OpenMPにおけるネストした並列性の実装と評価

田中義純† 田浦健次朗† 米澤明憲†

本論文では，OpenMPのタスクスケジューリングに遅延タスク生成に基づく動的負荷分散機構を導入することにより，入れ子状に並列化指示子が書かれている場合でも効率的な実行を可能にする実装を示し，その評価を行う．OpenMPは，共有メモリ計算機上でFortranやC/C++のプログラムを効率的に並列化するためのAPIであり，並列化したい部分にプラグマ指令などの並列化指示子を挿入させることによって，プログラムの並列化を表現する．しかしながら，並列化指示子が書かれている部分が再帰的に呼ばれるようなプログラムの場合，再帰するたびにスレッドを生成するような単純な実装だと，スレッド数が指数関数的に増加し，性能低下を招いてしまう．そのため，現在のOpenMPの仕様では，並列化指示子が入れ子状に出現した場合，その部分が並列に実行されるかどうかは実装依存となっており，現在の多くの処理系では並列には実行されない．この論文では，並列化指示子が入れ子状に出現する時の実装上の問題点を述べ，負荷分散が十分になされるための条件を示す．我々の方法では，OpenMPのタスクスケジューリングに，遅延タスク生成に基づく，細粒度スレッドを支援したStackThreads/MPライブラリを採用しているため，並列化指示子の出現ごとにスレッドを生成する単純な実装でも，最大限の並列性を引出し，かつ効率的に実行することができる．性能評価の結果，(1) 遅延タスク生成を導入したことによるオーバヘッドは5％程度に押さえられ，(2) 再帰的に並列化指示子が呼ばれるプログラムにおいても台数効果がでることが確認された．したがって，OpenMPに入れ子並列を実装することによる性能の低下はほとんどなく，また，StackThreads/MPのようなライブラリを用いることによって，実装の複雑さを非常に軽減することが出来ることが確認された．

Implementation and Evaluation of Nested Parallelism in OpenMP

Yoshizumi Tanaka,† Kenjiro Taura† and Akinori Yonezawa†

This paper describes an implementation scheme for efficient execution in dynamically-nested parallel regions in OpenMP. Our technique realizes low-cost dynamic load balancing based on Lazy Task Creation. OpenMP is an API for parallelizing sequential programs in Fortran or C/C++ on shared-memory multiprocessor machines. Programmers can parallelize their programs by annotating them with OpenMP parallelization directives. One of the most effective ways to enjoy a large amount of parallelism is to expose nested parallelism. The number of available parallel "logical tasks" grows exponentially if we extract parallelism in all nest levels. Thus, we would suffer from a large overhead in a naive implementation which straightforwardly maps a logical task to an OS-level thread. Since the OpenMP specification does not specify how parallel tasks are scheduled on multiprocessors, most of existent implementations fail to extract nested parallelism. This paper first explains problems in naive schemes for implementing nested parallelism and then gives a criterion to judge whether or not workload is distributed among processors effectively. Next we show our implementation that accomplishes efficient task scheduling even in the presence of dynamically-nested parallel regions. Our system utilizes a StackThreads/MP library, which supports fine-grain threads based on Lazy Task Creation. Since the library enables us to create a large number of threads with very small overhead, good performance can be obtained if we create a thread every time a logical task is created in nested parallel regions. Experimental results showed that (1) the overhead of dynamic load balancing was small (about 5% of whole execution time), and (2) we achieved high scalability even in programs calling parallel regions recursively. So we confirmed less performance fail by implementing nested parallelism in Open MP and we can implement very easily by using library like StackThreads/MP.

1. はじめに

並列計算機の発展に伴い，並列プログラミングの方式も多様化してきた．Message Passing Interface

† 東京大学大学院理学系研究科 情報科学専攻
Department of Information Science, Graduate School of Science, University of Tokyo

```
#pragma omp parallel for
  for (i = 0; i < n; i++) {
    work(i);
  }
```

図1 for ループの並列化
Fig. 1 Parallelization of a for-loop.

(MPI)[3]は，並列プログラミングの主な手法の1つであり，分散メモリ型並列計算機においては現在主流な手法である．一方，共有メモリ型並列計算機においては，近年スレッドプログラミングが一般的になってきており，多くの並列プログラムがスレッドを用いて作られるようになってきた．しかしながら，MPIやスレッドプログラミングは，プログラマがすべてのプロセス（スレッド）の挙動を理解してプログラムを書く必要があるため，並列プログラミングに慣れていない人にとっては大変な作業である．このようなプログラマの負担を軽減する1つの方向として自動並列化があるが，特に，並列化指示子を挿入しコンパイラにヒントを与えることによって自動並列化を行う手法は，High Performance Fortran[4]をはじめとして，様々な大学やベンダーで研究が行われている．

OpenMP[7]は，共有メモリ並列計算機上での並列プログラミングを容易に行うために開発されたAPIであり，プログラマは，C/C++やFORTRANのプログラムの並列に実行したいところにプラグマ指令である並列化指示子を挿入するだけで，その部分が自動的に並列化され実行される．例えば，スレッドプログラミングによってforループを並列化する時には，プログラマは，各スレッドが何番目のイテレーションを実行するかを考えながらプログラムを作成する必要があるが，OpenMPでは**図1**のように並列化指示子`parallel`を挿入してやるだけで自動的に複数個のスレッドが生成され並列実行される．もっとも単純にはここでプロセッサ台数分のLWPを生成すればよいと考えられるが，**図2**のように並列化指示子が入れ子状になっている（入れ子並列性と呼ぶ）ときは問題は複雑であり，並列化指示子`parallel`に遭遇するたびにプロセッサ台数分のスレッドを生成するような単純な実装では，性能が低下してしまう．一方でもちろん，`parallel`が入れ子になっていなければ，そこでわざわざプロセッサを余らせておく理由はない．

本論文は，遅延タスク生成[5]に基づく細粒度スレッドの機能を用いて，入れ子状に並列化指示子が出現する・しないにかかわらず，少ないオーバヘッドで常に十分な負荷分散がなされるようなOpenMPの実装方法について述べ，その性能を報告する．基本的な方式は，

```
#pragma omp parallel for
  for (i = 0; i < n; i++) {
    #pragma omp parallel for
      for (j = 0; j < n; j++) {
        work(i,j);
      }
  }
```

図2 入れ子状の for ループの並列化
Fig. 2 Parallelization of nested for-loops.

これまで多く研究されている，動的な並列度生成のためのプリミティブを持つ言語の実装方式に沿うもので，特に我々はStackThreads/MPライブラリ[9)~11)]を利用することで，基本的なスケジューリング機構を実現している．本論文の貢献は，OpenMPのセマンティクスに由来する微妙な問題の解決方法および，実装した結果得られた性能について報告することである．

以降の論文の構成として，2章でOpenMPの並列実行モデルについて説明し，3章で入れ子並列性を単純に実装したときの問題点を述べ，4章ではその解決方法としての我々の実装と，それに用いられているStackThreads/MPライブラリについて説明する．5章で実験内容について説明し，評価・考察を行い，6章で関連研究について触れ，最後に7章でまとめと今後の課題について述べる．

2. OpenMPの並列実行モデル

まず始めに，OpenMPの並列実行プリミティブと，典型的であると考えられている実装方式について述べる．

2.1 並列化指示子 parallel

プログラムが実行を開始する時には1つのLight Weight Process（以降LWP）が実行をはじめる（逐次実行）．逐次的に実行していたLWPが並列化指示子`parallel`に遭遇すると，複数のLWPが生成されて，それらのLWPがチームを構成する．そして，チーム内の各々のLWPが，指定された文を実行する．注意としては`parallel`自身はひとつの仕事を複数のプロセッサで並列に実行するためのプリミティブではなく，単に複数のLWPを生成するだけのプリミティブであるという点である．例えば**図3**を実行すると，作られたスレッドの数だけ文字列`hello`が表示される．

関数`omp_set_num_threads()`によって，`parallel`において生成されるLWPの数を指定することが出来るが，指定しなかった場合は実装依存である．

2.2 仕事共有子 for および sections

並列化指示子`parallel`によって生成されたLWPが仕事共有子`for`，`sections`に遭遇すると，それら

```
#pragma omp parallel {
  printf("hello");
}
```

図3 並列化指示子「parallel」の例
Fig. 3 Example of "parallel" construct.

```
#pragma omp parallel
  #pragma omp sections {
    #pragma omp section
      work1();
    #pragma omp section
      work2();
    }
```

図4 仕事共有子「sections」の例
Fig. 4 Example of work sharing construct "sections".

```
#pragma omp parallel
  #pragma omp for schedule(static,3)
    for (i = 0; i < n; i++)
      A[i] = B[i] + 1;
```

図5 仕事共有子「for」の例
Fig. 5 Example of work sharing construct "for".

の仕事はチーム内のLWPに分割されて実行される．注意としては，forやsectionsでは新たにLWPが生成されることはなく，parallelですでに生成されたチーム内のLWP間で，仕事を分割するだけであるということである．特に，forやsectionsは，必ずparallelの（動的な）スコープ内になくてはならず，そうでないforやsectionsはエラーである．

図4はsectionsの例で，parallelで生成されたLWPが各sectionを並列に実行する．

また，図5はforの例で，イテレーションをチーム内のLWPに分割して実行する．この時のスケジューリング方式として，static, dynamic, guidedの3種類を指定することができ，また分割するイテレーションの最低の単位をchunk_sizeとして指定することが出来る．これらはイテレーションをチーム内のLWPにどのように分割して実行するかを指定する．詳しくは文献7)を参照されたい．

2.3 入れ子並列性

先ほども説明したように，新たにLWPを生成するのはparallelだけであるので，OpenMPにおいて入れ子並列が出現するのは，parallel領域内を実行中に再びparallelが出現するときである．したがって，入れ子並列性の単純な実装法としては，parallelに遭遇するたびにLWPを生成するという方針が考えられるが，この方針では過剰なLWPが大量に生成されてしまい，生成やスケジューリングに大きなコストがかかってしまう．

したがって現在の仕様では，内側の並列性は無視してよく，実際，現在の多くの処理系ではサポートされていない．具体的には，parallelに遭遇した時に，すでにLWPのチームが生成されていたならば，そのparallelを単に無視するという方式が許されている．この方式は，最外側の並列化によって十分な並列性が引き出せ，かつ良好な負荷分散が得られるのであれば問題はないが，並列に樹状再帰呼び出しを行うプログラムなど，この方式では十分に負荷分散が行えないような問題も存在する．

2.4 スレッドIDのセマンティクスについて

これまで述べてきた基本的な実行方式：

- parallelにおいてLWPを生成し，
- forやsectionsにおいて仕事をLWP間で分割する，

はもちろん実行を理解するためのモデルであり，全ての実装が必ず従わなくてはいけない制約ではない．

例えばセマンティクスや性能という観点からは，parallelにおいてはLWPを作る代わりに1プロセッサが全プロセッサの動作を模倣し，forやsectionsに遭遇した時に初めてLWPを生成する，というような方針も可能である．極端に言えば，parallelにおいては1プロセッサが全プロセッサの動作を模倣し，forやsectionsは単に逐次的に実行する，というのもセマンティクスという観点からは正当である．実際我々は後にStackThreads/MPを用いて一定数のLWPを用いて入れ子並列性を抽出しようとしているので，このような実装方針の可能性について考察することは意味を持つ．

そのような，典型的ではない仕方でOpenMPを実装する時に保証しなくてはならないのは，プログラマから見た結果を，通常の実装と同等に保つことである．この時問題となるのがスレッドIDに関する問題である．

OpenMPには，いくつかのライブラリ関数が存在するが，その中でスレッドIDを得るための関数（omp_get_thread_num()）がある．スレッドIDというのは，スレッドチーム内における独自の番号のことで，単独で実行していたLWPがparallelに遭遇すると，スレッドIDが0のマスタースレッドになる．そして新たに生成されたLWPは，1から順番にID番号を振り分けられる．ここで，1番のLWPが再びparallelに遭遇したときには，新たに0番のIDを与えられて新しいスレッドグループのマスタースレッドになるのである．

典型的な実装法においては，単にチームを生成する

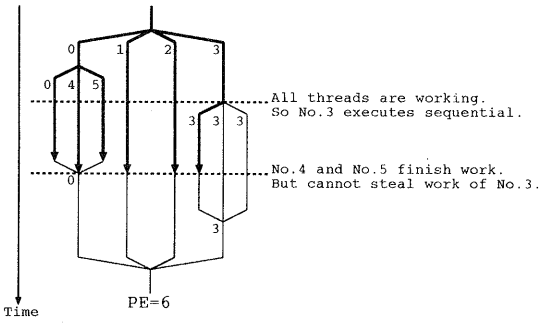


図6 並列化指示子に遭遇したときに分岐する方法
Fig. 6 A naive scheme to distribute tasks only at parallelization directives.

時に各LWPに番号を割り当て，LWP固有の領域にその番号を格納しておけば良い．そうでない実装法をする時は，この関数の実装を，セマンティクスに合致させるよう注意が必要である．

3. 入れ子並列性の既存の実装とその問題点

入れ子並列の実装方法として，入れ子並列に遭遇するたびにLWPを生成して並列に実行する方針は無駄なコストが多く，また，内側の並列性を無視する方針は負荷分散が不十分になる可能性がある．したがって，現実的な実装の方針としては，Processing Element（PE）と同数のLWPを生成し，各LWPに仕事を割り当てることが考えられるが，この方針も注意深く実装しないと負荷分散が十分になされない場合がある．

本章では，この方針にそった単純な実装方法を2つあげ，その問題点を指摘することにより，入れ子並列を実装する上で必要な条件を説明する．

3.1 実装例

3.1.1 並列化指示子に遭遇した時点で分岐する方法

まず最初にもっとも単純だと考えられる以下のような方針について考えてみる．プログラム開始時に固定数のLWPが生成され，そのうちの1つがプログラムの実行を開始する．残りのLWPは遊休状態であり，システムは遊休状態のLWPが格納されたプール（LWPプール）を管理する．その元で，

(1) 並列化指示子に遭遇するまでは単独で実行する
(2) 実行中のスレッドが並列化指示子に遭遇すると，その時点で遊休状態のLWPがプール内にいるかを調べ，`omp_set_num_threads()`で指定されている数だけの，またはその時点での全ての（大きくない方），遊休状態のLWPを用いてチームを生成する．
(3) 仕事共有子に遭遇すると，チームが生成されていればそれらの間で仕事を分割する．

各LWPは以下のような動作を繰り返す．

(1) 遊休状態では，実行中のどれかのLWPが並列化指示子に遭遇して，チームが作られるのを待っている．
(2) チームが作られると，必要な数だけのLWPがチームに加入し，そこでの仕事が終わると遊休状態に戻る．

この方針は，最外側で全てのLWPが使われるプログラムにおいては，入れ子並列性を無視する方針と同程度のオーバヘッドで実装でき，実装の複雑さも少ない．かつ，外側の並列性が極端に少ない場合（例えば樹状再帰のように，各並列化指示子においてはLWP数以下の並列度しか抽出されない場合など）には，ある程度，内側の並列性も抽出出来る利点があるが，負荷分散は不十分である．

なぜならば，一般にはある並列化指示子に遭遇した後に初めて遊休状態のLWPが出現することがあるからである．この方式では，並列化指示子に遭遇した時点で遊休状態のLWPがなければ，その並列化指示子を無視することに決めてしまい，後からその決断を取り消すことができない．

例えば，**図6**のような時には，3番LWPの仕事は分割されてさらに並列に実行されるべきであるのに，この方針では仕事が分割されていないために，4番，5番LWPが仕事を盗んで実行することが出来ないのである．

仕事をプロセッサ間で移動するコストが小さい共有メモリ計算機では，十分に負荷分散がなされていることを保証するための指針として，以下の**貪欲性**を保証するのが簡単である．

> **貪欲性：** 実行可能な仕事がn個，プロセッサの個数をTとすると，
> (1) $n > T$ なら T 個
> (2) $n \leq T$ なら n 個
> の仕事が常にプロセッサに割り当てられている．

つまり実行可能な仕事は，すべてのプロセッサがすでに他の仕事を実行しているのでない限り，常に実行されている，ということである．上で述べた方法は，明らかにこの条件を満たしていない．

3.1.2 並列化指示子に遭遇したら常に並列化する方法

先ほどの方法で負荷分散が不十分なのは，遊休状態のLWPが出現しても，すでに逐次実行を開始してし

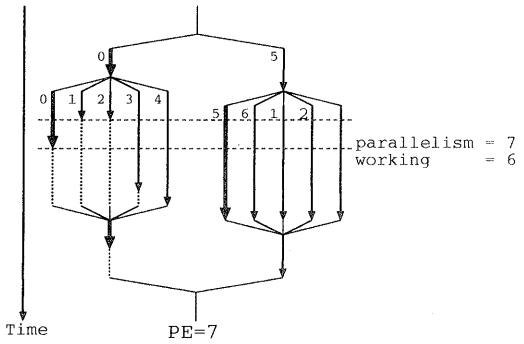


図7 すべての並列性を抽出する方法（しかしマスタースレッドは新たな仕事を獲得することが出来ない）
Fig. 7 A scheme to expose all parallelism. (A master thread cannot steal a new task.)

まった並列化指示子の仕事を獲得出来ないからである．そこで，次のような方針を考えてみる．LWP のプールを管理する点などは前節と同様である．その元で，

(1) 並列化指示子に遭遇するまでは単独で実行する
(2) 実行中のスレッドが並列化指示子に遭遇すると，その時点で遊休状態の LWP がいる，いないにかかわらず，常に `omp_set_num_threads()` で指定されている数を上限とする LWP からなるチームを作る．明らかに LWP が不足する場合があるので，その場合はその時点で利用可能なだけの LWP を用いて実行する．
(3) 仕事共有子に遭遇すると，共有メモリ上に仕事を生成し，その時点でそのチームに所属している LWP が，それらの仕事を実行する．後にチームに加わる LWP のために仕事は常に共有メモリ上に生成する．

各 LWP は以下の動作をする

(1) 遊休状態では，LWP 数が不足している（容量以下の LWP しか割り当てられていない）チームが出現するのを待っている．
(2) そのようなチームが現れると，そのチームに加入し，そこでの仕事が終わると再び遊休状態に戻る．

先の方式との大きな違いは，並列化指示子が実行を開始した後も，LWP がチームに加わることを許している点である．仕事共有子はこのことを考慮して，常に仕事を分割して共有メモリ上に配置しておく．そして後から LWP が加わった時に，その LWP が仕事を獲得できるようにしておく．

この方針では，無駄に遊休状態に陥るスレッドが無いという利点があるが，仕事共有子に遭遇するたびに仕事を生成するなど，明らかに大きなオーバヘッドがかかってしまう．また，仕事共有子の終了待ちの同期を実装するのは面倒である．なぜならば，終了待ちを行う LWP（チーム内の 0 番 LWP）を，単にその LWP ごと休眠させると，その LWP が他のチームに加わることができず，結果としてそこで並列度が 1 LWP 分だけ失われるからである．

例えば，**図7**のような場合を考える．0 番 LWP（左側のチームのマスタースレッド）は左側のチームで実行していた仕事共有子（例えば for 文）の終了を待っている．このとき，単にこの 0 番 LWP ごと休眠させたとする．すると，右側のチームにまだ実行されていない仕事が存在するにもかかわらず，この LWP は右側のチームに加わることができない．

これを防ぐには，大きく分けて 2 つの方法がある．

- 同期不成立時にも，LWP 自身を決して休眠させない方法．つまり，LWP は一旦その仕事だけを中断させて，LWP 自身は他の仕事を獲得できるようにする（仕事の中断・再開機構）
- 同期不成立時に，LWP 自身は休眠させるが，代わりに新たな LWP をひとつ生成してから休眠させる．これによって並列度の低下を防ぐ．一方同期が成立した時はかわりに 1 つ LWP を消す．これによってここでも LWP 数を一定に保つ．

性能上は前者の方式が好ましいが，一方で，仕事の中断・再開を実装するために，スレッドライブラリの同期機構を利用することができないので実装が複雑になる．

4. 実　　装

前章で述べた，負荷分散を十分にするための条件を満たす方法として，我々は StackThreads/MP[9)~11)] ライブラリを用いることにする．

4.1 StackThreads/MP

StackThreads/MP は，細粒度スレッドを，C コンパイラ（GCC）とライブラリの組合わせで実現した API である．

StackThreads/MP アプリケーションは，立ち上げ時に一定数（例えばプロセッサ数と同じ個数）の LWP を立ち上げ，以後 LWP の数は一定である．通常のユーザレベルスレッドライブラリ同様，アプリケーションは任意の個数の StackThreads/MP のスレッド（以下細粒度スレッド）を生成することができる，そして，それらのスレッドは自動的に一定数の LWP によって実行される．通常のスレッドライブラリとの大きな違いは，スレッドの生成コストが非常に小さいことおよび，スケジューリングに遅延タスク生成に基づいた方

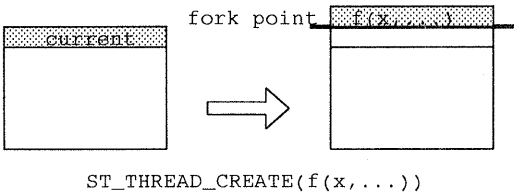


図 8 ST_THREAD_CREATE のスタック操作
Fig. 8 Stack management at ST_THREAD_CREATE.

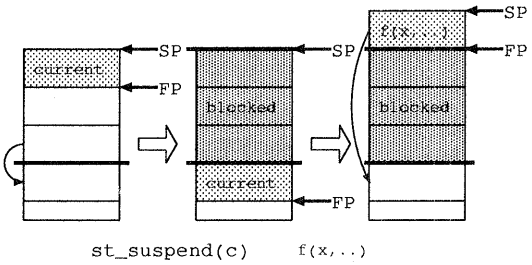


図 9 st_suspend のスタック操作
Fig. 9 Stack management at st_suspend.

式を用いることで，非常に多数のスレッド生成を許容できる点である．したがって，通常のスレッドライブラリを使うときに問題となるような，スレッドを過剰に生成することのオーバヘッドを気にしなくて良い．

StackThreads/MP の基本的な API は以下のものである．

ST_THREAD_CREATE$(f(x,y,\ldots))$：生成
ST_THREAD_CREATE が呼ばれると，**図 8** のようにこの点を境目にして新たな仕事が生成されるが，この時点では，その引数にとられた関数がスタックに積まれて実行されるので，普通の関数呼び出しと同等のコストしかかからない．普通の関数呼び出しと違う点は，後にこの点を境目にして他の LWP が仕事を盗めるということである．

st_suspend(c,n)：中断
st_suspend が呼ばれると，スタックポインタ，フレームポインタなどを保存しておくコンテクストを生成する．そして，**図 9** のように現在の仕事が中断されて切り放され，今の仕事を呼び出した仕事の続きをすることになる．この際，スタックポインタはそのままで，フレームポインタが現在のフレームを指すように移動する．つまり，スタックポインタは常にスタックの最上部を指している．この状態で，新たに関数呼び出しが起こると，呼び出した LWP のスタックの最上部に新たなフレームが作られて実行が継続される．

st_restart(c)：再開
st_restart が呼ばれると，保存したコンテクストを再開する．この時，他の LWP のコンテクストを再開するときには，st_suspend と同じようにスタックポインタはそのままにして，フレームポインタだけその LWP のスタック内を指すように変更する．

このように，st_suspend, st_restart は，下で走っている LWP を中断させることなく，StackThreads/MP レベルのスレッドを中断させることができる．また，スタック内の情報をヒープ領域に退避するようなことをしていないので，切り替えのコストを小さく押さえたまま仕事の中断・再開を行うことを可能にしている．

ST_POLLING()：負荷分散のタイミングを指定
ST_POLLING が呼ばれたときに他の LWP が仕事を盗もうとしていたら，**図 10** のように盗ませてやる．この時も st_suspend や st_restart と同様に，フレームポインタが他の LWP のスタックを指すことによって仕事の移動を実現する．このことによって，負荷分散がなされているのである．

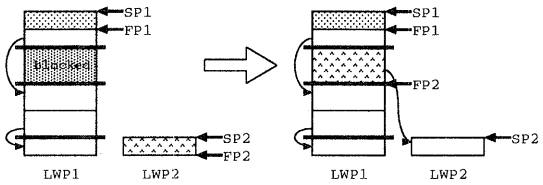


図 10 負荷分散時のスタック操作
Fig. 10 Stack management for load balancing.

StackThreads/MP のタスクスケジューリングにおいては，各 LWP は自分が生成した仕事があるうちはその仕事を深さ優先的に実行し，自分の仕事がなくなると，他の LWP のスタックの下の方から仕事を盗んでいく．この方式によって，多くの場合，非常に少ない仕事の移動で負荷分散を実現できる．したがって，ある仕事を並列化するのに，それを分割統治方式によって大量の細粒度スレッドに分割すれば，オーバヘッドを小さく保ったまま，均等な負荷分散を行うことができるのである．

4.2 我々の実装

3 章で取り上げた問題を StackThreads/MP を用いて解決するという方針のもとで，我々の実装の詳細を説明する．

4.2.1 並列化指示子 parallel

単独で実行していたスレッドが parallel に遭遇すると，常に（omp_set_num_threads() などで）指定された数の細粒度スレッドを生成し並列に実行する．

```
#pragma omp parallel for schedule(dynamic,3)
  for (i = 0; i < N; i++)
    work(i);
```

図 11 OpenMP の for ループ
Fig. 11 for loop in OpenMP.

注意としては，ここで生成された細粒度スレッドがどの LWP によって最終的に実行されるかは，この時点ではまだ分からないということである．これが一番外側の並列化指示子であればおそらく各 LWP が 1 つずつの細粒度スレッドを実行するだろうし，外側で十分に並列度が抽出されていれば，一台の LWP が全ての細粒度スレッドを実行するかもしれない．

4.2.2 仕事共有子 for および sections

我々の方式では，仕事共有子に遭遇した時にも単に必要なだけの細粒度スレッドを生成する．具体的には，

- `sections` においては，`section` の個数だけ細粒度スレッドを生成する
- `for` においては，イテレーション全体を分割統治方式によって指定された chunk_size まで分割して細粒度スレッドを生成する

つまり，`for` に遭遇したときには，各細粒度スレッドが，chunk_size で指定された数のイテレーションを実行する．また，`sections` に遭遇したときには，各細粒度スレッドが 1 つの `section` を実行する．これは，入れ子状の `for` や `sections` に遭遇したときも同様で，常に細粒度スレッドを生成する．

そして，`parallel` の時と同様，実際にそれらの細粒度スレッドがどの LWP で実行されるかは実行時の条件によって決まる．

`sections` や `for` の終了待ちの同期は単に StackThreads/MP プリミティブの `st_suspend` を利用することで行う．そこで結果的に起こることは下で走っている LWP が他のチームの仕事を盗んで実行する，ということである．

例として，**図 11** は**図 12** のように変換される．☆

4.2.3 スレッド ID

我々の実装では，負荷分散を十分に行うために，各々の LWP がさまざまなチームの細粒度スレッドを乱雑に実行する．つまり，LWP と細粒度スレッドの間の結びつきは動的である．すると，OpenMP プログラマから見た，スレッド ID（つまり，`omp_get_thread_num()` の返り値）をどう保つかというのが問題になる．LWP に ID を恒久的に割り当て，それを返すだけでは，

☆ 簡単のために以下の例は N=chunk_size $\times 2^K$ と書けることを仮定している．

```
void for_loop(int lb,int ub,int ch) {
  if (ub-lb <= ch) {
    for (i = ub; i < ub; i++)
      work(i);
  } else {
    int m = (lb+ub)/2;
    ST_THREAD_CREATE(for_loop(lb,m,ch));
    for_loop(m,ub,ch);
    /* 終了待ち同期（省略） */;
  }}
```

図 12 変換後の for ループ
Fig. 12 for loop after translation.

OpenMP のセマンティクスに違反してしまう．

これを直すための基本的な方針は単純で，各細粒度スレッドに，OpenMP のセマンティクスが規定するところのスレッド ID を渡してやり，どの LWP が実行しているかにかかわらず，その ID を返せばよい．ただし，OpenMP プログラムからはその渡された ID を無引数の関数呼び出し `omp_get_thread_num()` によって得られなくてはならないので，各細粒度スレッドは実行を開始する際に，その与えられた ID を LWP 固有の記憶領域に登録し，`omp_get_thread_num()` はそれを読むことで値を返す．

以上のようにして，各細粒度スレッドがどの LWP によって実行されていても，通常の OpenMP の実装で得られるスレッド ID が返されるようにすることができる．

ただし厳密にはこの実装でも正しくない場合がある．このようにすると，同じチーム内で同じスレッド ID を返すような複数の細粒度スレッドが，異なる LWP によって，「同時に」実行される，ということがありうる．一方，典型的な実装を行えば，同じスレッド ID が返される計算は 1 つの LWP で逐次的に実行されることになる．このことに依存して書かれたプログラムは，我々の実装では正しく実行できない．

残念ながら，これをセマンティクス違反と呼ぶべきかどうか判断できるほど，OpenMP の仕様は厳密ではないが，仮に違反だった場合は，ある程度のオーバヘッドを加えることで，1 つのチーム内で同一のスレッド ID を持つ細粒度スレッドを，排他的に実行するようにすることは可能である．だが，現在はこの問題は無視している．

5. 実　　験

5.1 計算機環境

計算機環境としては，共有メモリ並列計算機である Ultra Enterprise 10000 （Ultra SPARC 250MHz,

```
int A[N][N], B[N][N], C[N][N]
#pragma omp parallel for private(i)
  for (i = 0; i < N; i++)
    #pragma omp parallel for private(j, k)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
          C[i][j] += A[i][k] * B[k][j];
```

図 13 行列の掛け算のプログラム
Fig. 13 Code of Matrix Product.

8GB Main Memory, 1MB L2 Cache, 64CPUs) のうち，32CPUs を用いて実験を行った．

5.2 処理系

実験は，OpenMP の処理系の 1 つである Omni[8] と，Omni のコンパイラとランタイムライブラリに変更を加えることにより実装を行った我々の実装の 2 つで比較を行った．

Omni は，3.1.1 節の方針に基づいて実装されている．

5.3 アプリケーション

性能を測定するために，以下のアプリケーションを用いた．

行列の掛け算

$N \times N$ の整数要素の正方行列どうしの掛け算を行う．**図 13** のように，3 重ループのうち外側 2 つのループにプラグマを挿入して並列化する．測定の条件として，Omni では，for ループにおける仕事の割り当て方を static なブロック分割とし，最も外側のループに PE と同数の LWP を割り当てるようにする．我々の実装の方では，プラグマの挿入されているすべての for ループを，実行台数によらず N 個の仕事に分割し，各スレッドに分配する．Omni と我々の実装の実行時間を比較することにより，StackThreads/MP を用いることによるオーバヘッドを測定し，台数効果を調べることにより，それぞれの処理系の性能を評価する．なお台数効果は，それぞれの処理系で LWP 数を 1 とした時の実行時間で各実行時間を割ったものである．$N = 512$ とし，LWP 数を変化させて測定する．

クイックソート

整数 N 個のクイックソートを行う．再帰的に関数が呼ばれる部分を**図 14** のように `sections` を用いて並列化する．並列部分の並列度を 2 とすることにより，`section` を割り当てられずに `parallel` の終りで同期待ちをしているスレッドがないようにする．実行時間，台数効果を比較することにより，我々の実装において，どれだけ有効に負荷分散がなされているかを測定する．$N = 2^{24}$ (16M) 個とし，LWP 数を変化させて性能を評価する．

```
void QuickSort(int *A, int a, int b) {
  int p; /* pivot index */
  /* divide two portion by p */
  omp_set_num_threads(2);
  #pragma omp parallel sections
    {
    #pragma omp section
      QuickSort(A, a, p-1);
    #pragma omp section
      QuickSort(A, p, b);
  }
}
```

図 14 クイックソートのプログラム
Fig. 14 Code of Quick Sort.

分子動力学法によるシミュレーション

分子動力学法[6] に基づいて，N 個の分子の時系列に沿った運動エネルギーと位置エネルギーを計算する．この問題には，$N \times N$ の各イテレーション間に仕事量の差の無い入れ子並列性が存在する．$N = 500$ として，50 単位時間のシミュレートを行う．

5.4 結果と考察

まず，行列の掛け算についてであるが，このような問題の場合，内側のループ間の仕事量の差はほとんどないので，最も外側のループで負荷分散されていればスケーラブルな結果が得られるはずである．**図 15** より，Omni と我々の実装における絶対性能の差はほとんど無く，32 台の時には，Omni で 1824 (ms)，我々の実装で 1920 (ms) と 5%程度であり，我々の方法を用いることによるオーバヘッドはほとんど無いことが分かる．また，**図 16** より，Omni，我々の実装共に十分な台数効果が出ていることが確認される．なお，Omni，我々の実装において LWP を 1 とした時にかかった時間と，純粋な逐次版でかかった時間は，それぞれ，63946 (ms)，63202 (ms)，63649 (ms) であり，逐次実行における性能の差はほとんどない．

図 17 より，クイックソートの実験では，我々の実装の方は順調に時間短縮しているけれども，Omni の方はほとんど性能向上が見られない．また，**図 18** を見ると，我々の実装では 12 台ほどの台数効果がでており，このアルゴリズムにおける台数効果としてはかなりよい値である．それに対し Omni の実装では性能は 10 台で頭打ちになり，後はどんどん性能が悪くなる一方であった．その主な原因の 1 つとして，Omni では並列化指示子に遭遇したときに LWP が余っていると並列化されるわけであるが，このアルゴリズムでは，先に並列化指示子に遭遇する，つまり二分された

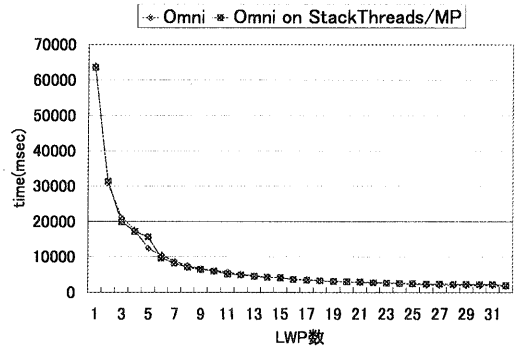


図 15 行列積の計測結果
Fig. 15 Elapsed time for Matrix Product.

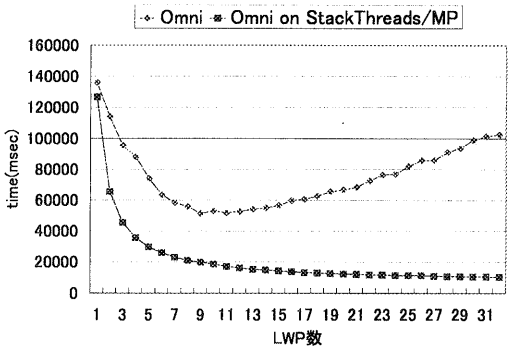


図 17 クイックソートの計測結果
Fig. 17 Elapsed time for Quick Sort.

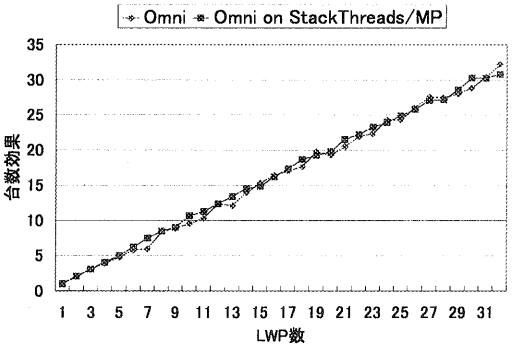


図 16 行列積の台数効果
Fig. 16 Speed up of Matrix Product.

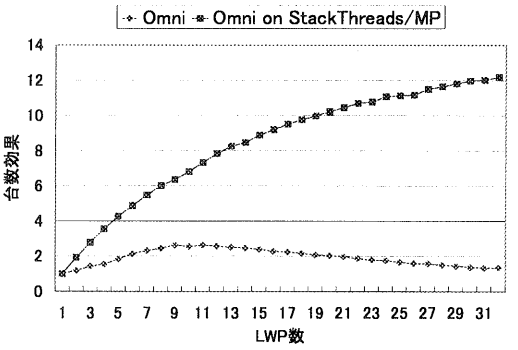


図 18 クイックソートの台数効果
Fig. 18 Speed up of Quick Sort.

要素数の少ない方がより並列化されやすいということがある．したがって，プロセッサ台数を増やしても，すぐに終わる仕事ばかり並列化されて，長くかかる仕事が逐次に実行されるためほとんど台数効果が得られないという結果になってしまったわけである．また，もう1つの原因として，データが不均等に割り振られるために，同期待ちのために遊休状態のスレッドがたくさん存在していることもあげられる．

したがって，イテレーション間の仕事量に差があるとき，また，再帰的に並列化指示子が出てくるような状況においては，我々の実装は非常に効果的であることが実証された．

最後に分子動力学法によるシミュレーションにかかった時間であるが，**図 20** より，Omni の方では台数効果が十分出ているのに対して，我々の実装では Omni の半分くらいしか出ていなかった．その原因としては，このアプリケーションでは2重の入れ子並列性があり，内側の並列領域における計算結果を共有変数に畳み込む操作のときにロックをとっていることが考えられる．Omni の場合には，とりあえず外側の `parallel` にすべてのスレッドを割り当てるため，内側の `parallel` では並列化されず逐次実行されるので，ロックをスレッド台数回しかとらないのに対し，我々の実装では，細粒度スレッドの個数の2乗回とっているためにその部分がボトルネックになっている可能性が考えられる．この問題は，畳み込みを樹状に行うことによって解決できるものと思われる．現在その処理を実装中である．

5.5 Polling のタイミング

我々の実装において `ST_POLLING` は，文献 9) を参考にして `ST_THREAD_CREATE` の前後に挿入している．こうすることにより，行列の掛け算の場合には，$512 \times 512 \times 2 = 2^{19}$ 回 `ST_POLLING` が呼ばれることになり，入れ子の for ループの回りでは少し過剰に，それ以外では少なめになってしまっているが，実行結果を見る限り特に問題はないと思われる．

6. 関連研究

コンパイラにヒントを与えることによって，自動並列化を行う研究としては High Performance Fortran

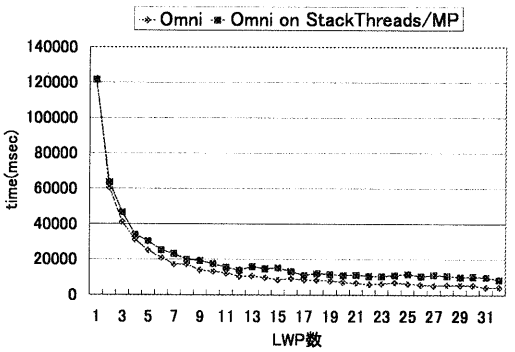


図 19 分子動力学法の計測結果
Fig. 19 Elapsed time for Molecular Dynamics.

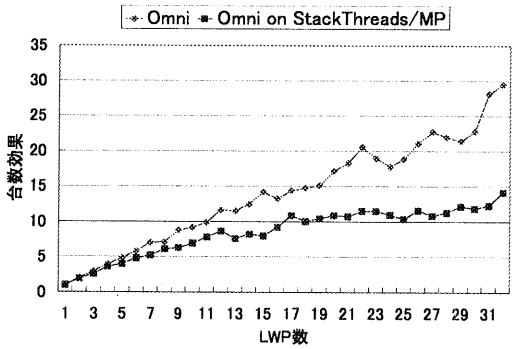


図 20 分子動力学法の台数効果
Fig. 20 Speed up of Molecular Dynamics.

(HPF) の研究がある．HPF の研究では主に `for` ループの並列化を主眼としており，データ間に依存関係があるような場合にも，いかに依存関係を壊さないで効率的に並列化するかという研究もなされている．しかしながら，我々 OpenMP の研究では，データ間には依存関係が無いことをプログラマが保証しているので，いかに並列性を効率よく抽出できるかということを考えればよく，また OpenMP ではループレベルの並列性だけでなく，タスクレベルの並列性を扱うことが出来る点が HPF とは大きく違う点である．

入れ子並列性の研究として，文献 2) では，C 言語にベクトルを導入した V という言語上で，入れ子状のベクトルを divide-and-conquer 方式で並列化しようとしているが，我々の方式では，より一般的な入れ子構造を効率的に並列化することが可能である．また Cilk[1] は，StackThreads/MP と同様に遅延タスク生成に基づいた動的負荷分散をする言語であるので，入れ子並列性が出てきても効率的に負荷分散をすることが出来るわけであるが，我々の場合は，細粒度スレッドライブラリを新しい処理系 OpenMP に組み込むことによって，その OpenMP を効率的に実行出来るようにした点が，他に類をみない例である．

7. まとめと今後の課題

我々は，OpenMP に，遅延タスク生成に基づくスケジューリングを行うスレッドライブラリを導入することによって，並列化指示子が入れ子状になっているような時にでも，効率的に実行するための実装方法を示し，OpenMP の処理系 Omni に実装した．その際，ライブラリを用いることにより，実装の複雑さを非常に軽減することが可能になった．さらに，Sun Enterprise 上での実験の結果，遅延タスク生成によるスケジューリングのオーバヘッドは 5 %程度に押さえられ，OpenMP に入れ子並列を実装することによる性能の低下はほとんどないことが確認された．また再帰的に並列化指示子が呼ばれ，各仕事量が不均等な場合にも台数効果が出ることが確認され，我々の方式の有効性が示された．

しかしながら，分子動力学法によって計算を行う問題の場合，台数効果が半分程度になってしまった．現在この問題の原因を究明し，解決法を実装中である．

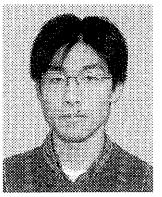
謝辞 Omni の利用に際し，多大なご協力をしていただいた，新情報処理開発機構の佐藤三久氏に心から感謝致します．また，東京大学大学院理学系研究科の米澤研究室・小林研究室の方々には，様々な協力をしていただきました．

参 考 文 献

1) Blumofe, R. D.: *Executing Multithreaded Programs Efficiently*, PhD Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology (1995).
2) Chakravarty, M. M. T., Schröer, F.-W. and Simons, M.: V—Nested Parallelism in C, *Programming Models for Massively Parallel Computers* (Giloi, W.K., Jähnichen, S. and Shriver, B. D.(eds.)), IEEE Computer Society, pp. 167–174 (1995).
3) Dongarra, J. J., Hempel, R., Hey, A. J. G. and Walker, D. W.: A Proposal for a Userlevel, Message Passing Interface in a Distributed Memory Environment, Technical report, Oak Ridge National Laboratory (1993).
4) Koelbel, C., Loveman, D., Schreiber, R., Steele Jr., G. L. and Zosel, M.: *The High Performance Fortran HandBook*, The MIT Press (1994).
5) Mohr, E., Kranz, D. A. and Halstead, Jr., R. H.: Lazy Task Creation: A Technique for In-

creasing the Granularity of Parallel Programs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 264–280 (1991).
6) OpenMP Architecture Review Board: A simple molecular dynamics simulation. `http://www.openmp.org/index.cgi?samples+samples/md.html`.
7) OpenMP Architecture Review Board: *OpenMP C and C++ Application Program Interface* (1998). `http://www.openmp.org/`.
8) RWCP: Omni OpenMP Compiler. `http://pdplab.trc.rwcp.or.jp/pdperf/Omni/`.
9) Taura, K.: *StackThreads/MP User's Manual*, University of Tokyo (1999). `http://www.yl.is.s.u-tokyo.ac.jp/sthreads/`.
10) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP:Integrating Futures into Calling Standards, *Symposium on Principles and Practice of Parallel Programming*, ACM SIGPLAN, pp. 60–71 (1999).
11) Taura, K. and Yonezawa, A.: Fine-grain Multithreading with Minimal Compiler Support—A Cost Effective Approach to Implementing Efficient Multithreading Language, *Programming Language Design & Implementation*, Las Vegas, Nevada, ACM SIGPLAN, pp. 320–333 (1997).

(平成11年 7 月 16 日受付)
(平成 11 年 12 月 29 日採録)

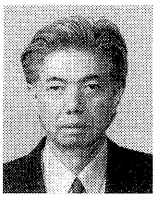


田中 義純

1976 年生．1999 年東京大学理学部情報科学科卒業．現在東京大学大学院理学系研究科情報科学専攻修士課程に在学中．主に並列プログラミングの研究に従事．

田浦健次朗（正会員）

1969 年生．1996 年より東京大学大学院理学系研究科助手．1997 年東京大学大学院より理学博士取得．並列プログラミング言語の設計，実装に関する研究に従事．



米澤 明憲（正会員）

1947 年生．1977 年 Ph. D. in Computer Science（MIT）．1989 年より東京大学理学部情報科学科教授．超並列・分散ソフトウェアアーキテクチャなどに興味を持つ．共著書「モデルと表現」等（岩波書店），編著書「ABCL」（MIT Press）等がある．1992–1996 年ドイツ国立情報処理研究所（GMD）科学顧問，ACM Transaction on Programming Languages and Systems 副編集長，IEEE Parallel & Distributed Technology および Computer 編集委員などを歴任，元日本ソフトウェア科学会理事長，ACM Fellow.