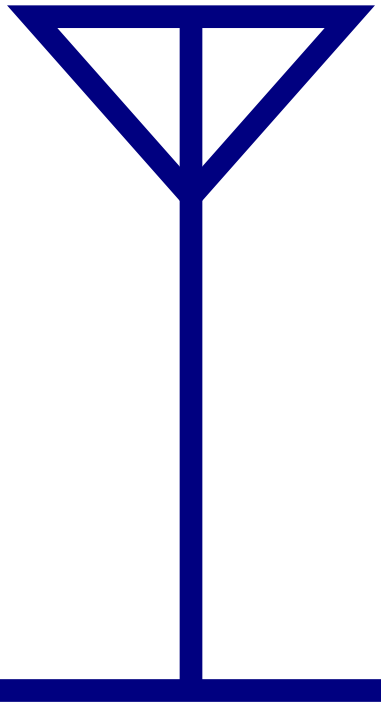
Journal of Hamradio Informatics No.5

D言語で実装する並列スケジューラ入門

Parallel Work Stealing Scheduler on D



無線部開発班　**平成 29 年 4 月 3 日改訂**

http://pafelog.net

目次

第1章　並列スケジューラの概念

第 1 章では、ワークスティーリング方式のタスク並列スケジューラを自作するために必要な知識を紹介する。

1.1　並列処理の記述

並列処理とは、多大な時間を要する処理を、複数の計算装置に分配することで、高速に実行する技術である。並列化の方法は、Algorithm 1 に示す**データ並列**と、Algorithm 2 に示す**タスク並列**の 2 種類が考えられる。

Algorithm 1 data parallelism.

procedure multiply(matrix A, B, C)
  for parallel i **do**
    for parallel j **do**
      for parallel k **do**
        c_{ij} += $a_{ik}b_{kj}$
      end for
    end for
  end for
end procedure

Algorithm 2 task parallelism.

procedure fibonacci(integer n)
  if $n > 1$ **then**
    $t_1 = \text{fork}(\text{fibonacci}(n-1))$
    $t_2 = \text{fork}(\text{fibonacci}(n-2))$
    return $\text{join}(t_1) + \text{join}(t_2)$
  else
    return n
  end if
end procedure

データ並列では、`for` 文のイテレーションを分割し、計算装置に分配して並列化するか、SIMD 命令により、複数の値に対する演算を同時に実行する。タスク並列では、`fork` 関数を呼ぶと指定の処理が並列実行され、結果は `join` 関数により回収する。データ並列とタスク並列は、それぞれ写像と再帰関数で定式化できるが、両者は真っ向から対立する概念というわけではなく、何れかが他方の糖衣構文として実装される場合もある。

1.2　並列処理の実行

タスク並列では、処理は任意個の単位に分割され、スケジューラによりプロセッサに分配され、実行される。分割された処理単位をタスクと呼ぶ。Fig. 1.1 に示す通り、FIFO に分配するスケジューラが普及している。

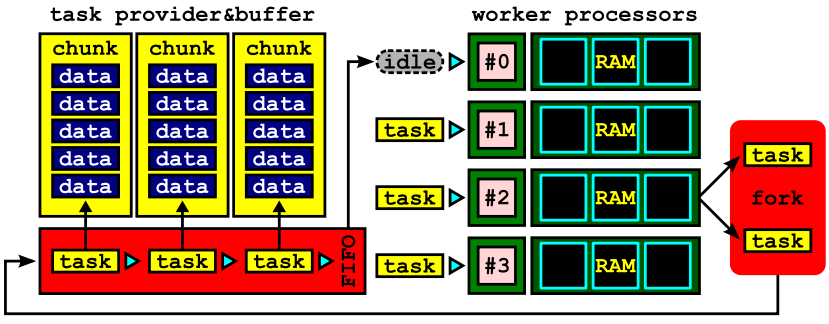


Fig. 1.1: FIFO scheduler.

D 言語の `std.parallelism`[1] も同様で、ワーカスレッドはキューから `Task` を先着順に取り出して実行する。負荷の不均衡（ロードインバランス）を解消できるし、実装も単純であるが、任意の入れ子構造を持つタスクの並列化には適さない。例えば、グラフ構造を再帰的に辿るような処理では、タスクの無闇な細分化を招き、参照の局所性を損ねる。

1.3 遅延タスク分岐

任意の入れ子構造を持つタスクを分配する際は、タスクの木構造を葉ではなく幹の部分で分割すべきである。幹の部分、即ち木構造の根に近い階層で分割することにより、大きなタスクの塊をプロセッサに割り当てる。Fig. 1.2 の遅延タスク分岐は、第 1.2 節で述べた問題点を解消する。

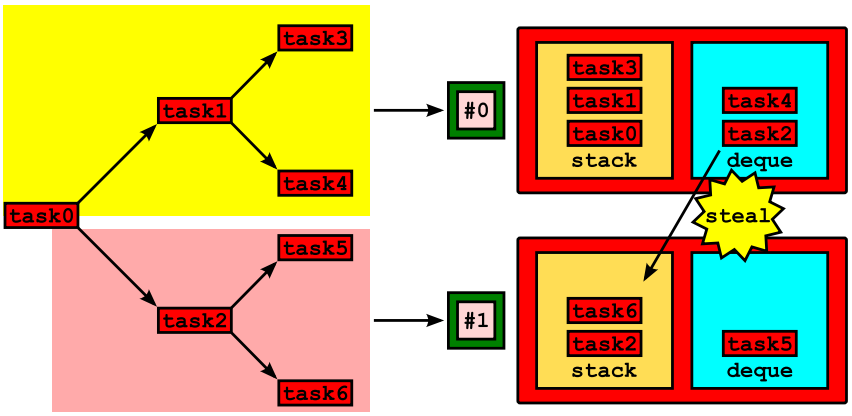


Fig. 1.2: work-stealing scheduler.

プロセッサは個別に両端キューを有し、各プロセッサで新たに生成されたタスクが古い順に保管されている。両端キューが空にならない限り、各プロセッサはタスクを FILO に取り出し逐次処理と同じ順番で実行する。有閑状態（アイドル）のプロセッサは、他のプロセッサの残存タスクを奪って実行する。これをワークスティールと呼ぶ。ワークスティール時のみ、タスクは FIFO に取り出され、木構造の根に近い階層で再分配されることになる。逐次処理において高い局所性を持つプログラムなら、並列化しても高い局所性を維持することが期待できる。

[1] `http://dlang.org/phobos/std_parallelism.html`

第2章　並列スケジューラの実装

第 1.3 節に述べた遅延タスク分岐による並列スケジューラへの理解を深めるため、D 言語で自作してみよう。

wing.d
```
module wing;
```

ソースファイルの名前は wing.d とし、モジュール宣言を行う。次に、必要なモジュールをインポートする。

wing.d
```
import std.parallelism;
import std.range;
```

次に、各プロセッサに一意に割り振られた ID を coreId と名付け、スレッド局所記憶として変数宣言する。

wing.d
```
private uint coreId = 0;
```

2.1　スケジューラ本体

続いて Wing クラスを宣言する。Ret と Args はテンプレート引数で、タスクの返り値と引数の型を表現する。

wing.d
```
public shared class Wing(Ret, Args...) {
  alias Ret function(Args) Func;
  private shared Deque[] stacks;
```

Wing クラスこそ、第 2 章で実装する並列スケジューラの本体である。続いて、コンストラクタを定義する。

wing.d
```
  this(uint numCores = totalCPUs) {
    foreach(id; iota(numCores)) {
      stacks ~= cast(shared) new Deque;
    }
  }
```

配列 stacks には、両端キューを実装する Deque のインスタンスが、論理プロセッサの個数だけ格納される。また、プロセッサが自分の Deque を参照する際の便宜のため、下記に示す通り stack メソッドを用意する。

wing.d
```
  auto stack(uint id = coreId) {
    return cast() stacks[id % numCores];
  }
```

スケジューラが管理する論理プロセッサ数を返す numCores メソッドも定義する。

wing.d

```
  uint numCores() {
    return cast(uint) stacks.length;
  }
```

2.2 並列タスクの実装

続けて Task クラスを宣言する。これは Wing の内部クラスであり、第 2.1 節のプログラムに続けて記述する。

wing.d

```
  private static final class Task {
    private bool done;
    private Func func;
    private Args args;
    private Ret value;
```

タスクが実行する関数とその引数はコンストラクタで受け取る。

wing.d

```
    this(Func func, Args args) {
      this.func = func;
      this.args = args;
      this.done = false;
    }
```

done はタスクの完了を保持するためのフィールドで、初期値は false である。isDone メソッドで取り出す。

wing.d

```
    public bool isDone() {
      return done;
    }
```

invoke メソッドは関数 func を呼び出し、完了すると返り値を value に代入し、done に true を代入する。

wing.d

```
    public void invoke() {
      value = func(args);
      done = true;
    }
```

タスクの返り値は、下記の result メソッドで取り出すことができる。

wing.d

```
    public auto result() {
      return value;
    }
  } // end of class Task
```

以上で、Task クラスの実装が完了した。

2.3 両端キューの実装

両端キューを表す Deque クラスを宣言する。Wing の内部クラスであり、第 2.1 節のプログラムに追記する。

wing.d

```
  private static final class Deque {
    private Task[] data;
```

動的配列 data で Task のインスタンスを管理する。add メソッドは、タスクを配列 data の末尾に追加する。

wing.d

```
    public Task add(Task task) {
      synchronized(this) {
        data ~= cast() task;
        return task;
      }
    }
```

続いて、プロセッサが自身の両端キューからタスクを FILO に取り出す際に用いる pop メソッドを定義する。

wing.d

```
    public Task pop() {
      synchronized(this) {
        if(!data.empty) {
          Task task = data.back;
          data.popBack;
          return task;
        } else return null;
      }
    }
```

poll メソッドは、有閑(アイドル)状態のプロセッサが他のプロセッサのタスクを FIFO に取り出すための関数である。

wing.d

```
    public Task poll() {
      synchronized(this) {
        if(!data.empty) {
          Task task = data.front;
          data.popFront;
          return task;
        } else return null;
      }
    }
  } // end of class Deque
```

両端キューの操作は、プロセッサ間で競合する可能性があるため、ミューテックスを用いて競合を防止する。排他制御の時間を D とし、待ち行列理論に従えば、待ち時間 w は式 (2.1) となる。

$$w = \left\{ N - 1 - \frac{1}{\rho b_{N-1}} \left(\sum_{k=0}^{N-1} b_k - N \right) \right\} D, \tag{2.1}$$

$$b_n = \sum_{k=0}^{n} \frac{(-1)^k}{k!} (n-k)^k e^{(n-k)\rho} \rho^k, \quad \rho = \lambda D. \tag{2.2}$$

遅延タスク分岐では、プロセッサが増加しても到着率 λ が抑制され、両端キューでの待ち時間は無視できる。

2.4 タスクの分岐合流

Wing クラスに、下記のメソッドを追記する。fork メソッドは、関数 func と引数 args で Task を生成する。

wing.d

```
  public auto fork(Func func, Args args) {
    return stack.add(new Task(func, args));
  }
```

join メソッドは、引数に指定された Task の実行が完了するまで、他のタスクを実行しつつ、繁忙待機（スピンウェイト）する。

wing.d

```
  public auto join(Task target) {
    while(!target.isDone) {
      if (!local) steal;
    }
    return target.result;
  }
```

join メソッドが参照する steal メソッドは、他のプロセッサを巡察し、タスクを FIFO に奪って実行する。

wing.d

```
  private auto steal() {
    foreach(i; iota(1, numCores)) {
      uint id = coreId + i;
      auto stolen = stack(id).poll;
      bool failed = stolen is null;
      if(failed) continue;
      return stolen.invoke;
    }
  }
```

join メソッドが参照する local メソッドは、自分の両端キュー内のタスクを FILO に取り出して実行する。

wing.d

```
  private bool local() {
    auto popped = stack.pop;
    if(popped is null) return false;
    else return popped.invoke, true;
  }
```

最後に、スケジューラのエントリーポイントとなる dawn メソッドを定義して、スケジューラは完成である。

wing.d

```
  public auto dawn(Func func, Args args) {
    auto root = fork(func, args);
    auto cpus = iota(numCores);
    auto pool = new TaskPool(numCores);
    foreach(c; pool.parallel(cpus, 1)) {
      coreId = c;
      join(root);
    }
    pool.finish;
    return root.result;
  }
} // end of class Wing
```

第3章　並列スケジューラの評価

第2節で実装した並列スケジューラの台数効果（スケーラビリティ）を確認するため、密行列積の計算時間を測定することにした。

dmm.d

```
import core.atomic;
import std.algorithm;
import std.datetime;
import std.stdio;
import wing;
```

まず、スケジューラを格納する shared 変数を宣言する。

dmm.d

```
shared auto sched = new Wing!(int, int, int, int, int, int, int)(8);
```

密行列積は $C = A\ {}^tB$ の形式とし、正方行列 A, B, C とその行数 N を変数宣言する。

dmm.d

```
const int N = 8192;
shared double [] A = new double[N * N];
shared double [] B = new double[N * N];
shared double [] C = new double[N * N];
```

行列積は、分割統治法により再帰的にタスクと空間を分岐して並列化する。これを dmm 関数として定義する。

dmm.d

```
int dmm(int i1, int i2, int j1, int j2, int k1, int k2) {
  int longest = max(i2 - i1, j2 - j1, k2 - k1);
  if(longest <= 128) {
    dmm_leaf(i1, i2, j1, j2, k1, k2);
  } else if(longest == i2 - i1) {
    int Im = (i1 + i2) / 2;
    auto t = sched.fork(&dmm, i1, Im, j1, j2, k1, k2);
    return dmm(Im, i2, j1, j2, k1, k2), sched.join(t);
  } else if(longest == j2 - j1) {
    int Jm = (j1 + j2) / 2;
    auto t = sched.fork(&dmm, i1, i2, j1, Jm, k1, k2);
    return dmm(i1, i2, Jm, j2, k1, k2), sched.join(t);
  } else if(longest == k2 - k1) {
    int Km = (k1 + k2) / 2;
    auto t = sched.fork(&dmm, i1, i2, j1, j2, k1, Km);
    return dmm(i1, i2, j1, j2, Km, k2), sched.join(t);
  }
  return 0;
}
```

最長軸を分割するのは、各階層の部分行列を正方行列に近付けてキャッシュヒット率を改善する意図である。
続いて、分割統治法の末端部において、$128 \times 128 \times 128$ の部分行列積を計算する dmm_leaf 関数を実装する。

dmm.d

```
void dmm_leaf(int i1, int i2, int j1, int j2, int k1, int k2) {
  for(int i = i1; i < i2; i++) {
  for(int j = j1; j < j2; j++) {
    const int iN = i * N;
    const int jN = j * N;
    double cij = 0.0;
    for(int k = k1; k < k2; k++) cij += A[iN + k] * B[jN + k];
    C[iN + j].atomicOp!"+="(cij);
  }
  }
}
```

時間測定プログラムの main 関数は、概ね下記の通りになる。行列 A, B も適当な実数で忘れずに初期化する。

dmm.d

```
void main() {
  A[0..$] = 0.1;
  B[0..$] = 0.1;
  auto sw = StopWatch(AutoStart.yes);
  sched.dawn(&dmm, 0, N, 0, N, 0, N);
  sw.stop();
  real sec = sw.peek().to!("seconds", real);
  writeln(2e-9 * N * N * N / sec, "GFLOPS");
}
```

Fig. 3.1 は、$N = 8192$ として Intel Xeon®E5-2699 v3 を 2 個搭載した共有メモリ環境で得たグラフである。std.parallelism.TaskPool で同様に分割統治法による並列化を行った場合に比べ、50%以上も優位である。

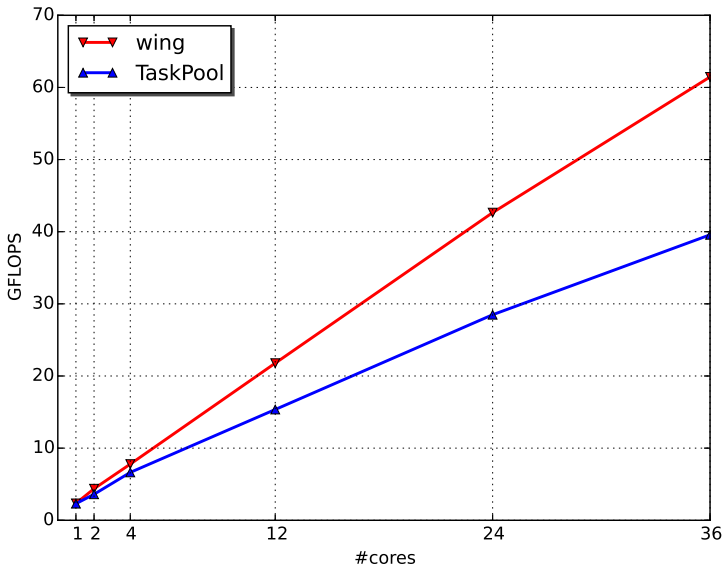


Fig. 3.1: dense matrix multiplication, $8192 \times 8192 \times 8192$, loop unrolled by 4

前掲のプログラムでは単純化のため省略したが、Fig. 3.1 の測定では、複数の最適化技法を盛り込んである。簡単で費用対効果の高い技法としては、4 倍のループ展開、並びに配列 A, B, C のパディングが挙げられる。単純な計算を行う for 文では、ループ展開は命令レベルの並列化を阻害するデータハザードの解消に役立つ。パディングとは、多次元配列の各行の末端にダミー領域を設け、各行をメモリ空間上で分離する技法である。プロセッサ間でのメモリのフォルスシェアリングを予防し、キャッシュコヒーレンシによる律速を回避する。

第4章　共有メモリ用の製品紹介

dusk は *nonuniform memory access* 環境に C++11 で実装された *lightweight workstealing scheduler* である。

4.1　利用方法

dusk は http://github.com にて BSD 3 条項ライセンスで頒布されており、下記のコマンドで導入できる。

```
$ git clone https://github.com/nextzlog/dusk
# make build install -C dusk
# ldconfig
```

dusk を自分の C++プログラムで利用する際は、dusk.hpp を下記の書式によりインクルードする必要がある。

dusk.hpp
```
#include <dusk.hpp>
```

sun::launch 関数は、所定の関数 pad を起点に並列処理を開始する。この間、スケジューラは稼働し続ける。

dusk.hpp
```
void sun::launch(void(*pad)(void));
```

sun::salvo 関数は、2 個のタスクを生成し、所定の関数 f 及び引数 a と b を与えて並列実行し、待機する。

dusk.hpp
```
template<typename Arg> void sun::salvo(void (*f)(Arg), Arg a, Arg b);
```

sun::burst 関数は、0 から round-1 までの通し番号を引数に、所定の関数 body を並列実行し、待機する。

dusk.hpp
```
template<typename Idx> void sun::burst(Idx round, void (*body)(Idx));
```

dusk は、特定のコンパイラに依存しない。プログラムをビルドする際は、必ず libdusk.so をリンクする。

```
$ g++ -ldusk -std=c++11 your_program.cpp
```

4.2　環境変数

dusk では、下記の環境変数により、論理プロセッサ数や負荷分散の戦術、両端キューの容量を変更できる。

```
$ export DUSK_WORKER_NUM=80
$ export DUSK_TACTICS=PDRWS
$ export DUSK_STACK_SIZE=64
```

DUSK_WORKER_NUM を POSIX スレッドが認識するプロセッサ数を超えて設定した場合の動作は未定義である。

4.3 負荷分散

環境変数 DUSK_TACTICS に設定可能な値と、その意味を Table 4.1 に示す。デフォルトの値は PDRWS である。

Table 4.1: DUSK_TACTICS

設定値	負荷分散	初期タスクの分配
QUEUE	FIFO	キューから全プロセッサに分配
PDRWS	遅延タスク分岐	繁忙状態のプロセッサから奪取
ADRWS	遅延タスク分岐	幅優先的に全プロセッサに分配

細粒度の分割統治では、QUEUE よりも PDRWS が優位である。ADRWS の PDRWS に対する優位性は些少である。

4.4 内部実装

第 4.1 節で取り上げた sun::launch 関数、sun::salvo 関数、sun::burst 関数は、下記の関数を隠蔽する。

dusk/main/internal/dehcs.hpp

```
void* sun::root(void*(*function)(void*), void* argument);
Task* sun::dawn(void*(*function)(void*), void* argument);
void* sun::dusk(Task* join_with);
```

sun::dawn 関数は、タスクを生成する。sun::dusk 関数は、所定のタスクが完了するまで、繁忙待機（スピンウェイト）を行う。

4.5 性能測定

Fig. 4.1 は、Intel Xeon®E5-2699 v3 を 2 個搭載した共有メモリ環境で密行列積を計算した際の性能である。PDRWS は第 3 章と同様に、3 軸を粒度 128 まで交互に分割して並列化し、AVX で SIMD 化した結果である。

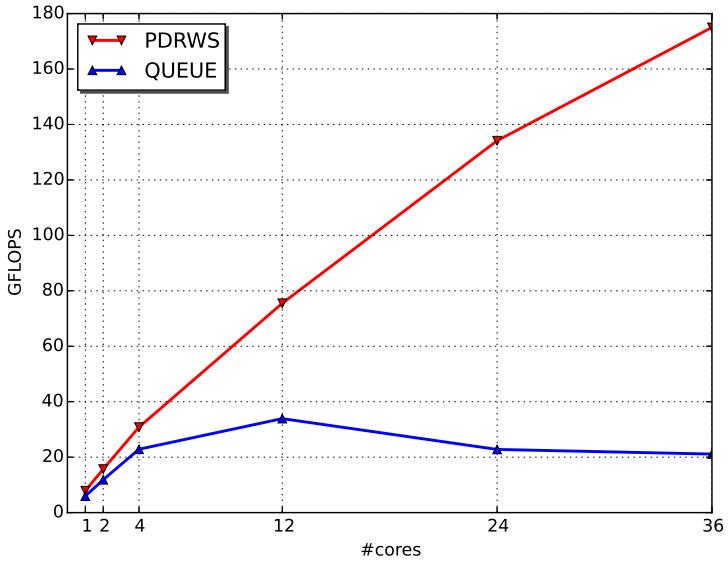


Fig. 4.1: dense matrix multiplication, 8192 × 8192 × 8192, vectorized by AVX.

QUEUE は第 3 章と違い、2 軸を粒度 128 まで格子状に分割して並列化し、AVX で SIMD 化した結果である。