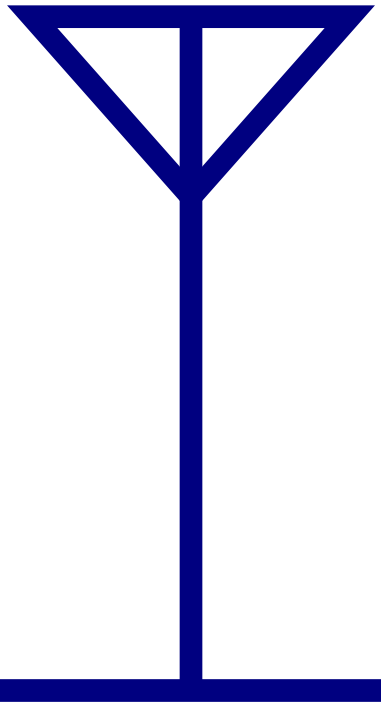
Journal of Hamradio Informatics No.5

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　**2019年7月25日改訂**

http://pafelog.net

目次

第1章　並列スケジューラの概念

並列処理とは、長時間かかる処理を複数のプロセッサで分担することで、処理速度の向上を図る技術である。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(matrices 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

なお、両者は対立する概念ではなく、例えばデータ並列がタスク並列の糖衣構文として実装される例がある。また、これらの並列性とは別に、プロセッサでは**命令レベルの並列性**による逐次処理の高速化が実施される。

1.1　データレベルの並列性

配列を分配しての並行処理による並列処理をデータ並列処理と呼び、特に配列から配列への写像が該当する。具体的には、OpenMP の並列 `for` 文や、CPU の *single instruction, multiple data* (SIMD) 命令を利用する。

simd.d

```
import core.simd;
import std.range;
import std.parallelism: parallel;

void dmm_simd(const size_t RANK, double *A, double *B, double *C) {
  foreach(i; parallel(iota(RANK), 32)) {
  foreach(j; parallel(iota(RANK), 32)) {
    double2 u = 0;
    foreach(k; iota(0, RANK, 2)) {
      auto w = simd!(XMM.LODUPD)(*cast(double2*) &A[i * RANK + k]);
      auto x = simd!(XMM.LODUPD)(*cast(double2*) &B[j * RANK + k]);
      w = simd!(XMM.MULPD, double2)(w, x);
      u = simd!(XMM.ADDPD, double2)(w, u);
    }
    C[i * RANK + j] = u[0] + u[1];
  }
  }
}
```

上記の例では OpenMP の代わりに D 言語に付属する `std.parallelism` を利用し、SIMD 命令を併用した。

1.2 タスクレベルの並列性

関数など処理単位の非同期実行による並列処理をタスク並列処理と呼び、分岐した処理単位を**タスク**と呼ぶ。キューを利用してタスクの実行を制御する仕組みを**スケジューラ**と呼び、特に**負荷の不均衡**の解消に役立つ。ここで注意すべきは、単に負荷を均等にするだけでなく、データの配置にも注意を払う必要がある点である。

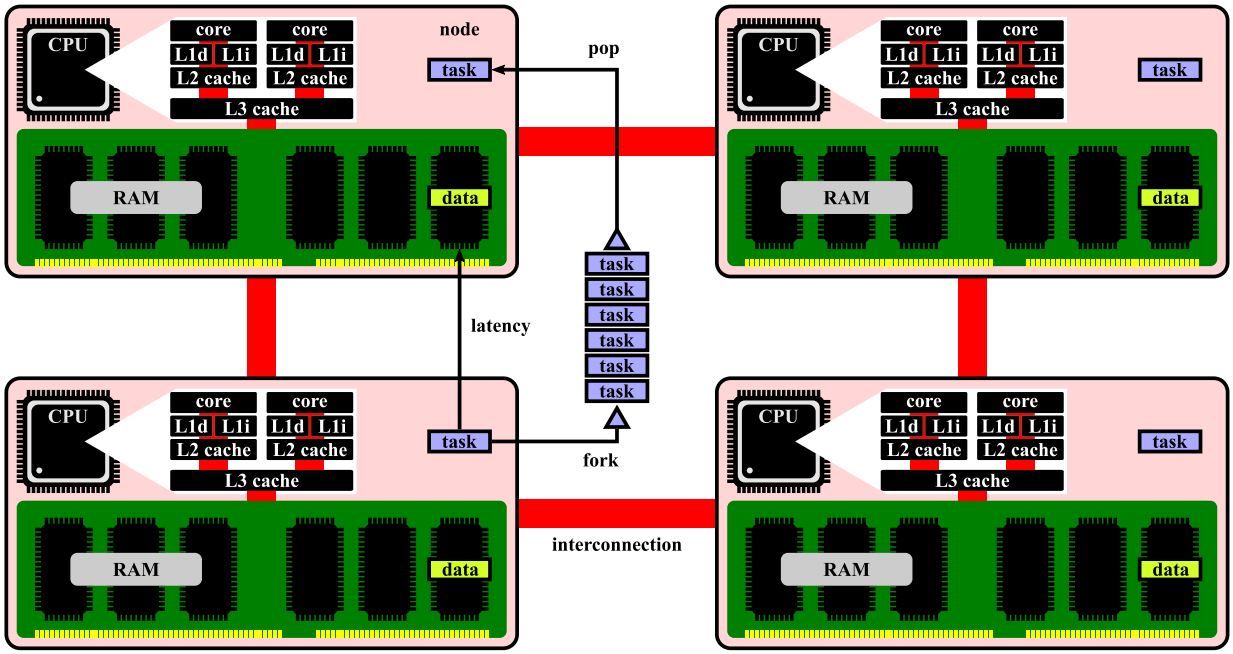


Fig. 1.1: FIFO scheduler on shared-memory architecture.

通常、並列計算機はプロセッサと主記憶が対をなす**ノード**の集合体であり、非対称な**共有メモリ**を構成する。他のノードのデータを参照すると効率が落ちる。故に、プロセッサは可能な限り参照を局所化すべきである。だが、FIFO 型の実装では、プロセッサに渡されるタスクが断片化されるため、局所性が大幅に損なわれる。

1.3 ワークスティーリング

タスクの断片化を抑制するには、再帰構造を持つタスクの末端部ではなく、粗粒度な塊を分配すべきである。故にこそ Fig. 1.2 に示す FILO 型の実装が役に立つ。まず、プロセッサはそれぞれ独自に両端キューを持つ。

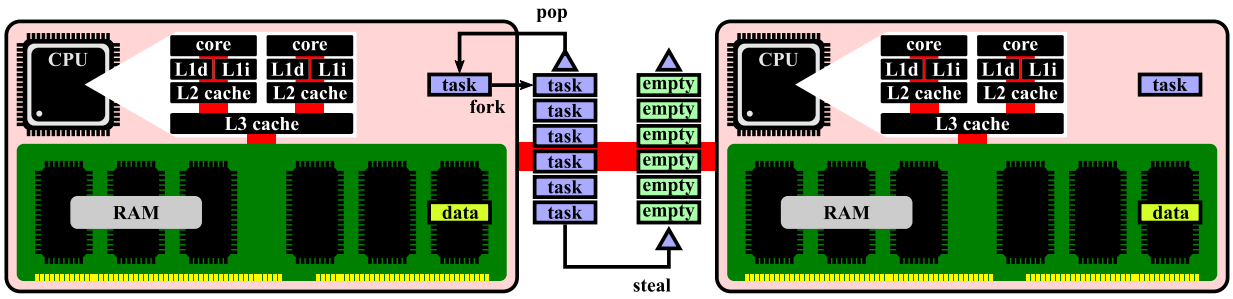


Fig. 1.2: work-stealing scheduler on shared-memory architecture.

プロセッサは自身が保有するタスクを FILO に実行し、両端キューが空になった時だけ他からタスクを奪う。奪う際は FIFO にタスクを選ぶことで、なるべく大きな塊を持ち去る。これを**ワークスティーリング**と呼ぶ。高度な実装では、特に PGAS 等の**分散メモリ**で、タスクの移動と同時にデータの再配置を行う場合がある。

第2章　並列スケジューラの実装

第 1.3 節に述べたワークスティーリングを D 言語で実装する。なお、第 4 章では C++での実装も紹介する。以下、dawn モジュールに実装する。なお、掲載するプログラム断片を順番に結合すると完全な実装になる。

dawn.d

```
module dawn;
import core.atomic, std.range: iota;
import std.parallelism: TaskPool, totalCPUs;
```

まず、並列処理を分担するプロセッサに識別番号を割り当てて coreId と名付け、**スレッド局所記憶**に置く。

dawn.d

```
private size_t coreId = -1;
```

2.1　スケジューラ本体

次に Dawn クラスを実装する。引数 Ret と Args は、スケジューラが扱うタスクの返り値と引数の型である。

dawn.d

```
public shared class Dawn(Ret, Args...) {
  alias Ret function(Args) Func;
  private shared Deque[] stacks;
  private const size_t numCores;
```

また、Dawn クラスのコンストラクタを定義する。並列処理を行うプロセッサの個数を引数 numCores に渡す。配列 stack は、プロセッサと同数の両端キューを格納する。この配列は他のプロセッサから参照可能にする。

dawn.d

```
  this(size_t numCores = totalCPUs) {
    this.numCores = numCores;
    foreach(i; iota(numCores)) {
      stacks ~= new shared Deque;
    }
  }
```

プロセッサが任意のプロセッサの両端キューを参照する際の便宜のため、下記の stack メソッドを用意する。

dawn.d

```
  private auto stack(size_t id = coreId) {
    return cast() stacks[id % numCores];
  }
```

以下、タスクの分岐と待ち合わせの処理を実装する。まず、タスクを分岐させる fork メソッドを実装する。

dawn.d

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

また、join メソッドは、引数で指示されたタスクが終了するまで、他のタスクを実行しつつ**繁忙待機**する。

dawn.d

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

繁忙待機で呼び出されている local メソッドは、自分が保有しているタスクを新着順に取り出して実行する。

dawn.d

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

同じく呼び出されている steal メソッドは、他のプロセッサを巡回して、タスクを FIFO に奪って実行する。

dawn.d

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

最後に、並列処理を開始して、終了するまで待機する dawn メソッドを定義して、全てのメソッドが揃った。

dawn.d

```
public auto dawn(alias func)(Args args) {
  auto root = new Task(&func, args);
  auto cpus = iota(numCores);
  auto pool = new TaskPool(numCores);
  foreach(c; pool.parallel(cpus, 1)) {
    if((coreId = c) == 0) root.invoke;
    else join(root);
  }
  pool.finish;
  return root.result;
}
```

ただし、タスクを実装する Task 構造体及び両端キューを実装する Deque クラスがこの段階で未完成である。

2.2 並列タスクの実装

続けて、Task 構造体を Dawn クラス内部に実装する。タスクは、実行予定の関数と引数と返り値を包み込む。

dawn.d

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

実行予定の関数と引数はコンストラクタで受け取る。この時、タスクの終了を表す変数 done も初期化する。

dawn.d

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

変数 done を外部から保護しつつ join メソッドにタスクの終了を通知するため isDone メソッドを定義する。

dawn.d

```
    public bool isDone() {
      return atomicLoad(*(cast(shared) &done));
    }
```

invoke メソッドは関数を呼び出した**直後**に、返り値を value に保存して、変数 done を true に切り替える。

dawn.d

```
    public void invoke() {
      value = func(args);
      atomicStore(*(cast(shared) &done), true);
    }
```

関数と代入の *out-of-order* 実行を防ぐ目的で、**メモリバリア**を使った。最後に result メソッドを定義する。

dawn.d

```
    public auto result() {
      return value;
    }
  }
```

以上で、Task 構造体の実装が完了した。この時点で、両端キューを表す Deque クラスの実装が残っている。

2.3 両端キューの実装

最後に Dawn クラスの内部に Deque クラスを定義して、実行予定のタスクを管理する両端キューを実装する。

dawn.d

```
  private static final class Deque {
    import std.container: DList;
    private DList!(Task*) queue;
```

実態は、双方向リストに**排他制御**を組み込むだけである。タスクを加えるには、下記の add メソッドを使う。

dawn.d

```
    public Task* add(Task* task) {
      synchronized(this) {
        queue.insertBack(task);
        return task;
      }
    }
```

また、プロセッサが自身の管理下にある最新のタスクを FILO に取り出す際は、下記の pop メソッドを使う。

dawn.d

```
    public Task* pop() {
      synchronized(this) {
        if(!queue.empty) {
          auto task = queue.back;
          queue.removeBack;
          return task;
        } else return null;
      }
    }
```

最後に poll メソッドを定義する。プロセッサが他のプロセッサから最古のタスクを FIFO に奪う際に使う。

dawn.d

```
    public Task* poll() {
      synchronized(this) {
        if(!queue.empty) {
          auto task = queue.front;
          queue.removeFront;
          return task;
        } else return null;
      }
    }
  }
}
```

両端キューの操作は、競合する他のプロセッサに対して**不可分な操作**である必要から、排他制御を実施する。なお、第 4 章で紹介する C++版の実装は、Arora et.al (1998) による排他制御なしの両端キューを採用した。

2.4 キューの待ち時間

両端キューを $M/D/1/N$ の**無記憶マルコフ過程**の待ち行列とし、排他制御に定数時間 D かかると仮定する。ここに最大 N 台のプロセッサが**ポアソン到着**すると考え、待ち時間の平均値 w を求めると、式 (2.1) になる。

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

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

ワークスティーリングは常に大きなタスクを奪うので、プロセッサ数 N が増えても到着率 λ が抑制される。反対に、FIFO 型のスケジューラでは待ち時間が増大する。式 (2.1)(2.2) は Brun & Garcia (2000) に従った。

第3章　行列積の並列処理の評価

第 3 章では、密行列積の処理速度を計測して、第 2 章で実装した並列スケジューラの**台数効果**を確認する。

dmm.d

```
import dawn, core.atomic;
import std.algorithm, std.parallelism, std.random, std.range, std.stdio;
```

変数 sched は、スケジューラを参照する。プロセッサ数は実行時に決まるので、インスタンス化は後で行う。

dmm.d

```
shared Dawn!(int, int, int, int, int, int, int) sched;
```

行列積は**キャッシュヒット率**を考慮して行列 B を転置した $C = A\,{}^tB$ の形式とし、正方行列を変数宣言する。並列処理の効果を高めるため、各行の末尾にダミー領域を設けて、メモリ空間で分離する**パディング**を行う。

dmm.d

```
const int N = 8192;
const int PAD = 32;
shared double[N * (N + PAD)] A = [0];
shared double[N * (N + PAD)] B = [0];
shared double[N * (N + PAD)] C = [0];
```

これは、プロセッサ間でメモリの**フォルスシェアリング**を防ぎ、**キャッシュコヒーレンシ**に伴う律速を防ぐ。行列積の計算は、分割統治により再帰的にタスクと行列を分割して行う。これを dmm_dawn 関数に実装する。

dmm.d

```
int dmm_dawn(int i1, int i2, int j1, int j2, int k1, int k2) {
  auto axes = [i2 - i1, j2 - j1, k2 - k1];
  if(axes[axes.maxIndex] <= 128) {
    dmm_leaf(i1, i2, j1, j2, k1, k2);
  } else if(axes.maxIndex == 0) {
    auto t1 = sched.fork!dmm_dawn(i1, (i1+i2)/2, j1, j2, k1, k2);
    auto t2 = sched.fork!dmm_dawn((i1+i2)/2, i2, j1, j2, k1, k2);
    sched.join(t1);
    sched.join(t2);
  } else if(axes.maxIndex == 1) {
    auto t1 = sched.fork!dmm_dawn(i1, i2, j1, (j1+j2)/2, k1, k2);
    auto t2 = sched.fork!dmm_dawn(i1, i2, (j1+j2)/2, j2, k1, k2);
    sched.join(t1);
    sched.join(t2);
  } else if(axes.maxIndex == 2) {
    auto t1 = sched.fork!dmm_dawn(i1, i2, j1, j2, k1, (k1+k2)/2);
    auto t2 = sched.fork!dmm_dawn(i1, i2, j1, j2, (k1+k2)/2, k2);
    sched.join(t1);
    sched.join(t2);
  }
  return 0;
}
```

最長軸を分割することで、部分行列を正方行列に近付けて局所性を高め、キャッシュヒット率を改善できる。分割統治の末端の最適な粒度に達すると、並列化をやめて dmm_leaf 関数を呼び出し、部分行列積を求める。

dmm.d

```
void dmm_leaf(int i1, int i2, int j1, int j2, int k1, int k2, int stride = N + PAD) {
  import std.numeric: dotProduct;
  foreach(i, iN; enumerate(iota(i1 * stride, i2 * stride, stride), i1)) {
  foreach(j, jN; enumerate(iota(j1 * stride, j2 * stride, stride), j1)) {
    C[iN + j].atomicOp!"+="(A[iN+k1..iN+k2].dotProduct(B[jN+k1..jN+k2]));
  }
  }
}
```

比較対象として、D 言語に標準的に付属する TaskPool を利用して、全く同じ手順での密行列積を実装する。

dmm.d

```
void dmm_pool(int i1, int i2, int j1, int j2, int k1, int k2) {
  auto axes = [i2 - i1, j2 - j1, k2 - k1];
  if(axes[axes.maxIndex] <= 128) {
    dmm_leaf(i1, i2, j1, j2, k1, k2);
  } else if(axes.maxIndex == 0) {
    auto t1 = task!dmm_pool(i1, (i1+i2)/2, j1, j2, k1, k2);
    auto t2 = task!dmm_pool((i1+i2)/2, i2, j1, j2, k1, k2);
    taskPool.put(t1);
    taskPool.put(t2);
    t1.workForce;
    t2.workForce;
  } else if(axes.maxIndex == 1) {
    auto t1 = task!dmm_pool(i1, i2, j1, (j1+j2)/2, k1, k2);
    auto t2 = task!dmm_pool(i1, i2, (j1+j2)/2, j2, k1, k2);
    taskPool.put(t1);
    taskPool.put(t2);
    t1.workForce;
    t2.workForce;
  } else if(axes.maxIndex == 2) {
    auto t1 = task!dmm_pool(i1, i2, j1, j2, k1, (k1+k2)/2);
    auto t2 = task!dmm_pool(i1, i2, j1, j2, (k1+k2)/2, k2);
    taskPool.put(t1);
    taskPool.put(t2);
    t1.workForce;
    t2.workForce;
  }
}
```

再帰的な分割統治を行わず、並列 for 文と同様に行列積を格子状に分割統治する dmm_gr3d 関数も実装する。

dmm.d

```
void dmm_gr3d(int i1, int i2, int j1, int j2, int k1, int k2, int grain = 128) {
  const int iN = (i2 - i1) / grain;
  const int jN = (j2 - j1) / grain;
  const int kN = (k2 - k1) / grain;
  foreach(ch; parallel(iota(0, iN * jN * kN), 1)) {
    const int i = i1 + ch / kN / jN * grain;
    const int j = j1 + ch / kN % jN * grain;
    const int k = k1 + ch % kN * grain;
    dmm_leaf(i, i + grain, j, j + grain, k, k + grain);
  }
}
```

また、第 3 の比較対象として dmm_gr2d 関数を実装する。k 軸の分割を行わずに i 軸と j 軸のみ並列化する。

dmm.d

```
void dmm_gr2d(int i1, int i2, int j1, int j2, int k1, int k2, int grain = 128) {
  const int iN = (i2 - i1) / grain;
  const int jN = (j2 - j1) / grain;
  foreach(ch; parallel(iota(0, iN * jN), 1)) {
    const int i = i1 + ch / jN * grain;
    const int j = j1 + ch % jN * grain;
    dmm_leaf(i, i + grain, j, j + grain, k1, k2);
  }
}
```

最後に main 関数を実装する。行列積に含まれる乗算と加算の回数を処理時間で割ると、flop/s 値が求まる。

dmm.d

```
void main() {
  defaultPoolThreads = totalCPUs;
  import std.datetime.stopwatch: AutoStart, StopWatch;
  sched = new shared Dawn!(int, int, int, int, int, int, int);
  for(int i = 0; i < A.length; i++) A[i] = uniform(0, 1.0);
  for(int j = 0; j < B.length; j++) B[j] = uniform(0, 1.0);
  auto sw = StopWatch(AutoStart.yes);
  sched.dawn!dmm_dawn(0, N, 0, N, 0, N);
  writeln(2.0 * N * N * N / sw.peek.total!"nsecs", "GFLOPS");
}
```

以上で行列積の実装が完成した。コンパイル時は、最適化を有効にしつつ、実行時の監視機能を無効化する。

```
$ ldc -O -release -boundscheck=off -of=dmm dawn.d dmm.d
$ for c in {0..35}; do taskset -c 0-$c ./dmm; done
```

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。

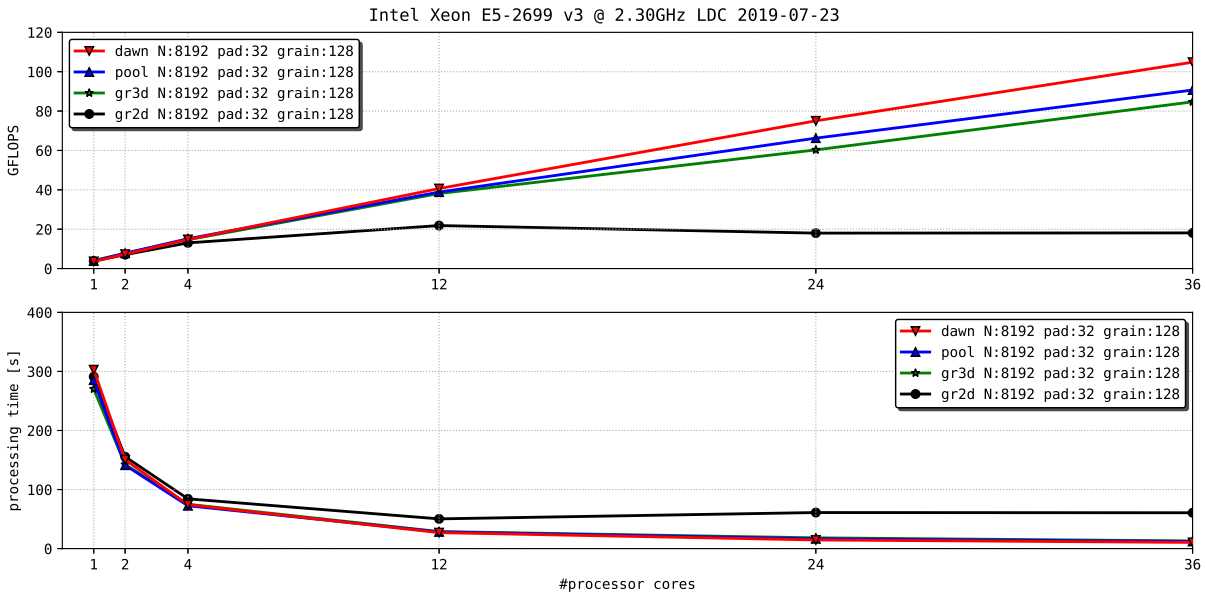


Fig. 3.1: dense matrix multiplication, $8192 \times 8192 \times 8192$.

これ以上の高性能化を図るには、行列積の末端における逐次処理の SIMD 命令による高効率化が必須である。

第 4 章　高性能並列処理系の紹介

dusk は *non-uniform memory access* (NUMA) 向けに C++11 で実装された軽量な並列スケジューラである。

4.1　利用方法

dusk は UNIX 用の実装が BSD 3 条項ライセンスで頒布されており、GitHub から下記の操作で導入できる。

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

dusk は Linux の NUMA 環境で gcc と clang と icc の全てで動作を確認した。本体は libdusk.so である。

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

dusk を C++プログラムで使う際は、dusk.hpp を読み込む。これで後述するテンプレート関数が利用できる。

dusk.hpp

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

sun::launch 関数は、所定の関数 pad を起点にスケジューラを起動して、並列処理が終わるまで待機する。

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));
```

以上の sun::launch 関数と sun::salvo 関数と sun::burst 関数は、実態としては下記の関数を隠蔽する。
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、繁忙待機を行う。

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);
```

dusk の実装は第 2 章とほぼ同様であるが、**ロックフリー**な両端キューを実装するなど、高性能化を図った。

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。
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(1)	キューから全プロセッサに分配
PDRWS	FILO(N)	繁忙状態のプロセッサから奪取
ADRWS	FILO(N)	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 に対する優位性は未知である。

4.3 性能測定

第 3 章と同様に、Intel Xeon®E5-2699 v3 を 2 個搭載した NUMA 環境で、密行列積の処理速度を測定した。

```
$ make -C dusk/test/dmm
$ ./dusk/test/dmm/dmm.plot
```

Fig. 4.1 に結果を示す。Meltdown や Spectre 等の対策を行う以前の 2015 年の測定である点に注意を要する。

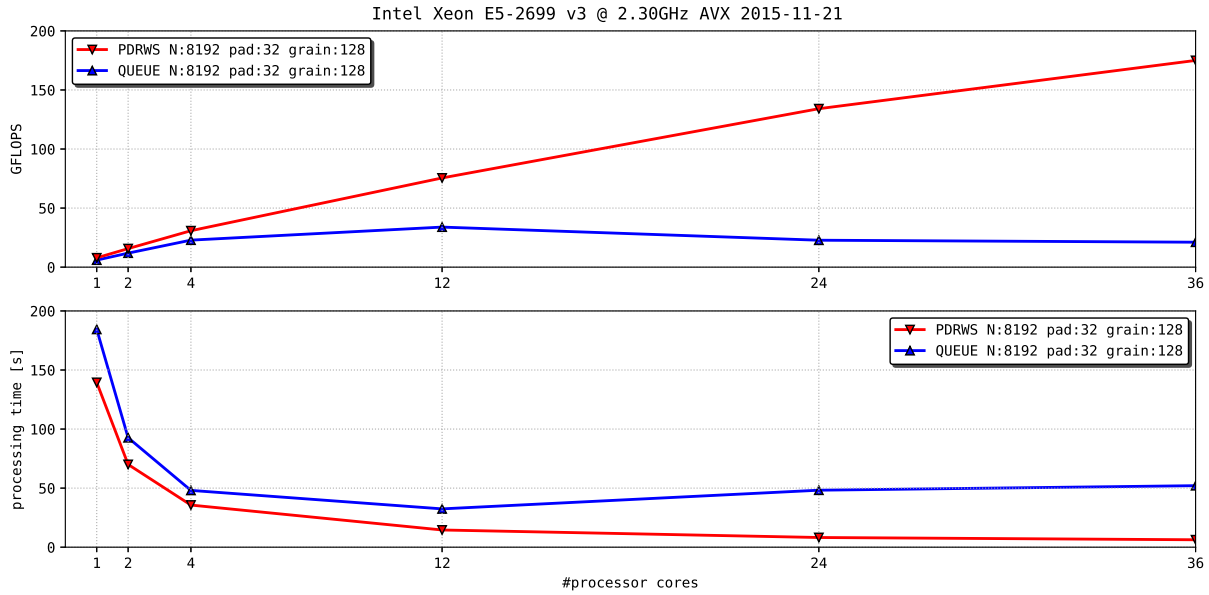


Fig. 4.1: dense matrix multiplication, 8192 × 8192 × 8192, vectorized by AVX.

PDRWS は dmm_dawn 関数に相当し、3 軸を粒度 128 まで順番に並列化して、末端では SIMD 命令を使用した。
QUEUE は dmm_gr2d 関数に相当し、2 軸を粒度 128 まで格子状に並列化して、同様に SIMD 命令を使用した。