The Implementation of the Cilk-5 Multithreaded Language

秋山 茂樹

全体ミーティング *(2010/6/23)*

The Implementation of the Cilk-5 Multithreaded Language

- Cilk-5: C のマルチスレッド拡張
  - spawn, sync, ...
- “work-stealing” scheduling の実装について解説
  - “work-first” principle に基づく
    - “two-clone” compilation strategy
    - ready deque のための排他制御プロトコル

Matteo Frigo, Charles E. Leiserson, Keith H. Randall,The Implementation of the Cilk-5 Multithreaded Language, PLDI’98

Outline

1. Cilk Language
2. Work-Stealing Scheduling
3. Work-First Principle
4. Cilk's Compilation Strategy
5. THE Protocol
6. Benchmarks

1. Cilk Language

- 共有メモリSMP のための C マルチスレッド拡張
- コンセプト
  - ポータビリティ: ANSI C へ変換可能
  - C による実装と遜色ない性能

1. Cilk Language

- 追加された機能
  - spawn によるスレッド生成
  - sync によるバリア同期
  - inlet, abort (略)
- Cilk キーワードを消すと(ほぼ) C コードと同じ

1. Cilk Language
例: spawn, sync

```
cilk int fib(int n) {
     if (n < 2) {
          return n;
     } else {
          int x, y;
          x = spawn fib(n-1);
          y = spawn fib(n-2);
          sync;
          return x + y;
     }
}
```

fib(n-1), fib(n-2) を
計算するスレッドを生成

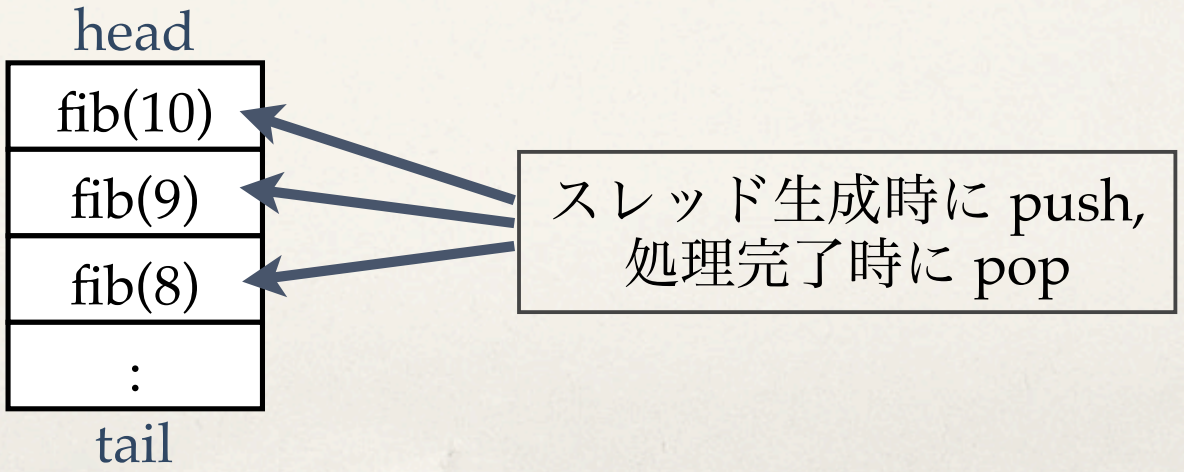
fib(n-1) と fib(n-2) の
計算が終わるのを待つ

2. Work-Stealing Scheduling 概要

- load-balancing scheduler の一種
  - プロセッサごとに ready deque を用意
  - プロセッサが idle 状態になったとき、他のプロセッサの deque からスレッドを盗む

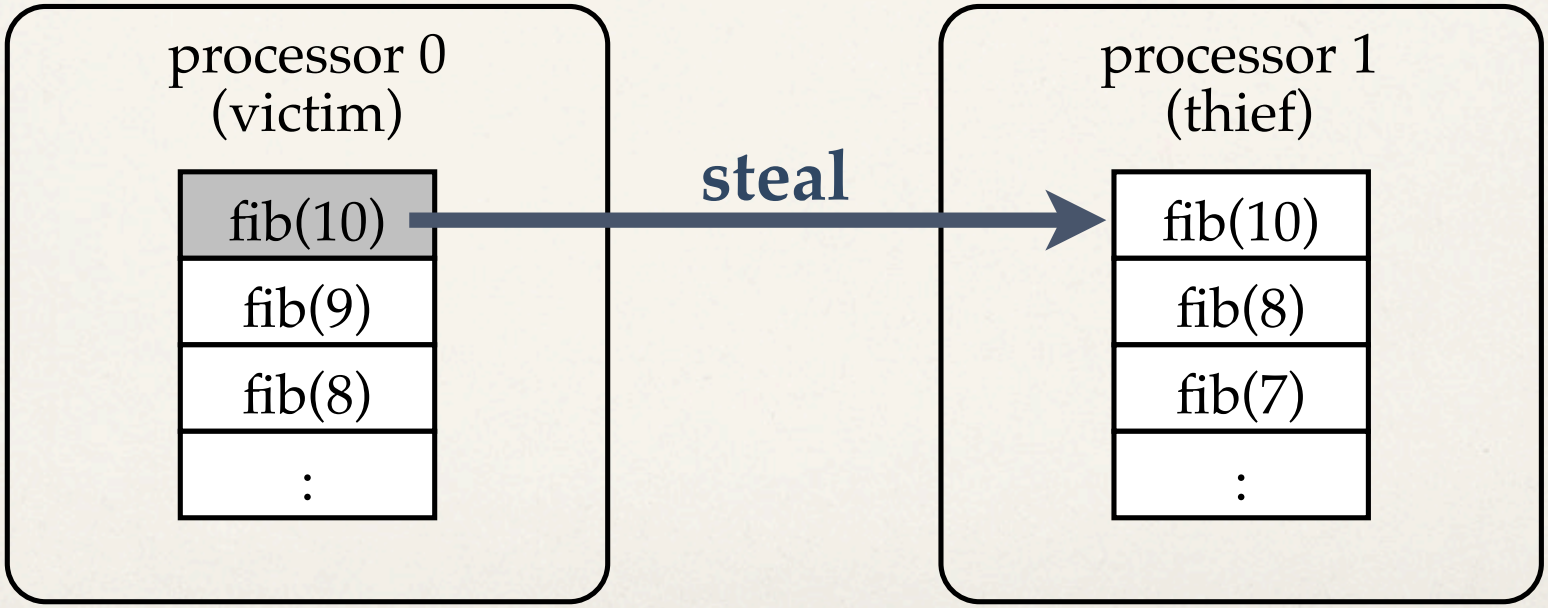
2. Work-Stealing Scheduling Ready Deque

- スレッドを管理するためのデータ構造
  - 末尾に対してフレームの push/pop が可能
  - 加えて先頭から要素を取り除く操作が可能



2. Work-Stealing Scheduling
work-stealing イメージ

- プロセッサが idle 状態になったら他のスレッドの ready deque の頭からスレッドを盗む



3. Work-First Principle 概要

- Cilk による並列計算を特徴づける数値
  - work: 1 processor での実行時間 T_1
  - critical-path length: ∞ processor での実行時間 T_∞
- work-first principle
  - プロセッサ数に対してスレッド数が十分あるなら critical-path length が大きくなったとしても work を最小化すべき

3. Work-First Principle
実行時間の下界

- Pプロセッサでの実行時間 T_P の下界
  - $T_P \geq T_1/P$
    - 最速でも 1プロセッサでの実行時間の 1/P
  - $T_P \geq T_\infty$
    - 最速でも ∞プロセッサでの実行時間

3. Work-First Principle critical-path overhead

- 論文[1] によると
  - $T_P = \underbrace{T_1/P}_{\text{work term}} + \underbrace{O(T_\infty)}_{\text{critical-path term}}$
- $T_P \leq T_1/P + c_\infty T_\infty$ なる最小の c_∞ を
  - critical-path overhead と定義する

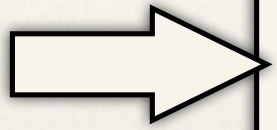
[1] Robert D. Blumofe, Charles E. Leiserson, Scheduling Multithreaded Computations by Work Stealing, FOCS'94

3. Work-First Principle parallel slackness

- average parallelism = maximum possible speedup
  - $P' = T_1 / T_\infty$
- parallel slackness
  - P' / P (1プロセッサが利用可能な並列性)
  - 並列性が十分かどうかを考えるのに使用

3. Work-First Principle

- 十分な parallel slackness があると仮定する
  - $P'/P >> c_\infty$
- $P' = T_1/T_\infty$ から
  - $T_1/P >> c_\infty T_\infty$
- $T_P \leq T_1/P + c_\infty T_\infty$ から
  - $T_P \approx T_1/P$



十分な parallel slackness を
仮定すれば
critical-path overhead が
性能に与える影響は小さい

3. Work-First Principle work overhead

- work overhead を $c_1 = T_1/T_S$ と定義すると
  - $T_P \leq c_1 T_S/P + c_\infty T_\infty$
- よって
  - $T_P \approx c_1 T_S/P$

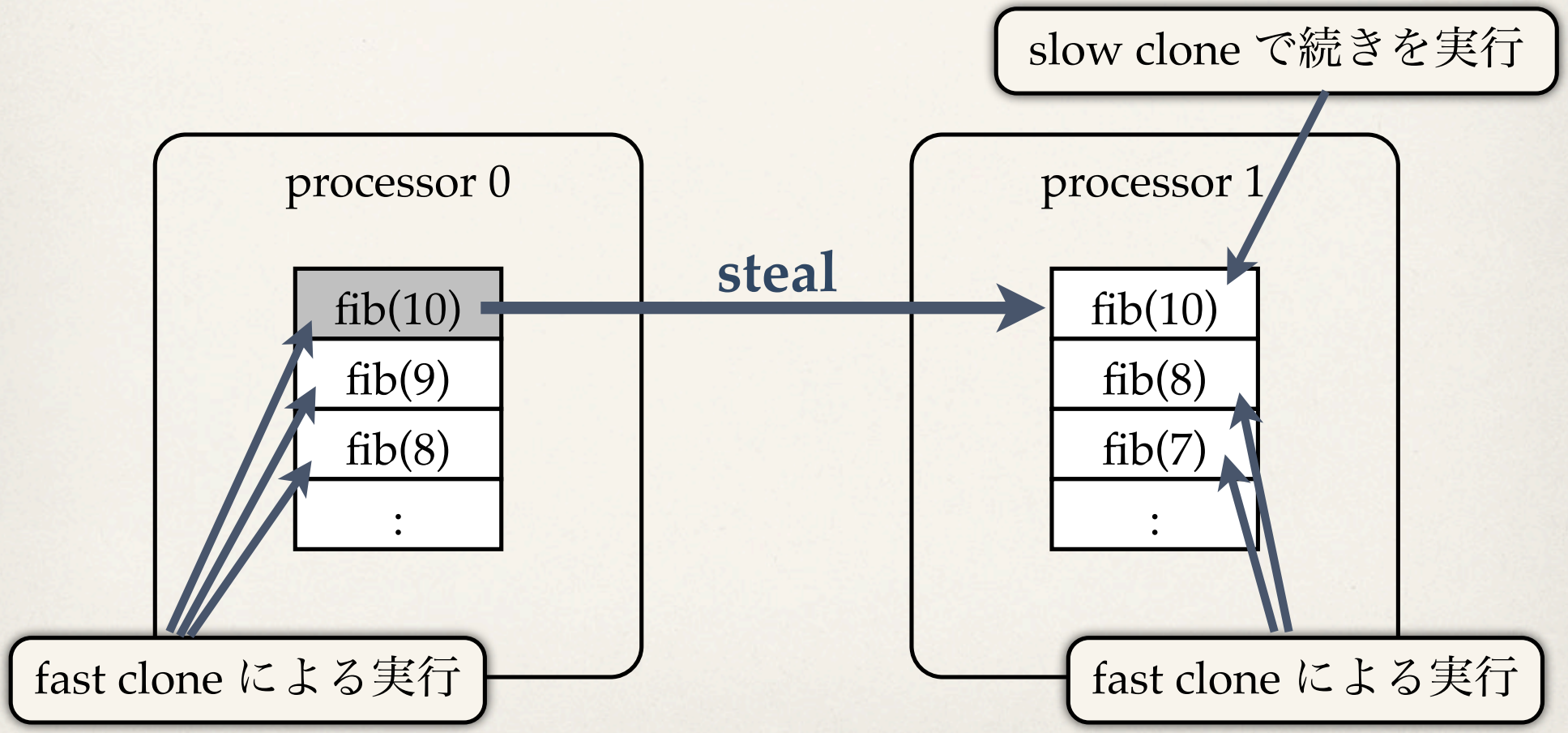
work-first principle

c_∞が大きくなっても c_1 を最小化すべき

4. Cilk's Compilation Strategy

- Cilk コードをどのように C コードに変換するか
  - "two-clone" strategy
    - work-first principle に基づく実装戦略
    - 通常は fast clone (逐次実装)で実行
    - thread migration 時に slow clone (並列実装) でスレッドの続きを実行
      - fast clone + スレッド再開 + sync サポート

4. Cilk's Compilation Strategy "two clone" strategy イメージ



4. Cilk's Compilation Strategy fast clone の概要

- fast clone: deque にスレッドを追加しつつ逐次実行
  - spawn 前: スレッドの状態を保存 (suspend)
    - 状態: PC, live variables
  - spawn: 単なる関数呼び出し
  - spawn 後: resume するスレッドが steal されていないか確認
  - sync は不要

4. Cilk's Compilation Strategy fast clone の実装

```
int fib(int n) {
    fib_frame *f = alloc(sizeof(*f));
    if (n < 2) { free(f); return n; } else {
        f->entry = 1; f->n = n;
        *T = f;
        push();
        int x = fib(n-1);
        if (pop() == FAIL) return DUMMY;
        ... (int y = spawn fib(n-2)) ...
        ; // sync
        free(f); return x + y;
    }
}
```

4. Cilk's Compilation Strategy slow clone

- slow clone: steal したスレッドの続きを実行
  - 状態を復元し、処理を再開
  - spawn は fast clone と同様
  - sync では spawn したすべてのスレッドが終了しているかどうか確認

4. Cilk's Compilation Strategy sl

```
int fib_slow(fib_frame *f) {
    if (f->entry == 1) goto ENTRY1;
    if (f->entry == 2) goto ENTRY2;
ENTRY1:
        int n = f->n;    // restore live var
        f->entry = 2;
        f->n = n;
        *T = f;
        push();
        int y = fib(n-2);
        if (pop() == FAILURE) return DUMMY;
ENTRY2:
        ... sync 処理 ...
        free(f); return x + y;
    }
}
```

※ 論文からの推測です

5. THE Protocol
deque へのアクセスの競合

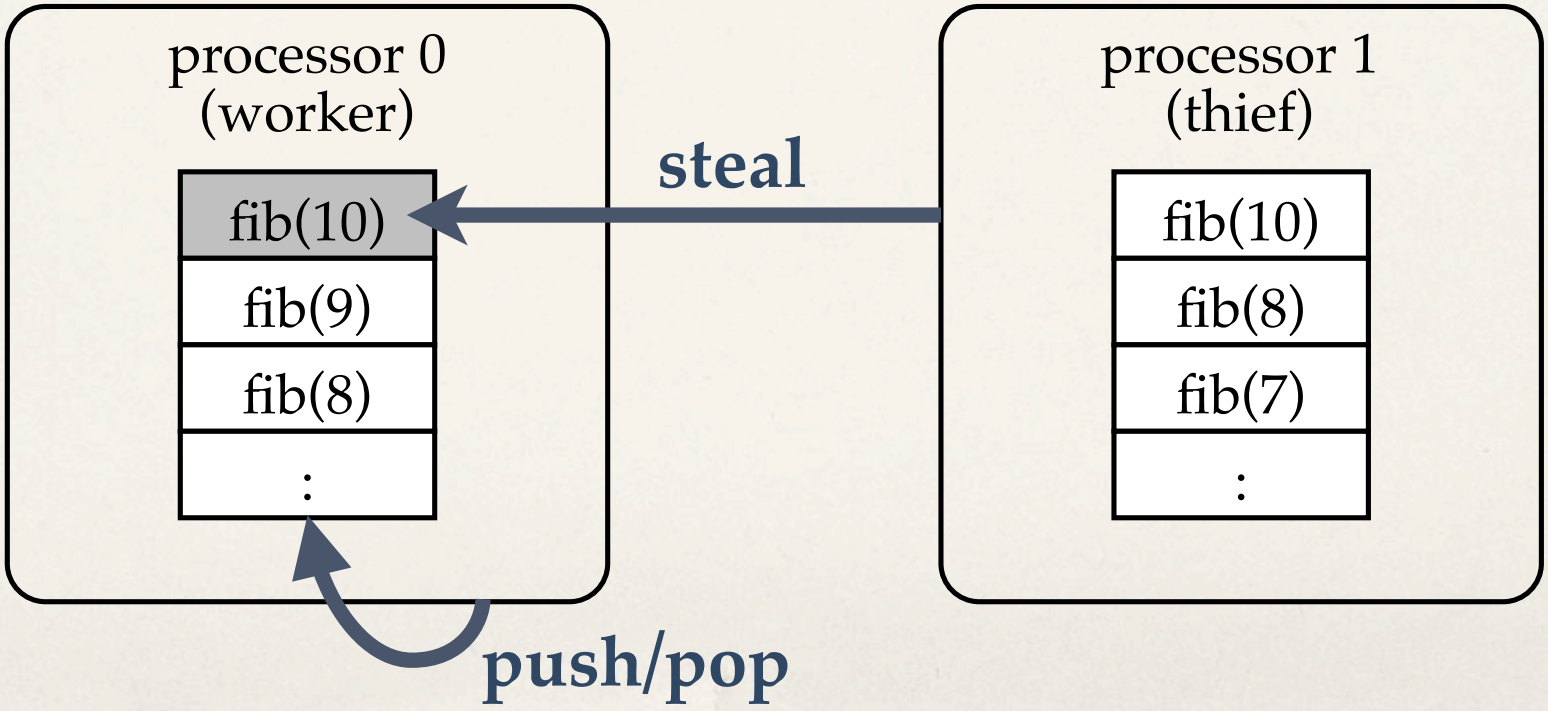
- victim による pop と thief による steal が同じフレームに対して行なわれたとき race condition になりうる
  - ready deque に対する排他制御が必要

5. THE protocol

- ***THE*** protocol:
  - deque のための共有メモリ排他制御プロトコル
  - work overhead を最小化するように設計
  - T: TAIL index
  - H: HEAD index
  - E: Exception

5. THE protocol
ready deque へのアクセス

- thief は ready deque の先頭しか触らない
- worker は ready deque の末尾しか触らない



5. THE protocol
方針

- コストを worker から thief へ移転
  - worker が pop するのは work overhead
  - thief が steal するのは critical-path overhead
- 方法
  - worker の pop では lightweight protocol
  - thief の steal では heavyweight hardware lock

5. THE protocol Simplified version (T, H のみ)

```
push() { T++; }
pop() {
  T--;
  if (H > T) {
    T++;
    lock(L);
    T--;
    if (H > T) {
      T++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

```
steal() {
  lock(L);
  H++;
  if (H > T) {
    H--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

5. THE protocol Exception を追加

```
push() { T++; }
pop() {
  T--;
  if (E > T) {
    if (E = ∞)
      (exception handler)
    T++;
    lock(L);
    T--;
    if (E > T) {
      T++; unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

```
steal() {
  lock(L);
  E++;
  if (E > T) {
    E--;
    unlock(L);
    return FAILURE;
  }
  H++;
  unlock(L);
  return SUCCESS;
}
```

例外状態になったら E は ∞ に設定される

6. Benchmarks
実験環境

- Sun Enterprise 5000 SMP
  - 167 MHz UltraSPARC
  - 8 processor
  - L1 cache: 16KB, L2 cache: 512KB
  - Solaris 2.5
  - gcc 2.7.2 (-O3)

6. Benchmarks

Program	*Size*	T_1	T_∞	$\overline{P}$	c_1	T_8	T_1/T_8	T_S/T_8
`fib`	35	12.77	0.0005	25540	3.63	1.60	8.0	2.2
`blockedmul`	1024	29.9	0.0044	6730	1.05	4.3	7.0	6.6
`notempmul`	1024	29.7	0.015	1970	1.05	3.9	7.6	7.2
`strassen`	1024	20.2	0.58	35	1.01	3.54	5.7	5.6
*`cilksort`	$4,100,000$	5.4	0.0049	1108	1.21	0.90	6.0	5.0
†`queens`	22	150.	0.0015	96898	0.99	18.8	8.0	8.0
†`knapsack`	30	75.8	0.0014	54143	1.03	9.5	8.0	7.7
`lu`	2048	155.8	0.42	370	1.02	20.3	7.7	7.5
*`cholesky`	BCSSTK32	1427.	3.4	420	1.25	208.	6.9	5.5
`heat`	4096×512	62.3	0.16	384	1.08	9.4	6.6	6.1
`fft`	2^{20}	4.3	0.0020	2145	0.93	0.77	5.6	6.0
`Barnes-Hut`	2^{16}	124.	0.15	853	1.02	16.5	7.5	7.4

6. Benchmarks
average parallelism: $P' = T_1 / T_\infty$

- プロセッサ数8 で割っても十分な値になっている

Program	*Size*	T_1	T_∞	$\overline{P}$	c_1	T_8	T_1/T_8	T_S/T_8
fib	35	12.77	0.0005	25540	3.63	1.60	8.0	2.2
blockedmul	1024	29.9	0.0044	6730	1.05	4.3	7.0	6.6
notempmul	1024	29.7	0.015	1970	1.05	3.9	7.6	7.2
strassen	1024	20.2	0.58	35	1.01	3.54	5.7	5.6
*cilksort	4,100,000	5.4	0.0049	1108	1.21	0.90	6.0	5.0
†queens	22	150.	0.0015	96898	0.99	18.8	8.0	8.0
†knapsack	30	75.8	0.0014	54143	1.03	9.5	8.0	7.7
lu	2048	155.8	0.42	370	1.02	20.3	7.7	7.5
*cholesky	BCSSTK32	1427.	3.4	420	1.25	208.	6.9	5.5
heat	4096×512	62.3	0.16	384	1.08	9.4	6.6	6.1
fft	2^{20}	4.3	0.0020	2145	0.93	0.77	5.6	6.0
Barnes-Hut	2^{16}	124.	0.15	853	1.02	16.5	7.5	7.4

6. Benchmarks
work overhead: $c_1 = T_1 / T_S$

- ほとんどの場合で数%程度の増加

Program	*Size*	T_1	T_∞	$\overline{P}$	c_1	T_8	T_1/T_8	T_S/T_8
fib	35	12.77	0.0005	25540	3.63	1.60	8.0	2.2
blockedmul	1024	29.9	0.0044	6730	1.05	4.3	7.0	6.6
notempmul	1024	29.7	0.015	1970	1.05	3.9	7.6	7.2
strassen	1024	20.2	0.58	35	1.01	3.54	5.7	5.6
*cilksort	$4,100,000$	5.4	0.0049	1108	1.21	0.90	6.0	5.0
†queens	22	150.	0.0015	96898	0.99	18.8	8.0	8.0
†knapsack	30	75.8	0.0014	54143	1.03	9.5	8.0	7.7
lu	2048	155.8	0.42	370	1.02	20.3	7.7	7.5
*cholesky	BCSSTK32	1427.	3.4	420	1.25	208.	6.9	5.5
heat	4096×512	62.3	0.16	384	1.08	9.4	6.6	6.1
fft	2^{20}	4.3	0.0020	2145	0.93	0.77	5.6	6.0
Barnes-Hut	2^{16}	124.	0.15	853	1.02	16.5	7.5	7.4

6. Benchmarks
work overhead の詳細 (fib)