細粒度マルチスレッド処理系 MassiveThreads

秋山 茂樹

全体ミーティング (2011/6/27)

コンピュータにおける並列度の増大

CPUのマルチコア化、SMT機構の採用

- 例:
  - Xeon E7-8820: 10コア (SMTで20スレッド)
  - Opteron 6176 SE: 12コア
  - Cray XMT: SMTで128スレッド
- マルチプロセッサ構成でコア数が数倍になりうる

不規則問題 (irregular application)

複雑なデータ構造・領域分割の必要な計算

- 特徴
  - 静的なタスクの分割が困難
  - 不規則なメモリアクセスが発生
- 例: 適合細分化格子(AMR)法,
  高速多重極展開(FMM)法

性能の良いプログラムを実装するためには
- **動的負荷分散**
- **通信コスト隠蔽**

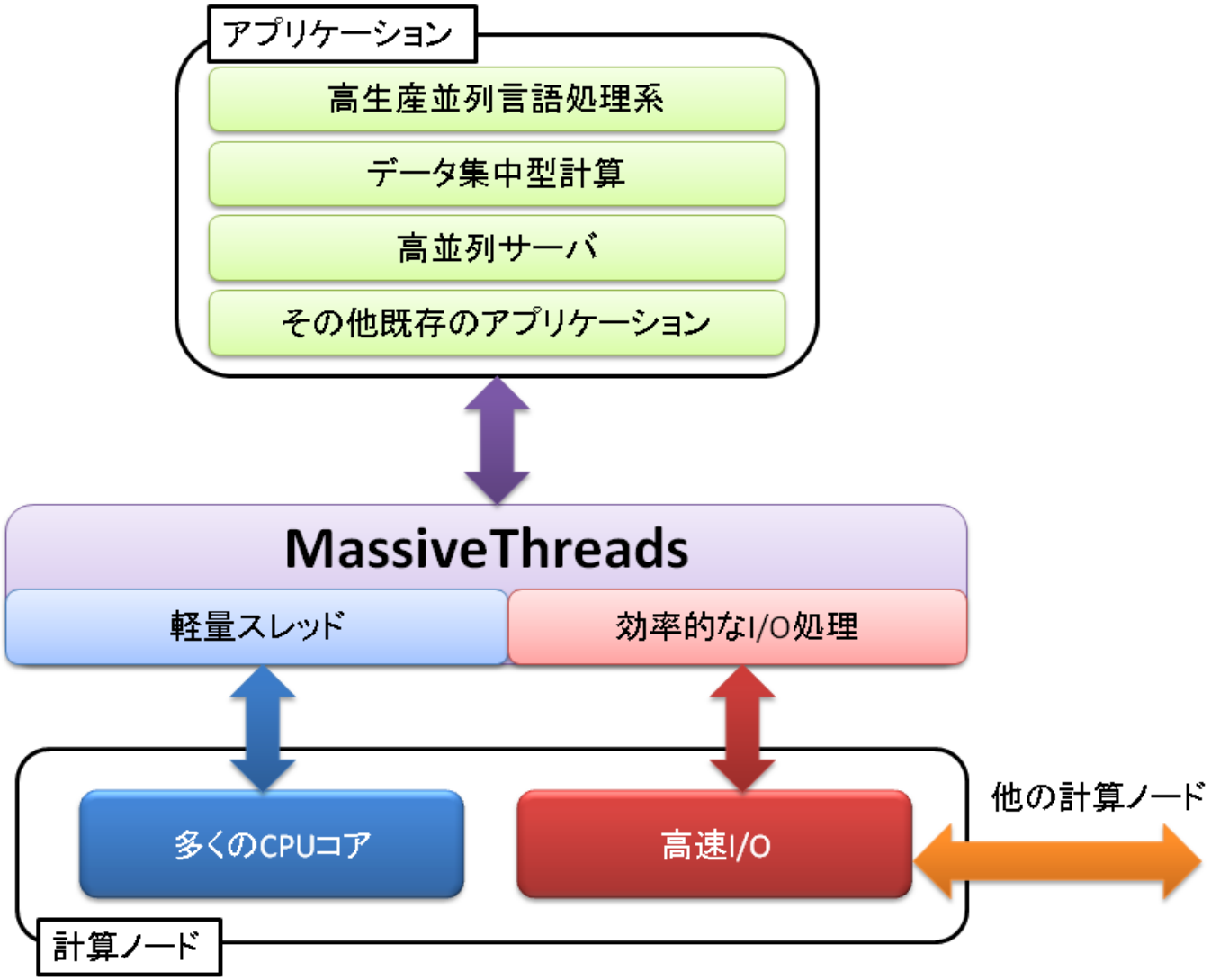
を明示的に記述しなければならない

MassiveThreads

計算ノード内での動的負荷分散を実現する マルチスレッド処理系

- 目標
  - スレッドの軽量性
  - スケーラブルな動的負荷分散
  - コンテキスト切り替えによるI/Oコスト隠蔽
  - 既存のpthreadプログラムを再利用できること
- 軽量スレッドと効率的I/O処理を同時に満たす マルチスレッド処理系は存在しない

MassiveThreads



プログラミングモデル

pthreadと同様のAPI, read/write syscall で通信

- pthread_create, pthread_join, etc...
- ただしスレッドスケジューリングの公平性は保証されない

```
void *fib(void *n) {
  if (n < 2) return n;
  else {
    pthread_t th0, th1;
    pthread_create(&th0, NULL, fib, n-1);
    pthread_create(&th1, NULL, fib, n-2);
    int result0, result1;
    pthread_join(th0, &result0);
    pthread_join(th1, &result1);
    return result0 + result1;
  }}
```

MassiveThreads の実装

Lazy Task Creation ベースの細粒度スレッド実装 + 軽量スレッド処理系におけるI/Oコスト隠蔽

- 特徴
  - Lazy Task Creation ベースのスレッドスケジューリングを**Cライブラリの形式**で実現している
    - ソースコード変換, アセンブリポストプロセスなどの方法を利用していない
  - スレッド切り替え + I/O多重化 を用いてI/Oコスト隠蔽を実現している

今日はこちらの話だけ

マルチスレッド処理系の基本的な実装

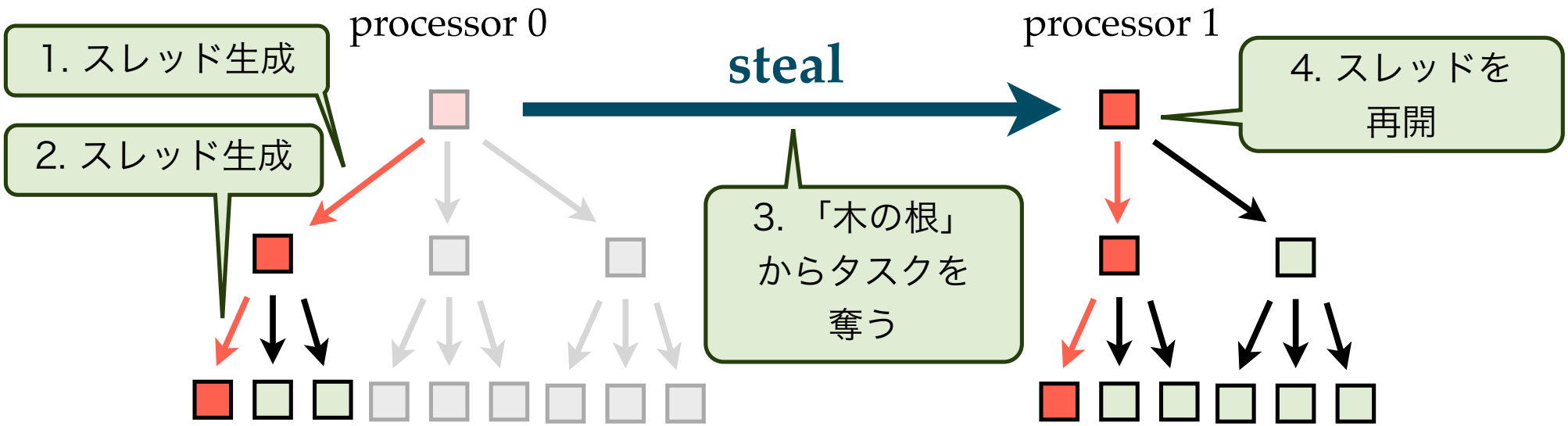
“ランキュー”を用いて実行可能なスレッドを管理

- スレッド生成:
  - ランキューにスレッドを追加
- スレッド実行終了:
  - スレッド管理領域に結果を書き込む
  - ランキューからスレッドを取り出し、実行する
- スレッド合流(join):
  - 合流するスレッドが終了していれば、結果を取り出す
  - 合流するスレッドが終了していなければ、終了するまでランキュー中のスレッドを実行する

Lazy Task Creation

再帰的なスレッド生成を効率よく扱える軽量スレッド+動的負荷分散実装手法

- LIFO形式のスレッドスケジューリング
- 「木の根」からタスクを奪う動的負荷分散手法



Lazy Task Creation

FILO形式のスレッドスケジューリング

- スレッド生成:
  - **実行中のスレッド**をランキューの**先頭**に追加し、新しいスレッドに切り替える
- スレッド終了:
  - ランキューの**先頭**からスレッドを取り出し、実行する

動的負荷分散

- 実行可能スレッドがなくなったら「木の根」からスレッドを奪い、実行する
  - 別ワーカーのランキューの**末尾**からスレッドを奪い、実行

Lazy Task Creation の利点

局所性が利用可能

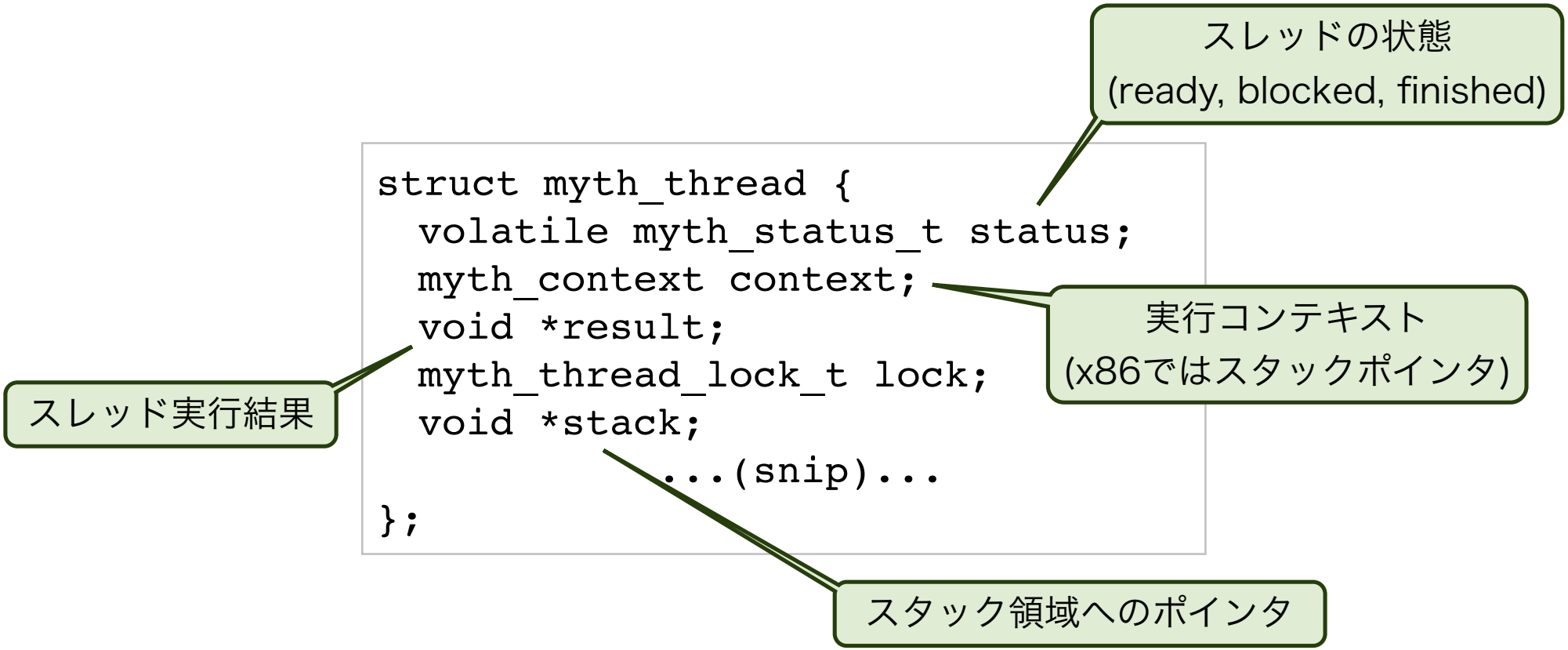
- スレッドの実行順序が関数呼び出しと同様

メモリ消費量が少ない

- 再帰の深さdについてO(d)で済む
  (FIFOなスケジューリングであればO(2^d))
  - cf. 深さ優先探索 vs. 幅優先探索

LTCをCライブラリとして実装 (MassiveThreads)

スレッド管理領域のデータ構造



LTCをCライブラリとして実装 (MassiveThreads)

スレッド生成:

- スレッド管理領域とスタックの割り当て、初期化
- 新しいスレッドに**コンテキスト切り替え**
- 実行していたスレッドをランキューの先頭に追加
- 新しいスレッドでスレッド関数を実行する

LTCをCライブラリとして実装 (MassiveThreads)

コンテキスト切り替え:

1. callee-saveレジスタの退避
2. 現在のスタックポインタを実行中スレッドの管理領域に保存
3. 切り替え先スレッドの管理領域に保存されているスタックポインタを復元
4. callee-saveレジスタの復元

```
asm volatile(PUSH_CALLEE_SAVED()
             “push $1f\n”
             “mov  %%rsp,%0\n”
             “mov  %1,%%rsp\n”
             “ret”
             "1: " POP_CALLEE_SAVED()
) ...;
```

実行中スレッドの実行コンテキスト

切り替え先スレッドの実行コンテキスト

LTCをCライブラリとして実装 (MassiveThreads)

スレッド終了:

1. スレッド管理領域をロック
2. 管理領域に結果を格納し、状態を finished にする
3. スタック領域を解放
4. このスレッドを join したスレッドがあれば、
   そのスレッドを実行
5. ランキューの先頭からスレッドを取り出し、実行

LTCをCライブラリとして実装 (MassiveThreads)

スレッド合流(join(th)):

1. スレッドthの管理領域をロック
2. スレッドthの状態が finished:
    1) 結果を取得
    2) スレッドthの管理領域をアンロック
    3) スレッドthの管理領域を解放
3. スレッドの状態が finished でない:
    1) 実行中スレッドの状態を blocked に変更
    2) スレッドthの管理領域に実行中スレッドを登録
    3) ランキューの先頭からスレッドを取り出し、実行

LTCをCライブラリとして実装 (MassiveThreads)

タスクスティール:

- スレッドを盗むワーカーをランダムに選ぶ
  - ランダムに選ぶのが最良[Frigo,1998]
- そのワーカーのランキューの末尾からスレッドを取り出し、再開する

その他実装について

スレッド管理領域・スタックのメモリ管理

- フリーリストを利用
- 管理領域とスタックの分離により、メモリ領域の再利用率を改善

ランキューの実装

- Java fork/join framework と同様のアルゴリズム
  - 先頭への追加/削除操作はメモリバリアのみ
  - 末尾からの取り出しは CAS が必要

評価

実験で次の項目を評価

- スレッドの軽量性
- 負荷分散性能

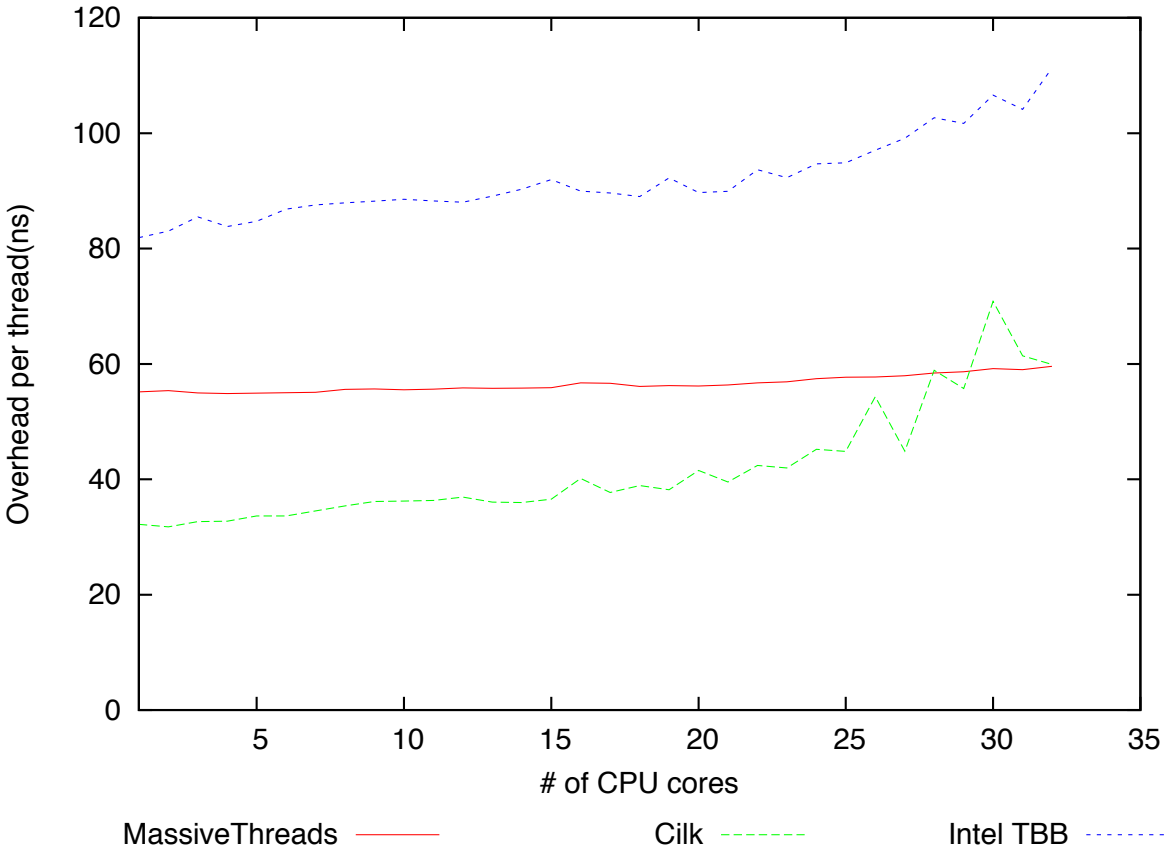
評価環境

- CPU: Opteron 2380 (2.5GHz, 4コア) * 8ソケット
- コンパイラ: GCC 4.4.1

評価1: スレッドの軽量性

スレッドの生成・合流にかかる時間を計測 (fib(40))

- コア数の増加に対してスケーラブルになっている



評価1: スレッドの軽量性

1コア使用時のオーバーヘッド内訳 (単位はcycles)

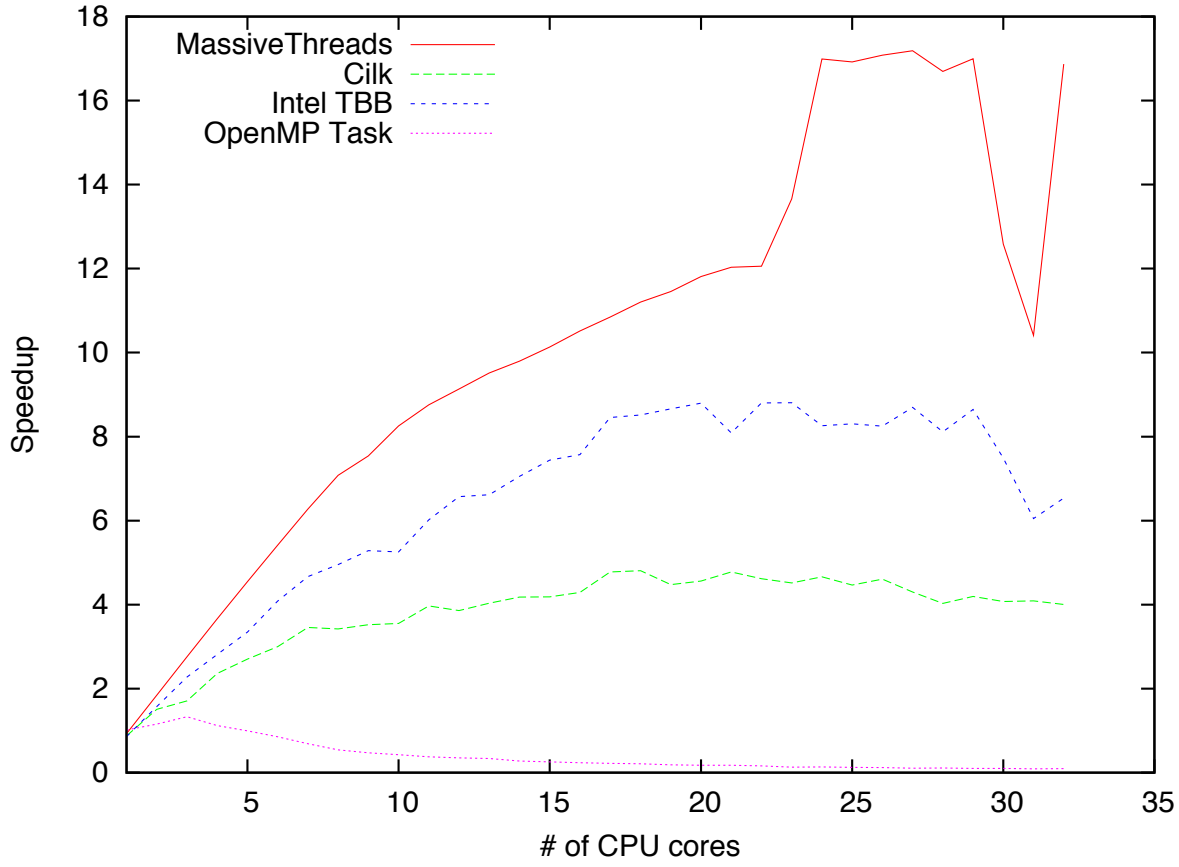
	MassiveThreads(Shared lib)	MassiveThreads(Header)	Cilk
スタック確保・解放	11.81	12.08	14.80
ランキュー操作	23.74	24.49	2.09
コンテキスト切り替え	22.65	22.55	2.03
合流処理	34.67	32.79	37.85
その他 (関数呼び出しなど)	33.70	11.86	5.57
合計	126.57	103.76	62.23

- Cilkと比較して次が10倍以上遅くなっている
  - ランキュー操作
  - コンテキスト切り替え
- 共有ライブラリ版では関数呼び出しの分だけ遅い

評価2: 負荷分散性能

UTS benchmark で優れたスケーラビリティ

- UTS: 不均衡な木のノード数を数えるベンチマーク



評価2: メモリ管理の改善効果

管理領域とスタック領域を分離 (UTS bench)

- 負荷分散のスケーラビリティが大幅に向上