アルゴリズムと並列

アルゴリズムで並列を考える

アルゴリズム ＝ 計算手順

問題意識１

並列化しにくいアルゴリズムが存在する？

問題意識２

アルゴリズムが、必要以上に順序を指定している？

$S^{(*)} = 0$
$S^{(0)} = S^{(*)} + a[0]$
$S^{(1)} = S^{(0)} + a[1]$
$S^{(2)} = S^{(1)} + a[2]$
$S^{(3)} = S^{(2)} + a[3]$
…
$S^{(99)} = S^{(98)} + a[99]$

$\sum_{i=0}^{99} a[i]$ を求めたい

```
S = 0;
for (i=0; i<100;i++){
      S = S + a[i];
}
```

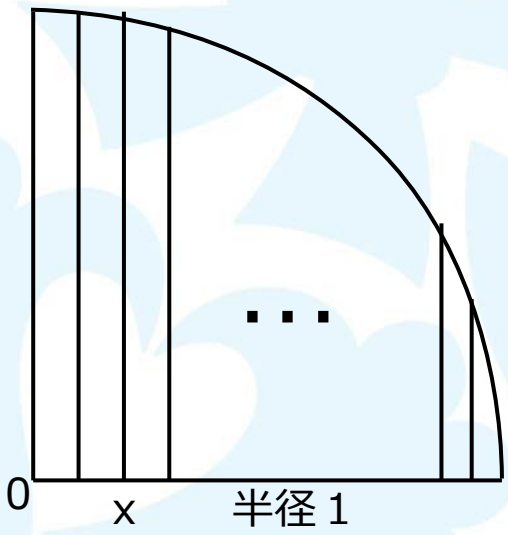
？？

並列化しやすいアルゴリズム例

- (たとえば) シミュレーションで
  - 時刻 t=0 から 1 ずつ進めながら
  - 　３次元メッシュの各点について
    t=t の時刻の状態を元にして
    t+1 の状態を計算する
  - 全ての点の計算が終わったらt=t+1に進める
- この中では、ある時刻t=t での各点の計算はお互いに独立なので、並列処理可能
  - ３次元メッシュの点の数が多い ⇒ 並列度大

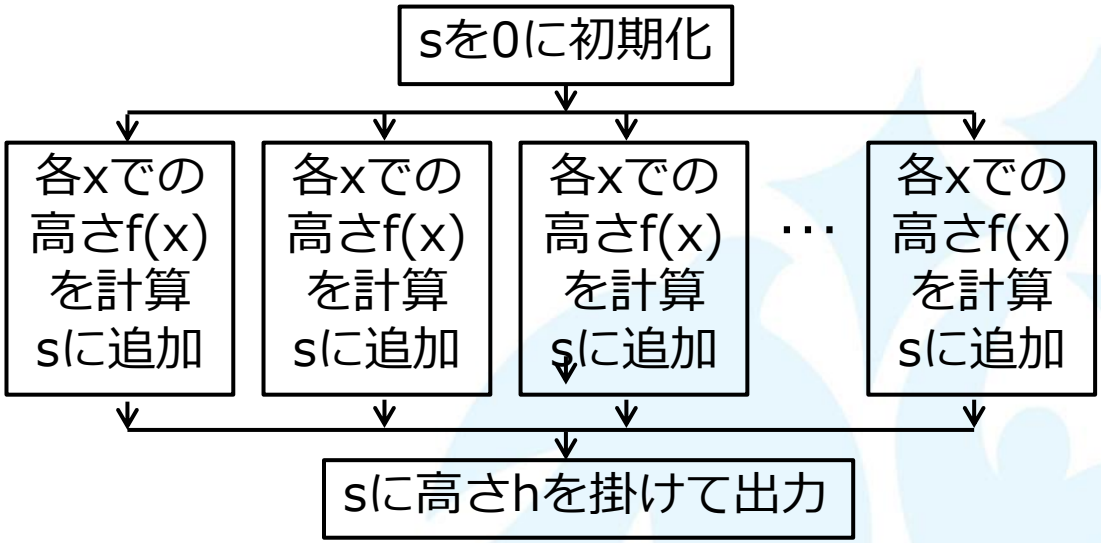
並列化しやすいアルゴリズム例

- (非常に人工的な例だが) 数値積分の例
  - 数値積分 (区分求積法) でπを計算
  - 案１：右の1/4円の面積S
    - $\int_0^1 \sqrt{(1-x^2)}\,dx = \pi/4$
  - 案２：定理
    - $\int_0^1 \frac{1}{x^2+1}dx = \frac{\pi}{4}$
    - 平坦なので端の誤差が小



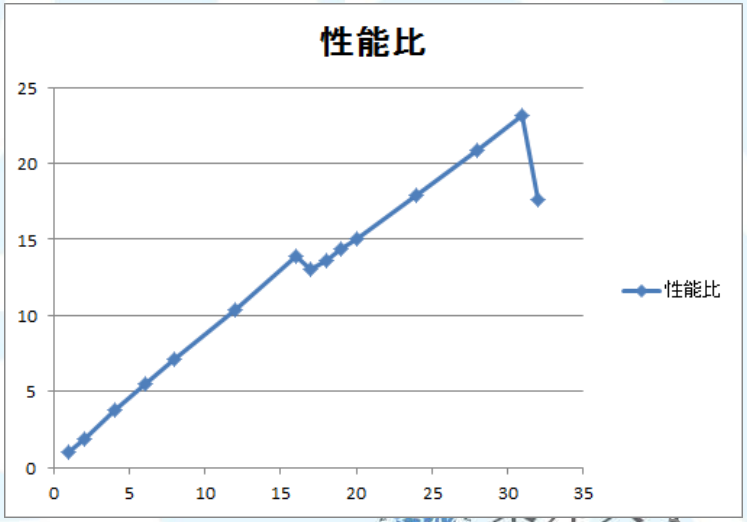
- 並列化
  - 長方形がそれぞれ独立に計算可能 (依存性無し)

区分求積の例




```
double f(double x) { return sqrt(1.0-x*x); }
double s = 0;
#pragma omp parallel for reduction(+:s)
for (i=0; i<=DIV; i++) {
    s = s + f(((double)i)/(double)DIV);
}
s = (s - (f(0.0) + f(1.0))/2.0) / ((double)DIV);
printf("pi=%2.18f¥n", s*4);
```

16コア×2チップでの実行例



並列化しづらいアルゴリズム例

- 動的計画法 (Dynamic Programming)
  - 対象となる問題を帰納的に解く場合にくり返し出現する小さな問題例について、解を表に記録し表を埋めていく形で計算をすすめ、冗長な計算をはぶくアルゴリズムのことをいう。
    特定のアルゴリズムを指すのではなく、上記のような手法を使うアルゴリズムの総称である。

    (ウィキペディア　動的計画法)
    (https://ja.wikipedia.org/wiki/%E5%8B%95%E7%9A%84%E8%A8%88%E7%94%BB%E6%B3%95)

- 単純例 ⇒ フィボナッチ数列の計算
  - f(n) = f(n-1) + f(n-2)　　但しn≧2, f(0)=0,　f(1)=1

動的計画法

出発点は
問題を分割し
再帰的に計算

$$f(n) = f(n-1) + f(n-2)$$

定義通りに計算

```
int fib(int n) {
    if (n==0) then return 0;
    else if (n==1) then return 1;
    else return fib(n-1)+fib(n-2);
}
```

途中結果をメモし
再計算を避ける

```
int fib(int n) {
    int M[1000] ;  M[0] = 0;  M[1] = 1;
    for (i = 2; i <= n; i++) {
        M[i] = M[i-1] + M[i-2];
    }
    return M[n];
}
```

```
 fib(4)
= fib(3) + fib(2)
= (fib(2) + fib(1)) + fib(2)
= (((fib(1) + fib(0)) + fib(1)) + ((fib(1) + fib(0))
= (((1) + (0)) + (1)) + ((1) + (0))
= 3
```

```
M[0] = 0;  M[1] = 1;
M[2] = M[1] + M[0] = 1
M[3] = M[2] + M[1] = 2
M{4] = M[3] + M[2] = 3
```

nが増えると$O(2^n)$で計算量が増える

nが増えても比例して増えるだけ

東邦大学

動的計画法

$$f(n) = f(n-1) + f(n-2)$$

出発点は
問題を分割し
再帰的に計算

定義通りに計算

```
int fib(int n) {
    if (n==0) then return 0;
    else if (n==1) then return 1;
    else return fib(n-1)+fib(n-2);
}
```

途中結果をメモし
再計算を避ける

```
int fib(int n) {
    int M[1000] ;  M[0] = 0;  M[1] = 1;
    for (i = 2; i <= n; i++) {
        M[i] = M[i-1] + M[i-2];
    }
    return M[n];
}
```

```
 fib(4)
= fib(3) + fib(2)
= (fib(2) + fib(1)) + fib(2)
= ((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))
= (((1) + (0)) + (1)) + ((1) + (0))
= 3
```

nが増えると$O(2^n)$で計算量が増える

```
M[0] = 0;  M[1] = 1;
M[2] = M[1] + M[0] = 1
M[3] = M[2] + M[1] = 2
M[4] = M[3] + M[2] = 3
```

M[2]が無いと
M[3]できない

M[3]が無いと
M[4]できない

それぞれのM[i]の計算は
並列にはできない

東邦大学

つまり

- 問題を、小さな問題に分割し、再帰的に
  解決することで、解けるケースがある
- その時、途中結果をうまく記憶できれば
  指数時間⇒線形時間にできる
  この工夫を「動的計画法」と呼ぶ
- ところが、記憶を次のステップで使うので
  ステップ間の並列化はできない

動的計画法の形で、並列化は難しそうだ。
別の方法を思いつくだろうか？

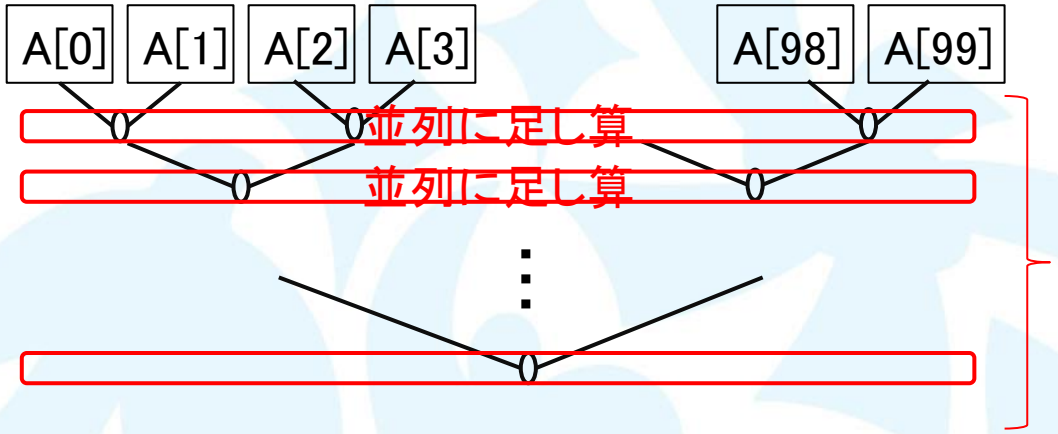
アルゴリズム＝問題を解く手順

- 「手順」って何だろう？
- 例：　バブルソート
  - 左端から順番に、
  - 隣同士を比較し、逆順なら左右を入替える
  - 右端に到達したら、もう一度左端から同じ処理
  - １回も左右入替えが起こらなければ、終了
- 「順番に」処理することを前提？？
  - 従来は直列手順　（並列を考える場合を除き）
    ⇒　並列で何が起こるかは考慮の範囲外？

足し算のアイデア

$\sum_{i=0}^{100} a[i]$ を求めたい

```
S = 0;
for (i=0; i<100;i++){
    S = S + a[i];
}
```



直列に99回足し算

$Log_2100(=6.6)\fallingdotseq 7$段足し算

並列度 (横幅) は最大50

50⇒25⇒12⇒…

（注）　ソフトでこれを制御すると
制御のオーバーヘッドが大きい

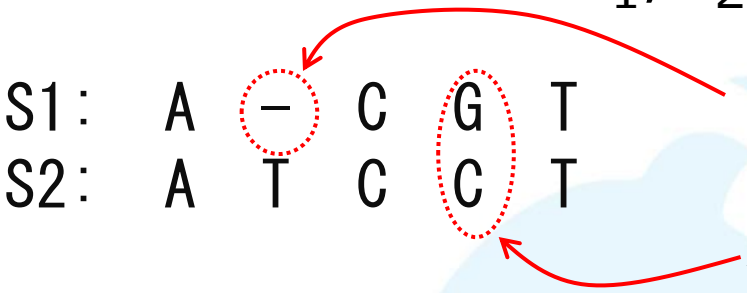
ソートの例

是非、参考書「並列コンピューティング技法」の
第８章「ソート」pp131-182、第９章「サーチ」pp183-198　を
自分で細かく見てみてください

- バブルソート？
  - データを分割？
  - データはそのまま、ただswapを複数並列に？
  - それぞれ問題がある⇒バブルは並列に向かない？
- 奇遇転置ソート？
  - Swapを奇数フェーズと偶数フェーズに分ける
  - フェーズ内のデータ並列性が大⇒うまくゆく
- クイックソート？
  - 再帰呼出しを並列化する工夫
  - さらにいろいろな工夫をする余地（が書かれている）
- 基数（ラディックス）ソート？

動的計画法をむりやり並列化

- ２つのDNA塩基配列S_1,S_2の位置合せ問題



- 不一致のペナルティ↓を足し合わせる

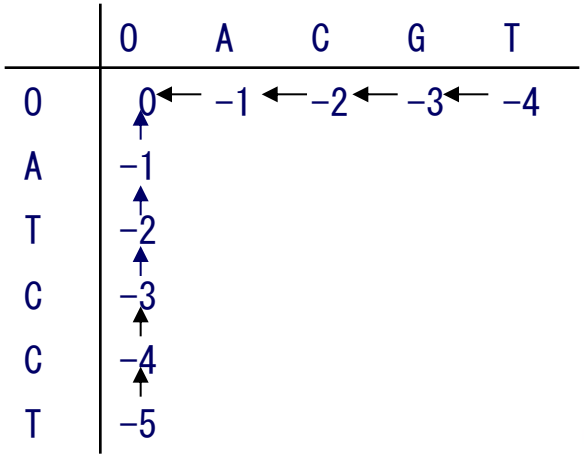
w(x, −) = −1,　w(−, x) = −1,　一方が空欄なら-1
w(x, x) = +1,　w(x, y) = −1 if x≠y　一致なら+1、不一致なら-1

D[i, j] = max {　←ある点[i, j]の評価Dは、下記３ケースの最大値
　D[i−1, j] + w(S1[i], −)　←S2の側が空欄、ペナルティ-1
　D[i, j−1] + w(−, S2[j])　←S1の側が空欄、ペナルティ-1
　D[i−1, j−1] + w(S1[i], S2[j]) }　←両方文字、一致なら+1
　　不一致なら-1

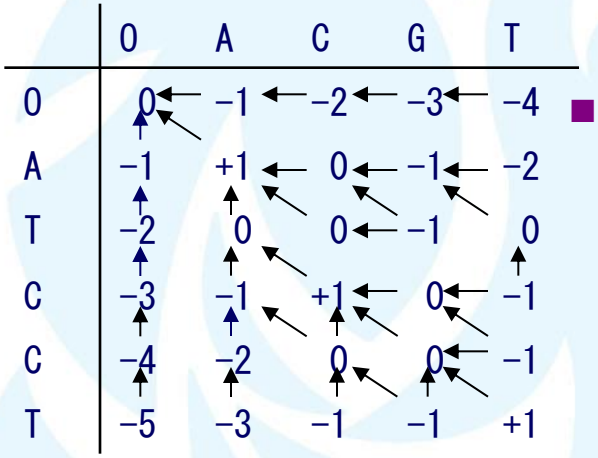
表に書く（前に計算したものを残す）

- 空の配列　⇒　文字0で表す
- 点数はわかったものを加える
- 矢印はmaxを選んだ向き

・w(x,−) = w(−,x) = −1
・w(x,x) = 1
・w(x,y) = −1　(x≠y)



初期条件（縦端・横端）を計算



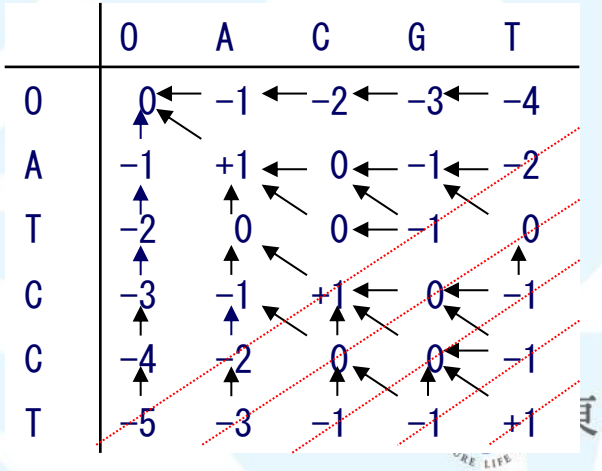
内部を次々に計算
左上から右下に向かって
計算する（前の結果利用）

- D[i,j] = max {
  D[i−1,j] + w(S1[i], −)
  D[i,j−1] + w(−, S2[j])
  D[i−1,j−1] +
  w(S1[i], S2[j]) }

動的計画法をむりやり並列化

- 表を埋める処理を並列化したい？
  - 矢印の前の位置の結果を使うので
    計算が済まないと次の位置が計算できない
    （データ依存のため、並列化できない）

- データ依存性は波（リップル）状に斜めに伝搬する
  - 原点から同じ距離にある
    点は、同時に計算できる

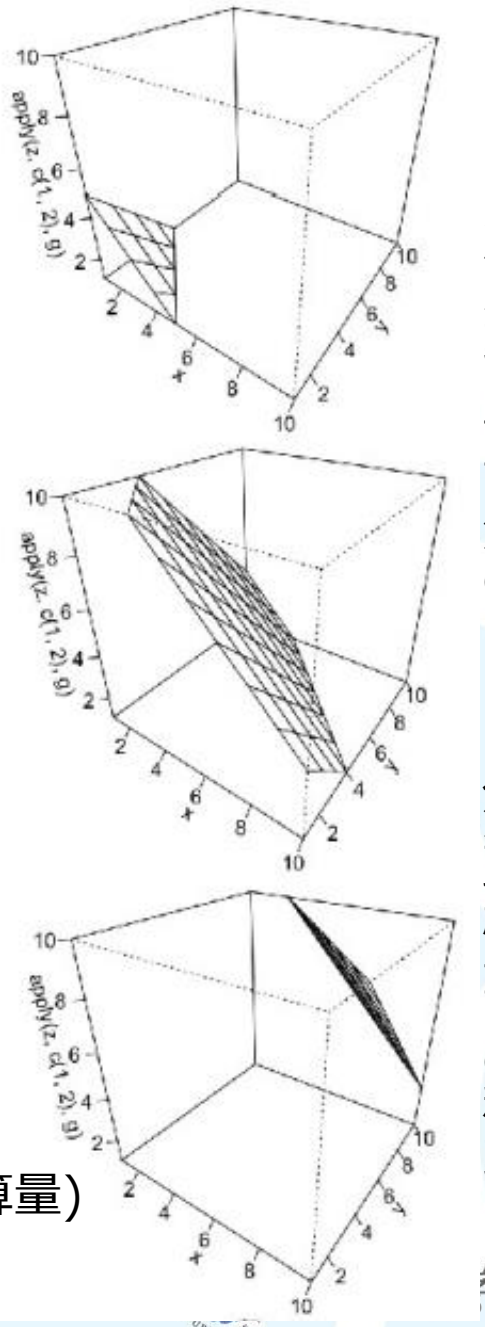


動的計画法を むりやり並列化

- 前頁（２つの配列の位置合せ）は斜め線上（リップルの波頭）の
  要素数は少ない ⇒ 並列度小

↓

- 多数(≧３)の配列の位置合せなら
  N次元空間での動的計画法になり
  リップルの波頭は超平面になり
  要素数は多いだろう
  ⇒ 並列度を稼げる



3次元動的計画法のリップル波頭超平面の移動

この問題では、
(超平面に載る点の抽出の手間) ＞ (それぞれの点での計算量)
だったため、並列の効果は出なかった

アルゴリズム？

- KnuthのThe Art of Programming か？