応用数学工学特論

山本有作

２００６年4月２０日

目 次

1　序論

今日、プロセッサの動作周波数の向上は飽和に達しつつあり、また専用スーパーコンピュータの設計コストの増加も無視できない程になっている。そこで考案されたのが並列計算機である。並列計算機とは複数のプロセッサを並列に繋いだもので、プロセッサ数を増やすことでピーク性能を無制限に向上させることが可能である。また汎用のプロセッサを使用することで設計コストの大幅な削減が可能である。

並列計算機を有効に利用するためには、複数のプロセッサに均等に演算を分担し効率良く働かせることが重要になるため、良い並列化アルゴリズムが必要になる。良い並列化アルゴリズムは計算環境によって異なり、並列計算機の種類に合わせたプログラミングが必要となる。並列計算機には、共有メモリ型並列計算機、分散メモリ型並列計算機、SMP クラスタがある。

2　共有メモリ型並列計算機

共有メモリ型並列計算機 (SMP) は、複数のプロセッサユニット (PU) がバスを通してメモリを共有したものである (図 1)。PU はそれぞれキャッシュを持つ。メモリ空間が単一であるためプログラミングが容易である一方で、PU の数が多すぎるとアクセス競合によって性能が低下するという欠点がある。そのため共有メモリ型並列計算機を運用するときは 4〜8 台程度の PU を用いる場合が多い。

プログラミング言語には OpenMP(FORTRAN/C/C++ + 指示文) を使用する。

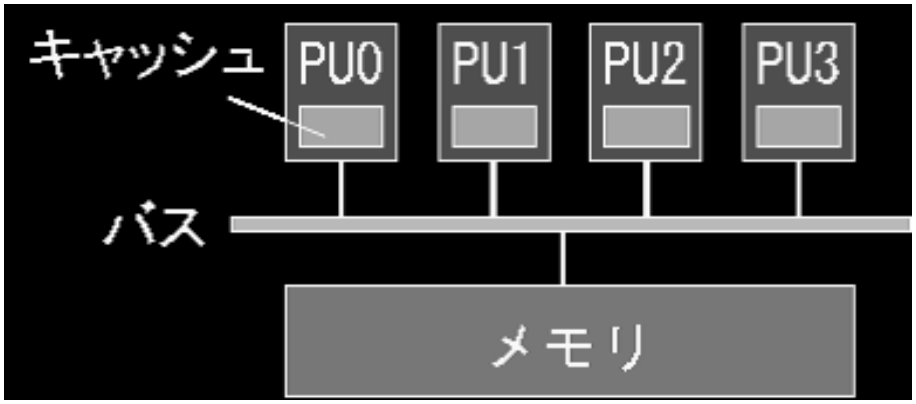

図 1: 共有メモリ型並列計算機

3　分散メモリ型並列計算機

分散メモリ型並列計算機は、各々がメモリを持つ複数のプロセッサユニット (PU) をネットワークで接続したものである (図 2)。PU はそれぞれキャッシュを持つ。各々がメモリを持つためアクセス競合が起きることはなく、数千〜数万 PU 規模の並列化が可能である。ただしメモリ空間が単一でないため、PU 間のデータ通信が必要であ

る。そのため PU 間のデータ分散を意識したプログラミングが必要になる。
プログラミング言語には FORTRAN/C/C++ + MPI を使用する。

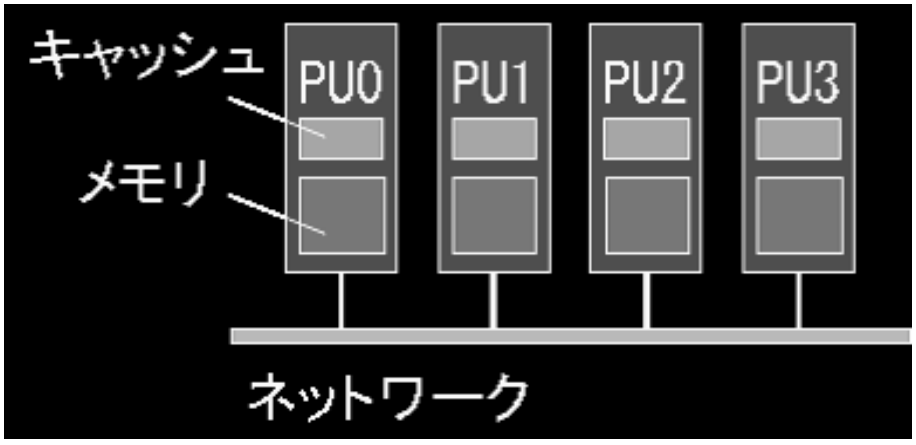

図 2: 分散メモリ型並列計算機

4 SMP クラスタ

SMP クラスタは、複数の共有メモリ型並列計算機 (SMP) をネットワークで接続したものである (図 3)。各ノードの性能を高くできるため、比較的少ないノード数で高性能を達成することができる。

プログラミングは、ノード内部の計算では共有メモリ型並列計算機として、ノードをまたがる計算では分散メモリ型並列計算機として行う。すなわち MPI と OpenMP とを組み合わせて使用する。

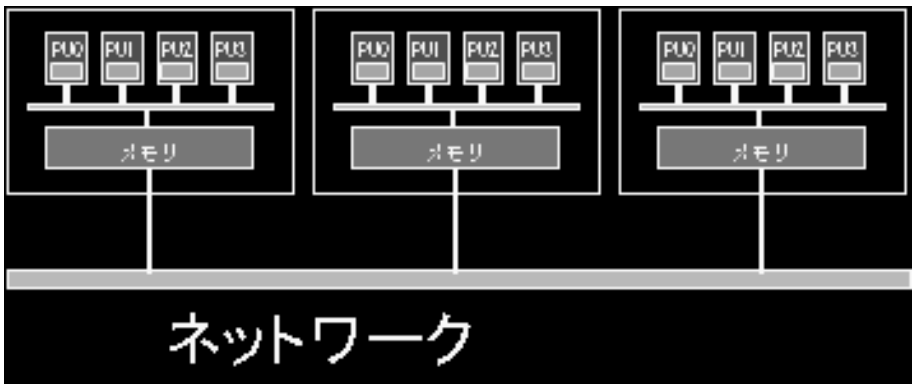

図 3: SMP クラスタ

5　並列化効率

並列計算の実行時間は、

$$\text{並列計算実行時間} = \text{演算時間} + \text{通信時間} + \text{待ち時間}$$

であり、単純に演算時間に等しくはならない。通信時間は PU 間のデータ通信にかかる時間、待ち時間は PU 間のデータ通信が必要になったが一方の PU の演算が途中までしか済んでない場合の演算処理待ち時間である。

p 台のプロセッサで並列化した場合の並列化効率は、

$$\text{並列化効率} = \frac{1\,\text{プロセッサでの実行時間}}{p\,\text{プロセッサでの実行時間} \times p}$$

で表される。並列化効率は 1 になるのが理想であるが、実際には通信時間や待ち時間が発生するため 1 より小さい値になる。並列化効率が 1 に近いほど、良い並列化アルゴリズムと言える。

6　OpenMP とは

OpenMP とは、共有メモリ型並列計算機上での並列プログラミングのためのプログラミングモデルである。ベースとなるプログラミング言語 (FORTRAN/C/C++) 中にディレクティブ（指示文）を挿入することで、元のプログラムを並列プログラムに拡張することが出来る。

6.1　OpenMP の実行モデル

OpenMP の実行モデルとしては、Fork-join モデルが用いられている。これは、並列化を指定しない部分は逐次的に実行され、指示文で指定された部分（並列構文）のみを複数のスレッドで実行する、というモデルである。この OpenMP で記述されたプログラムは、マスタスレッドと呼ばれる 1 つのプロセスとして実行を開始し、並列構文に突き当たるまで、逐次的にプログラムを実行する。並列構文に突き当たった時には、マスタスレッドはスレッドの team を生成し、マスタスレッドはその team のマスタとなる。並列構文に囲まれた文は、その中の文から呼び出されたスレッドルーチンを含めて、team のそれぞれのスレッドによって並列に実行される (fork)。並列構文の終わりでスレッドは同期し、マスタスレッドのみが実行を続ける (join)。このような並列構文を 1 つのプログラムで複数回用いることが出来る。したがって、プログラムは実行中に何度も fork/join を行うことになる。ただし、各スレッドは同じメモリ空間をアクセスすることになる。

6.2　OpenMP プログラムの構成要素

OpenMP を用いたプログラムは、簡単に以下のような構成要素から成る。

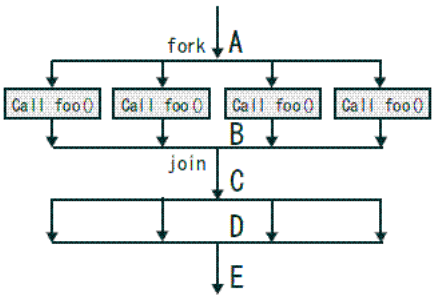

図 4: Fork-join モデル

・元のプログラム　通常の FORTRAN または C/C++によって書かれたプログラム。

・指示文　並列化すべき場所 (parallel region)、並列化方法を明示、指定する文。

・ライブラリ関数　並列実行部分で様々な働きを行う関数。
（例）omp_get_thread_num ･･･ 関数を実行したスレッドのチーム内の番号を返す。
omp_get_num_thread ･･･ 並列リージョンを実行しているチームのスレッド数を返す。
omp_set_num_thread ･･･ 並列リージョンで使用するスレッド数を指定する。

・環境変数　プログラムの並列実行に関する様々な環境を指定する変数。
（例） OMP_NUM_THREADS ･･･ 並列実行中に使用するスレッド数を指定する。

6.3 OpenMP を用いた簡単なプログラム例

6.3.1 並列実行を行うスレッド番号を出力するプログラム

下記のプログラムをコンパイルし、実行する。

```
program hello
integer omp_get_thread_num        ← スレッド番号を取得するライブラリ関数
write(6,*) 'program start.'
!$omp parallel        ← 指示文：並列実行部分の開始
write(6,*) 'My thread number = ',omp_get_thread_num()
!$omp end parallel        ← 指示文：並列実行部分の終了
stop
end
```

環境変数OMP_NUM_THREADS を 2 に設定したとき、実行結果は以下のようになる。

```
My thread number = 0
```

```
My thread number = 1
```

6.3.2 ベクトルの加算 $\mathbf{z} = a\mathbf{x} + \mathbf{y}$

下記のプログラムをコンパイルし、実行する。

```
program axpy
integer i
double precision a,x(100),y(100),z(100)
（a,x(100),y(100),z(100) の値を設定）
!$omp parallel do        ← 指示文：直後の do ループの並列化を指示
do i = 1,100
    z(i) = a*x(i) + y(i)        ← ベクトルの加算 z = ax + y
end do
（z(100) の値を表示）
stop
end
```

環境変数OMP_NUM_THREADS を 2 に設定すると、$1 \leq i \leq 50$ がスレッド 0、$51 \leq i \leq 100$ がスレッド 1 で計算される。

6.4 共有変数とプライベート変数

OpenMP を用いたプログラムでは、全ての変数、配列に対してそれぞれ、共有変数 (shared) かプライベート変数 (private) かのどちらかの属性が与えられる。各属性について、以下に簡単に説明する。

共有変数 この属性を持った変数、配列は、各スレッドで同じ値を参照もしくは変更する。すなわち、並列実行中のチーム内のどのスレッドからも書き込み、もしくは読み出しが可能な変数を意味する。変数、配列に共有変数の属性を指定する際には、並列化指示文の後に shared 節を追加すれば良いが、各変数、配列のデフォルト属性が shared であるために通常は指定する必要は無い。

プライベート変数 この属性を持った変数、配列は、各スレッド各々で異なった値を持つ。ループインデックス変数（i,j,k 等）は、異なったスレッドで共有してしまうと正しい制御ができない。そこで、このようなループインデックス変数についてはプライベート変数とする必要がある。変数、配列にプライベート変数の属性を指定するには、並列化指示文の後に private 節を追加すれば良い。

上で述べたように、何も指示をしない変数については基本的に共有変数となる。しかし、並列化されたループのインデックス変数のように、明らかにプライベート変数

でなければならない変数については、特に指示をしなくてもプライベート変数となる（ただし多重ループの場合は、並列化対象ループのインデックス変数のみプライベート変数となる。）。

6.5 変数の属性指定のプログラム例

6.5.1 2 重ループの並列化（行列ベクトル積）

下記のプログラムをコンパイルし、実行する。

```
program gemv
integer i,j
double precision a(100,100),x(100),y(100)
（a(100,100),x(100) の値の設定）
!$omp parallel do private(j)        ← j をプライベート変数に指定
do i = 1,100
    y(i) = 0.0d0
    do j = 1,100
       y(i) = a(i,j) * x(j)
    end do
end do
（y(100) の値を表示）
stop
end
```

上プログラムでは、a,x,y は自動的に共有変数となっている。また、指示文!$omp parallel do の直後にくるループのインデックス変数i は、明らかかにプライベート変数でなければならないとみなされ、自動的にプライベート変数となる。しかし、プライベート変数となるべきもう一つのループインデックス変数j は、並列化対象ループのインデックス変数ではないために、自動的にはプライベート変数とはならない。よって、明示的な指定が必要となる。

6.6 リダクション変数

以下のベクトルの内積のプログラムを考える。

```
program dot
integer i
double precision x(100),y(100),c
（x(100),y(100) の値を指定）
c = 0.0d0
do i = 1,100
```

```
    c = c + x(i)*y(i)
end do
（c の値を表示）
stop
end
```

このプログラムにおいて並列化を行う際の、ベクトルの内積の結果を表す変数c の属性について考える。

変数c を共有変数と指定した場合、ベクトルの総和は正しく計算できない。これは、共有変数に対して各スレッドが同時にアクセスするために起こる問題で、排他制御（指定した一連の処理について同時に実行できるスレッド数を１つに制限すること。）を行うことで回避できるが、この方法を用いると、ループ内で同時に１つのスレッドしか処理を進めることが出来ないため、並列化することによる処理時間の短縮は望めない。

また、プライベート変数と指定した場合を考える。このとき、c の値は各スレッドで異なった値を持つが、並列実行部分が終了し、マスタスレッドにプログラムの実行が移る時に、c の値はどのスレッドが持っていたものか解らなくなってしまう。よってこの時も、ベクトルの内積を正しく行うことが出来ない。

このような問題に対して、並列化による処理時間の短縮を有効にしつつ正しい値を求めるための方法としてリダクション変数の利用が挙げられる。

リダクション変数　ベクトルの総和や最大値の探索等の「リダクション計算」と呼ばれる計算を並列に実行するための変数。リダクション変数は、共有変数とプライベート変数の両方の性質を持っている。並列ループが開始され、各スレッドがループを実行している間はプライベート変数として扱われる。そのため、各スレッドはそれぞれのリダクション変数に対して独自の値を持つことになる。その後、全スレッドが並列ループの処理を終了すると、各スレッドのリダクション変数の値は１つにまとめられ、共有変数のようにマスタスレッドの対応する変数に格納される。このリダクション変数を１つにまとめる際に、和、積、最大値等の計算処理が行われる。リダクション変数としたい変数がある場合、指示文に reduction 節を追加する。上述のベクトルの内積を例にして、リダクション変数を用いると、以下のようなプログラムになる。

```
program dot
integer i
double precision x(100),y(100),c
（x(100),y(100) の値を指定）
c = 0.0d0
!$omp parallel do reduction(+:c)        ← c をリダクション変数（総和型）に指定
do i = 1,100
    c = c + x(i)*y(i)
end do        ← ここでc の値が合計される。
```

```
（c の値を表示）
stop
end
```

このようにリダクション変数を用いることで、総和型のループも正しく並列化することができる。

6.7 ループへのスレッド割り当ての指定

並列実行する際に、インデックス変数の値の範囲をどのように各スレッドに分割し割り当てるか、という問題が生じる。デフォルトではインデックス変数はスレッド数分に分割され、各部分が各スレッドに割り当てられる。行列とベクトルの積の計算を考える場合、その行列が密行列の場合はデフォルトのままで問題ないが、行列が疎行列、特に上三角行列のような場合、スレッド間に負荷の不均等が生じ、並列計算の効率の面から考えると、多くの無駄が生じることになる。このような場合は、一定の長さ l のブロックを周期的にスレッドに割り当てるブロックサイクリック割り当ての採用が有効である。

6.7.1 ブロックサイクリック割り当てのプログラム例

上三角行列とベクトルの積を並列計算するプログラムを考える。

```
program gemv
integer i,j
double precision a(100,100),x(100),y(100)
（a(100,100),x(100) の値を設定）
!$omp parallel do private(j) schedule(static,10)
                  ← ブロック幅 10 のブロックサイクリック割り当て
do i = 1,100      y(i) = 0.0d0
    do j = 1,100
       y(i) = a(i,j) * x(j)        ← 上三角行列とベクトルの積
    end do
end do
（y(100) の値を表示）
stop
end
```

上記のプログラム例では、コンパイル時に割り当てを決める静的 (static) 割り当てが行われる。この他に、実行中に適宜割り当てが行われる動的 (dynamic) 割り当てや、環境変数による割り当て方法の指定も可能である。

6.8 セクション型の並列化

これまで述べてきたのは主に do ループの並列化であるが、各スレッドに別々の仕事を並列に行わせたい場合は、セクション並列化を指定することができる。

```
program main
（逐次処理の部分）
!$omp parallel sections        ← セクション並列化の開始
!$omp section
（スレッド 0 の処理内容を記述）
!$omp section
（スレッド 1 の処理内容を記述）
!$omp end parallel sections        ← セクション並列化の終了
（逐次処理の部分）
stop
end
```