

図5　Jacobi法のOpenMPプログラム(主要部)

このプログラムは並列領域内で複数のスレッドに参照されるデータが近似解のベクトルxと残差平方和errのみであるため、実行時に一貫性制御の必要なデータが少なく、我々の方式の特徴を示しやすい。

5.2 SMPクラスタ COMPaS

評価に用いるプラットホームは、我々の開発したPrentiumProを用いたSMPクラスタCOMPaSである[9]。各ノードは200MHzのPentiumProプロセッサが4個共有バスを介して接続されている。そのようなノードが8個、Myrinetを介して接続されている。OSはSolaris 2.5.1であり、ノード内のマルチスレッドライブラリとしてSolarisスレッドを用いた。ノード間の通信は我々の開発した通信ライブラリNICAM[10]を用いて行う。NICAMはリモートコピーと、ノード間のバリアをサポートしている。

5.3 実行性能

前述のJacobi法のOpenMPプログラムをSMPクラスタ用のコードに変換し、6144元の連立一次方程式を解いた。方程式$Ax = b$において、係数行列Aは収束が保証される対角優位行列を自動生成し、ベクトルbは解xの全ての要素が1.0となるように与えた。xの初期値は0ベクトルとした。実験を行った全ての場合に10回の反復で解が得られた。

図6に反復計算部分の実行時間を、ノード数とノード内のプロセッサ数を変えて測定した結果を示す。なお、1スレッドの場合と元のOpenMPプログラムを逐次プログラムとしてコンパイルした場合の実行時間の差はわずかである。

いずれの場合もスレッド数の増加に応じて性能が向上している。ノード数の増加に対しては線形に近い性能向上を示しているが、ノード内のプロセッサ数の増加に対しては低い性能向上しか得られない。これは共有バスのバンド幅の不足によるものと考えられる。

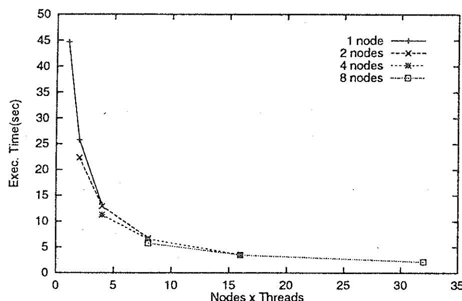

図6　実行時間

5.4 考　　察

我々の方式では、Jacobi 法の OpenMP プログラムに対して次のような最適化が行えた。

(1) 行列 A とベクトル b, y は、各スレッドで参照する範囲が重ならないため、要素単位では共有されていない。そのため、反復計算部で A, b, y に対する一貫性制御のためのチェックやリモートコピーを全く行わずに済んだ。

(2) ベクトル x は、その全要素が全てのスレッドにより参照される。そのため各ノードで値を更新する必要があるが、SMP クラスタであるために、一回の更新で同一ノード内の複数のスレッドが最新のコピーを得ることが出来る。

つまり、Jacobi 法のプログラムではデータ並列な for ループによる並列処理が主体であるために、A, b, y に対しては一貫性制御をコンパイル時に解決し、x については実行時オーバーヘッドの少ない方法で動的な一貫性制御を行うように最適化できた。これらの最適化は、ループの各反復が各スレッドにどのように割り当てられ、どの配列要素が参照されるかをコンパイル時に解析できるために行うことが出来た。マルチスレッドライブラリを直接用いて記述したプログラムでは、このような最適化をコンパイラで行うのは困難である。

6. ま と め

OpenMP プログラムを SMP クラスタ上で実行するための、コンパイラおよび実行時ライブラリを設計・開発した。本方式は、以下の特徴を持つ。

- OpenMP プログラムから共有データのデータフローを解析し、正確な解析が可能な場合には、ノード間で共有されるデータに対してリモートコピーを用いて静的な一貫性制御を行う。正確に解析できない場合には、ソフトウェア DSM によって実行時に一貫性制御を行う。
- OpenMP プログラムはマルチスレッドライブラリを直接用いた並列プログラムよりもスレッド間の依存関係を解析するのが容易であるため、最適化をより効果的に行える。
- SMP クラスタであるために、同一ノード内のスレッド同士では物理的にメモリを共有していることを利用し、同一ノード内の複数のスレッドで共有されるデータの一貫性制御のためのオーバーヘッドを削減できる。

OpenMP で書かれた簡単なデータ並列プログラムを用いてこれらの特徴を具体的に示した。今後は、広範囲な OpenMP プログラムによる評価と、コンパイラによる最適化および実行時ライブラリの改良を行って行く予定である。

参 考 文 献

1) OpenMP: Simple, Portable, Scalable SMP Programming (http://www.openmp.org/).
2) OpenMP Architecture Review Board, OpenMP Fortran API Ver.1.0, Oct. 1997.
3) OpenMP Architecture Review Board, OpenMP C and C++ API Ver.1.0, Oct. 1998.
4) Honghui Lu, Y.Charlie Hu, and Willy Zwaenepoel, OpenMP on Networks of Workstations, Supercomputing '98.
5) OdinMP: A Free, Portable OpenMP Implementation for Fortran (http://www.it.lth.se/~d92jh/odin.html).
6) Daniel J. Scales and Kourosh Gharachorloo, Design and Performance of the Shasta Distributed Shared Memory Protocol, ICS '97.
7) 丹羽純平 稲垣達氏 松本尚 平木敬, 非対称分散共有メモリ上における最適化コンパイル技法の評価, 情報処理学会論文誌, Vol.39, No.6, 1998.
8) 草野和寛 佐藤三久 佐藤茂久, OpenMP コンパイラの試作と評価, 情報処理学会研究報告 99-HPC-76, 1999.
9) 田中良夫 松田元彦 久保田和人 佐藤三久, COMPaS: Pentium Pro を用いた SMP クラスタとその評価, 情報処理学会論文誌, Vol.40, No.5, 1999.
10) 松田元彦 田中良夫 久保田和人 佐藤三久, SMP クラスタ向けネットワーク・インターフェース AM 通信, 情報処理学会論文誌, Vol.40, No.5, 1999.