ハイパフォーマンス
コンピューティング 82−12
（2000．8．4）

OpenMPによる粗粒度タスク並列実行方式

福岡岳穂　本多弘樹　弓場敏嗣
電気通信大学大学院情報システム学研究科

本論文では、プログラム全体を粗粒度タスク（マクロタスク）に分割し、マクロタスクの並列性が実行開始条件として表現されている際に、そのマクロタスク集合を OpenMP の指示子とライブラリ関数を用いて並列実行する方式とその予備的評価について述べる。プログラム中のマクロタスクは条件分岐文等によりすべてが実行されるとは限らず、マクロタスクのスケジューリングをコンパイル時に静的に行うのが困難であるため、ダイナミックスケジューリングを用いる方法が提案されている。本方式ではマクロタスクのダイナミックスケジューリングによる並列実行を OpenMP の指示子とライブラリ関数を用いて実現する。予備的な評価として、本方式を SMP マシン上に実装し、ダイナミックスケジューリングのオーバヘッドを評価した。

Coarse Grain Parallel Execution Scheme With OpenMP

Takeaki FUKUOKA, Hiroki HONDA, Toshitsugu YUBA
Graduate School of Information Systems, The University of Electro-Communications

In this paper, we discuss a scheme for parallel execution of coarse grain tasks (macro-tasks) by using OpenMP directives and library functions. This scheme requires conditions that source program is divided into some macro-tasks and each macro-task's executable condition is known. Considering branch-conditional among macro-tasks, not all macro-tasks can't be executed in parallel. As a result, it is difficult to execute macro-tasks in parallel by static assignment which done at compile-time. Therefore, on this scheme, macro-task's assignment are done at run-time dynamically.

We evaluate this scheme on SMP machines. The preliminary results show that under condition the size of macro-tasks is not small, this scheme shorten program's execution-time.

1. はじめに

並列計算機上でひとつのプログラムを並列実行する際のタスク分割の粒度として、ひとつの算術代入文をタスクとした細粒度タスク、ループの１イタレーションをタスクとした中粒度タスク、ループ等の基本ブロックをタスクとした粗粒度タスク等が考えられる。本論文では粗粒度タスク（マクロタスク）の並列実行方式について述べる。

マクロタスク並列実行方式について、[1]では OSCAR [3] 上での、また [2] では FX4 と KSR1 上での実装・評価について述べられている。[1] [2] では、マクロタスク集合の並列性は実行開始条件 [4] として表現されており、条件分岐文等によりすべてのマクロタスクが実行されるとは限らないため、マクロタスクのプロセッサへの割り当てには、プログラムの実行時に動的に行うダイナミックスケジューリングが採用されている。

本論文では [2] で提案されている方式をコモディティ SMP マシン上で実現することを目的とし、マクロタスク集合の並列実行制御コードに OpenMP [5] の指示子とライブラリ関数を用いて実装することを試みる。

OpenMP は共有記憶型並列計算機のためのコンパイラ指示子統一規格として提案されているものであり、単一プログラム並列実行のための指示子とライブラリ関数が定義されている [6]。

OpenMP で記述されたプログラムは、逐次コンパイラを利用してコンパイルすればそのまま逐次実行可能であることや、プログラムに新たに指示子を追加することが容易であること等、パフォーマンスのチューニングが比較的容易であることが OpenMP を利用した理由である。

本論文では、まず 2 章において本方式の前提条件、概要、実装方法を示し、3 章でコモディティ SMP マシンでの予備的な性能評価を行う。そして 4 章でまとめを述べる。

2. OepnMP による粗粒度タスク並列実行方式

2-1. 前提条件

本方式では、その前提として、プログラム全体が粗粒度タスク（マクロタスク）に分割されており、かつマクロタスク間の並列性が実行開始条件 [4] で表現されているとする。マクロタスクを次のように定義する。

[マクロタスク]　プログラムの一部分（ブロック）であり、そのブロックを実行開始する文が、そのブロックの先頭の行だけであるもの。ブロック途中への飛び込み、ブロック途中からの飛び出しはない [1]。

マクロタスク間の制御フローとデータ依存関係をひとつのグラフで表現したものをマクロフローグラフと呼ぶ [1]。図 1 にマクロフローグラフの例を示す。図中の方形はマクロタスク、方形内の円は条件分岐、点線は制御フロー、実線はデータ依存を示す。

マクロタスクの実行開始条件とは、プログラム実行中にどのような条件がそろったらそのマクロタスクの実行を開始してよいか、ということを論理式で表現したものである [4]。論理式は、＜マクロタスクの終了＞、＜分岐方向決定＞、＜マクロタスクの分岐方向付終了条件＞の 3 種類の原子条件と論理演算子∧（論理積）、∨（論理和）から構成される。各原子条件の初期値は偽である。表 1 に図 1 のマクロフローグラフに対応する、各マクロタスクの実行開始条件を示す。例えば表中の MT7 の実行開始条件は、MT2 の終了、MT6 の終了、MT3 から MT4 への分岐方向決定のいずれかが成立した時点で MT7 が実行開始可能になることを表現している。

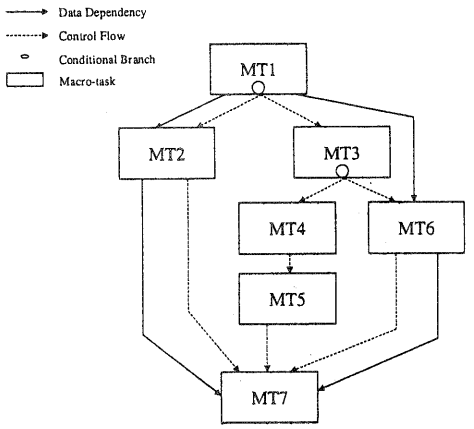

図 1.マクロフローグラフ

表 1. 実行開始条件

マクロタスク	実行開始条件
MT1	TRUE
MT2	1(1,2)
MT3	(1,3)
MT4	(3,4)
MT5	(3,4)
MT6	(3,6)∧1(1,3)
MT7	2∨6∨(3,4)

注：表中の表記について。

MT0の終了が条件の場合には"0"、MT0からMT1への分岐が条件の場合には"(0,1)"、MT0が終了し、かつMT0からMT1への分岐が条件の場合には"0(0,1)"と表現する。

2-2. マクロタスクダイナミックスケジューリング

実行開始条件により並列性が表現されたマクロタスク集合の並列実行の際には、条件分岐文等によりすべてのマクロタスクが実行されるとは限らないため、[1] [2] ではマクロタスクダイナミックスケジューリングを行っている。

マクロタスクダイナミックスケジューリングではマクロタスクの処理が進行するにともない、実行開始条件の原子条件を "true" に更新し、実行開始条件が真となったマクロタスク（レディマクロタスク）をマクロタスクの実行を行っていないプロセッサ（アイドルプロセッサ）に順

次割り当てていく。具体的にはスケジューラがある時点でマクロタスクの実行開始条件を評価し、あるマクロタスクがレディマクロタスクとなった場合には、そのマクロタスク番号をレディマクロタスクキューに登録する。スケジューラはレディマクロタスクキューからマクロタスク番号を取り出し、その番号に対応するマクロタスクをアイドルプロセッサへ割り当てる。マクロタスクを割り当てられたプロセッサはそのマクロタスクの実行を行い、<マクロタスクの終了>、<分岐方向決定>、<マクロタスクの分岐方向付終了条件>の事象が生じた際に、対応する原子条件を "true" に更新する。マクロタスクダイナミックスケジューリングでは上記のマクロタスクの実行と実行開始条件の評価をマクロタスク集合の実行を終了するまで繰り返す。

マクロタスクダイナミックスケジューリングの方式には、スケジューラの機能をはたすスケジューリングコードの実行を各プロセッサに分散させる分散スケジューラ方式 [2] と単一プロセッサに集中させる集中スケジューラ方式 [1] [2] が考えられる。

分散スケジューラ方式では、スケジューリングコードを各マクロタスク処理の前後に埋め込み、各プロセッサはスケジューリングコードが埋め込まれた同一のプログラムを実行する。各プロセッサはプロセッサ間で共有されるスケジューリング用共有データ（レディマクロタスクキューや実行終了したマクロタスクの番号、マクロタスク内の条件分岐文の分岐先マクロタスク番号を登録するデータ）を排他アクセスしながらマクロタスクの割り当て、実行を行う [2]。

集中スケジューラ方式ではスケジューリングコードをマクロタスク実行コードとは別に生成し、1 台のプロセッサがスケジューリングコードを、他のプロセッサはマクロタスクの実行コードを実行する。スケジューリングコードを実行するプロセッサは、実行開始条件を評価するとともに、各プロセッサへのマクロタスク割り当てを行う。マクロタスク実行コードを実行するプロセッサは割り当てられたマクロタスクを実行するとともに、実行中のマクロタスクの分岐方向、マクロタスクの実行終了等の情報をスケジューリングコードを実行するプロセッサへ通知する [2]。

本方式では分散スケジューラ方式を採用し、そのコードを OpenMP 指示子とライブラリ関数を用いて実装した。

2-3. OpenMP 指示子とライブラリ関数利用方法

OpenMP は fork-join 型の実行モデルを採用しており、プログラム中で並列化指示子があると、そこで指定された数のスレッドを生成し、並列実行が終了するとマスタスレッド以外のスレッドは消滅する。並列実行時にはループイタレーション間の並列性を利用した "for" 指示子や、プログラマが指定したブロック（プログラムの一部分）間の並列性を利用した "sections" 指示子等が利用可能である。

マクロタスク並列処理の実装に "sections" 指示子を利用することも考えられる。しかし "sections" 指示子では通常、 "sections" 有効範囲内で記述される各 "section" を実行する際のスケジューリングは静的に決定しているので、マクロタスク間に依存関係があるマクロタスク集合の並列実行には都合が悪い。

そこで本方式では、マクロタスク集合実行時に OpenMP 並列化指示子 "parallel" を用いて複数のスレッドを起動し、[2] の分散スケジューラ方式と同様、各スレッドがスケジューリング用の共有データを排他アクセスをしながら実行開始条件の評価、マクロタスクの実行を行う。本方式で排他アクセスされるデータはレディマクロタスクキューのみである。排他アクセスは OpenMP ライブラリ関数 "omp_set_lock()" と "omp_unset_lock()" を用いて実現した。

2-4. 並列実行プログラム

マクロタスク集合として表現されるプログラムから本方式を用いた並列実行プログラムへの変換方法を以下に示す（図 2 を参照）。

① プログラム中のマクロタスク集合を並列実行部分とする。つまりマクロタスク集合全体を OpenMP の並列化指示子 "parallel" の

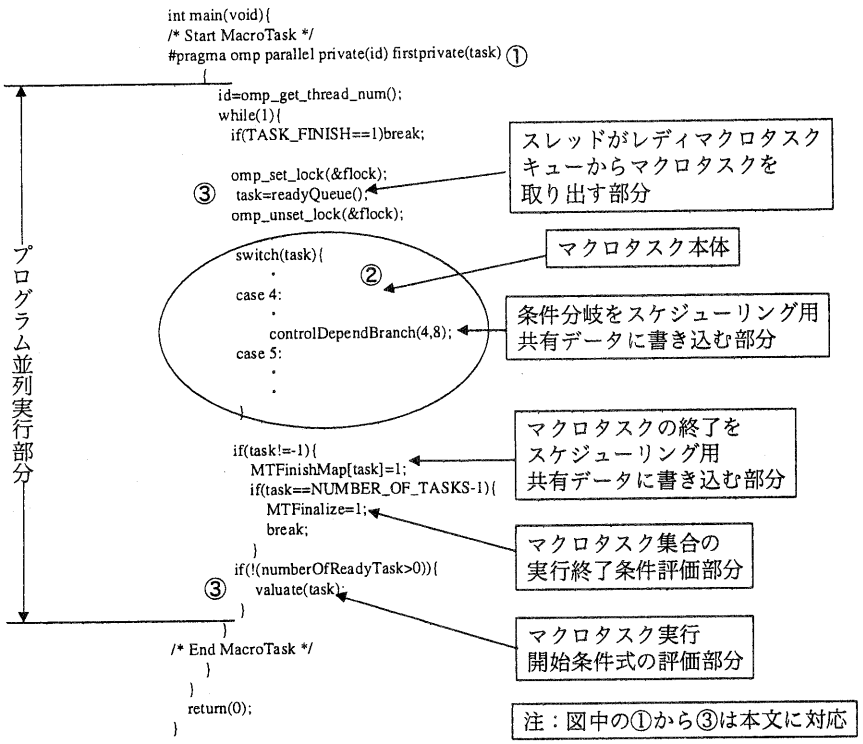

図 2. 並列実行プログラム

有効範囲内に入れる。

② ひとつのマクロタスクをひとつの "case" 文とした "switch-case" 構文にする。

③ レディマクロタスクキューに排他アクセスするためのクリティカルセクションとマクロタスク実行開始条件評価を行う部分を作成する。

並列実行プログラム実行時の各スレッドの挙動を以下 (1) から (6) に示す。マクロタスク集合の並列実行は、全スレッドが (1) から (6) を繰り返すことにより実現している。

(1) マクロタスク集合終了フラグが真ならば、マクロタスク集合の実行を終了する。

(2) スレッドは、レディマクロタスクキューが空でなければレディマクロタスクキューを排他アクセスし、実行可能なマクロタスクをひとつ取り出す。レディマクロタスクキューが空の場合にはレディマクロタスクキューに排他アクセスせずに (6) へ。

(3) スレッドは、レディマクロタスクキューから取り出したマクロタスク番号に対応するマクロタスクを実行する。マクロタスク内で条件分岐が発生した場合には、条件分岐先が決定した時点で分岐方向をスケジューリング用の共有データに書き込む。

(4) マクロタスク処理を終了したスレッドは、そのマクロタスクの終了をスケジューリング用共有データに書き込む。

(5) マクロタスク集合の終了条件を評価し、それを満たしていれば、マクロタスク集合終了フラグを真とし、そのスレッドはマクロタスク集合の実行を終了する。

(6) もしレディマクロタスクキューが空ならば実行開始条件を評価する。まだ真となっていない実行開始条件をすべて評価し、レディマクロタスクキューに排他アクセスし、新たに実行可能になったマクロタスクすべての番号をレディマクロタスクキューに登録する。

3. 評価

本章では、本方式によるタスク割り当てオーバヘッドの評価のために作成した例題プログラム（プログラムとしての処理に意味はない）に対する性能評価を示す。例題プログラムのマクロフローグラフを図 3 に示す。マクロタスク 1 の条件分岐により、実際に実行されるマクロタスクはマクロタスク 2 から 16（またはマクロタスク 17 から 31）にマクロタスク 1、32 を加えた 17 個である。マクロタスク 2 から 16（またはマクロタスク 17 から 31）の間には依存関係がなく、並列に実行することができる。

各マクロタスク内の処理は以下のコードであり、N の値を変化させることによりマクロタスクの処理量を変化させた。

```
for (i=0;i<N;i++)
  a [i] =b [i] +i+4;
```

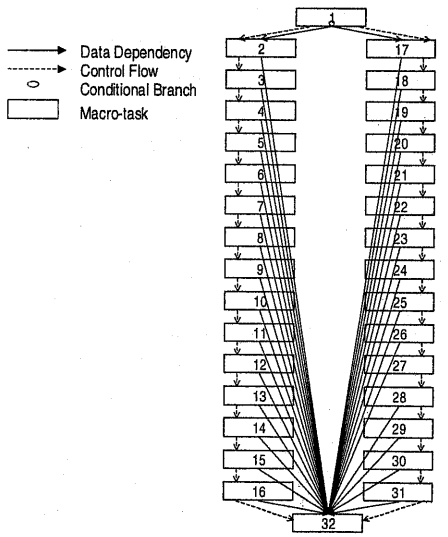

図 3. 例題プログラムのマクロフローグラフ

この例題プログラムに本方式を適用し、表 2 の実験環境で、マクロタスクの処理量を N=100、N=1000 としたときの実行時間を図 4、5 に示す。図中の横軸は時間軸である。グラフ中の白塗りの部分は割り当てられたマクロタスクを実行している状態を示しており、黒塗りの部分はそれ以外の処理（実行開始条件の評価等）を示している。

表 2. 実験環境

OS	Linux 2.2.16
Processor	Intel Pentium II 400[MHz] x 2
Thread	Posix Thread
OpenMP Complier	Omni1.1[7]

この例題プログラムは、マクロタスク 1 を実行後、マクロタスク 1 での条件分岐にしたがってマクロタスク 2 から 16（又はマクロタスク 17 から 31）を実行、そしてそれら 15 個のマクロタスクの実行が終了した後にマクロタスク 32 を実行する。つまりオーバヘッドが無い理想的な実行状態であれば 2 並列実行時には、実行時間が逐次実行時の約 0.59 倍になる。

N=100 で本方式を適用し並列実行したものは、マクロタスクの実行時間よりも、マクロタスク割り当てのためのオーバヘッドの方が大きく、2 並列実行時において、本方式を適用せずに逐次実行したものに対して約 1.30 倍、本方式を適用したプログラムを逐次実行したものに対しては約 0.93 倍の実行時間となっている。

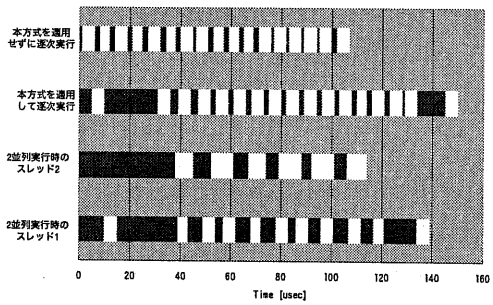

図 4. 例題プログラムの実行時間 (N=100)

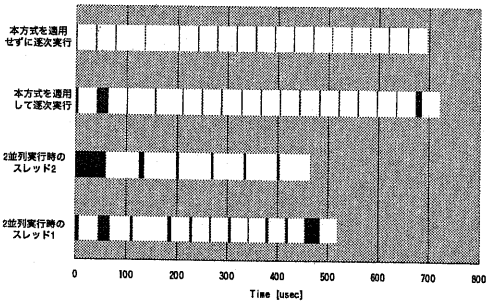

図 5. 例題プログラムの実行時間 (N=1000)

N=1000 では 2 並列実行時において、本方式

を適用せずに逐次実行したものと比較して約0.74 倍、本方式を適用したプログラムを逐次実行したものに対しては約 0.72 倍の実行時間となっている。

なお、実行時間は純粋にマクロタスク集合の並列実行を開始してから、マクロタスク集合の処理を終了するまでの時間を計測したものであり、OpenMP 指示子によるスレッドの起動やマクロタスク集合の並列実行の初期設定等の時間は含まれていない。

この例題プログラムにおいては、N=1000 の場合では実行時間を短縮できたものの、マクロタスク処理量が小さい、N=100 ではダイナミックスケジューリングオーバヘッドが目立ってしまい、並列処理効果が得られていない。オーバヘッドが大きい原因としては、次のことが考えられる。

(1) スレッドがレディマクロタスクキューからレディマクロタスク番号を取り出す際のレディマクロタスクキューへの排他アクセス。
(2) 実行開始条件の評価。
(3) 実行終了したマクロタスク番号をスケジューリング用の共有データに書き込み。

4. まとめ

本論文では、プログラムをマクロタスクに分割し、マクロタスクの並列性が実行開始条件として表現されている際のマクロタスク並列実行を、OpenMP 指示子とライブラリ関数を用いた分散スケジューラで実現し、その性能の予備的な評価を行った。

本方式の性能をコモディティ SMP マシン上で例題プログラムを用いて予備的に評価した結果、ダイナミックスケジューリングのオーバヘッドとの関係で、どの程度のタスク粒度であれば並列処理効果が得られるかが明らかになった。

今後は [8] を参考にしながらスケジューリングオーバヘッドの詳細な分析をするとともに、スケジューリングオーバヘッド削減手法の検討、さらにダイナミックスケジューリングのオーバヘッドに見合った粒度のタスク生成手法の検討を行う予定である。さらに、プロセッサ台数を増やした SMP 上でのダイナミックスケジューリングオーバヘッドの評価も行う予定である。

謝辞 Omni の利用に際し、御助言・御協力を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いたします。

なお本研究の一部は文部省科学研究補助金 (12680336) による。

5. 参考文献

[1] 本多弘樹, 合田憲人, 岡本雅巳, 笠原博徳, "Fortran プログラム粗粒度タスクの OSCAR における並列実行方式," 電子情報通信学会 D-I, Vol. J75-D-I, No.8, pp.526-535, 1992 年 8 月.

[2] 合田憲人, 岩崎清, 岡本雅巳, 笠原博徳, 成田誠之助, "共有メモリ型マルチプロセッサシステム上での Fortran 粗粒度タスク並列処理の性能評価," 情報処理学会論文誌, Vol.37, No.3, pp.418-429, Mar., 1996.

[3] 笠原博徳, 成田誠之助, 橋本親, "OSCAR (Optimally Scheduled Advanced Multiprocessor) のアーキテクチャ," 電子情報通信学会論文誌 D, Vol.J71-D, No.8, pp.1440-1445, 1988 年 8 月.

[4] 本多弘樹, 岩田雅彦, 笠原博徳, "Fortran プログラム粗粒度タスク間の並列性検出手法," 電子情報通信学会論文誌, D-Ⅰ, Vol.J73-D-Ⅰ, No.12 pp.951-960, 1990 年 12 月.

[5] OpenMP : Simple, Portable, Scalable SMP Programming (http://www.openmp.org)

[6] OpenMP Consortium, "OpenMP C and C++ Application Program Interface Version 1.0," October, 1998.

[7]Omni : RWCP OpenMP Compiler Project (http://pdplab.trc.rwcp.or.jp/pdperf/Omni/home.html)

[8] 草野和寛, 佐藤茂久, 佐藤三久, "Omni OpenMP コンパイラと実行時ライブラリの性能評価," JSPP2000 論文集, pp.229-236, June, 2000.