Vol.2013-HPC-140 No.16
2013/8/1

6種のタスク並列処理系の比較評価

田浦健次朗[1,a)]　中島潤[1,b)]

概要：タスク並列プログラミングモデルは, 関数や文などを実行する「タスク」を動的に生成することで並列処理を記述するプログラミングモデルである. タスク並列処理は高並列な環境で高生産性と性能を両立させるための, 今後有望なパラダイムであると信じられている一方で SPMD や静的な負荷分散と異なり, 実行方式がプログラマから見えない「ブラックボックス」的な要素が多く, 性能が第一義な HPC 分野の開発者にとって使いづらいという一面もある. そこで本論文は, 種々のタスク並列処理系のベンチマークを通して, プログラマがタスク並列プログラムの性能を理解する際の助けとなる情報を提供すること, 今後の処理系実装者に有用な情報を提供することを目指す. 48 プロセッサの Opteron および 61 物理コア (244 プロセッサ) の Intel Xeon Phi 上で, MassiveThreads, Cilk, OpenMP, TBB, Qthreads, Nanos の 6 種類の並列処理系, 10 種類の実装の評価を行い, 各処理系の性能や特徴を明らかにする.

キーワード：タスク並列処理, クリティカルパス, 台数効果

A Comparative Study of Six Task Parallel Programming Systems

Kenjiro Taura[1,a)]　Jun Nakashima[1,b)]

Abstract: Task parallel programming models express parallel programs by dynamically creating "tasks," entities that execute functions or statements. While they are believed to be a promising paradigm for achieving high programmability and high performance, they tend to hide performance-relevant factors behind "black boxes," making it difficult to reason about performance of task parallel programs. Performance is also highly dependent on implementation of task parallelism. These factors render task parallel systems difficult to adapt for HPC programmers, whose primary concerns are performance. This paper tries to give useful information with which programmers can understand performance of task parallel programs, and a useful methodology to evaluate/contrast the difference between task parallel implementations. We evaluate ten implementations of six task parallel systems (OpenMP, Intel Threading Building Block, Cilk, MassiveThreads, Qthreads, Nanos++) on 48 processors Opteron and 61 physical cores (244 processors) Intel Xeon Phi systems.

Keywords: task parallel processing, critical path, scalability

1. はじめに

タスク並列プログラミングモデルは, 関数や文などを実行する「タスク」を動的に生成することで並列処理を記述するプログラミングモデルである. 一般に, 並列処理の単位 (タスク) をプログラムの任意の時点で動的に生成できることと, タスクの負荷分散をプログラマではなく処理系が行うことを特徴とする. Cilk (spawn/sync 構文)[6], OpenMP (task/taskwait 構文) [10], Intel Threading Building Block (`task_group` クラス) [12], Chapel (begin 構文) [3], X10 の (async/finish 構文) などがこれをサポートしている他, タスク並列を容易に実装する軽量スレッドライブラリとして, MassiveThreads [8], Qthreads [13], Nanos++ [1] などがある.

これに対して, MPI, UPC [5] のように, 固定した数のプロセスをプログラマに意識させ, 負荷分散を明示的に記述

[1] 東京大学
University of Tokyo, 7-3-1 Hongo Bunkyo-ku, Tokyo 113–0033, Japan
[a)] tau@eidos.ic.i.u-tokyo.ac.jp
[b)] nakashima@eidos.ic.i.u-tokyo.ac.jp

させる Single Program Multiple Data (SPMD) モデルや, HPF のようにひとつのループネストの並列実行だけを対象としたプログラミングモデルがある. タスク並列処理の, (1) 任意の時点でタスク (並列性) が動的に生成できる, (2) タスクの負荷分散が処理系によって動的に行われる, という特徴により, 任意に入れ子になった並列ループ, 再帰呼び出しの並列化などを容易に記述・並列実行することが可能になる. それらは木構造を用いた計算, 負荷が一様でない計算, 行列計算における再帰的なブロッキング, 直交再帰分割などの効率的な空間分割などで効力を発揮する. また, 近年の高並列な環境では OS ノイズや修正可能なメモリエラーなど, 制御・予測困難な事象に対する耐性をあげるために, 処理の細分化 (細粒度タスク) および動的な負荷分散が有用であるという報告もなされている.

以上のような性質から, タスク並列処理は高並列な環境で高生産性と性能を両立させるための, 今後有望なパラダイムであると信じられている. しかしその一方で SPMD や静的な負荷分散と異なり, 実行方式がプログラマから見えない「ブラックボックス」的な要素が多く, 性能が第一義な HPC 分野の開発者にとって使いづらいという一面もある. 性能も処理系の基本性能 (タスク生成や負荷分散の速度), タスク配置などに大きく左右され, 実際同じプログラムでも処理系により性能が大きく異なる.

本論文は, 種々のタスク並列処理系のベンチマークを通して, プログラマがタスク並列プログラムの性能を理解する際の助けとなる情報を提供すること, 今後の処理系実装者に有用な情報を提供することを目指す. また, 全タスク並列処理系に対して共通のソースコードでプログラムが記述できるための最大公約数的なモデルとその実装について簡単に触れる. ベンチマークは, Opteron (48 PU) および Xeon Phi (244 PU) 上で行った. PU は OS から認識されるプロセッサの数 (仮想コア, ハードウェアスレッド) の数である. 比較した処理系は, OpenMP (GCC および Intel), Cilk (MIT 版, Intel 版), Intel Threading Building Block, MassiveThreads, Qthreads, Nanos++の 6 種である. 詳しくは 4 節で述べる.

本論文の以降の概要は以下のとおりである. 2 節で, 対象とするプログラミングモデルおよびそれらを共通に記述できる枠組みについて説明する. 3 節で, タスク並列プログラムの性能の理解に関する一般論を述べ, 処理系の違いを生み出しうる要因について考察する. 4 節で実験の内容および結果について述べる. 6 節で関連研究について述べ, 7 節で結論と今後の課題を述べる.

2. 共通タスク並列モデル

多数の処理系を比較評価するにあたって, 前処理系に共通のタスク並列モデル—最大公約数とでも言うべきモデル—と, その記述法を設定する. そしてそれを C/C++のマクロだけで実現する方法について述べる.

2.1 モデル

想定するタスク並列のモデルは, タスクを生成する, その終了を待つというふたつのプリミティブだけをサポートする最低限のもので, オリジナルの Cilk と同等のものである. Cilk ではそれらを以下の構文でサポートしている.

spawn: タスクを生成するプリミティブで,

spawn 関数呼び出し;

または

変数 = spawn 関数呼び出し;

の表記により, 「関数呼び出し」を実行するタスクを生成する.

sync: タスクの終了を待つプリミティブで,

sync;

の表記により, 現在のタスク (親) がそれまでに spawn した全ての子タスクが終了するまで, 親タスクの実行を中断する.

オリジナルの Cilk では, spawn で呼び出せる関数は,「Cilk 関数である」と宣言されなくてはならず, それには関数定義の先頭に, `cilk` というキーワードを付ける. *1 以下が最も簡単な Cilk の例題である.

```
1  cilk int fib(int n) {
2    if (n < 2) return 1;
3    else {
4      int x, y;
5      x = spawn fib(n-1);
6      y = spawn fib(n-2);
7      sync;
8      return x + y;
9    }
10 }
```

並列度を抽出する上では, 2 個目の再帰呼び出し時の spawn は無意味である. これは, fib が 5 行目で spawn されているために, cilk 関数でなくてはならず, そうするとすべての fib 関数の呼び出しは spawn を用いて行われなくてはならないという, Cilk 特有の構文上の制限からくるもので, 本質的な意味はない.

2.2 各処理系でのモデルの実現方法

この, spawn/sync 構文は, 各種の処理系上で簡単に実装できる. 以下に方針を述べる.

OpenMP: OpenMP の task/taskwait pragma 構文にほぼ一対一に対応している. OpenMP の task pragma は, 任意の文をタスクで実行できる分, より柔軟である. 例えば上記の例題の OpenMP 版は,

```
1  int fib(int n) {
2    if (n < 2) return 1;
```

*1 この制限も Intel Cilkplus においては取り除かれている.

```
  else {
    int x, y;
#pragma omp task shared(x)
    x = fib(n-1);
    y = fib(n-2);
#pragma omp taskwait
    return x + y;
  }
}
```

である．

Intel Threading Building Block (TBB): TBB は C++用に，言語処理系自身を拡張せずライブラリとして実現されたタスク並列処理系である．TBB でタスク並列を記述するためには，task_group クラスを使うのが簡単である．task_group クラスのオブジェクトは，run，wait という 2 つのメソッドを持ち，意味としては前者が spawn，後者が sync に対応する．run の引数は，オペレータ op() を持つデータであればよく，関数ポインタでも，オペレータ op() を定義したクラスのオブジェクトでも良いが，前者は自由変数を暗黙的にタスクに渡せないという制限があり，後者はタスクを作る度にいちいちクラスを定義しなくてはいけないという煩わしさが存在する．これを解決するために，C++0x で導入されたラムダ式 (クロージャ) を用いることができる．GCC (g++) ではバージョン 4.5 以降でサポートされている．[*2] これを用いると，上記の例題はこのように記述できる．

```
#include <tbb/task_group.h>

int fib(int n) {
  if (n < 2) return 1;
  else {
    tbb::task_group tg;
    int x, y;
    tg.run([=,&x] { x = fib(n-1); });
    y = fib(n-2);
    tg.wait();
    return x + y;
  }
}
```

8 行目の，[=,&x] { x = fib(n-1); } がラムダ式の記法で，&x は，変数 x を親と子で共有すること，その前の=は，それ以外の変数は親から子にコピーすることを示している．この場合，文内で用いている自由変数 n がコピーされる．この仕組みを使えば OpenMP の task 構文と同じく，任意の文をタスクとして実行できる上，その変換を C/C++のマクロ程度の機能で機械的に行うことができる．

MassiveThreads, Qthreads, Nanos++: これらはいずれも新しい構文を提供するものではなく，C/C++言語 API を提供するスレッドライブラリである．いずれのライブラリも最終的には命令アドレス (C 言語の関数ポインタ) とひとつの引数を受け取って，それをタスクとして実行する機能を提供する．特に MassiveThreads は POSIX スレッドと同一の関数・引数名で軽量スレッドを提供する．

そのような API 上に簡便なタスク並列 API を実現する際の問題点と解決方法は，TBB の場合とほぼ同じである．すなわち問題点とは，タスクとして実行できるのは命令アドレス (= 自由変数を持たない関数へのポインタで，引数の個数も固定されている) だけであるということであり，解決方法は C++のラムダ式を使うことである．具体的には，

(1) タスクを表すデータ構造 task を作り，ラムダ式をメンバとして格納する
(2) task へのポインタを唯一の引数として受け取り，その中のラムダ式を起動するだけの関数 invoke_task を定義する．
(3) タスク生成は invoke_task を実行するスレッドを (個々のライブラリのスレッド生成 API を用いて) 作成することで行う．

ラムダ式を受け取り，task 構造体にそれをパッケージ化し，新たにスレッドを生成し，そのスレッドの中でラムダ式を取り出して呼び出す，という一連の操作を一つの API で実現すれば，ユーザにとっては TBB の task_group クラスの run メソッドと同等の簡便な API となる．我々はこの手法で，TBB の task_group と同一の API を持つクラスを，MassiveThreads, Qthreads, Nanos++の 3 種のスレッドライブラリ上に実装した．3 種に共通した実装概要は，説明のためにやや簡略化して書くと以下のとおりとなる．

```
class task_group {
  int n_tasks;
  task tasks[NTASKS]; // spawn されたタスクを格納
  ...
  void run(std::function<void ()> lambda_exp) {
    // ラムダ式を受け取り，task 構造体に格納し，スレッドを生成する
    task * t = &tasks[n_tasks++];
    t->lambda_exp = lambda_exp;
    t->tid = thread_create(invoke_task, (void*)t);
  }
  ...
}
// スレッドにより起動される関数
void * invoke_task(void * arg) {
  // タスクからラムダ式を取り出し，呼び出す
  task * t = (task*)arg;
  t->retval = t->lambda_exp();
  return NULL;
}
```

[*2] http://gcc.gnu.org/projects/cxx0x.html

8 行目の thread_create は各スレッドライブラリにおける, スレッド生成関数を呼び出すところである. MassiveThreads では, pthread_create, Qthreads では qthread_fork, Nanos++では, nanos_submit を用いている. また, 上記の説明では task_group 内に格納できるタスク数 (つまり, ひとつの task_group で連続して spawn できるタスク数) が, NTASK という一定値になっているが, 実際の実装では必要に応じて動的に配列を拡張している.

2.3 構文の統一

以上により, Cilk の spawn/sync に相当するモデルは, どの処理系の上でも簡単に実現できることがわかった. それでも表面的な構文としては Cilk, OpenMP, (MassiveThreads, Qthreads, Nanos++の 3 種の処理系も含む) TBB, の 3 通りが存在する. 我々はそれを土台に, それらの表面的な違いを吸収し, 全く同一の構文でベンチマークやアプリケーションを記述できるマクロを定義した. そのマクロを用いた記述例を以下に示す. これまで述べたどの記述方法にも機械的に変換できるよう, 若干冗長になっているが, 新たな言語処理系を開発することなく実現できる利点がある.

```
1  __cilk__ int fib(int n) {
2    if (n < 2) return 1;
3    else {
4      spawn_tasks {
5        int x, y;
6        spawn_task1(x, x = spawn fib(n-1));
7        call_task(y = spawn fib(n-2));
8        sync_tasks;
9      }
10     return x + y;
11   }
12 }
```

3. タスク並列プログラムの性能理解

3.1 基本

タスク並列プログラムでは, プログラマではなく処理系が実行時に負荷分散を行う. そのため不確定な要素が多く, 性能が予測しにくいという側面がある. しかし基本はむしろ簡単で, まずは以下の事実から出発する [4].

> タスク並列プログラムを貪欲なスケジューラで実行すると, 以下が成り立つ.
>
> $$T_p \leq T_1/P + T_\infty \qquad (1)$$

ここで, P はプロセッサ数, T_1 プログラムの総仕事量, T_∞ はクリティカルパス長である. T_1 は一台での実行時間, T_∞ は台数を制限せずに理想的に実行した場合の実行時間ということもできる. 証明は [4] を参照.

上記定理の重要な点として以下があげられる.

(1) T_1, T_∞ はプログラムの実行方法によらない尺度として, 多くの場合抽象的なアルゴリズムの記述から簡単に導ける点.

(2) 一度それができれば, $A = T_1/T_\infty$ で定義される, 「平均並列度」とでもいうべき値を用いて, 期待できる台数効果が簡単に評価できる点.

2 点目に関してより具体的に言えば, 理想的な P 台の台数効果の, $(1 + P/A)$ 倍程度しか悪くならないということが言える. 例えば, 平均並列度が 500 のプログラムを 100 台で実行すれば, 台数効果は理想 (100 倍) と比べてせいぜい, $1+100/500 = 1.2$ 倍程度にしか悪くならない—すなわち約 80 倍の台数効果が出る—と, 簡単に予測できる点である.

貪欲なスケジューラとは, システム中のどこかに実行可能な仕事がある限り, 決してプロセッサを遊休させないスケジューラのことである. 貪欲なスケジューラを厳密に実現するには, システム中のどこかに実行可能なタスクがある限り, 即座にを発見し, 実行を開始しなくてはならない. したがって上で述べた通りの「理想的な」貪欲なスケジューラを実現するのは難しい. しかし, 遊休状態になったら積極的に他のプロセッサからタスクを盗むスケジューラであれば, 盗むタスクを発見するまでの時間が十分短ければ近似的に貪欲と言って良い. その意味で, ほとんどの動的負荷分散を実現するスケジューラは, 貪欲なスケジューラを近似していると言って良い.

例えば, 効率的な動的負荷分散処理系の元祖である Lazy Task Creation[7] では, 各プロセッサが固有のなタスクキューを保持し, プロセッサが遊休状態になったら他のプロセッサをランダムに選んで, そこでタスクが見つかればそれを奪う (ランダムワークスチーリング [2]). 選んだプロセッサが奪えるタスクを持っていなければ, 他のプロセッサを継続して探す. これも貪欲なスケジューラを近似的に実現していると言って良い. Blumofe らはランダムワークスチーリングに基づいたスケジューラでは, 実行時間が高い確率で $T_1/P + O(T_\infty)$ に収まることを証明している [2].

3.2 基本式からの乖離

式 (1) は動的に負荷分散を行う処理系で, どのような性能を期待すべきか・すべきでないかということを教えてくれる重要な式である. 一方で上記を導くためにいくつかの理想化が行われており, 現実の処理系やマシンではそれが満たされていないという場合がある. その理想化とは, (1) 全てのアイドル時間は, 実行可能なタスクが存在しないことによってのみ生ずる, (2) システム関数の呼び出し (タスクの生成や同期など) は完全にスケールする, つまりそれらのオーバーヘッドは台数によらず一定である, (3) アプリケーション固有の仕事 (システム関数などを除いた, 純粋なアプリケーションの処理) は, どのプロセッサで実行しても, 同じ時間で行われる, というものである. 現実のシステムと, 式 (1) との乖離も, 基本的にはこれらのどれかに帰着でき

表 1　実験環境

機種	アーキテクチャ	周波数	PU 階層	キャッシュ
Opteron 6172	Magny Cours	2.1 GHz	4/8/48	64KB/512KB/10MB
Xeon Phi 7120P	MIC	1.2 GHz	61/244	32KB/512KB

表 2　評価対象処理系

通称	名称	環境	バージョン
omp	OpenMP	opteron	GCC ver 4.7.2
tbb	TBB	opteron	4.1.6102
cilk	Cilk	opteron	Cilk 5.4.6
mth	MassiveThreads	opteron	0.9
qth	Qthreads	opteron	1.9
nanox	Nanos++	opteron	0.7a 2013/06/14
iomp	OpenMP	mic	Intel C Compiler 13.0.0
itbb	TBB	mic	TBB 4.1.6100
icilk	Cilk	mic	Intel C Compiler 13.0.0
imth	MassiveThreads	mic	0.9

る. 言い換えれば乖離の要因は以下のように分類できる.

本質的でないアイドル時間 (non-inherent idle time): 実行可能なタスクと遊休なワーカが同時に存在しているにも関わらず, そのタスクが実行されないまま放置されている時間. ワークスチーリングを行う処理系では, ワークスチーリングにかかる時間 (タスクを見つけ, プロセッサ間で移動させ, その実行を開始するまでの全時間) が, 実行時間に影響する.

API のボトルネック (API bottlenecks): タスク生成や同期など, API を呼び出して行われる操作の実行時間が, 台数と共に増加する. 一部の処理系ではプロセッサ間の共有キューを用いてタスクを管理したり, 一つのプロセッサがスケジューリングを担当するなどにより, このようなことがおこる.

仕事時間の伸び (work time stretch): タスク生成や同期などを含まない, アプリケーション固有の部分を実行している場合でも, 1 台で逐次実行した場合と並列に実行した場合とで, 時間が異なることがある.

最後に述べた仕事時間の伸びの原因について補足する. 最も典型的かつ動的負荷分散を行う処理系につきまとう問題は, タスクの移動によるキャッシュミス数の増加である. これは, 1 台での実行においては必然的に同じプロセッサ上で時間的に連続して実行される 2 つのタスク $A; B$ が, 並列実行では異なる 2 つのプロセッサ上で実行され得る, ということに起因して発生する. このシナリオでは, 1 台の実行では, A によってキャッシュに持ち込まれたデータ B が再利用可能であるのに対し, B が他のプロセッサに移動することでそれがまず期待できなくなる.

4. 評価実験

4.1 環境

用いた環境を表 1 に, 処理系を表 2 に示す.

実験中にわかったこととして, Qthreads のデフォルトのスケジューラは貪欲ではないということがわかった. 具体的には, spawn cache と呼ばれる仕組みがあり, 生成されたタスクはしばらくの間, 生成したプロセッサ内の spawncache に置かれ, 他のプロセッサによって実行することはできない. これは多数が負荷分散する遅延を犠牲にしてスループットをあげることを意図した仕組みだが, プログラマから挙動がわかりにくい上, 本論文のベンチマークに対しては悪い作用しか持たない. そこで, 処理系の configure 時に —disable-spawn-cache を追加してこの機能を無効化している.

Nanos++には多数のスケジューラがあるが, 後に述べる bin ベンチマークの結果が最良であったものを選び, 結果として wf (work first) スケジューラを用いた.

5. ベンチマーク

5.1 タスク生成

本ベンチマークでは, タスク生成のオーバーヘッド及びそのスケーラビリティを評価する. ベンチマークの核は以下のような, 何もしない関数 (`null_task()`) を呼び出すタスクを繰り返し生成しては直後に終了を待つというループである.

本ベンチマークは以下を全コアで行う.

```
1 void null_task() { }
2 void create_many() {
3   spawn_tasks {
4     for (i = 0; i < many_times; i++) {
5       spawn_task(null_task());
6       sync_tasks;
7     }
8   }
9 }
```

タスク生成・終了検出のオーバーヘッドが小さいことは以下の観点から重要である.

生産性: タスク生成・終了のオーバーヘッドを一定以内に抑えたければ, タスクの粒度はタスク生成・終了のオーバーヘッドに比例して決める必要がある. タスク生成・終了のオーバーヘッドが小さいことは, タスクの粒度を決める際の制約が緩くなり, プログラマにとって自然な記述を許しやすくなることを意味する.

スケーラビリティ: タスク並列処理では多くの場合, 3 節述べた理由により, 平均並列度, すなわち T_1/T_∞ をプロセッサ数 P よりも, 「十分」大きくしておくことが推奨される. タスクの粒度を小さくすることは多くの場合, T_∞ を小さくすることに貢献する.

5.2 1 プロセッサのオーバーヘッド (ベストケース)

図 1 に, 1 プロセッサで上記のコードを実行した時の, タスク一回 (ループ一周) あたりの時間 (クロック数) を示す. 一回の測定は 1000 回の繰り返しで測定しており, その測定一般にどの処理系も, プログラム開始直後の測定は例外的

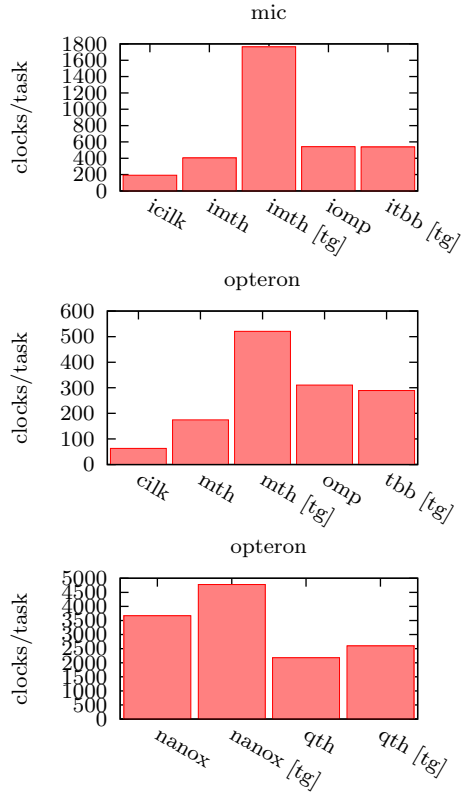

図 1 1 プロセッサでのタスク生成オーバーヘッド

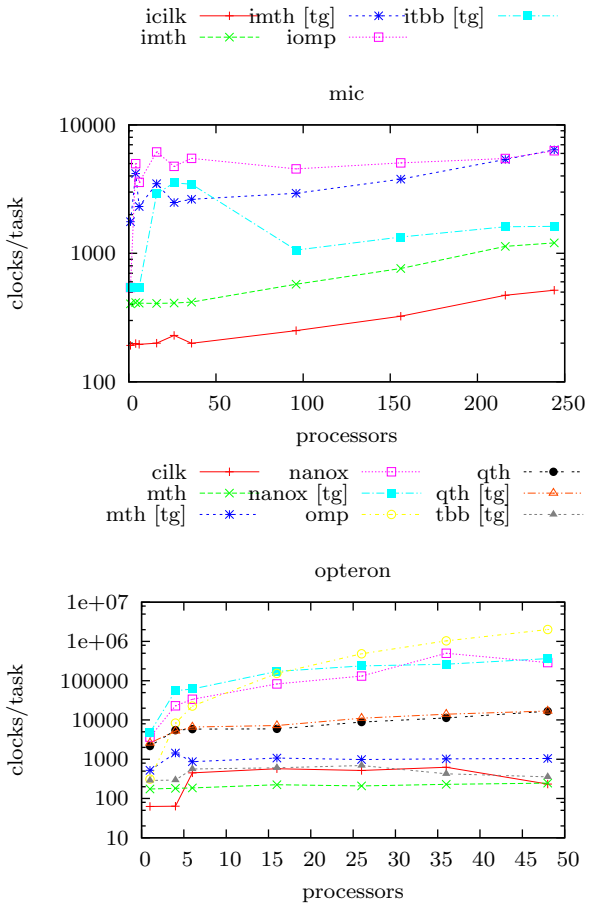

図 2 複数プロセッサでのタスク生成オーバーヘッド

に悪い場合が多い. これはメモリ管理やディマンドページングによるオーバーヘッドなどが含まれていると想像されを 31 回繰り返し, 最初の 5 回を除いた平均を示している.

基本的には上述のコードがマクロ展開によって 2 節で述べたそれぞれのシステムのプリミティブへ展開されるが, MassiveThreads, Qthreads, Nanos++に関しては,`task_group` クラスのオーバーヘッドを排除し, 直接各ライブラリのスレッド生成関数を呼び出すコードも別途記述し, 評価に含めている. グラフ中の横軸で, [tg] がついているものが, `task_group` クラスを用いたものである. Qthreads, Nanos++は他の処理系と比べて 1,2 桁遅いため, 別のグラフにしてある.

5.3 スケーラビリティ

次に, 上記のコードを多数のプロセッサで同時に実行した時, 各プロセッサが観測したタスク一回あたりのクロック数が, プロセッサ数と共にどう変化するかを測定した. プログラムとしては, 上記 create many 関数を実行するタスクを, プロセッサ数と同じだけ作り, かつタスク生成のループに入る前にバリア同期をして, 実行タイミングが重なるように制御している. 結果を図 2 に示す. 多くの処理系でプロセッサを増やすとタスク生成のオーバーヘッドは著しく増えた. グラフの縦軸は log スケールであることに注意されたい. 中でも顕著なのは,

(1) omp (Opteron 上の GCC OpenMP; GOMP): 48 プロセッサ時に, 1 プロセッサ時の 6000 倍以上であり, グラフを見てもさらに伸び続けることが予想される. 絶対値としても, 200 万クロック (1 ms 程度) に達している.

(2) nanox (Opteron 上の Nanos++): 48 プロセッサ時に 1 プロセッサ時の 70-80 倍である. 絶対値としては, 30 万クロック (150μ 秒) 程度である.

個々の処理系の性能の原因を探ることは, 筆者の現在の余力及び本論文の範囲外であるが, GOMP はタスク管理を共有キューで行なっており, それがスケーラビリティを阻害する一因であることは想像しやすい. MIC 上では, どの処理系も増大しているものの, 物理コア数を越えた所での増大は, プロセッサ資源の競合の可能性もある. 物理コア数を超えない範囲での伸び率を見ると, Intel OpenMP が 10 倍程度になっているが, 絶対値としてはそれでも 5000 クロック (5 μ s) 程度である. これらの結果は, それぞれの処理系で適切なタスクの粒度を考える際に, タスク生成のコストを一定と考えてはいけない, またはそれがあまりにも複雑であれば, 最悪値を考えるべきである, ということを意味する. 例えば GOMP 処理系で良好な台数効果を得るためには, 200 万クロックのタスク生成・終了オーバーヘッドでも問題にならない, その数倍の大きさのタスクを

作る必要があり, これは応用範囲を妨げる.

5.4 負荷分散速度

5.4.1 最低遅延

タスクを作っても実際に並列度が抽出されるためにはそのタスクないし別のタスクが他のプロセッサに移動しなければ並列処理は行われない. 移動に多くの時間を要すればその時間はアイドル時間と同じことであり, 台数効果を下げる要因となる.

どのような戦略でタスクを移動させるかは処理系によって異なるが, それに依存しない基本的なベンチマークとして以下の測定を行う.

(1) 親タスクが時刻を記録 (t_0 とする). 計時開始直後に子タスクを一つ作る.
(2) 子タスクが実行を介した時点で時刻を記録する (t_1 とする)
(3) 親タスクが子タスク生成直後に時刻を記録する (t_2 とする)
(4) $(t_1 - t_0)$, $(t_2 - t_0)$ の大きい方が, 「並列度を 1 増やすのに要する最低時間」である.

コードとしては以下のようになる.

```
1  volatile int parent_started = -1;
2  volatile int child_started = -1;
3  __cilk__ void child() {
4    t1 = 時刻 ();
5    parent_started = 1;
6    while (parent_started == -1) ;
7  }
8  __cilk__ void parent() {
9    parent_started = 1;
10   child_started = 1;
11   spawn_tasks {
12     t0 = 時刻 ();
13     spawn_task(spawn child());
14     parent_started = 1;
15     while (child_started == -1) ;
16   }
17 }
```

タスク並列処理系はその実装によって, この状況で子タスクが直ちに開始されて, 親が他のプロセッサに移動するもの (work-first または child-first), その逆 (help-first または parent-first) が存在するが, 前者においては得てして, $(t_2 - t_0)$ に大きな値が, 後者においては $(t_1 - t_0)$ に大きな値が観測されることになる. そのためこれは処理系がどちらの戦略をとっているかの, 大雑把な判定方法としても機能する.

図 3 がその結果である. 測定は 10000 回繰り返した上で最初の 1000 回を除いている. なお以降では, MassiveThreads, Qthreads, Nanos++においては, task group の API を用いたもののみを評価の対象とする.

際立っているのは, Cilk は Opteron, MIC 双方で, 子が

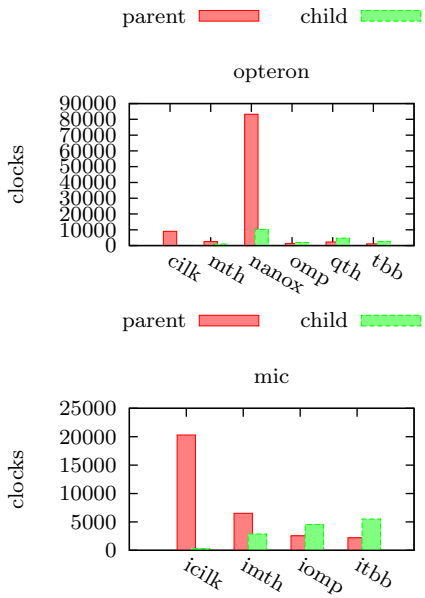

図 3 タスクの移動時間

開始するまでの実行時間が極めて短いことである. これは Cilk の実装方式を考えるとうなづける結果である. Cilk では spawn 時に子タスクを即座に実行する. Cilk の言語処理系によって, この処理は関数呼び出し + 中断時にコンテクストを保存するためのヒープフレーム割り当て程度になっている (Opteron で 80 クロック, MIC で 285 クロック程度. 一方で親タスクを別プロセッサに移動した上で再開させるまでの時間はそれとは比較にならないほど大きい (Opteron 上 9000 クロック, MIC 上で 20000 クロック).

対して MassiveThreads は, スケジューリング方針自体は Cilk と同じく, work-first であるが, 子タスクが開始するまでの時間は Cilk よりも一桁以上大きい (Opteron 上で約 1000 クロック, MIC 上で約 3000 クロック). これは前節で示した, task group API を用いた場合のタスク生成・終了のオーバーヘッド (Opteron 上で約 500 クロック, MIC 上で約 1800 クロック) を考えればうなづける. 一方, 親が再開するまでの時間は Cilk よりも数倍短い. これは, 平均並列度がプロセッサ数に比べてそれほど豊富でないプログラムにおいて効果を発揮する性質である.

また, OpenMP, TBB, Qthreads においては子タスクのほうが後から実行されており, work-first ではなく, help-first であると予測できる.

5.4.2 多数プロセッサへのタスク拡散

前節では「一つの」タスクが別のプロセッサに移動する時間を測定したが, ここではそれが組み合わさって, 多数のプロセッサへ多数のタスクが拡散するのに要する時間を測定する. コードとしては, 2 分木状にタスクを作る. 木は, プロセッサ数と同じだけの末端ノードを持つような (不完全)2 分木である. 末端では充分長い時間を消費して, 確実に各プロセッサが一つの末端タスクを実行するようにする.

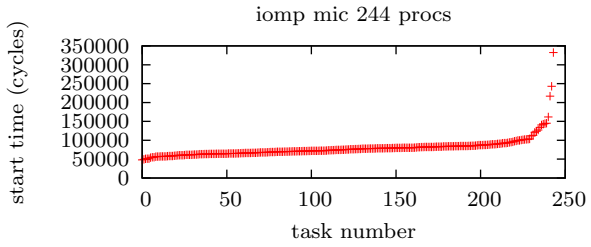

図 4　MIC 244 コアへのタスクの拡散

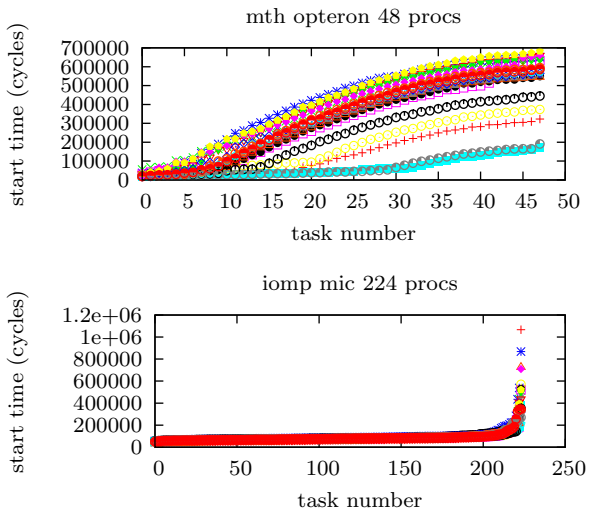

図 5　タスクの拡散

擬似コードは以下の通りである.

```
_cilk__ double create_tasks(int l, int r) {
  if (r - l == 1) {
  時刻を記録 ();
  長時間消費 ();
} else {
  spawn_tasks {
    int c = (l + r) / 2;
    spawn_task(spawn create_tasks(l, c));
    call_task(spawn create_tasks(c, r));
    sync_tasks;
  }
}
```

図 4 は, MIC 上の OpenMP で 244 個プロセッサを用いたある一回の試行の結果を表示している. 各タスクが開始時に取得した時刻を整列している. グラフにおける一つの点 (x, y) は, x 番目開始の早かったタスクの開始時刻があったことを示している. グラフを 90 度回転させてみれば, 時間と共に並列度がどのように上昇していったかを表していると言える.

図 5 は, 30 回の試行を行い, 最初の 10 回を 20 回分の試行を重ねて表示している. スペースの都合で, MassiveThreads (Opteron) および OpenMP (MIC) での結果を示している.

多くの処理系で共通して現れる現象として, グラフの右端に外れ値的に大きな値が出現するという点があげられる (qth, tbb, iomp, itbb, icilk, imth) で現れているが, 紙面の都合上載せていない他のグラフを見ると, 頻度の差こそあれ全ての処理系で起きている現象である. 原因は追求していないが, OS ノイズの影響も考えられる.

注意すべき点はこのプログラムは P 個のプロセッサに対してちょうそ P 個のタスクを作るという人工的な例であって, タスク並列モデルにおいては推奨されない (T_1/T_∞ が P に比べて十分大きくない) プログラムだということである. T_1/T_∞ が十分大きければ, たとえ最後の何% かのプロセッサがノイズその他の影響で遅れても, 他のプロセッサがそれを補うことができる. 逆にこの例は, そうすることの重要性を示しているとも言える. また, ほぼすべてのプロセッサにタスクを行き渡らせるのに要する時間として, 絶対値を頭に入れておくこと, そのばらつきの大きさが大きいことも, 処理系の性能を理解する上で重要である. たとえば Opteron の 48 のプロセッサに行き渡らせるのに MassiveThreads ではおよそ 100000 から 500000 クロック程度を要している. これは 50μ 秒から 250μ 秒程度の値で, 48 プロセッサでのプログラムの処理時間がこれより十分長い必要がある. 仮に 10 倍長いことを安全圏と考えるとすると, 一台での時間が, $250\mu \times 10 \times 48 = 120$m 秒程度の仕事であれば, タスク拡散速度自体が問題とはならないであろうと期待できる.

5.5　スケーラビリティベンチマーク

これまでの顕微鏡的な測定ではなく処理全体のスケーラビリティの違いを理解するベンチマークとして, 人工的な 2 分木状のタスク生成プログラムを考える.

```
void bin(int n) {
  CPU を消費(A);
  if (n > 0) {
    spawn_tasks {
      for (i = 0; i < K; i++) {
        spawn_task(spawn bin(n - 1));
        bin(n - 1);
        sync_task;
        CPU を消費(A);
      }
    }
  }
}
```

このプログラムの仕事量およびクリティカルパスは, 以下の 3 つのパラメータを用いて表現できる.

- 「CPU を消費 (A)」が呼ばれるごとに消費する時間 A
- 再帰の深さ n
- 各段で再帰を行う回数 K

T_1, T_∞ それぞれについて漸化式をたててとけばよく, 結果だけを示すと,

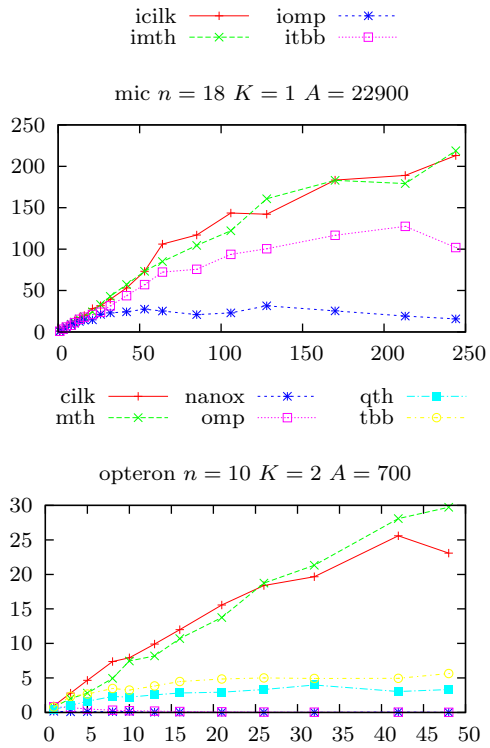

図 6 2 分木状タスク生成プログラムの台数効果

$$T_1(n) = \frac{A}{2K-1}(3K(2K)^n - (K+1))$$

$$T_\infty(n) = \begin{cases} 2An & \text{if } K = 1, \\ \frac{A}{K-1}(2K^{n+1} - (K+1)) & \text{if } K > 1 \end{cases}$$

となり, 平均並列度は,

$$\frac{T_1(n)}{T_\infty(n)} \approx \begin{cases} \frac{3}{2} \cdot \frac{2^n}{n} & \text{if } K = 1 \\ \frac{3(K-1)}{2(2K-1)} 2^n & \text{if } K > 1 \end{cases}$$

以下の 2 つの場合の結果を示す. A の単位はおおよそのクロック数である.

- $A = 23000, n = 18, K = 1 \rightarrow T_1 = 786430A, T_\infty = 36A, T_1/T_\infty \approx 21845$
- $A = 750, n = 10, K = 2 \rightarrow T_1 = 2097151A, T_\infty = 4093A, T_1/T_\infty \approx 512$

平均並列度という点では両者とも 48 プロセッサであれば十分大きく, 前者は 244 プロセッサに対しても十分大きい. 図 6 に, 前者の Opteron での台数効果, 後者の MIC での台数効果を示した. Opteron での結果は, 図 2 に示したタスク生成のオーバーヘッドが台数と共に増大する処理系の傾向を考えれば, 納得できる (少なくとも驚くには値しない) 結果と言える. たとえば nanox, omp とも, 1 プロセッサではせいぜい数千クロックであったタスク生成のオーバーヘッドが数十万クロックになっており, タスクの粒度 ($A = 750$ 程度) と比較してはるかに大きい. Qthreads も 2000 から 20000 程度に増大している. それだけでは理解できないのは TBB の台数効果の悪さである. TBB のタスク生成オーバーヘッドは台数が増えても悪くならないことを既に見ている. 今後調査をする予定であるが, 他のパラメータでの実験結果から TBB は $K = 2$ の場合に台数効果が悪いことがわかっている.

次に MIC での結果についてだが, OpenMP の結果が悪いことが目につく. 再び図 2 を参照すると, 確かに OpenMP はプロセッサ数が 1 から 244 まで増えるに従い, タスク生成のオーバーヘッドが 500 から 5000 以上に増大している. したがって粒度 23000 程度のタスクがそれに影響を受け効率が悪くなることは, ある程度予測できる. しかしそれで説明できるのは $23500/28000 \approx 82\%$ 程度であり, OpenMP のスケーラビリティの悪さはそれをはるかに下回っている. 今後, このベンチマークにおいて実際に起きていることの考察を進めていく.

6. 関連研究

タスク並列処理系を比較した研究としては, [9], [11] がある. 論文 [9] は UTS ベンチマークの比較を OpenMP, Cilk, Cilk++, Intel Thread Building Blocks に対して 16 プロセッサまで行なっている. [11] はマイクロベンチマーク及び, Bacelona OpenMP Task Suite 中のいくつかのアプリケーションの評価を OpenMP, Cilk++, Intel Thread Building Blocks に対して行なっている. 評価環境は前者が 16 コア, 後者が 8 コアまでである. 本研究ではより大規模な環境で, マイクロベンチマークから処理系のスケーラビリティ特性を抽出し, アプリケーションの台数効果の理解につなげることを目指した.

7. 終わりに

タスク並列プログラミングモデルの性能を理解するための基本的なモデルを示し, 現実の環境や処理系において, 基本的なモデルと乖離する原因についてのべた. それをもとに 6 種のタスク並列処理系のベンチマークを行い, 特にいくつかの処理系は, 台数と共にタスク生成オーバーヘッドが極めて悪化することがわかり, それが多くの場合, 台数効果の妨げとなるであろうことが予測できた. 実際のスケーラビリティ測定においてもそのような傾向が見られることが確認できた. 今後は, 得られたスケーラビリティのより詳細な説明, より複雑な現実のアプリケーションに対する評価を行なって行く.

謝辞 本研究の一部は科学技術振興機構, CREST 研究課題「高性能・高生産性アプリケーションフレームワークによるポストペタスケール高性能計算の実現」の助成を得て行われた.

参考文献

[1] Nanos++. http://pm.bsc.es/nanox.
[2] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.
[3] Bradford Chamberlain, David Callahan, and Hans Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.
[4] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on*, 38(3):408–423, 1989.
[5] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley & Sons Inc., 2005.
[6] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, New York, New York, USA, May 1998. ACM Press.
[7] Eric Mohr, David A. Kranz, and Robert Jr. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
[8] Jun Nakashima, Sho Nakatani, and Kenjiro Taura. Design and implementation of a customizable work stealing scheduler. In *ROSS '13 Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*.
[9] Stephen L. Olivier and Jan F. Prins. Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. *International Journal of Parallel Programming*, 38(5-6):341–360, June 2010.
[10] OpenMP Architecture Review Board. OpenMP Application Program Interface. (July), 2011.
[11] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A Comparison of some recent Task-based Parallel Programming Models, January 2010.
[12] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.
[13] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, April 2008.