高度なN体シミュレーション法

押野翔一 (東大宇宙線研)

2019/02/06

目次

- タイムステップの工夫
- 高精度積分法
- 相互作用の取り扱い方
- 並列計算

様々なN体計算

今回の実習では、
リープフロッグ法+直接法+固定時間刻み
の実装を行った

- 惑星形成：エルミート法+GRAPE+階層時間刻み
- 土星環：リープフロッグ法+GRAPE+local shearing box
- 銀河形成：リープフロッグ法+ツリー法+並列計算+SPH法
- 構造形成：リープフロッグ法+TreePM法+並列計算

調べたい現象によって適した計算法を用いる

タイムステップの工夫

- 共有時間刻み
- 可変時間刻み
- 独立時間刻み
- 階層時間刻み

衝突系では近接遭遇を精度よく計算する必要がある

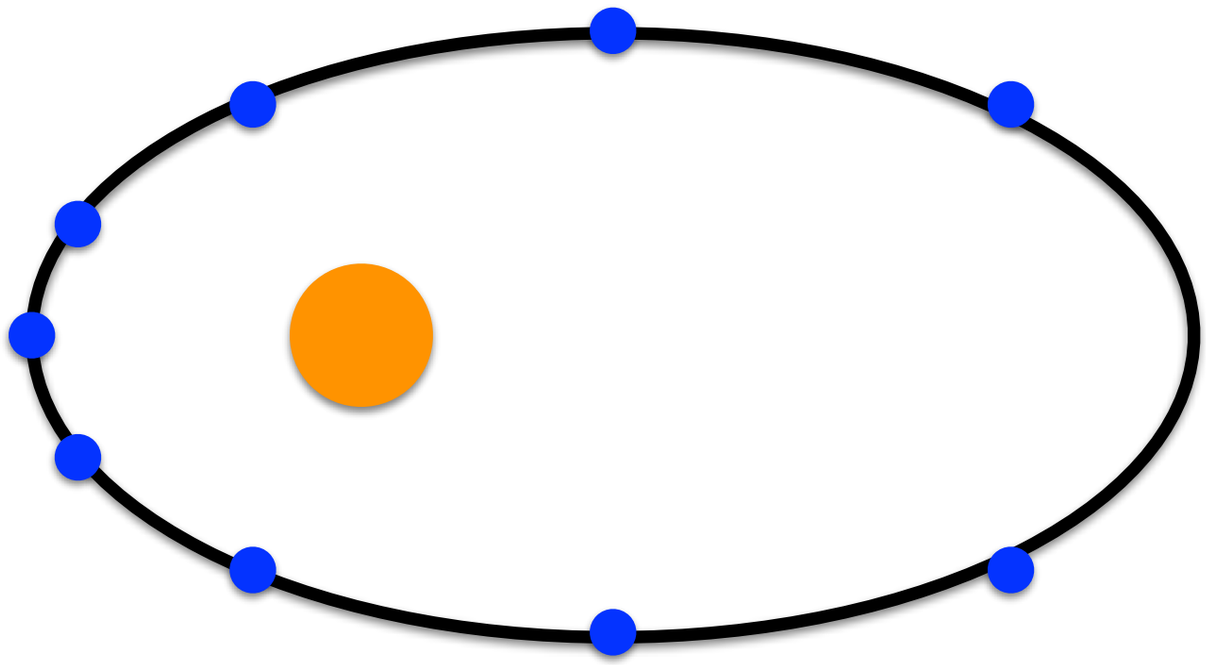
→　短い時間刻みが必要

→　計算量が増加してしまう

共有時間刻み

- すべての粒子のタイムステップを同じだけ進める
  - 高精度積分法でも簡単に実装できる
- タイムステップの大きさがシミュレーション全体で同じ場合は固定時間刻みと呼ぶ
  - 近接遭遇している粒子以外も小さいタイムステップにしなければならないので、衝突系では計算量が大きく なってしまう。

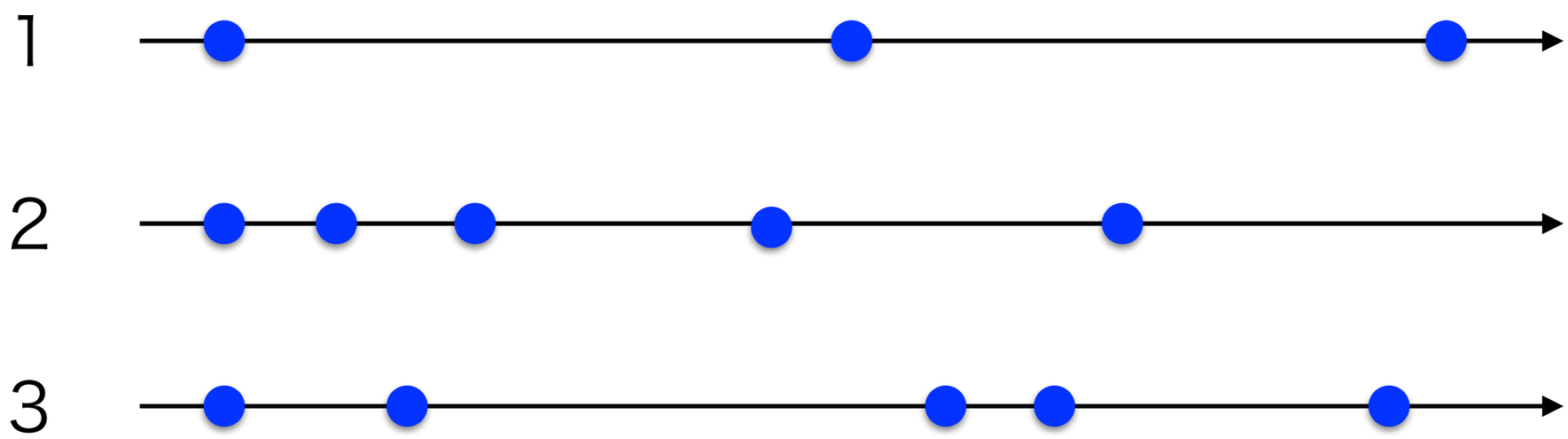
可変時間刻み



軌道の変化が大きいときのみ短いタイムステップを使用

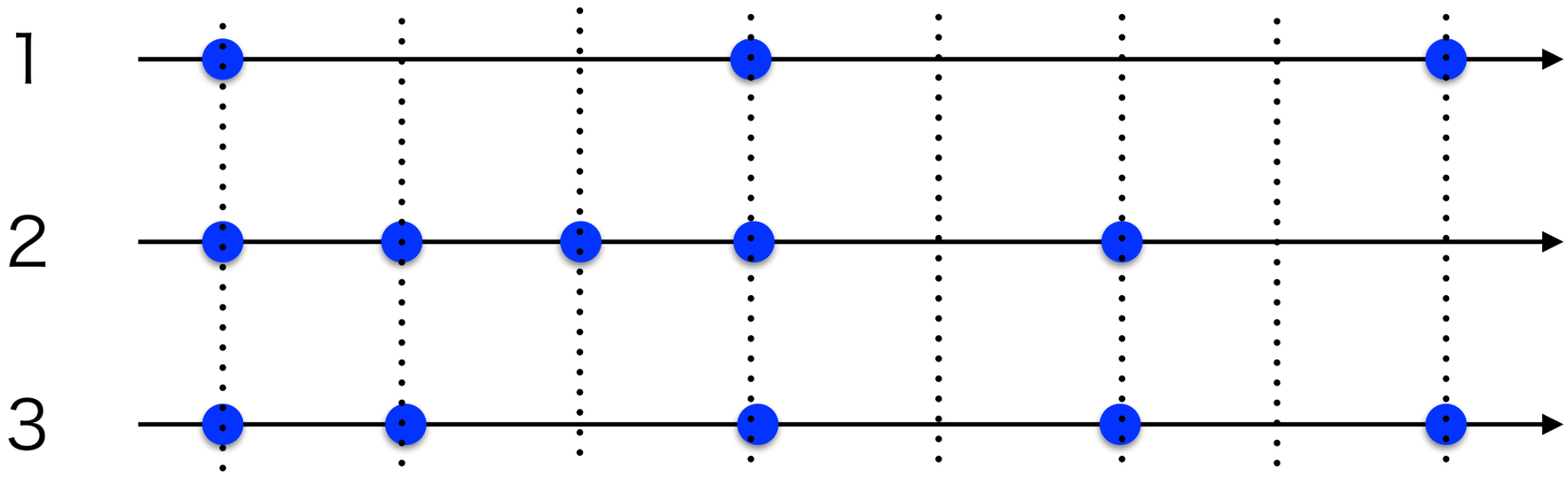
→　すべて短いタイムステップで計算するより精度は悪くなる、
　　計算量の削減の効果が大きい

独立時間刻み



- ここの粒子をそれぞれ異なるタイムステップで積分していく。
- 近接遭遇している粒子のみを短いタイムステップで計算する。
- 軌道の変化が大きい粒子は短いタイムステップで、変化が小さい粒子は大きいタイムステップで計算する。
- 予測子修正子法と組み合わせた場合、予測子の計算量が大きくなってしまう。

階層時間刻み



タイムステップの大きさを 2^n とする。
同時刻に積分する粒子が出てくるため、その分計算量を減らせる。
タイムステップの加算での丸め誤差がなくなる。

目次

- タイムステップの工夫
- 高精度積分法
- 相互作用の取り扱い方
- 並列計算

高精度積分法

- 粒子軌道を正確に計算する必要がある時、タイムステップを短くしていくとその分計算量が増加してしまう。そのため、高次の次数を使った積分法が必要となってくる。
  - 高次シンプレクティック積分法
  - エルミート法

高次シンプレクティック積分法

Yoshida 1990 参照
惑星系の長期安定性などを計算するとき、リープフロッグ法 (2 次シンプレクティック積分法) では精度が足らなくなることがある。この時、より高次のシンプレクティック積分法が使用される。

1. 長時間積分ではエネルギー誤差の増加が抑えられるシンプレクティック積分が有用。
2. プログラミングが簡単。4次シンプレクティック法の場合、リープフロッグ積分を時間刻みを変えて 3 回呼ぶことで求めることが出来る

エルミート法

高精度の N 体シミュレーション法として広く使われている。

4 次エルミート法 (Makino 1991)

1. 加速度と加速度の1階微分を計算する。
2. 位置と速度の予測子を計算する。
3. 予測子から加速度と加速度の微分を計算する。
4. エルミート補間多項式から高次の展開係数を計算する。
5. 位置、速度の修正子を計算する。

エルミート法

加速度と加速度の 1 階微分 (加加速度) の計算

$$a_i = -\sum_{i \neq j}^{N} G m_j \frac{x_j - x_i}{|x_j - x_i|^3}$$

$$\dot{a}_i = -\sum_{i \neq j}^{N} G m_j \left[\frac{v_j - v_i}{|x_j - x_i|^3} - \frac{3((v_j - v_i) \cdot (x_j - x_i))(x_j - x_i)}{|x_j - x_i|^5}\right]$$

加速度の 1 階微分のことを jerk とも呼ぶ

エルミート法

予測子

$$x_p = x_0 + v_0\Delta t + \frac{a_0}{2}\Delta t^2 + \frac{\dot{a}_0}{6}\Delta t^3$$

$$v_p = v_0 + a_0\Delta t + \frac{\dot{a}_0}{2}\Delta t^2$$

修正子

$$x_c = x_p + \frac{a_0^{(2)}}{24}\Delta t^4 + \frac{a_0^{(3)}}{120}\Delta t^5$$

$$v_c = v_p + \frac{a_0^{(2)}}{6}\Delta t^3 + \frac{a_0^{(3)}}{24}\Delta t^4$$

展開係数

$$a_0^{(2)} = \frac{-6(a_0 - a_1) - \Delta t(4\dot{a}_0 + 2\dot{a}_1)}{\Delta t^2}$$

$$a_0^{(3)} = \frac{12(a_0 - a_1) + 6\Delta t(\dot{a}_0 + \dot{a}_1)}{\Delta t^3}$$

エルミート法

6 次、8 次エルミート法 (Nitadori & Makino 2008)

- より高精度なエルミート法。
- 6 次は加速度の 2 階微分、8 次は 3 階微分まで使用する。
- 高次の積分法を使うことで、粒子の近接遭遇時にも長いタイムステップを使用できる。

目次

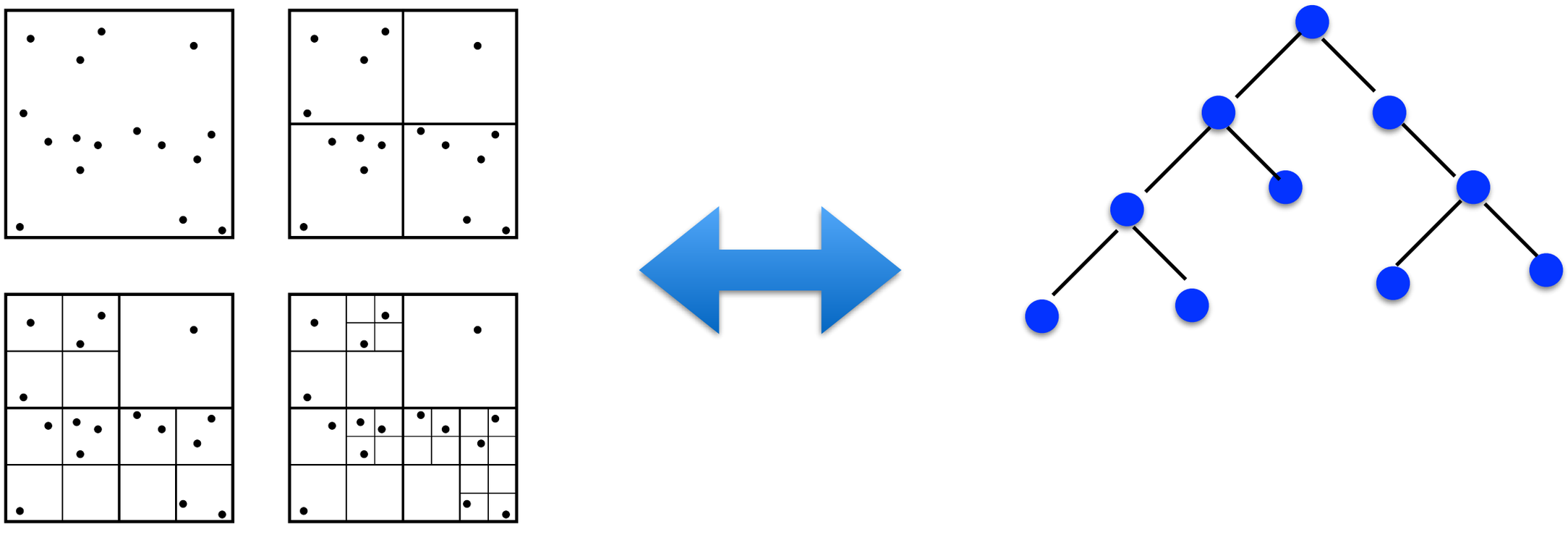
- タイムステップの工夫
- 高精度積分法
- 相互作用の取り扱い方
- 並列計算

相互作用の取り扱い方

- ツリー法
- PM法
- TreePM 法
- PPPT法

N 体シミュレーションにおいて最も時間がかかるのは粒子間相互作用の計算である。そのためこれまで様々な相互作用計算を軽くする手法が開発されてきた。

ツリー法

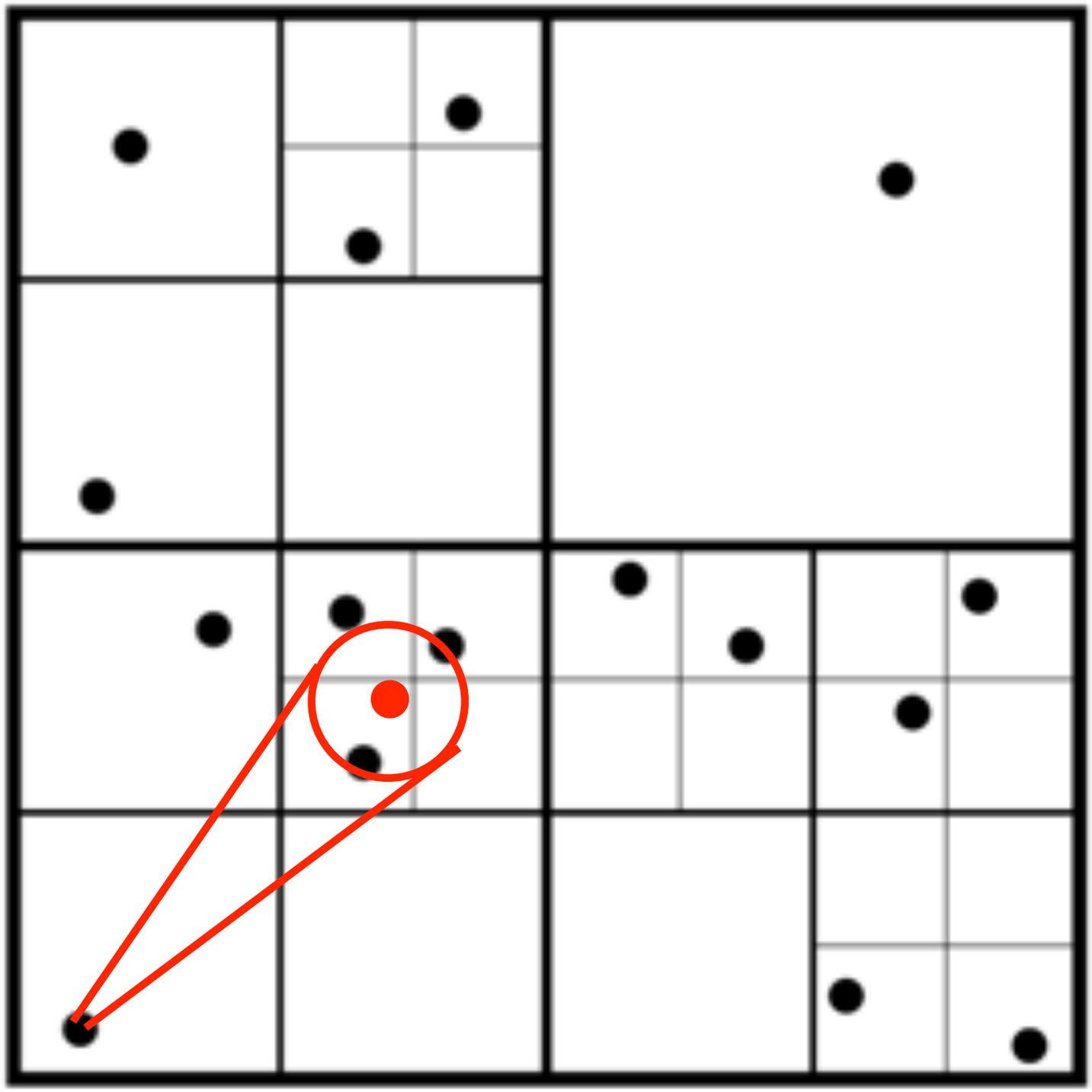


Barnes & Hut 1986
重力相互作用演算を O(N log N) にできる。
遠方の粒子からの重力をまとめて計算することで演算量を抑えている。
プログラミングが容易なので銀河形成などで広く使われている。

ツリー法

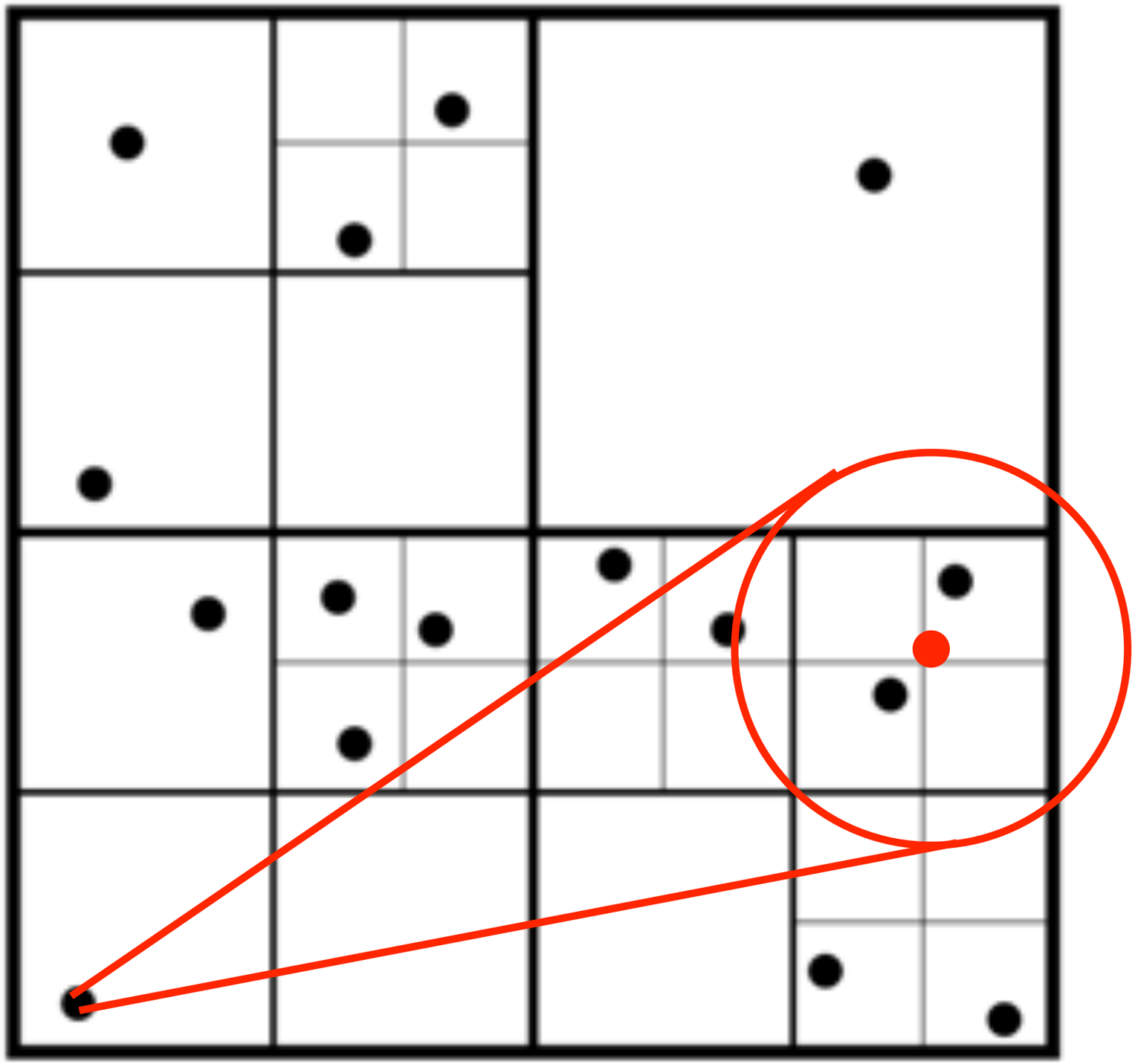
1. ツリー構造の作成
   (1) 全粒子を囲む正方形を作成
   (2) 内部に複数個の粒子がある場合は４分割
   (3) 粒子が一つになるまで繰り返す
2. セル内の全ての粒子の質量の総和と重心を計算
3. 力の計算
   (1) セルが十分遠くにある場合、セルの重心からの力
   (2) セルが近くにある場合、子セルからの力の総計

ツリー法



$\frac{l}{d} > \theta$　セルが近くにある場合：より小さなセルからの力を計算

ツリー法



$\frac{l}{d} < \theta$　　セルが遠くにある場合：セルの重心からの力を計算

PM法

空間をメッシュ分割して重力をポアソン方程式を用いて解く

$$\nabla^2\Phi = 4\pi G\rho$$

- 高速フーリエ変換(FFT)を用いることで、速く計算できる
- FFTはライブラリとして提供されている
- メッシュ内の粒子は直接計算するPPPM法もある

TreePM法

$$F = F_{PP} + F_{PM}$$

- 宇宙論的 N 体シミュレーションで用いられている方法
- 粒子数を最も多く扱える
- 空間をメッシュで区切り、 遠方のメッシュからの重力は FFT を用いて計算し、近傍の メッシュからの重力はメッシュ内の粒子をツリー法を用いて計算

PPPT法

近傍：エルミート法+直接法

遠方：リープフロッグ法+ツリー法

これまで見てきた相互作用の工夫は主に無衝突系で用いられてきた。なぜなら、これらの手法と衝突系において重要となる近接遭遇の取り扱いを両立させるのが難しいためである。

ここで紹介する PPPT 法は異なる積分法を組み合わせることで近接遭遇を正確に計算し、なおかつ粒子数の増加にも対応できる手法である。

目次

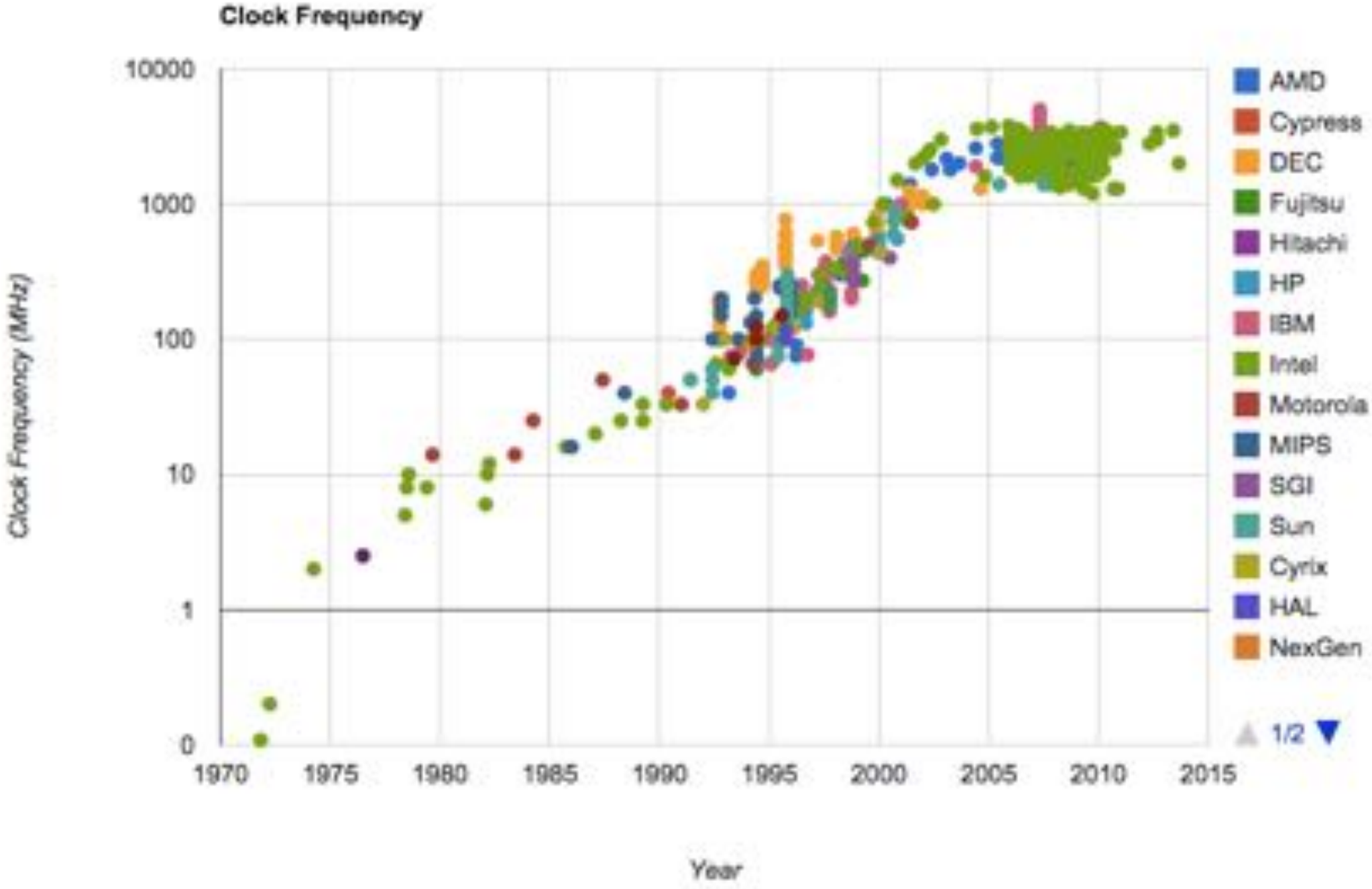
- タイムステップの工夫
- 高精度積分法
- 相互作用の取り扱い方
- 並列計算

並列計算

最近のCPUは1コア当たりの計算速度の向上が鈍化している
→　高速に計算するには多数のコアを接続して
　　並列計算をする必要がある

近年、Top500にランキングされるスーパーコンピュータは
ほとんどがこのような並列計算を行う計算機

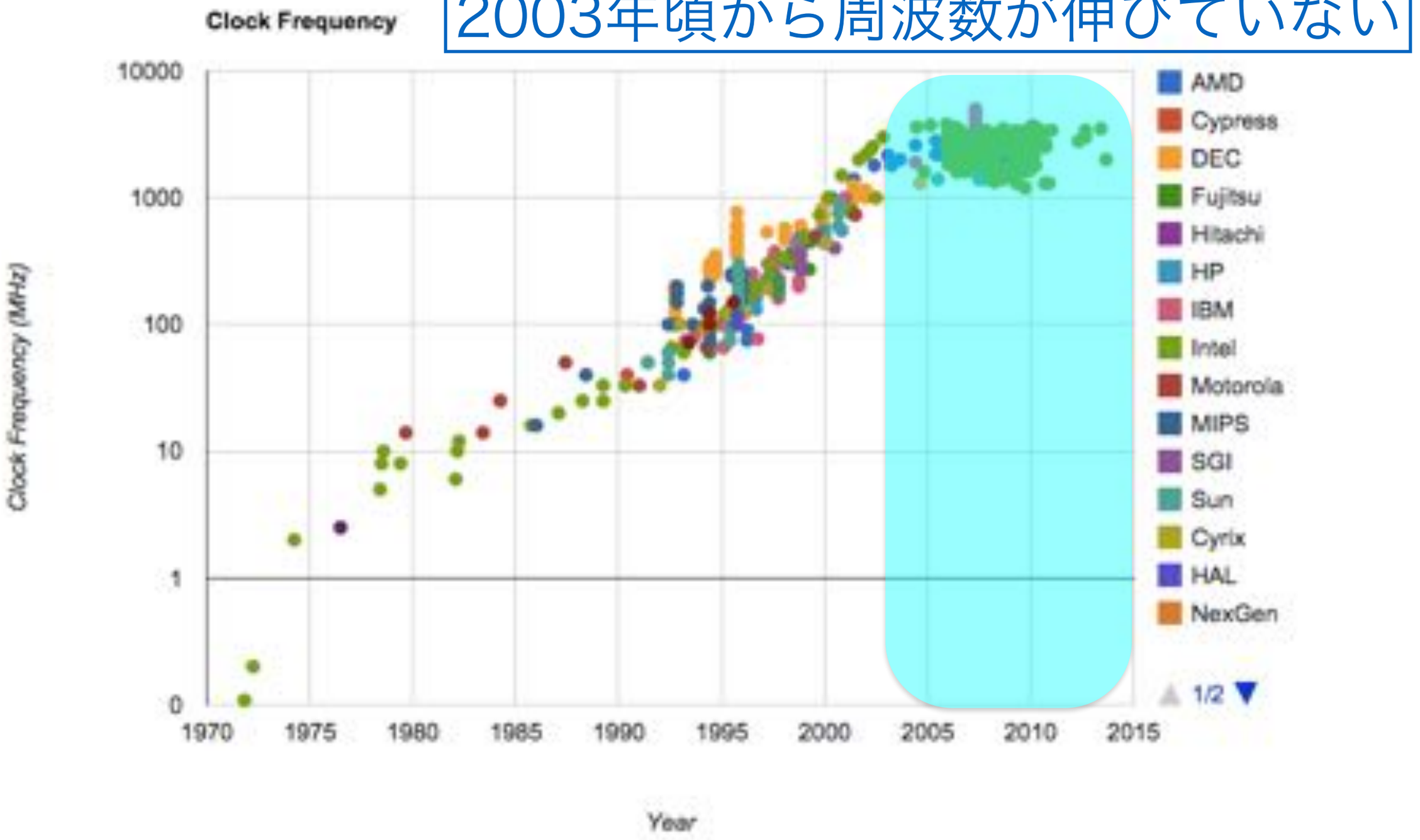
CPUの周波数の進化



http://cpudb.stanford.edu/

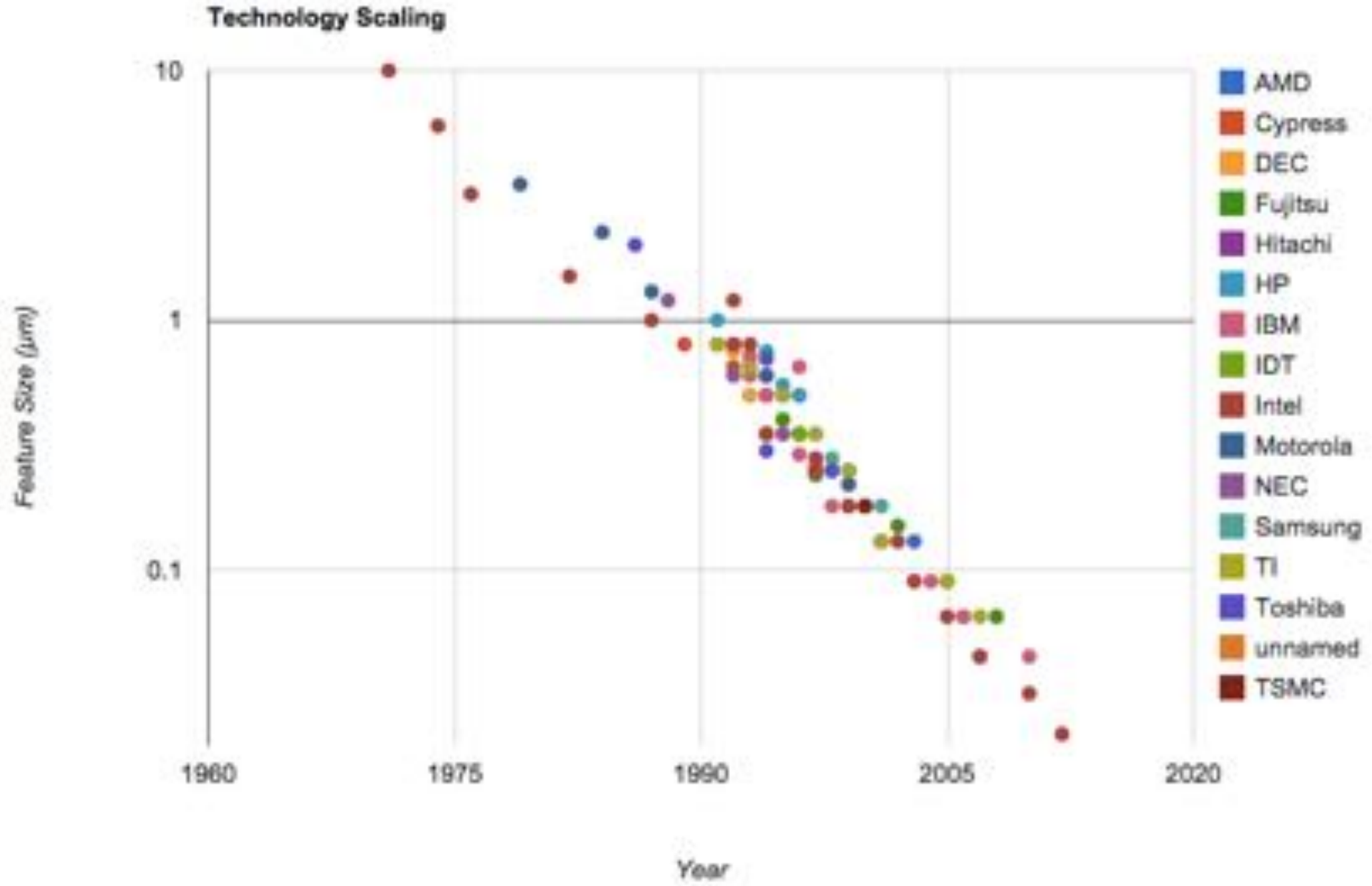
CPUの周波数の進化

2003年頃から周波数が伸びていない



http://cpudb.stanford.edu/

プロセスルールの進化

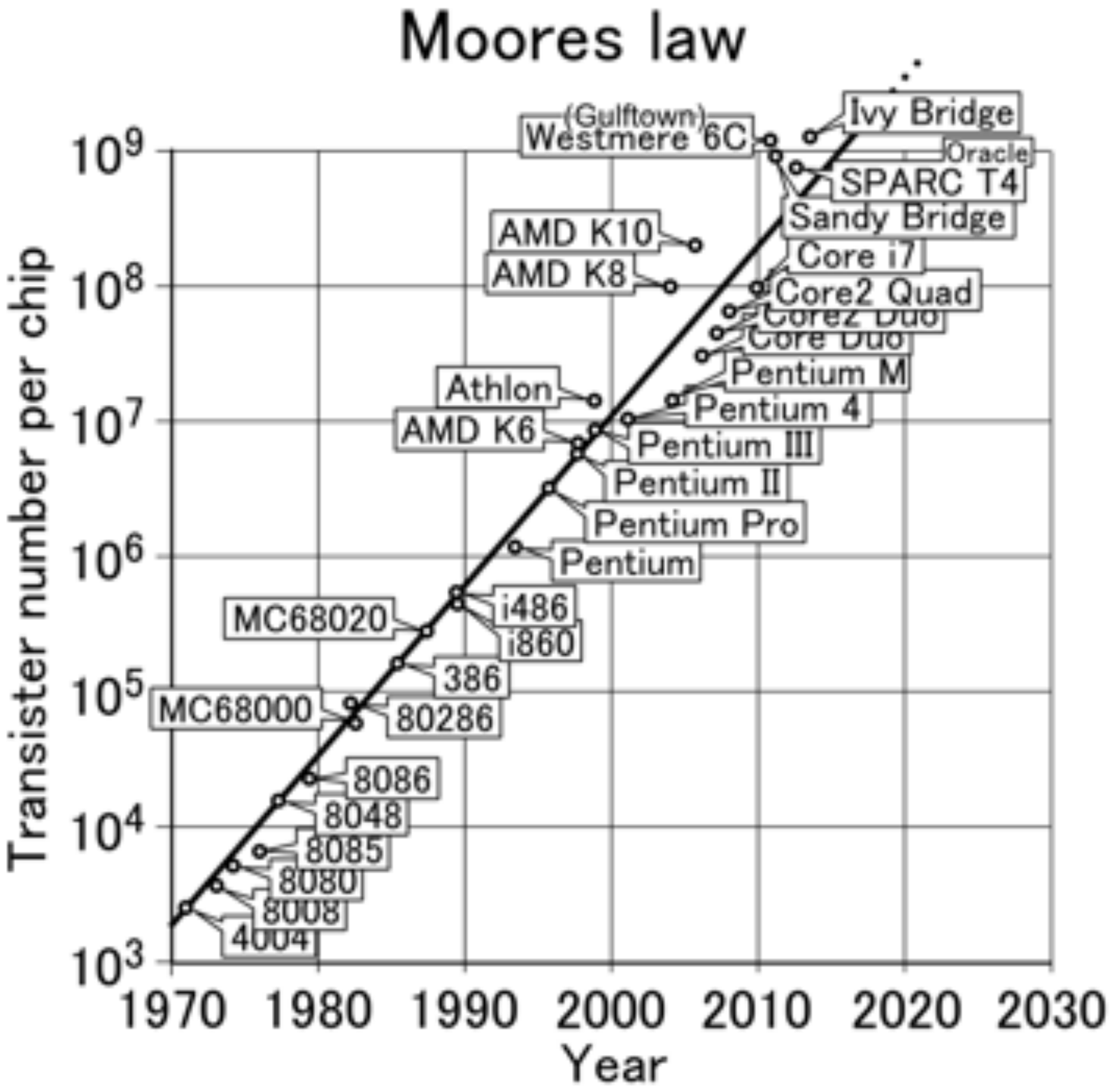


プロセスルール：回路の配線幅

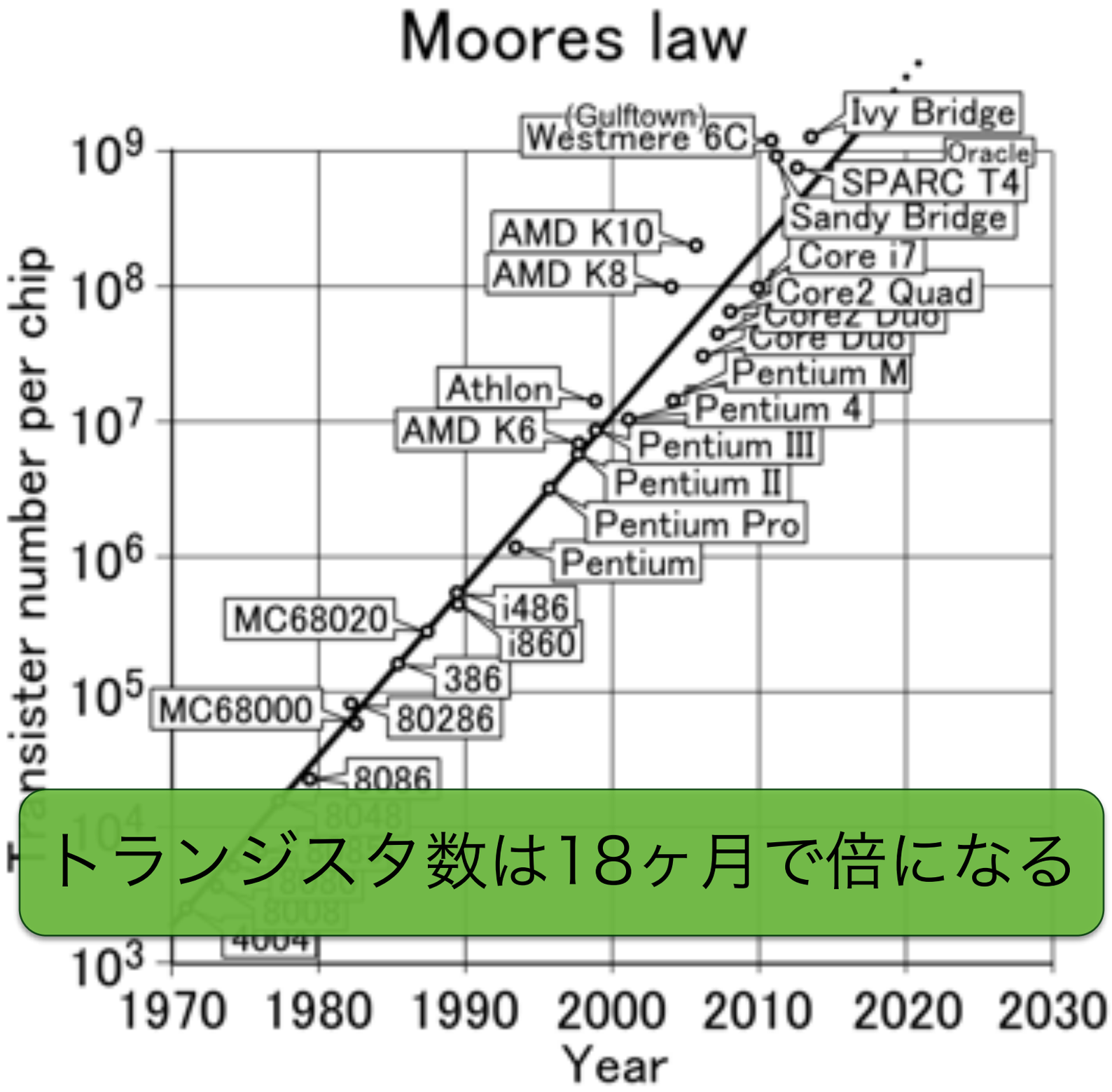
プロセスルールが小さい → より多くのトランジスタを作成可能

http://cpudb.stanford.edu/

ムーアの法則

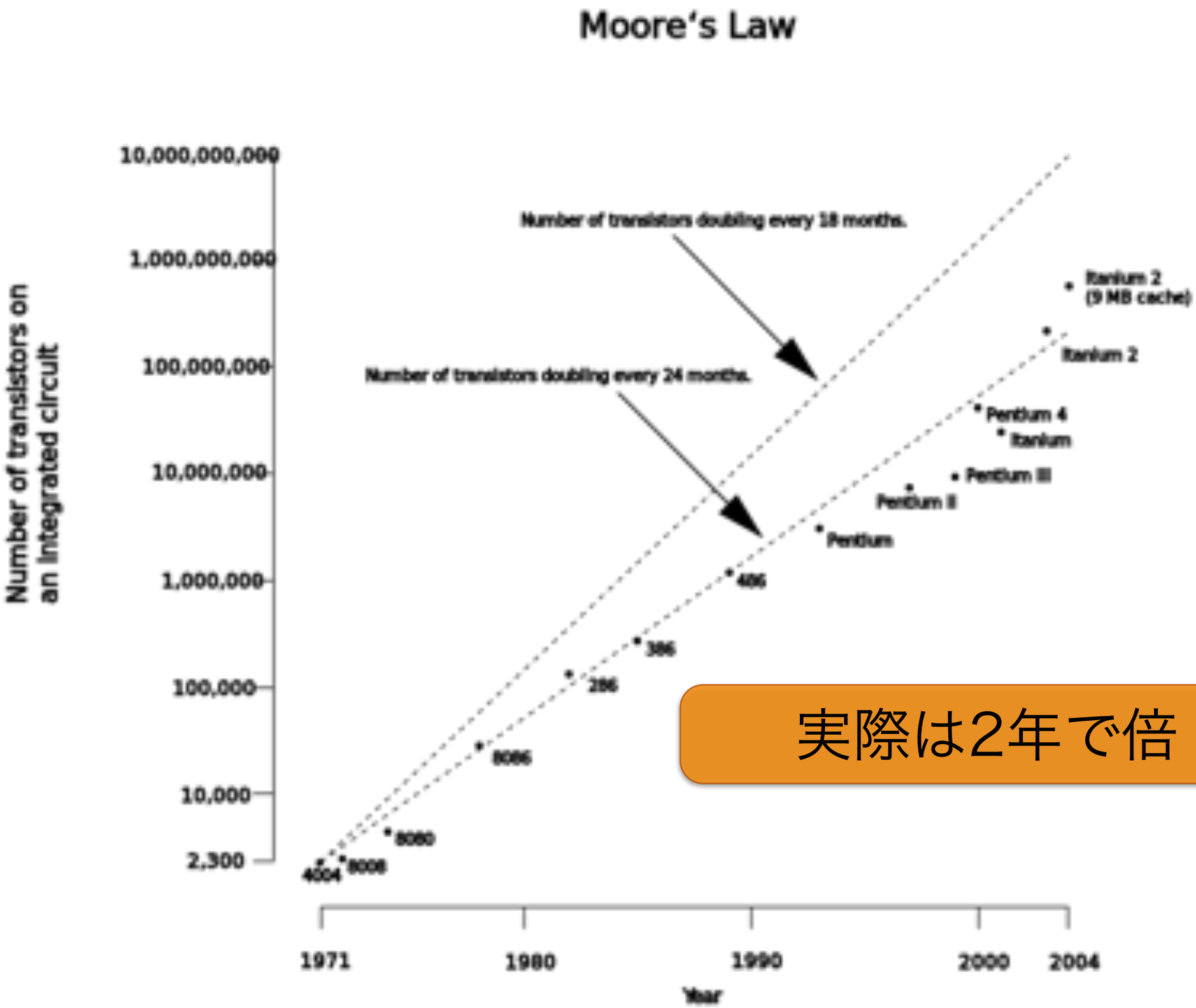

Moores law
Transister number per chip
Year
10^3
10^4
10^5
10^6
10^7
10^8
10^9
1970
1980
1990
2000
2010
2020
2030
4004
8008
8080
8085
8048
8086
80286
386
MC68020
i486
i860
Pentium
Pentium Pro
Pentium II
Pentium III
Pentium 4
Pentium M
Core Duo
Core2 Duo
Core2 Quad
Core i7
Sandy Bridge
SPARC T4
Oracle
Ivy Bridge
(Gulftown)
Westmere 6C
MC68000
AMD K6
Athlon
AMD K8
AMD K10

ムーアの法則



トランジスタ数は18ヶ月で倍になる

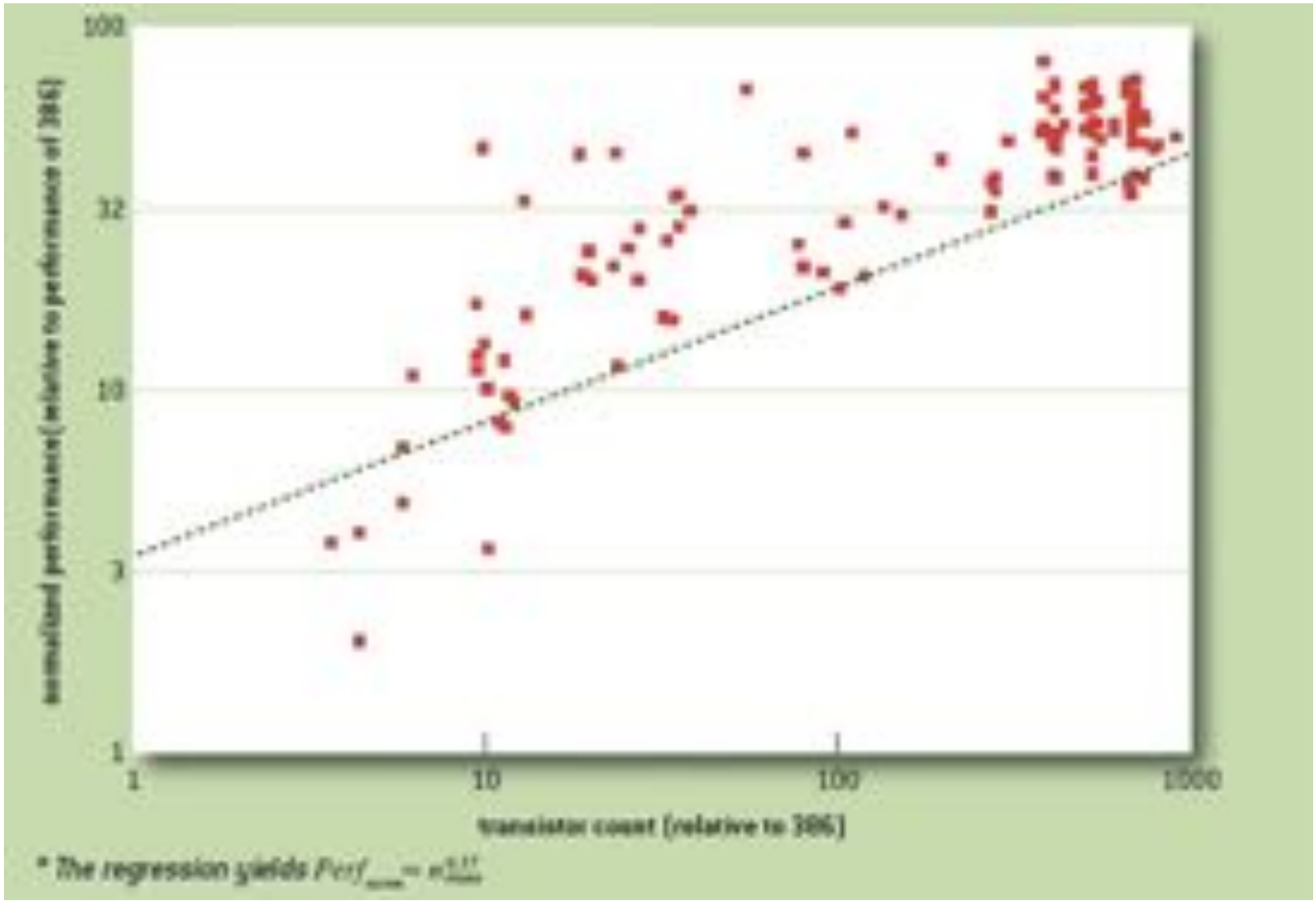
ムーアの法則



実際は2年で倍

Clayton Hallmark

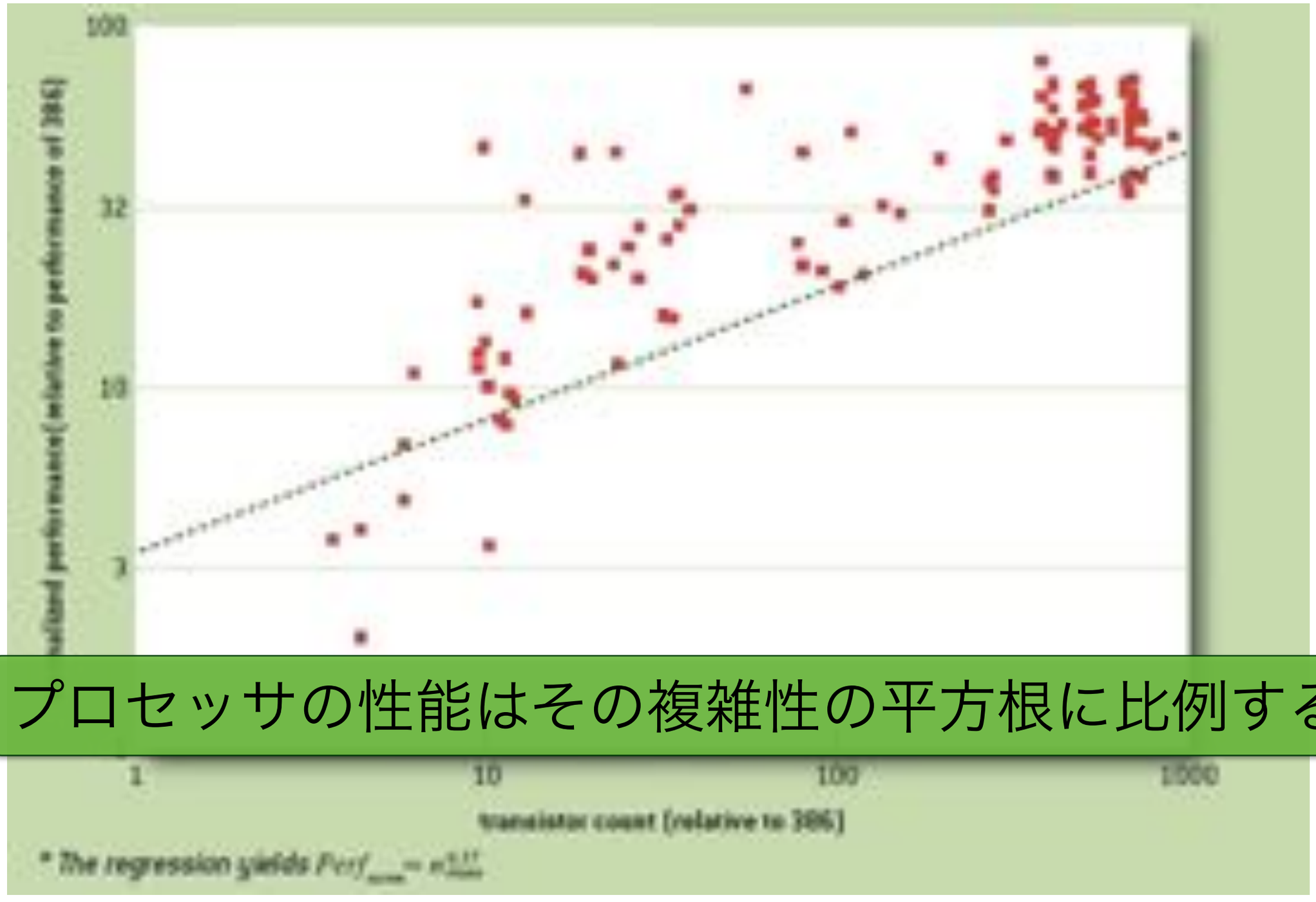
ポラックの法則



「プロセッサの性能はその複雑性の平方根に比例する」という経験則

http://cpudb.stanford.edu/

ポラックの法則



プロセッサの性能はその複雑性の平方根に比例する

http://cpudb.stanford.edu/

CPUの進化

- 最近のCPUでは周波数による性能の向上は見られない
- トランジスタ数は増加し続けている
  - トランジスタ数を倍にしても性能は約1.4倍
  - トランジスタ数を倍にすると消費電力も倍

↓

マルチコア化

- 発熱・消費電力の低減
- トランジスタの有効活用

スーパーコンピュータ(ベクトル型)

ベクトル型計算機

- 一つの命令で複数のデータ配列の演算を行う
- 命令読み出しの時間を短縮できる

スーパーコンピュータ(スカラ型)

多数のスカラ型CPUを
ネットワークで繋ぎ一つの計算機とする

スカラ型CPUの価格低下 → 現在のスーパーコンピュータの主流

大規模もしくは高速な計算を行う場合、
スーパーコンピュータ向けのプログラミングが必要

並列化

並列化レベル

- マシンレベル並列
- プロセスレベル並列
- スレッドレベル並列
- ベクトルレベル並列
- 命令レベル並列

プログラミングモデル

- メッセージパッシングモデル
  - データ空間と処理部を分割
  - 分散メモリ型計算機
  - ex. MPI
- データパラレルモデル
  - データ空間を共有
  - 共有メモリ型計算機
  - ex. VPP Fortran, OpenMP

並列化

並列化レベル

- マシンレベル並列
- **プロセスレベル並列**
- **スレッドレベル並列**
- **ベクトルレベル並列**
- 命令レベル並列

プログラミングモデル

- メッセージパッシングモデル
  - データ空間と処理部を分割
  - 分散メモリ型計算機
  - ex. MPI
- データパラレルモデル
  - データ空間を共有
  - 共有メモリ型計算機
  - ex. VPP Fortran, OpenMP

ベクトルレベル並列

- 旧来のベクトルプロセッサと同様に、一つの命令で複数のデータを処理する
  - SIMD演算、GPGPU等
- トランジスタ数に余裕ができ、スカラ型プロセッサに搭載されるようになってきた

プログラミング方法

- コンパイラによるベクトル化
  intrinsic命令等
- 言語拡張
  Cilk, OpenMP 4.0, OpenACC

スレッドレベル並列

- 基本的にCPUのコアごとに計算が実行される
  - 異なる命令を同時に実行できる
  - データパラレルモデル
- トランジスタ数に余裕ができ、1CPU当たりのコア数が増加している

プログラミング方法

- TBBなどのライブラリを利用
- 言語拡張
  Cilk, OpenMP

プロセスレベル並列

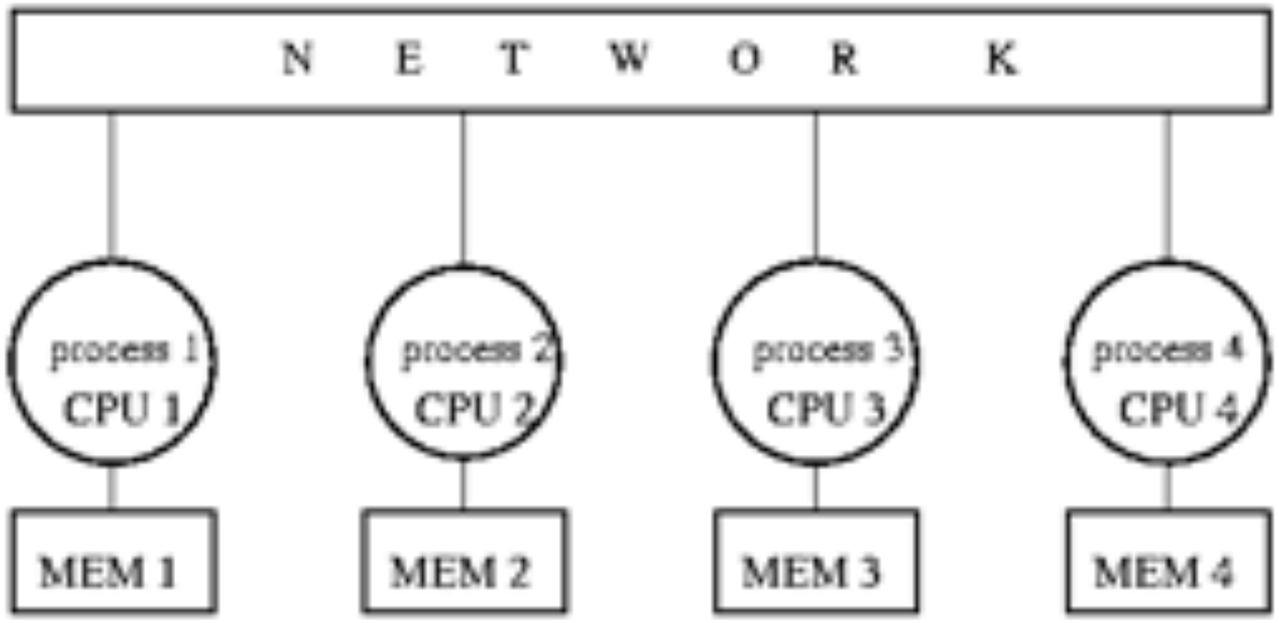
- スーパーコンピュータでのプログラミングでは主にこの部分の並列性を考える
- スカラ型スーパーコンピュータは分散メモリ型が主流
- メッセージパッシングモデルによるプログラミングが
- 行われている

プログラミング方法

- 言語拡張

  MPI, PGAS(Coarray Fortran, UPC)

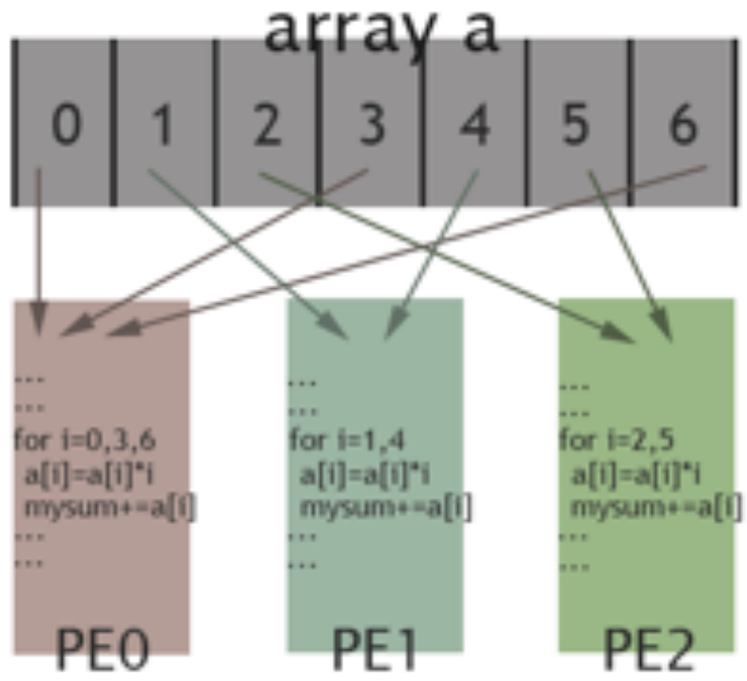
Message Passing Interface(MPI)



CPU間のデータを互いにやり取りするための規格

- ライブラリとして提供されているため、
  言語を問わず利用可能
- 一方、データの通信について利用者が考えなければならない

Partitoned Gloabal Address Space (PGAS)



分散メモリを共有メモリに仮想化し単一のデータ空間として利用する

- データの通信を考えなくてよいので、プログラミングが簡単
- 使用できる計算機が限られている

Unified Parallel C (UPC)

C言語を拡張したPGASモデルのプログラミング言語

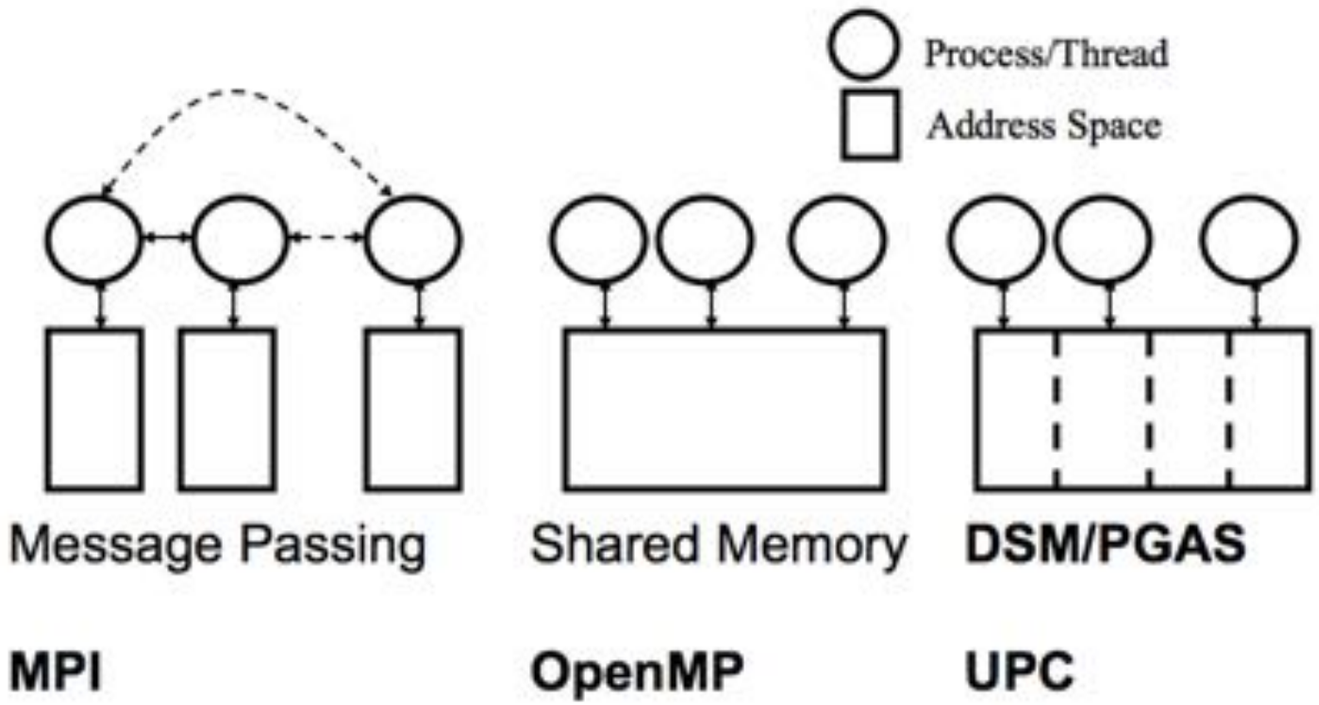
W. W. Carlson et. al. 1999

変数を共有化されたメモリ空間に割り当て、
どのプロセッサからでも参照できるようにする

データは物理的には個々のプロセッサに割り当てられる

C言語を拡張して並列性についての機構を加えた言語

Partitioned Global Address Space (PGAS)



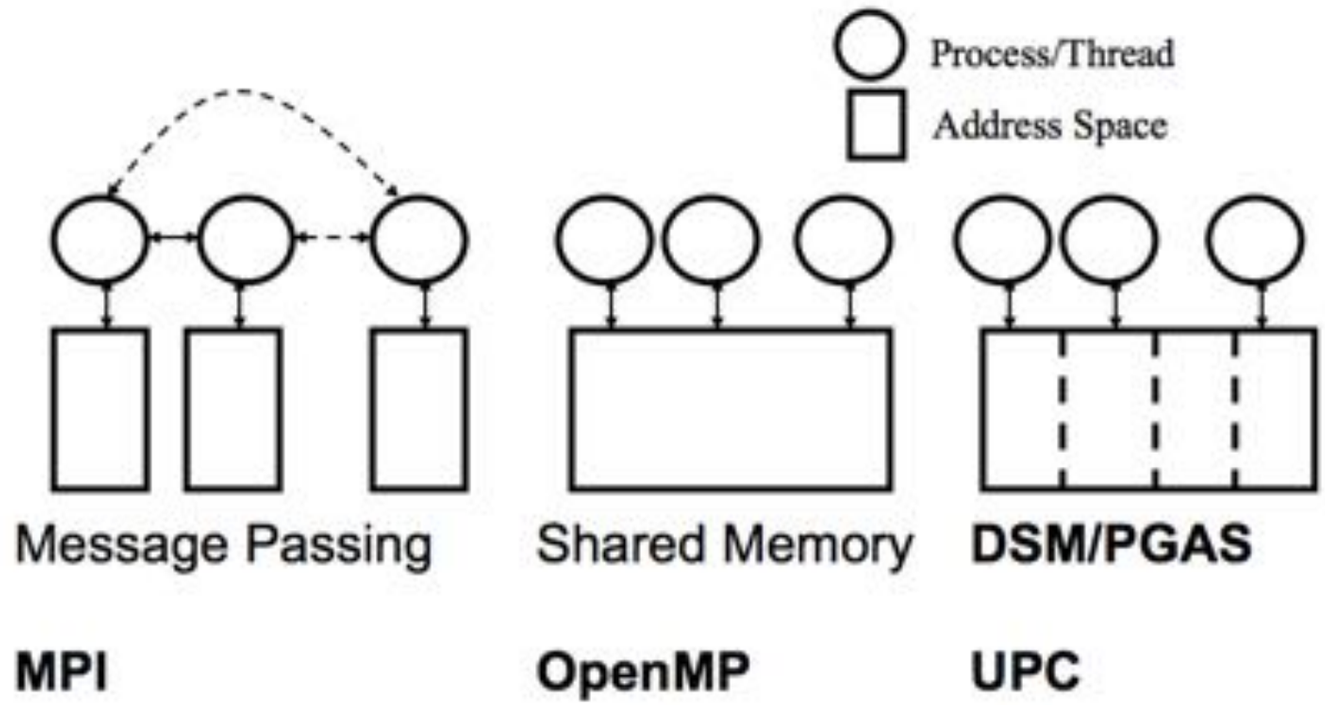
PGAS：区分化大域アドレス空間

物理的には各プロセッサにデータが存在するが、

仮想的な共有アドレスを使ってアクセスできる

例:UPC, Co-array Fortran

MPIとの比較



MPIでは明示的にデータの送受信を行う必要があるが、
UPCはどのプロセッサからでも共有メモリ空間に
アクセスできる

一方、MPIがライブラリとして提供されているのに対し
UPCは言語であるため使用できる計算機が限られる

まとめ

- 衝突系シミュレーションでは高精度時間積分法を用いることでタイムステップが小さくなることを防ぐ。
- 独立時間刻みを用いることで近接遭遇している粒子以外のタイムステップは小さくしなくてすむ。
- 相互作用を工夫することで、N 体シミュレーションで最も計算に時間がかかる重力計算を軽くしている。
- 大規模な計算を行うには並列計算が必須