論理スレッドの纏め上げ方式を用いた超軽量スレッド実行（メタスレッド）方式

明石　健太郎†　松尾　啓志†

SMP 計算機上で効率よく，また簡潔にプログラムを記述するためによく使われる方法はスレッドを用いた記述方法である．しかし，生成されるスレッドの数が非常に多い場合には，スレッド生成などのオーバーヘッドは非常に大きい．そこで，スレッド生成などの処理を OS ではなくライブラリが行うことにより記述の効率性を極力保ったまま効率の良い処理を行うメタスレッド方式を提案する．これは，プログラマによるスレッド生成に対して実際にはスレッドを生成せず，少数の実スレッド上で逐次に実行を行うものである．

また，プログラムの記述を分かりやすくするためにスレッド生成方法として良く知られる pthread ライブラリに似たインターフェースを持つライブラリとして実装を行った．しかし，pthread ライブラリではスレッド間の依存関係を記述することは難しい．そこで，スレッド間の依存関係を記述するために拡張を行った．

The super lightweight thread execution (meta-thread) method using the logical thread collecting

Kentarou AKASHI† and Hiroshi MATSUO†

The description method using thread is commonly used to describe a program efficiently and briefly on an SMP computer. However, when too many threads are generated, it will lead to large overheads such as thread generation. In this study, we propose a meta-thread system that performs efficient processing, which is not OS, but a library that processes thread generation Moreover, the efficiency of description is maintained as much as possible. This does not generate a thread in fact to the thread generation by the programmer, but performs sequentially on a small number of real thread.

Moreover, we implemented as a library with the interface similar to the pthread library well known as the thread generation method in order to make description of a program intelligible. However, in a pthread library, it is difficult to describe the dependency between threads. To solve this problem, we propose an extension to describe the dependency between threads.

1. まえがき

近年，計算機の低価格化に伴い，SMP 環境などの並列処理環境の構築がより容易なものとなっている．

しかし，SMP 環境で並列処理プログラムを記述するためには，一般的には並列化可能な箇所をすべてスレッドとして記述する，または実行順序を考慮してスケジュールを行いながら実行するプログラムを記述する必要がある．

すべて並列化する方法は，並列数が非常に大きくなった場合，スレッド生成やコンテキストスイッチなどのスレッド管理に非常に多くの時間が必要となり，また OS により管理可能な限界数がありこれを越えてスレッド生成を行えないという問題がある．プログラマによりスケジュールを行う方法は，スレッド管理の問題は無視できるが，プログラムが煩雑になる可能性がある．

これらの問題を解決するため，特に大規模な並列性を有する問題に対する手法が多数提案されている．多田らは longjmp 命令を用いることにより，汎用性が高く，軽量な疑似スレッドライブラリを提供している [1]．Cilk[2]，StackThreads[3] では，再帰並列などに見られる，タスク間の指向性を利用し，大規模並列処理に対応している．また VIOS[4] では，スレッドを単純なループによる連続処理へ置き換える．しかし，これらの手法はプログラムの記述という点では，C/C++言語に似せるなど既存のプログラミング言語を参考にしているが，制御関数を多く記述する必要がある．また，専用言語を利用しているため，独特な記述が必要とな

† 名古屋工業大学電気情報工学科
Nagoya Institute of Technology, Electrical and Computer Engeneering

り，プログラム記述に対する汎用性は高いものではない．また，これらの手法はデータ並列のみ，再帰問題のみといった単純な問題のみを対象としており，対象としない問題に対しては実行に時間がかかる，または記述不可能である手法も存在する．

そこで，本論文では，

- 大量のスレッド生成に対して実際にスレッド生成を行うことなく，少数の実スレッド上で逐次的に実行を行うことによりスレッドの大量生成時の問題を克服する．
- 論理スレッド間に対する依存関係の記述を可能とするために記述方式に拡張を導入する．
- 一般的に良く知られるスレッドライブラリである pthread ライブラリと似たインターフェースを導入することにより汎用性を持ったプログラム記述を可能とする．

を実現するメタスレッドライブラリを提案する．

本論文では，2 章でまずメタスレッド方式について示し，3 章でメタスレッドライブラリにおけるプログラミングモデルについて示し，4 章で pthread ライブラリとのスレッド生成能力に対する比較，実行速度に対して性能評価を行い，5 章で現時点におけるメタスレッドの問題点についての考察を行い，6 章でまとめる．

2. メタスレッド方式

非常に多くのスレッドを実際に生成することは，スレッド生成やコンテキストスイッチなどのオーバーヘッド，OS によるスレッド生成数の限界といった問題が存在する．そこで，これらの問題を解消するため，メタスレッド方式を提案する．

メタスレッド方式は，プログラマによる多量のスレッド生成により，

(1) 生成される論理スレッドを少数の実スレッド上に纏め上げる．
(2) 実スレッド上において逐次的に論理スレッドを実行する．

方式である．

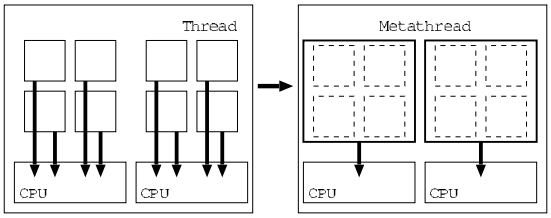

図 1 メタスレッドの概念
Fig. 1 The concept of the Metathread

これはスレッド多数生成の問題があるために，図 1 に示すように，プログラマにより生成される論理スレッドが独立にスレッド上にて実行される方式ではなく，実スレッドに対して複数の論理スレッドを割り当て，論理スレッドを逐次的に実行する方式である．

このような纏め上げを行うことにより，OS に管理されるスレッドの数を少数に保つことが可能であるため，実行速度の向上が期待できる．

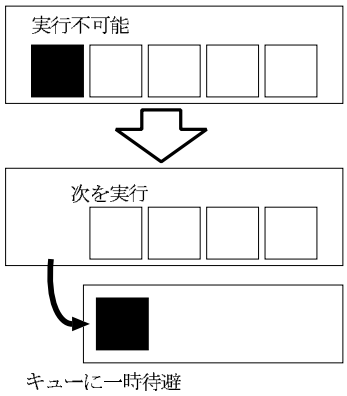

図 2 論理スレッドが実行不可能な場合のキュー操作
Fig. 2 The queue operation is executed when logical thread cannot execute

また，本手法では，スレッド間に対する依存関係を記述可能としている．依存関係が存在する場合には，メタスレッド上での論理スレッドの単純な逐次実行はデッドロックを引き起こす．そこで，依存関係判定を行い，図 2 のように依存関係により実行が不可能であると判定された論理スレッドのコンテキストをいったんキューにため，次の論理スレッドの実行を継続する．また，論理スレッドの逐次実行が終了した後，キューにためた論理スレッドの実行を行う．このようにキューを利用することによりデッドロックの危険性を回避する．

2.1 論理スレッドの纏め上げ

メタスレッド方式では生成される論理スレッドの纏め上げを行うが，纏め上げを行うことにより，

- スレッドを生成することなく，論理スレッドを実行する必要がある．
- 問題の特性を考慮することで効率のよい論理スレッド実行が可能となる．

が達成される．

例えば，画像フィルタ等，非常に単純な処理であり完全に並列化可能な処理では，CPU キャッシュなどの OS，システムに由来する問題を除けば，論理スレッドの実行順序の変更を行ったとしても実行時間は変化しない．しかし，DP マッチングなどの論理スレッド間に依存関係が存在する場合などにおいては，実行順

序の変更が実行時間に対して大きく影響を及ぼす．

依存関係が存在し実行不可能である場合，キュー操作を行う，または実行不可能な論理スレッドが実行可能となるまで実行不可能な論理スレッド群全体の処理を停止するといった対策が可能ではあるが，実行効率の観点から良い方法ではない．このことより，より問題に適した分割を選択する必要がある．メタスレッドライブラリでは，多次元に整列した論理スレッド群に対して各次元に対してブロック分割もしくはサイクリック分割をプログラマが選択する．

2.2 処 理 方 式

本手法におけるメタスレッド方式の実現方法について述べる．

スレッド生成命令が実行されると，CPU 数に合わせてどの実スレッドがどれだけの論理スレッドを実行するかを決定し，実スレッドに割り当て範囲を伝える．その後，図 3 に示すように，実スレッド上では与えられた論理スレッドすべてをループを用いて逐次実行する．この時，3.3 節に述べる依存関係判定により，実行不可能と判定された場合には，キューに入れ，他の論理スレッドを実行する．逐次実行が終了したとき，キューが空になるまでキュー内の論理スレッドを実行する．

3. プログラミングモデル

本手法におけるプログラミングモデルは，原則として C/C++ 言語上で pthread ライブラリを用いた記述法に準ずる．pthread ライブラリは C/C++ 言語上でのスレッド生成法として一般に良く知られており，プログラムの記述性は良いと言える．そこで，本手法においてこの記述法を選択した．

3.1 スレッド生成

スレッド生成のプログラム記述について説明する．

図 4 に pthread ライブラリを用いた場合と同等の記述方式で記述したメタスレッドライブラリを用いたプログラム例を示す． 図 4 では，実スレッド生成数を指定後，ループを用いて h× w 個のスレッドを生成し，その後すべてのスレッドの終了を待つプログラムを記述している．

しかし，大量のスレッド生成を行う場合，スレッド生成 API 等のオーバーヘッドが問題となるため，ループ内でのスレッド生成は非効率である．そこでスレッド生成法を拡張した．図 5 に，メタスレッドライブラリのスレッド生成拡張を用いたプログラム例を示す．

メタスレッドライブラリによるスレッド生成拡張では，ループを用いる多数スレッド生成ではなく，図 5 の 5 から 9 行目に示すように，ループに関するパラメータを関数の引数として与えることにより，一度の関数呼び出しで多数の論理スレッドの生成を行う．この時，メタスレッドライブラリ内部での動作は，パラメータを用いてループに展開し，ループを用いて多数の論理スレッドを実行する．また，このときのループ変数を各論理スレッドに渡す．

```
thread_execute( ){
  mt_index p;
  for(int i=b_h; i<e_h; i+=s_h ){
    for(int j=b_w; j<e_w; j+=s_w){
      Set_Index( p , i , j);
 //論理スレッドの実行
      int result = UsrFunc(task ,p);
      if(result == EXIT_FILURE)
 //実行不可能ならばキューに格納する
        queue[i].enqueue(p);
    }
  }
//キューが空になったら終了
  while( queue[i].num != EMPTY){
    p = queue[i].dequeue();
//論理スレッドの再実行
    int result = UsrFunc(task ,p);
    if(result == EXIT_FILURE)
//実行不可能ならばキューに戻す
      queue[i].enqueue(p);
  }
}
```

図 3 メタスレッド方式内部処理
Fig. 3 Inside processing of a meta-thread system

また，図 5 の

```
mthread_t tid;
```

は，複数の論理スレッド ID を扱うための構造体である．

3.2 スレッド実行

図 5 のスレッド生成に対するスレッドプログラム例を図 6 に示す．

図 5 のスレッド生成によって，生成された論理スレッドそれぞれに一意のループ変数が渡されるため必要であれば， GetIndex() を用いてループ変数を取り出す．その後，本来の目的である並列処理を記述する．

3.3 依存関係判定

pthread を用いた場合，スレッド間の依存関係を記述するためには，フラグ変数によりスレッドの実行可能性を判定し，実行不可能である場合，条件変数を用

```
int main(){
//初期化等は省略
//実スレッド数の指定
  mthread_setconcurrency( thread_num );
//スレッドの生成
  for(int i = 0;i < h;i++){
      for(int j = 0;j < w;j++){
          arg.h=h;arg.w=w;arg.m=matrix;
          mthread_create(&tid[i][j],NULL
          ,(pUsrFunc)UsrFunc,(void*)arg);
      }
  }
//生成したスレッドをすべて待つ
   for(int i = 0;i < h;i++){
      for(int j = 0;j < w;j++){
          mthread_join(tid[i][j] , NULL);
       }
  }
  return 0;
}
```

図 4 pthread ライブラリの記述法に準拠した並列プログラム
Fig. 4 The parallel program based on the method of describing a pthread library

```
int main(){
//初期化等は省略
  mthread_setconcurrency( thread_num );
  mthread_t tid;          //スレッド識別子
  mt_index p;
  p.dimension = 2;      //何重ループかを指定
  p.index[0][0] = 0;      //ループ範囲を指定
  p.index[0][1] = h-1;
  p.index[1][0] = 0;
  p.index[1][1] = w-1;
  mthread_create(&tid,NULL //スレッドの生成
    ,(pUsrFunc)UsrFunc,(void*)matrix,&p);
//スレッドの終了待ち
  mthread_join(tid , NULL);
  return 0;
}
```

図 5 メタスレッドライブラリを利用した並列プログラム
Fig. 5 The parallel program using the meta-thread library

いて実行可能となるまでスレッドの実行を停止する．また，フラグ変数に対して排他制御が必要である．このように pthread を用いたプログラミングにおける

```
int UsrFunc(void* data , mt_index * p ){
    ////初期化は省略
//論理スレッドに割り当てられたループ変数を取り
出す
    int h = GetIndex( p , 0);
    int w = GetIndex( p , 1);

    result[h][w]= 255 - matrix[h][w];
    return EXIT_SUCCESS;
}
```

図 6 並列処理を記述したプログラム
Fig. 6 The program which described parallel processing

スレッド間の依存関係の記述は煩雑である．そこで，スレッド間の依存関係を記述するために独自の拡張を行った．

```
int UsrFunc(void* data , mt_index * p ){
    ////初期化は省略
    int h = GetIndex( p , 0);
    int w = GetIndex( p , 1);

    CHECK(p,h,w-1);        //依存関係判定

    matrix[h][w]=2*matrix[h][w-1];
    return EXIT_SUCCESS;
}
```

図 7 依存関係の存在する並列プログラム
Fig. 7 A parallel program with a dependency

図 7 に示されるように依存関係判定に CHECK() を用いる．引数には実行位置情報に続き，依存するスレッドの位置情報を与える．このように記述することで，対応するスレッドが終了したかを判定する．

依存関係判定を行い，終了していないと判定された場合には，この関数は終了し，再びこの関数の最初から開始される．つまり依存関係判定は，図 7 のように目的の並列処理より前に記述する必要がある．なぜなら処理の副作用により，プログラムの結果が変わる可能性があるからである．

4. 性能評価

本手法を取り入れたメタスレッドライブラリを用い，2 つの実験を行った．

- pthread ライブラリとのスレッド生成能力につい

ての比較

- CPU 数による速度向上率

4.1 pthread ライブラリとの比較

pthread ライブラリ，メタスレッドライブラリを用いた DP マッチングプログラムを作成し，スレッド生成能力について比較を行った．

評価環境は CPU:Pentium3 866MHz x 2 ，Memory:512Mbyte , OS:Solaris8 の SMP 計算機を用いた．

DP マッチングの並列化手法は，DP マッチングの

$$g(i,j) = min\begin{cases} g(i-1,j) + d(i,j) \\ g(i-1,j-1) + 2d(i,j) \\ g(i,j-1) + d(i,j) \end{cases}$$

式の 1 計算に対して 1 スレッドを割り当てた. 依存関係を解決するために pthread では条件変数，メタスレッドではメタスレッド拡張，依存関係判定を利用した．

表 1 DP マッチングの実行時間 (sec)
Table 1 Execution time of DP matching

問題のサイズ	40x40	80x80	161x161	4096x4096
pthread	0.36	1.37	4.84	-
metathread	0.018	0.018	0.018	48.7

問題サイズの最大値が， 161x161 とした理由は，pthread ではこれより問題サイズを大きい場合に実行不可能であったからである. 実行不可能となる理由として，OS によるスレッド管理数の限界に達したと考えられる.OS によるスレッド管理数の限界は本評価環境の場合は，スレッド生成後スレッドが条件変数を用いて停止するという，スレッド生成数限界測定実験を行った結果， 3160 程度であった．ここでは，問題サイズの最大値がスレッド管理数の限界を大きく越えているが，一度にすべてのスレッドの生成を行わず，スレッド生成とスレッド実行，終了が平行して実行されているからであると考えられる. 従って，OS，CPU の性能，実行するプログラムなどにより実行可能な問題サイズの最大値は異なると考えられる．

pthread ライブラリを用いた場合，161x161 個のスレッド生成が限界であったが，メタスレッドライブラリを用いた場合には，4096x4096 以上 の論理スレッド生成を行うことが可能であり， pthread を用いた場合よりも非常に多くのスレッド生成を行うことが可能であった．

また，高速に実行可能であることが確認できる. ここで，メタスレッドライブラリを用いたプログラムがどのデータサイズにおいても同じ実行時間となっているのは実際にスレッド生成を行わないため，論理スレッド生成にかかるオーバーヘッドが小さく，DP マッチングに必要な時間もこれらのサイズでは非常に小さいためであると考えられる. そのため，メタスレッドライブラリの前準備に必要な時間が大きく影響を与える結果となっている．

また，pthread ライブラリを用いた DP マッチングプログラムから演算や排他制御を取り除き，純粋なスレッド実行時間の計測を行ったが実行時間はほぼ変わらなかった. このことより実行時間のほぼすべてを，スレッド生成処理に費やしていると考えられる．

4.2 SMP 環境での性能評価

メタスレッドライブラリの速度向上率を評価するために DP マッチング処理，画像フィルタ処理について評価を行った．評価環境は，富士通 汎用計算サーバ GP7000F model900 上で実験を行った．各処理は，一辺の長さが 1024，2048，4096 のそれぞれ 3 段階のデータサイズ，CPU の数が 1，2，4 のそれぞれ 3 段階に対して評価した．図 8，9 の横軸は CPU 数であり対数である，縦軸は速度向上率である．速度向上率は,「CPU を N 台利用した場合の実行時間 / CPU を 1 台利用した場合の実行時間」とした．

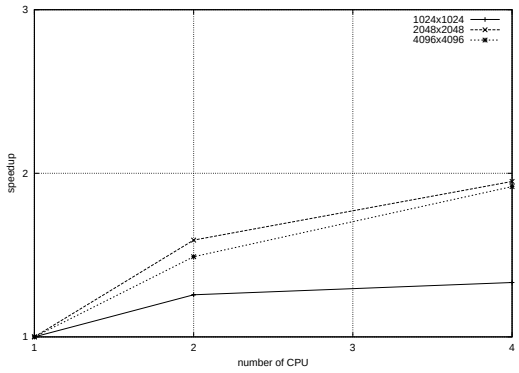

図 8 メタスレッドライブラリを用いた画像フィルタ処理結果 (横軸 CPU 数，縦軸台数効果)
Fig. 8 The filter processing result using the meta-thread library.

図 8 より，フィルタ処理において理論的には台数効果は CPU 数に比例する. しかしこの実験における台数効果は CPU 数 4，問題のサイズ 2048x2048 の場合に速度向上率は 1.95 でありこの結果は理論値に至らない. これはフィルタ処理に必要な計算量が非常に小さいため，メタスレッドライブラリの内部処理の比率が大きいためであると考えられる．

図 9 より，DP マッチング処理においてフィルタ処理と比べて台数効果は CPU 数 4，問題のサイズ 2048x2048 の場合に速度向上率は 2.95 となりフィルタ処理と比較し性能の向上が見られる. これは DP マッチングの処理にかかる計算量がフィルタ処理よりも大

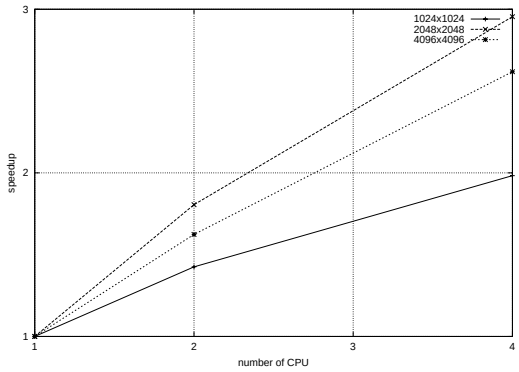

図 9 メタスレッドライブラリを用いた DP マッチング処理結果 (横軸 CPU 数，縦軸台数効果)
Fig. 9 The DP-matching processing result using the meta-thread library.

きく，メタスレッドライブラリの内部処理に対する DP マッチング処理の比率が大きいためであると考えられる.

また，4096x4096 の台数効果は 2048x2048 と比較し性能の低下が見られる. これはメモリ使用量が多く，その影響が出ていると考えられる.

5. メタスレッドの問題点と考察

本論文で議論しなかった、現時点におけるメタスレッドの問題点について以下に考察する。

5.1 スケジュール

論理スレッドが実行不可能である場合に，キューを用いているが，CPU 数の増加に伴い，後に開始した実スレッドが前に開始した実スレッドを追い越してしまうことがあり得る.DP マッチングなどの場合，この追い越しにより追い越したスレッドの残りの論理スレッドはすべて実行できず，キューに蓄えられる. キュー操作のオーバーヘッドは比較的大きいため実行速度は遅くなる. これを解決するためには依存関係を回避するために実行順序を自動的に変更する必要がある. その方法として，静的予測，動的予測などを導入することが考えられる.

5.2 再 帰 並 列

メタスレッドライブラリの現状においては，再帰並列を利用不可能である. メタスレッド方式は一度に大量のスレッドを生成する場合にスレッド間の依存関係の記述の簡潔性，実行処理の簡約化などによる高速な処理速度を求めるものであるため，再帰並列の適用は難しい.

しかし，メタスレッド方式とは別の枠組みで再帰並列を達成することは可能である. その方法として，論理スレッド内からのスレッド生成は，メタスレッドの枠組みを利用し、再帰並列を行う際には実スレッドを生成することにより新しい論理スレッドを実行するなどの方法が考えられる.

6. ま と め

メタスレッド方式を用いたスレッド生成ライブラリであるメタスレッドライブラリの実装と評価を行った.

メタスレッドライブラリを実際に用いたプログラムを用いた実験により，SMP 環境下における速度面における有効性を確認し，スレッド大量生成が可能であることを確認した.

今後の課題として、5 章で示した問題点を解決する手法について検討を行う予定である。

参 考 文 献

1) 多田好克, 寺田実: 移植性・拡張性に優れた C のコルーチンライブラリーの実現法, 電子情報通信学会論文誌,D-I, Vol. J73-D-I, No. 12, pp. 961–970 (1990).
2) Matteo Frigo, Charles E. Leiserson, and Keith H. Randall: *The Implementation of the Cilk-5 Multithreaded Language*, ACM SIGPLAN Conference on Programming Language (1998).
3) 田浦健次郎, 米澤明憲: 最小限のコンパイラサポートによる細粒度マルチスレッディング — 効率的なマルチスレッド言語を実装するためのコスト効率の良い方法, 情報処理学会論文誌, Vol. 41, No. 5, pp. 1459–1469 (2000).
4) 川脇智英, 松尾啓志 : 分散画像処理環境 VIOS-III, 電子情報通信学会論文誌 D-II, Vol. J84-D-II, No. 6, pp. 955–964 (2001).