計算機の話

牧野淳一郎

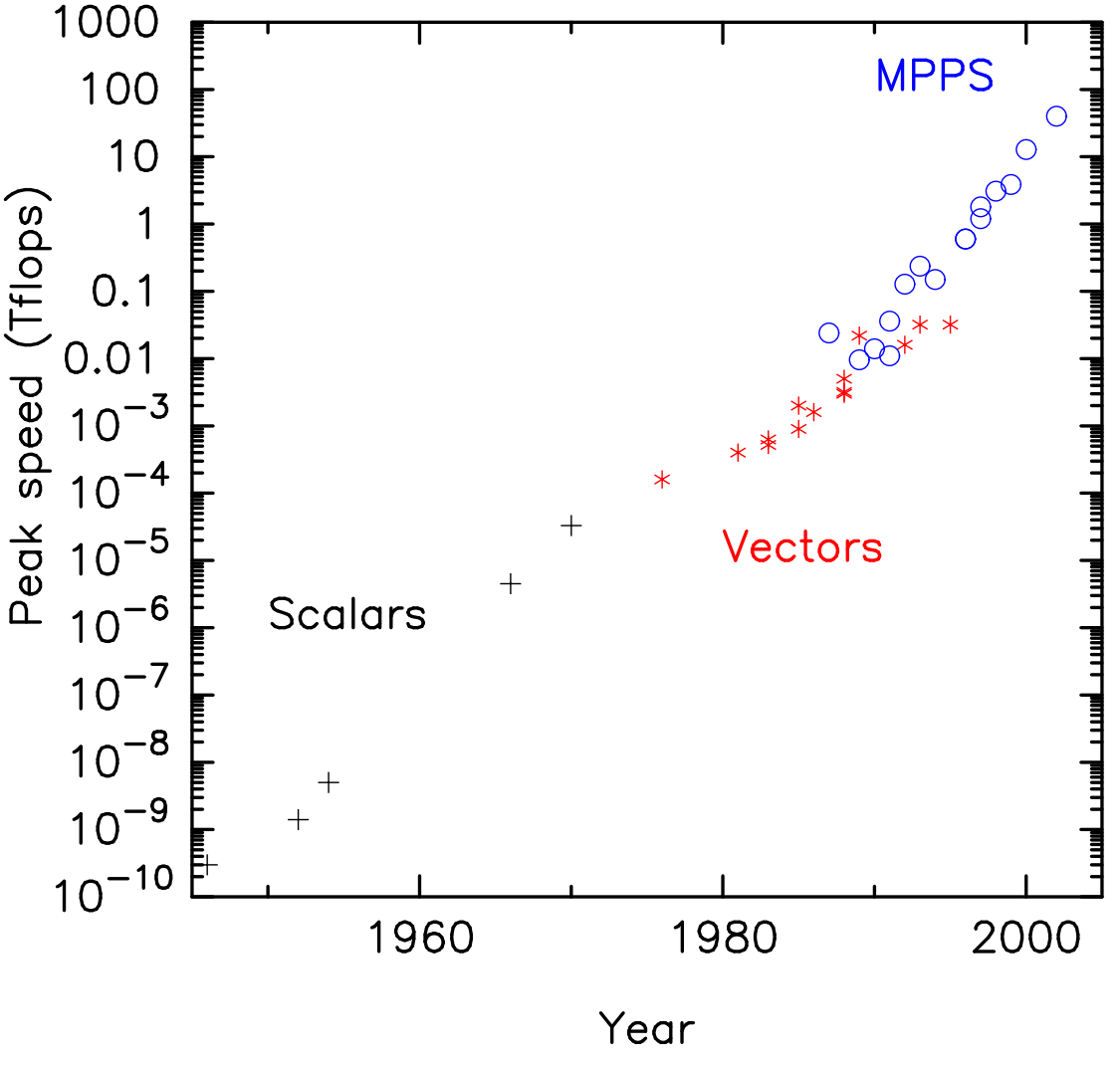
今日の話の概要

自分でちゃんと研究したネタがないので昔話

計算機の発展

速度の進化: 50 年で 10^{10} 倍

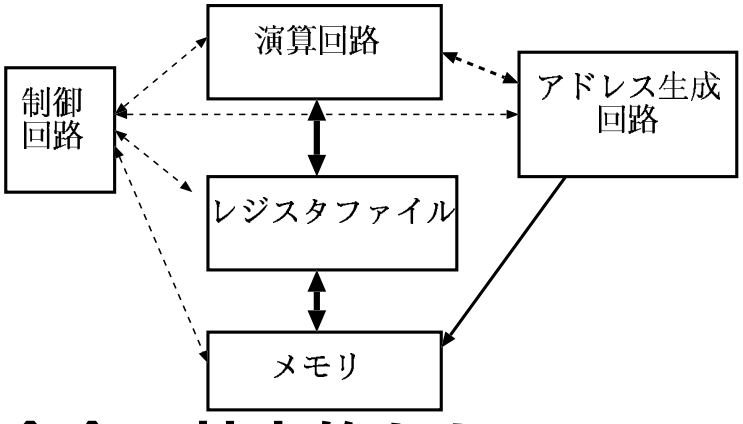
ほぼ時間の指数関数



指数関数的進化を可能にしたもの

1. ムーアの法則:トランジスタの大きさ: 3 年で半分
   - トランジスタの数: 4 倍
   - 速度: 2 倍
2. 計算アーキテクチャの革命
   スカラー計算機 $\rightarrow$ ベクトル $\rightarrow$ 並列

スカラー計算機: 1946–1975



要するに普通に「計算機」だと思うも
の。
「プログラム」があって、それを 1 命
令づつ実行する。
「フォン・ノイマン」アーキテクチャ

命令は基本的なもの:
load: メモリからレジスタへ
store: レジスタからメモリへ
演算: レジスタからデータを演算器に、結果をまたレジスタに
分岐: 次に実行する命令のアドレスを (条件により) 変更

スカラー計算機を速くする方法

- クロックを速くする
  - トランジスタの動作速度
  - 配線遅延
- 命令実行に必要なクロック数を減らす
- 複数の命令を同時に実行する
  - パイプライン化
  - 「スーパースカラー」

必要クロック数を減らす

(数値計算には) もっとも重要なのは乗算
計算機の内部で乗算する方法: 基本的には人間が筆算するのと同じ

```
          101
        x 101
-------------
          101
        101
------------
        11001
```

2進数で計算
かけられる数と、かける数の各ビットとの積を作って、シフトして足していく。

乗算を速くする

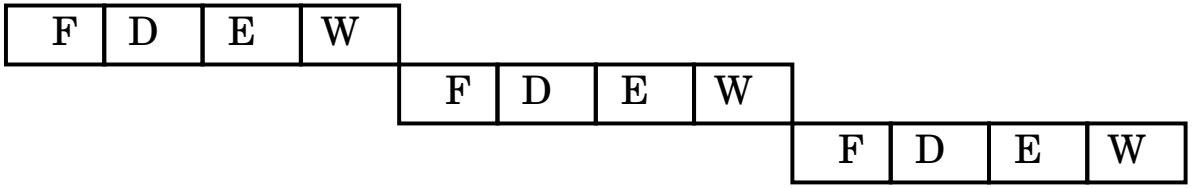
普通の (昔の計算機): 32 ビットなら、32 ビットの加算器を 1 つ使って、32 回足し算をする。
速くする: 32 個の部分積を一度に作って、「一度に」足す (トーナメント方式で足す)。
演算器は 32 (細かくいうともうちょっと多い) 個
全部加算するのにかかる時間: $\log_2 32$ くらい。
だいぶ速くなる

パイプライン化

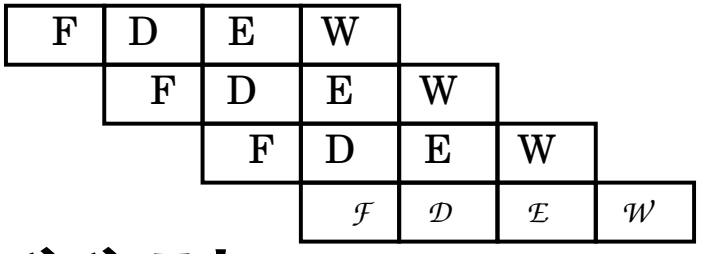
「一つの命令を実行」:実際にはいろんなことをしている。例えば:
命令読み込み (フェッチ F)
命令解釈 (デコード D)
命令実行 E
結果の書き戻し W
それぞれに 1 クロック使うと 4 クロックかかる。
これらはそれぞれ別の回路: 一つの命令を始めた次のクロックで次の命令を始めれば速くなる。

パイプライン化

非パイプライン



パイプライン



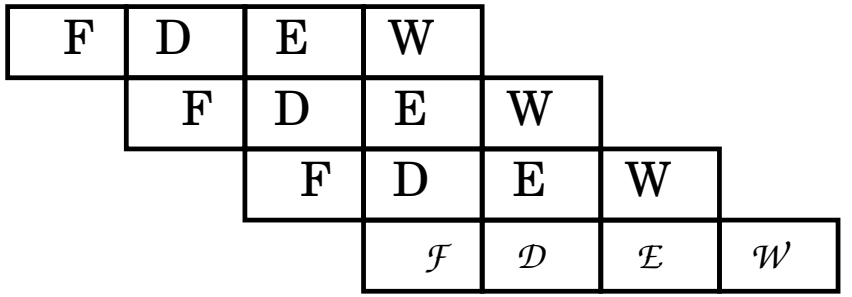
分岐するとややこしい

スーパースカラー

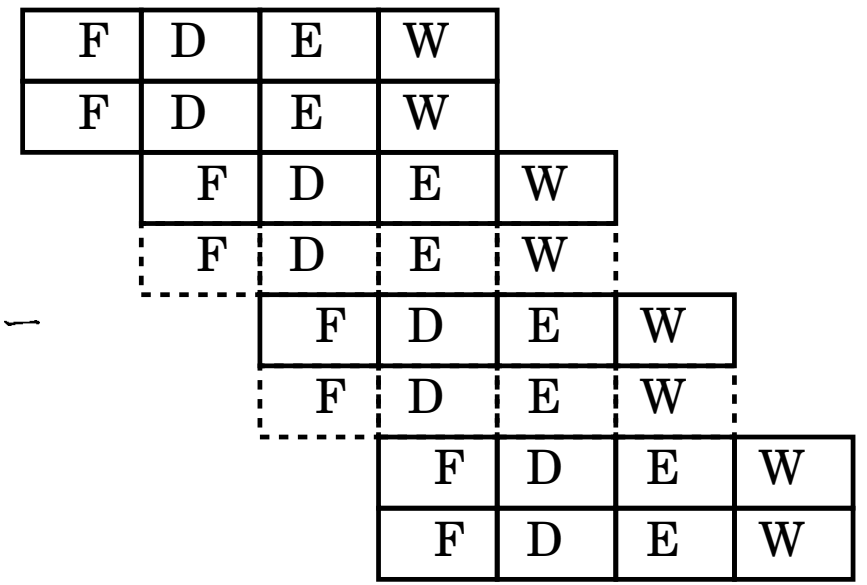
フェッチ・デコード：複数命令を同時に処理
実行ユニットは複数準備
「同時に実行できる命令があれば」同時に実行

スーパースカラー

パイプライン



スーパースカラー



もうちょっと速くする

- パイプラインの段数を増やして、その分クロック周波数をあげる
  - Intel P4 では20段から31段
- 命令を実行する順番を、意味が変わらない範囲で実行中に適当に変更
  Out-of-order execution
- 命令自体を再解釈して、パイプライン化/並列実行しやすくする

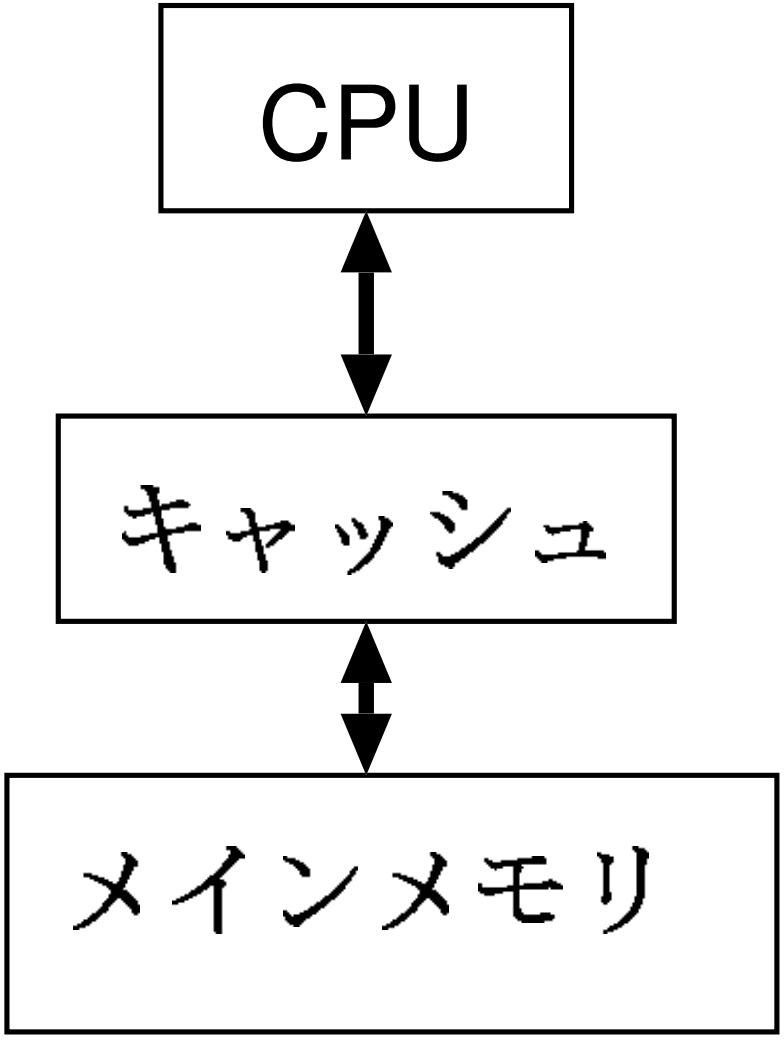
実行速度の制限要因: memory wall

今までの話: 演算命令とかの実行は速くできる
速くするのが難しい: メモリからの読み込み (load)

- DRAM のサイクルタイムはあまり速くない
- CPU とメモリの間のデータ幅 (信号線の数) はあまり多くできない

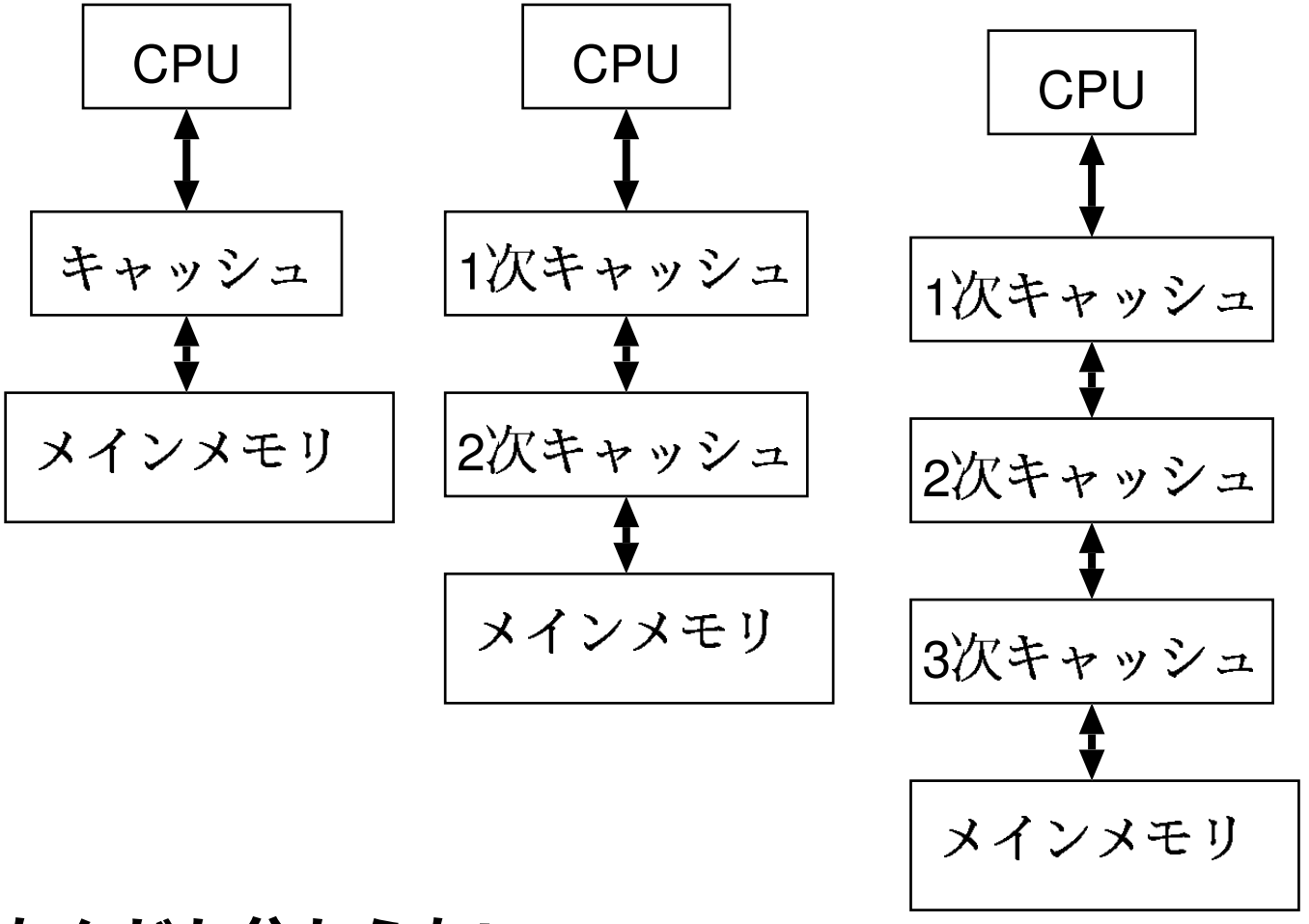
例:
Intel P4, C2D データ 64 bit, クロック 1066 MHz (DRAM チップ内部はもっと遅い)
AMD K8 データ 128 bit, クロック 667 MHz
CPU のクロックに比べると 5-10 分の 1

姑息な対応: キャッシュメモリ



CPU とメモリの間に、容量は小さ
けど速いメモリを置く (キャッシュメ
リ)。
適当なアルゴリズムで、最近使っ
データをキャッシュに残す。
データがキャッシュにあればそっちを
う。

キャッシュの「進化」



なんだか分からない。

キャッシュによる対応の限界

本当にメインメモリにアクセスする必要があればどうしようもない。

```
do i=1,10000000
      a(i) = b(i) + c(i)
enddo
```

こういうのを速く処理するにはメモリを速くするしかない。
いわゆる「スーパーコンピュータ」:この問題があらわになったのは 1970 年前後 (CDC 7600)

ちょっと余談

最近のマイクロプロセッサの「新技術」:大抵は、いわゆる「スーパーコンピュータ」に 30 年くらい前に導入されたもの

- スーパースカラー CDC 6600
- load-store アーキテクチャ (CDC)
- キャッシュ IBM 360/85
- OOO 実行 CDC 6600

真似っこというわけではなく (そういう面もあるが)、技術的な要請が同じようなものであるため
匡体一台 — チップ一つ
という大きさの変化だけ

メモリを速くする

遅いものは速くできないので、沢山並べて

- 順番にアクセス (マルチバンク)
- データ幅を広くする

「ベクトル計算機」の考え方:沢山並べて、並列にアクセスする: ずーっと先まで次に何をするかがわかっていれば簡単
キャッシュでは具合の悪い

```
do i=1,10000000
      a(i) = b(i) + c(i)
enddo
```

みたいなのが理想的

ベクトル計算機

単純なレジスタの代わりに「ベクトルレジスタ」
例えば要素数 64 のベクトルを 8 本とかしまう。

- メモリ—ベクトルレジスタ転送
- 演算

の両方を、「ベクトル」単位で実行 $=$ ベクトル命令
ベクトル命令が使えるようにプログラムを書くことが出来れば速い

スカラー、ベクトル計算機の発展

これまでみた計算機の構成の発展: ハードウェアがどんどん大がかりになる方向
何故そんなことが出来たか: ムーアの法則のおかげで使えるトランジスタ数がむやみに増えたため。
8086 : 10^4 トランジスタくらい。
P4 : 10^8 に近づく。
しかし、いろいろ限界、、、、
限界の具体現れ: 並列計算機の発展

ベクトル計算機の高速化

最初の実用ベクトル計算機: Cray-1 (1976)
使えるトランジスタはもっとどんどん増える。

- **ベクトル演算パイプラインを沢山付ける**
  CDC Cyber 205, **初期の日本の機械**
- CPU+**ベクトル演算パイプライン の組合せ自体を沢山付ける**
  Cray-2, Cray-(X/Y)MP

メモリ構成

前のどちらの方向も「共有メモリ」の構成をとった。
メモリのどこにあるデータも。どの CPU (あるいはパイプライン) からでも同じ速さで読み書きできる。
言い換えると: メモリと演算器がクロスバースイッチみたいなものでつながる。

- 長い配線が避けられない
- スイッチのハードウェア規模、遅延が大きくなる

性能をあげるのが難しくなる
共有メモリ: 16-32 CPU くらいが物理的な限界?
1980 年代後半から 90 年代になって、ベクトル計算機の性能向上は遅くなった。

並列計算機

並列計算機といっても、共有メモリならベクトル計算機と同じ問題が発生。

共有メモリにしない $\rightarrow$ 問題が (とりあえず) 回避される。
つまり: $1\ \mathrm{CPU} +$ メモリの簡単な計算機を構成要素にして、それを沢山並べる。
その間は、なんか適当なネットワークとかでつながる
という方向だと、ややこしい問題を (計算機を作る人は) 考えなくてもいい。
$=$分散メモリ型並列計算機

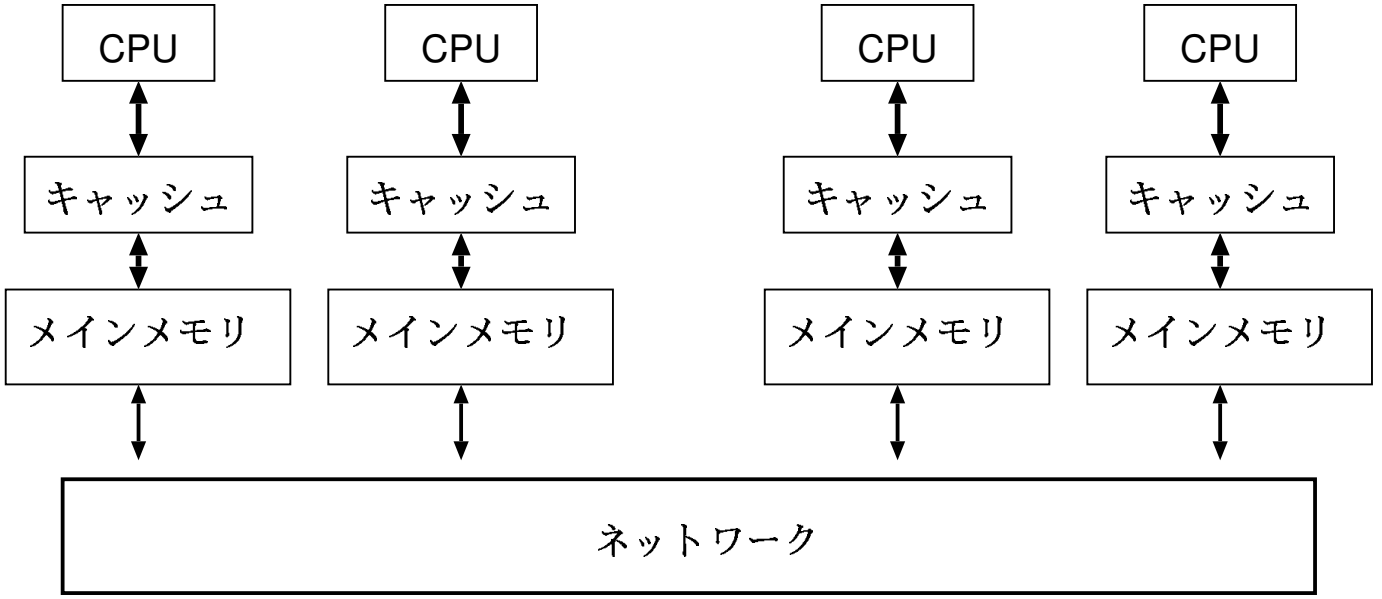
分散メモリ型並列計算機

開発の重要な源流は

- QCD 計算のための計算機 (Caltech Hypercube)
- 星野-川合らによる PACS/PAX
- イリノイ大学の ILLIAC IV

どれも、割合単純な連続系 (偏微分方程式系) を解くことだけを目標に設計・製作された。

分散メモリ型計算機の構成



どれもおなじようなもの。ILLIAC IV は特殊: SIMD 型 (全部の CPU が同じ演算を実行)。
あとの流れは MIMD (みんなバラバラに計算する)
分散メモリ型の中でも特に台数が多いもの: Massively-Parallel とか Highly-Parallel と呼んでいた。

分散メモリ型計算機の歴史

1970 年代: 汎用マイクロプロセッサを使った計算ノード
1980 年代: 多様

- **汎用マイクロプロセッサ/専用プロセッサ**
- SIMD/MIMD
- **分散メモリ/分散共有メモリ**/AllCache/...

分散メモリ型計算機の歴史 (2)

1990 年代: 淘汰

- **ベクトル型分散メモリ— 富士通、NEC**
- **Cray T3x — Alpha**
- **Beowulf — Intel x86**

2000 年代: ???

- **Beowulf、SMP クラスタ以外は死滅?**

死滅した理由

- 価格性能比
- プログラミングモデル

価格性能比

消費者マーケットに出るもの: 生産数が何桁も違う。
それだけ、開発費を投入して量産コストを下げることができる。
2002 年時点で:
生産数 $< 10^4$ 個のカスタム LSI: 開発費 数億、量産コスト チップあたり 5-10 万動作クロック 300-500 MHz
Intel Pentium 4: 開発費 ??? 売値: 安いものなら 1 万、動作クロック: 2GHz 以上
同じようなものを作ると、価格性能比で 10 倍は損

プログラミングモデル

多様なアーキテクチャ → プログラミングモデルへの要求:

- 移植性
- 性能の高さ

生き残ったモデル: message-passing
要するに: 各ノードでは普通にプログラムが走る。必要に応じて、通信ライブラリを使って通信する。
自動並列化とか並列処理言語とかそういったものには頼らない。
もっとも安直なハードウェアに有利な状況

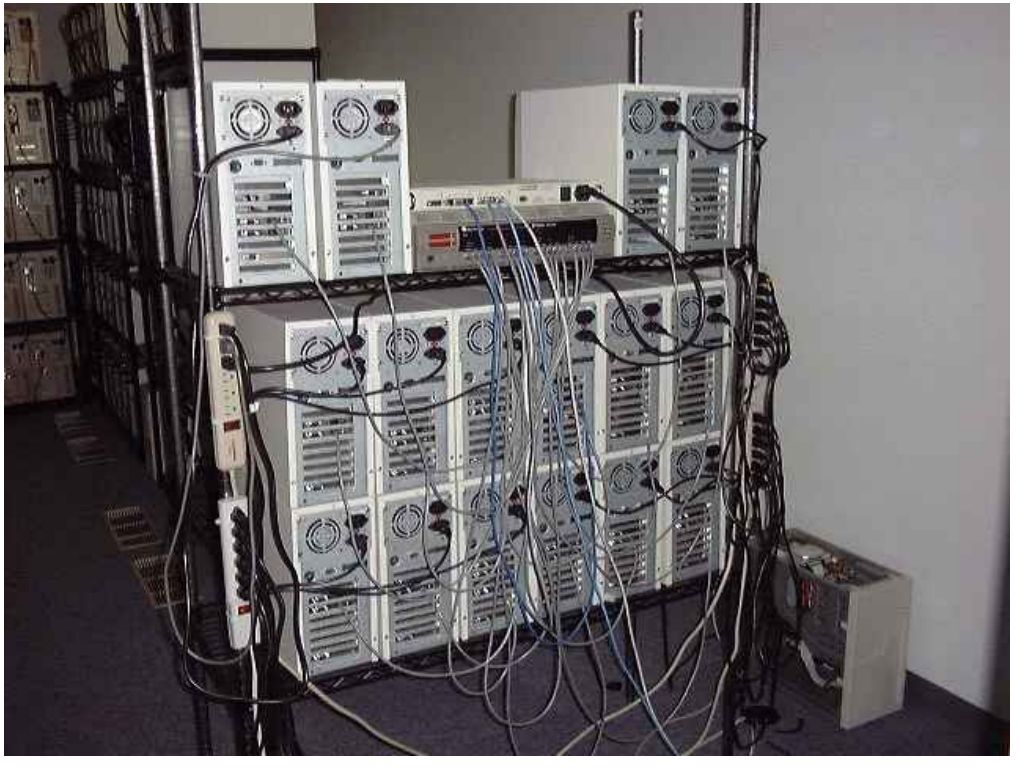
Beowulf とは？

- なるべく価格性能比のいいパソコンを沢山買う
- それを必要最小限の性能のイーサネットを使った普通のネットワークでつなぐ

それだけ。
安くあげる: OS, コンパイラにお金を使わない: Linux, gcc, MPICH

Early Beowulfs

http://beowulf.gsfc.nasa.gov/overview.html



Beowulf 上の並列プログラム

境界条件

- メッセージパッシングによるプログラム
- ネットワークを通したデータ転送
  - レイテンシは (特殊なハード、ソフトを使わないと) 遅い。 $> 100\mu$s
    「遅い」主記憶に比べてもさらに 3 桁遅い
  - バンド幅はどうやっても細い。今なら GbE で 100MB/s
    主記憶に比べてもさらに数十倍遅い

アルゴリズムに要求される条件
通信回数、通信量が十分に少ないこと。

まとめ

こんな話をもっと無限に長くだらだら書いた本を出すかもしれないので暇な人はそちらにコメント下さい。

多体シミュレーションの並列化

- 独立時間刻み
- ツリー・FMM

独立時間刻みの並列化

実はそれ以前の問題: 「全粒子から全粒子への力の計算」の並列化

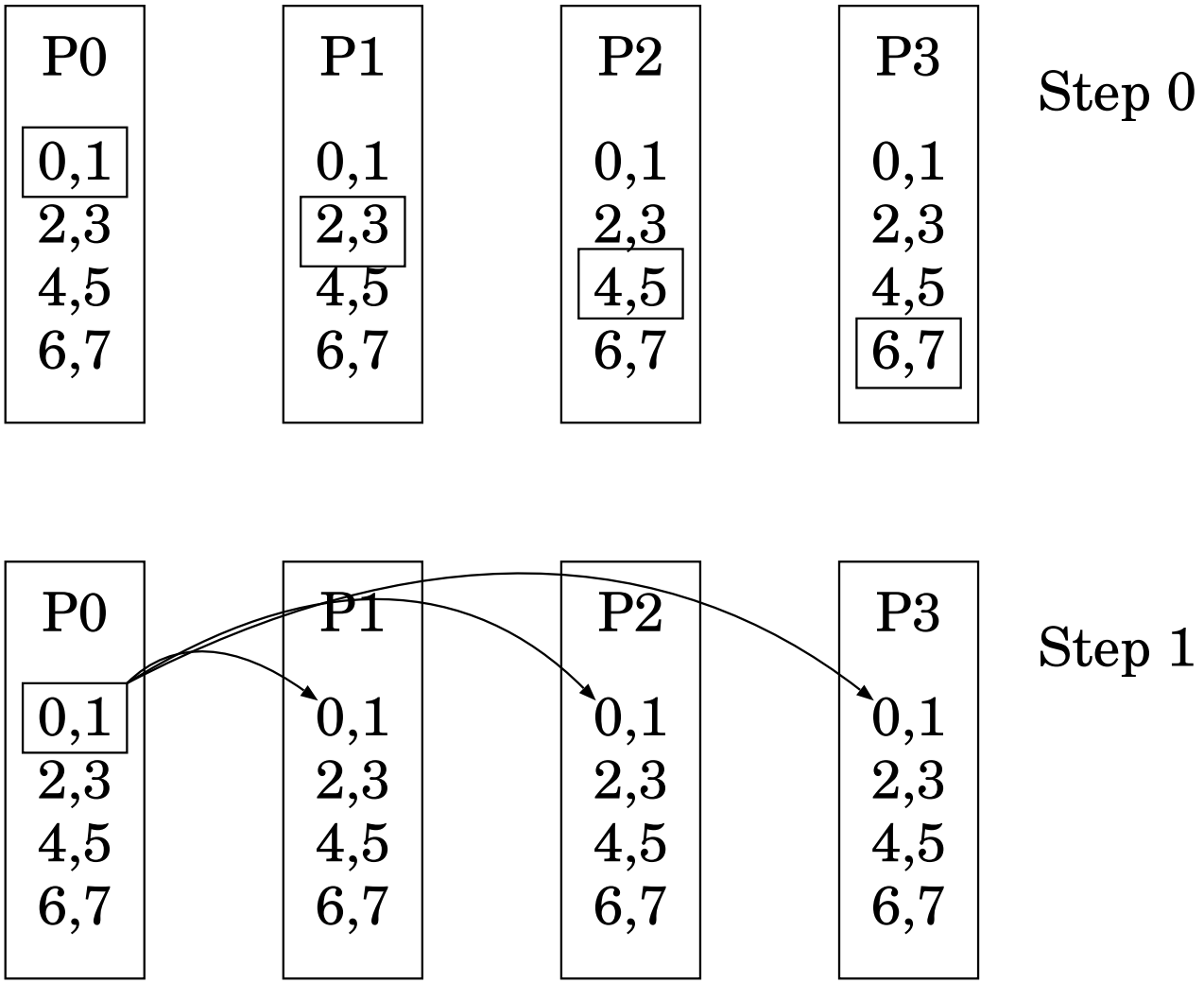
並列化の 2 方法

- 各ノードが全粒子のコピーを持って、重力計算、時間積分は分担する
- 各ノードが自分の分担の分だけを持つ。重力計算のためには他のノードから粒子を貰ってくる。

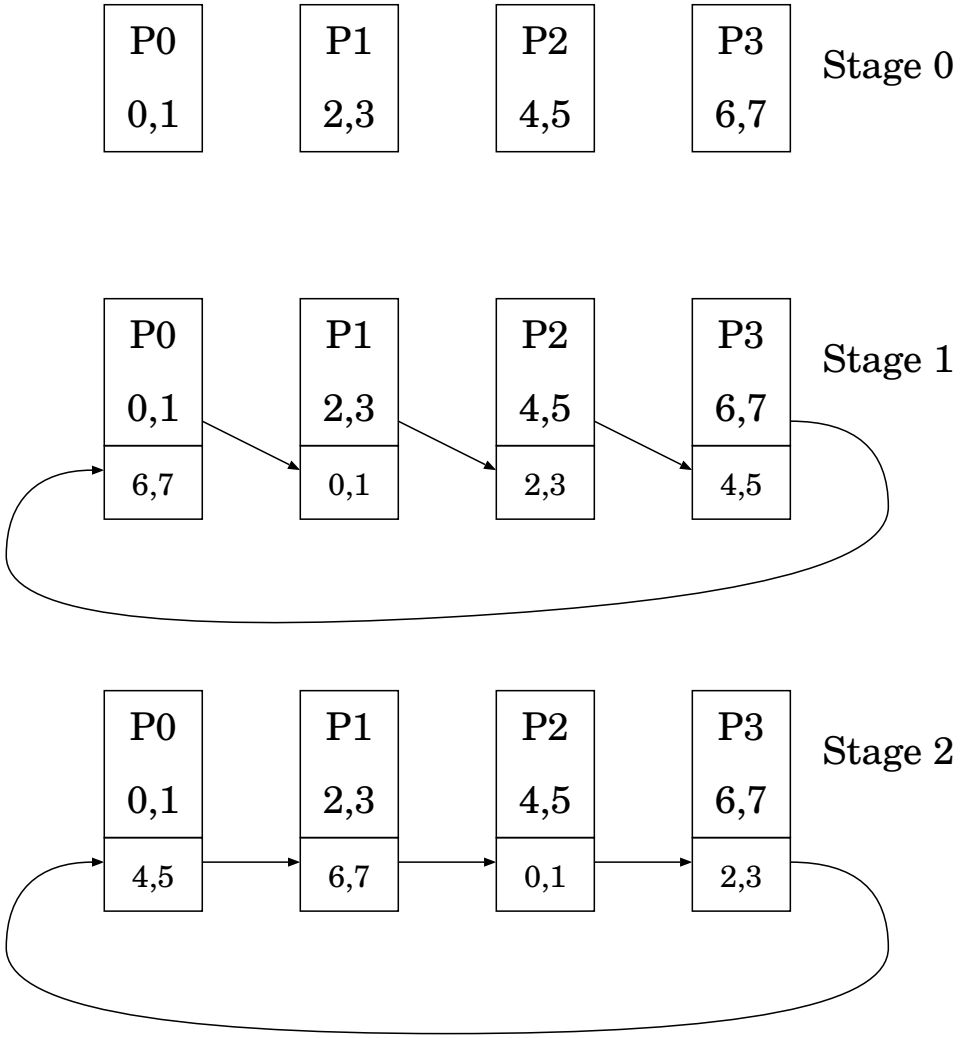
どちらにしても、各ノードが全粒子の情報を使って力を計算する。

ノード一つの通信バンド幅でスケーラビリティが制限される。

The copy algorithm



The ring algorithm



計算時間のスケーリング

N 粒子、p ノード

$$T_{\mathrm{ring}} = C_f N^2/p + C_c N + C_s p. \tag{1}$$

C_f　力の計算にかかる時間

C_c　1 粒子を転送するのにかかる時間

C_s　通信ライブラリのオーバーヘッド

通信時間は p によらない $\rightarrow O(N)$ ノードしか使えない。

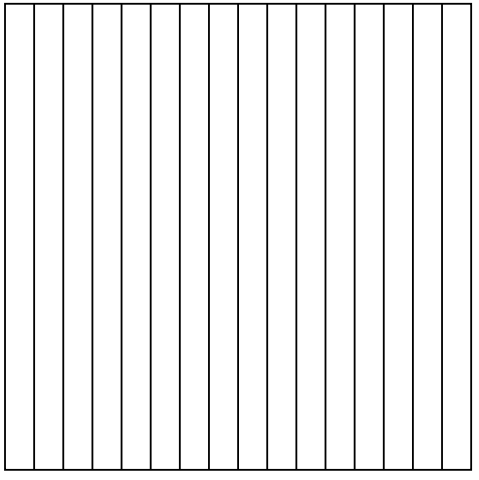
独立時間刻みの場合

- １ステップで動かす粒子数が減る
- 通信ライブラリのオーバーヘッドが無視できなくなる

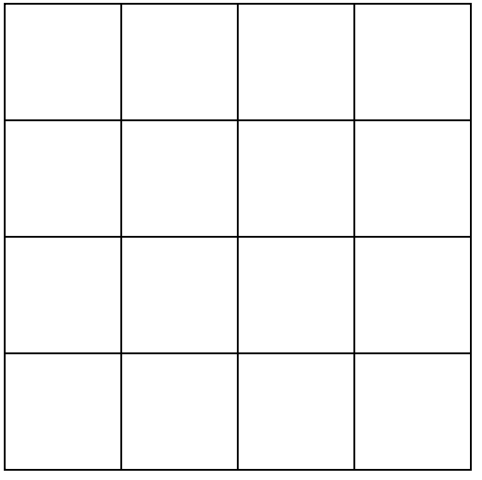
スケーラビリティが一層悪くなる

もうすこし違う考え方はないか？

近接相互作用の時（2次元の例）



1-D decomposition

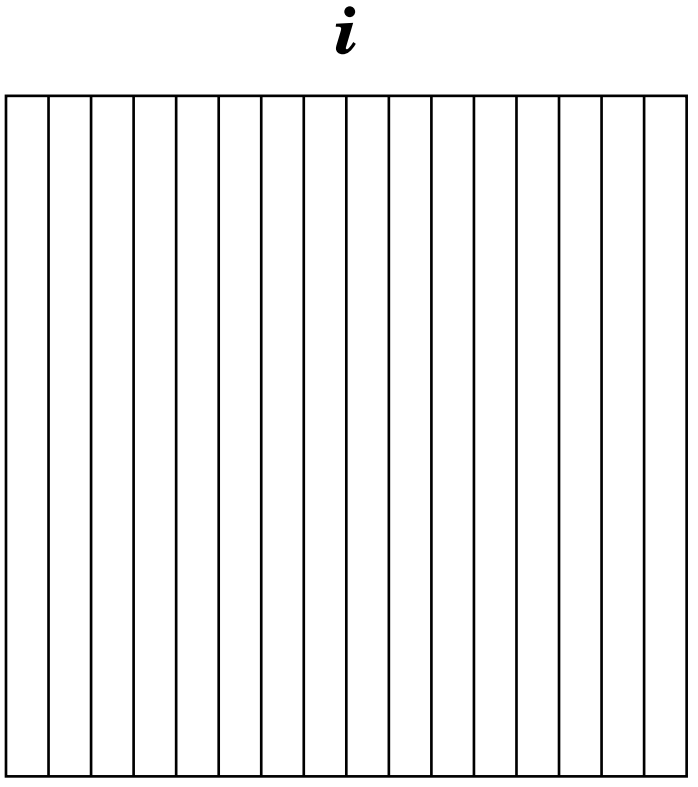


2-D decomposition

1次元空間分割：スケーラブルでない
ノード数 $\propto$ 1次元方向のメッシュ数

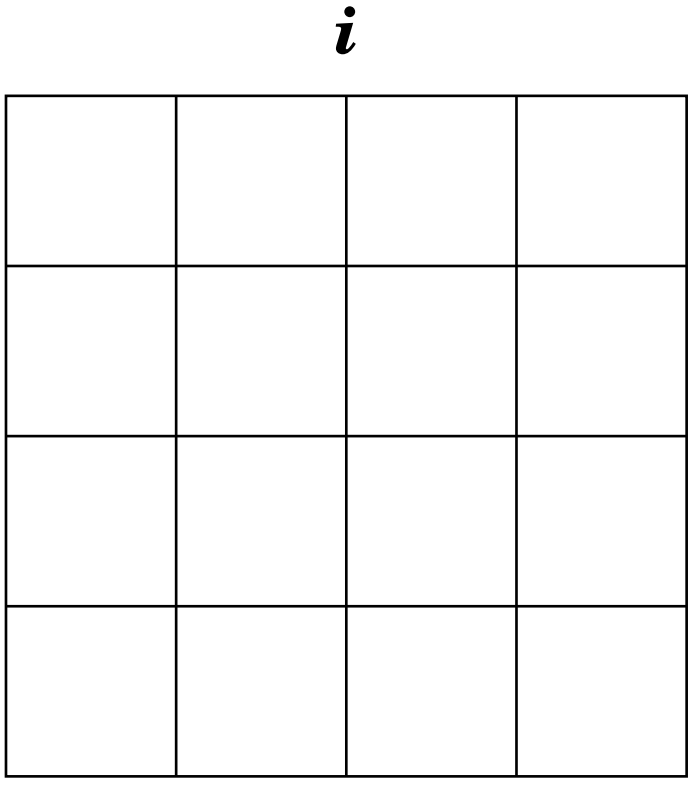
2次元空間分割：スケーラブル
ノード数 $\propto$ 全体のメッシュ数

遠距離相互作用 — 相互作用行列を考える



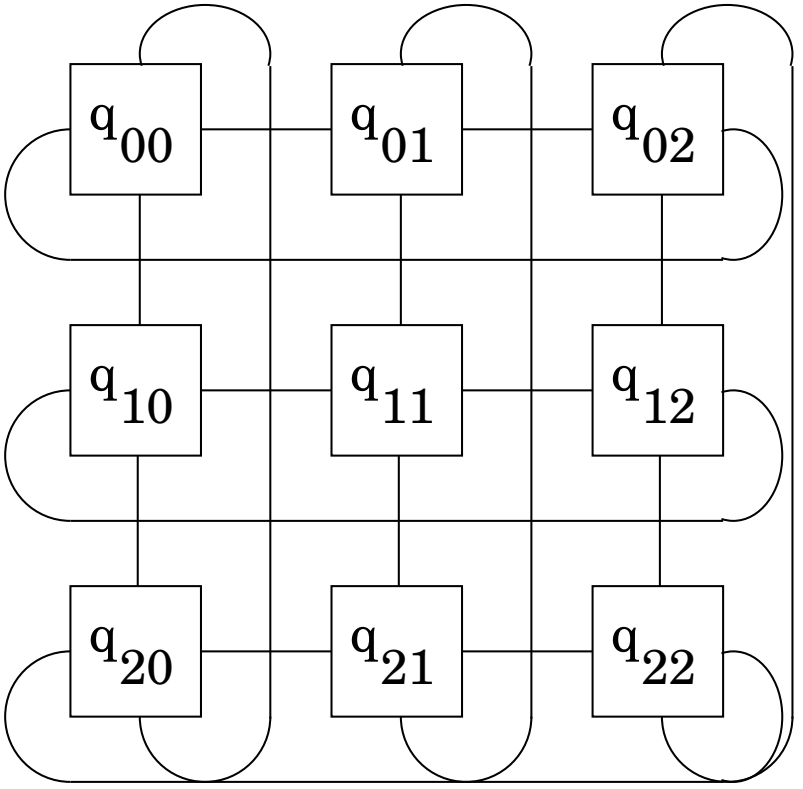
- 相互作用の行列を見ると、普通のアルゴリズムは1次元分割になっている。
- これがスケーラブルでない理由
- 2次元分割は可能か？

相互作用の2次元分割



- 形式的には左のような絵を書ける。
- 実際にはこれはなにを意味しているのか？

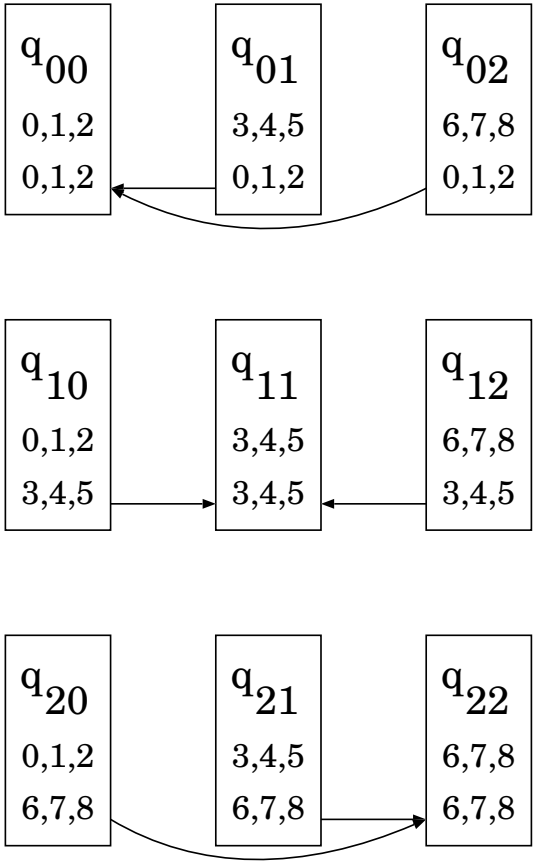
ハードウェアとしては、、、



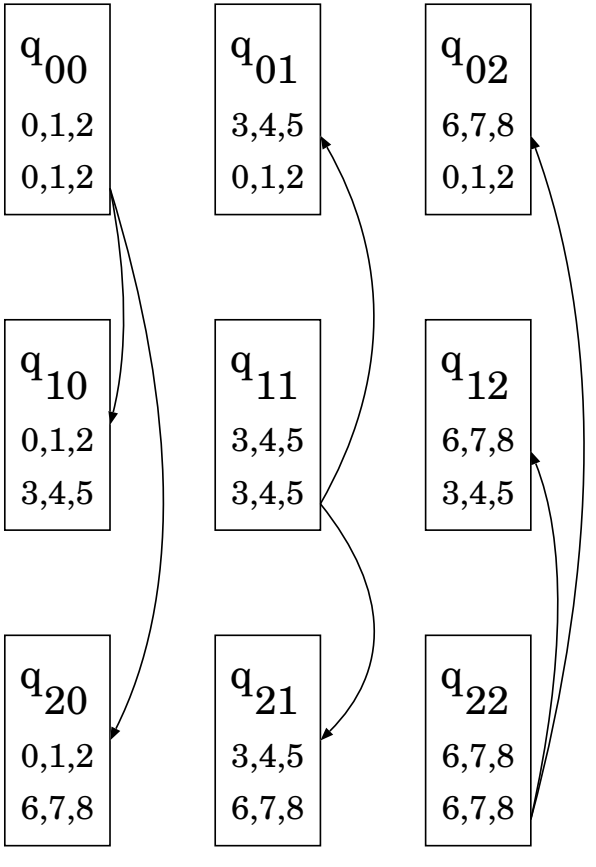
- 分割した相互作用行列にプロセッサを対応させる。
- トポロジ：2D トーラスなり「ハイパークロスバー」なり、、、

計算手順

Step 1



Step 2



横に総和、縦に放送

スケーリング（理論）

r^2 プロセッサ

$$T_{\mathrm{2Dbcast}} = C_f N^2/r^2 + 2(C_c N/r + C_s \log_2 r). \quad (2)$$

計算時間は $(N/r)^2$、通信は N/r になる。従って、 $r \sim N$、つまりプロセッサ数を N^2 にとれる。（スタートアップが無視できれば）

使えるプロセッサ数

- 効率が $50\ \%$ になるプロセッサ数

$$p_{\mathrm{half,2Dbcast}} \sim (NC_f/2C_c)^2 \tag{3}$$

(通信のスタートアップが無視できる場合) ここから先でも速度は向上する（プロセッサ数の平方根に比例）

独立時間刻みの時

$$p_{\mathrm{half,ind,2D}} \sim N^{5/3} C_f / [C_s \log_2 (N^{5/3} C_f / 2C_s)] \qquad (4)$$

大抵の場合にスタートアップ時間がスケーラビリティを制限する。
N^2 まではいかないが、 1 次元分割に比べれば圧倒的に良い。

こんなのは昔から知られてないか？

- N^2 個の（仮想）プロセッサを使うアルゴリズム
  Barnes and Hillis (1987?) に触れられている。
- Hyper-systolic algorithm (Lippert *et al.* 1998)
  1 次元リング + non-unit-stride communication
  なんだか難しい数学を使って通信パターンを最適化

同様の考え方自体は知られてはいた。
スケーラビリティ等は明確には議論されていない。

ツリー法のベクトル化、並列化

ベクトル化、並列化の技法はツリー法も FMM も同じ。

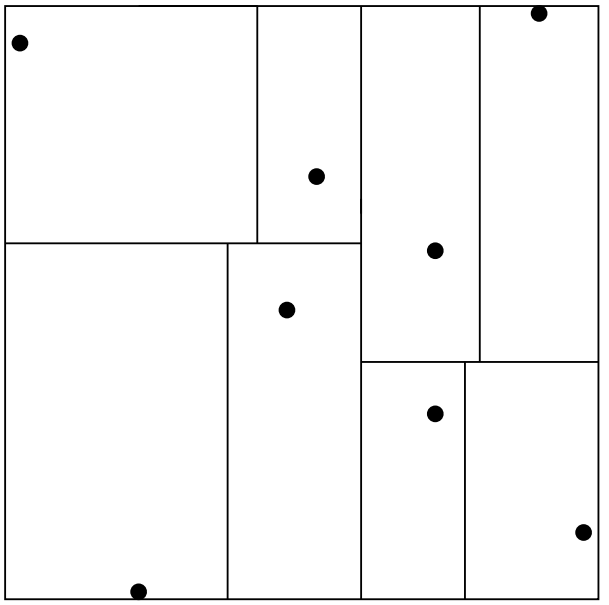
- 共有メモリの機械では容易（アルゴリズムの並列度は実際上 N）
- メッセージパッシングの機械：効率の良い空間分割が必要。Salmon & Warren が様々な方法をテストした。プログラムは面倒だが、作ってしまえば効率はよい。

従来の並列化アルゴリズム

実用になっている並列化法は以下の2つ。どちらももとCaltech Hypercubeのグループの Salmon と Warren によるもの

- Orthogonal Recursive Bysection (ORB)
- Hashed Oct Tree (HOT)

ORB



- まず、粒子が半分ずつにわかれるように x 軸に垂直な平面で切る
- 切ったそれぞれについて、また半分になるように y 軸に垂直な平面で切る 3 次元なら次に z、2 次元なら次は x 軸になる

というのを、プロセッサ数になるまで繰り返す

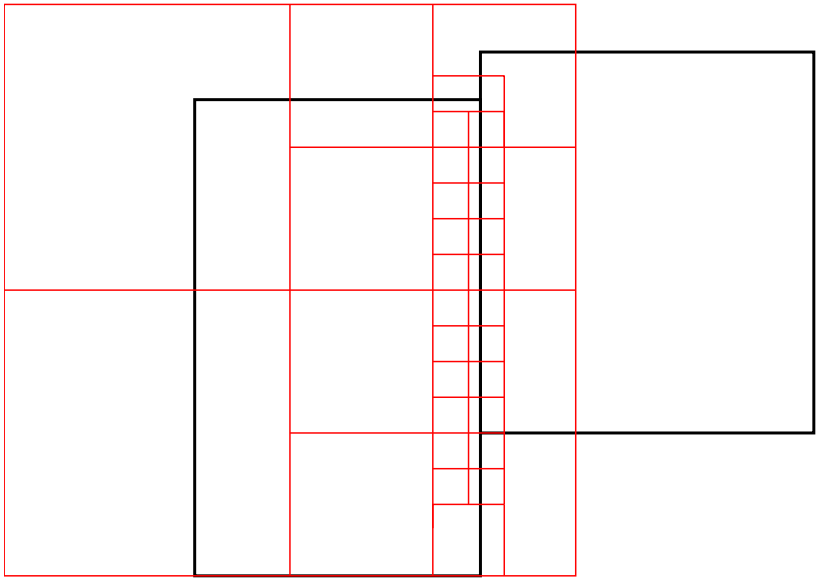
ORB 空間分割の実装

ORBも実装はいろいろ。良く知られているのは Dubinski の論文によるもの。粒子はすでに各プロセッサに適当に分配されているとする。

1. まず2分法で分割点を捜す。各プロセッサで粒子を数えた後、関係するプロセッサ全部での合計をとって新しい分割点を決める。
2. 各プロセッサで分割点の右と左に粒子を振り分ける
3. 各プロセッサは、 ORB ツリーの現在の階層で自分とペアになるプロセッサと粒子を交換する

というのを、階層分繰り返す。
ツリーはとりあえず各プロセッサでバラバラに作る。

他のプロセッサにある粒子からの力の計算

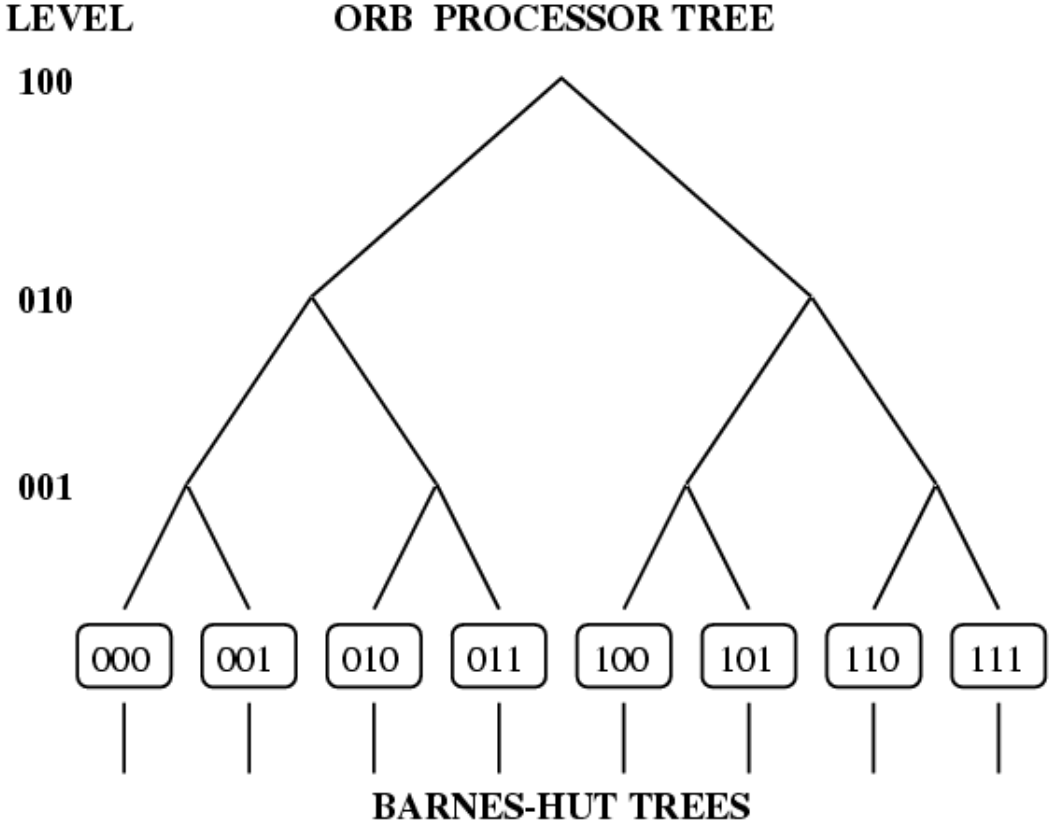


他のプロセッサから、「必要ないところをカットした」ツリーをもらってくる (local essential tree, LET)

それらと自分のところを適当にまとめて全体ツリーを作る

「適当にまとめる」とは？

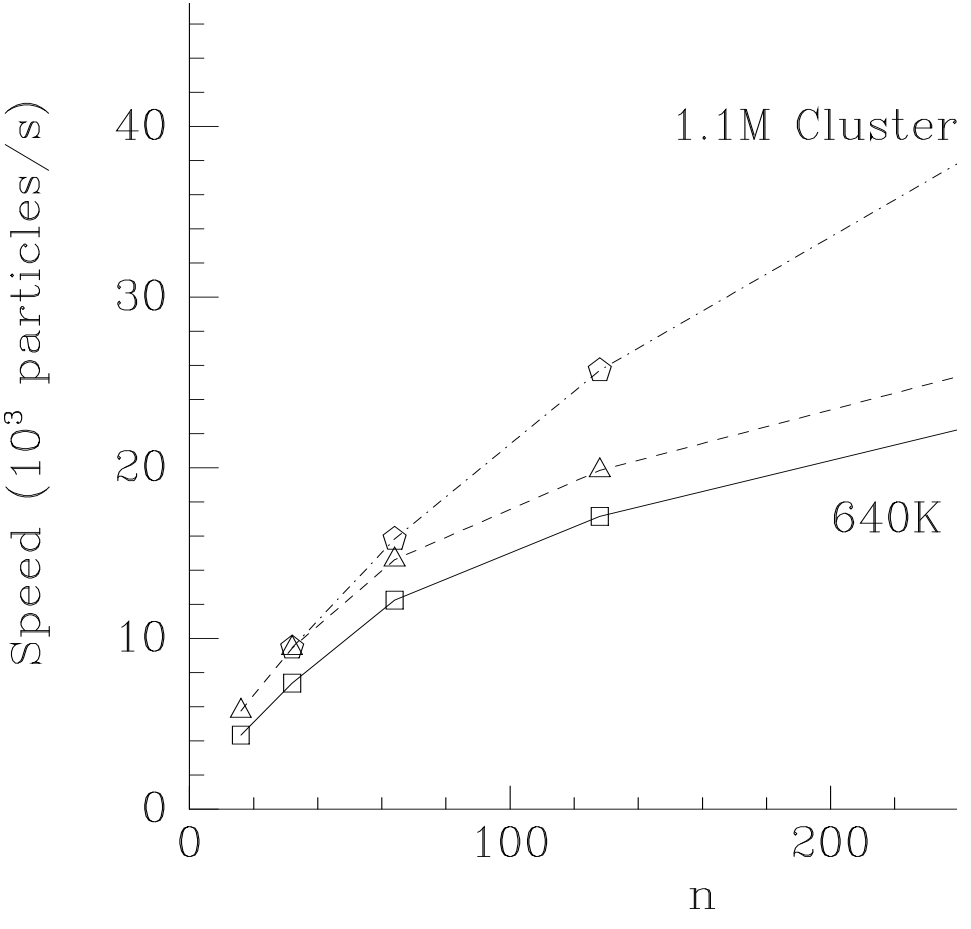
Dubinski の実装：この上の階層は ORB ツリーそのもの



ORB tree の問題

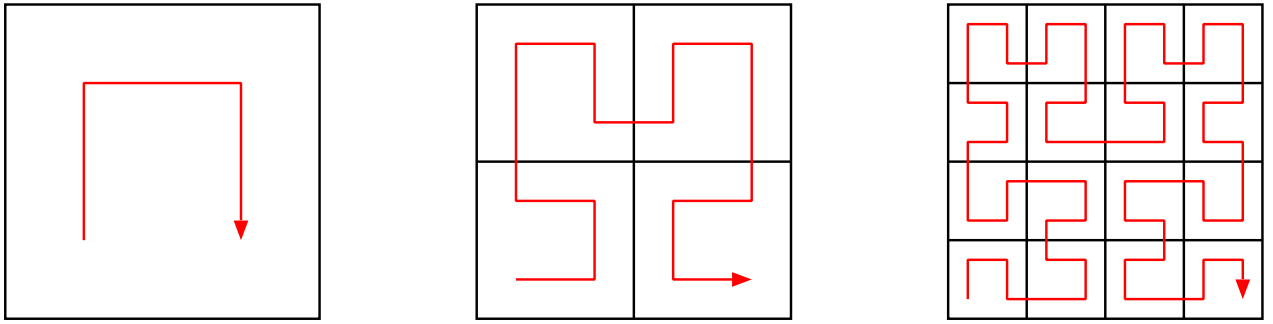
- 実装が面倒
  - ORB tree のところと local tree のところでツリーの構造が違う
  - LET はツリー構造（ポインタとか）もちゃんと保持して送る必要あり
- スケーラビリティが悪い
  - 通信回数がプロセッサ数に比例
  - 空間分割の形が悪くなると通信量も計算量も増える
- シリアルコードと結果が丸め以上に違う

スケーリング



HOT

ペアノ曲線：N 次元空間を 1 次元にマッピング



- **粒子をペアノ曲線上の順番で並べる**
- **各プロセッサに連続した粒子を割り当てる**

HOT の例

ただしこれはペアノ曲線ではなくて Morton Ordering

HOT でのツリー構成・重力計算

ツリー構成

- ペアノ曲線を作るところでツリーが構成される
- 並列ソートが必要

重力計算

- LET のようなことは困難
- 計算中に必要になった情報を他のプロセッサにリクエストする。

メッセージコンバイニング、計算と通信の非同期実行などの複雑なテクニックが必要になる。

HOT の利点・欠点

利点

- 任意のプロセッサ数・プロセッサトポロジに適用可能
- プロセッサ数が増えても計算量は増えない
- シリアルコードと結果が一致

欠点

- 実装が非常に大変
- 速い通信ライブラリ and/or 通信を隠蔽する「何か」が必要

ORB と HOT

- ORB
  - 実装は比較的簡単。速度もまあまあ
  - スケーリングが悪い（計算量自体が増える)
  - プロセッサ数が 2^n でないといけない
- HOT
  - プロセッサ数は任意
  - 速度は？
  - 実装は大変

我々の使うことにした方法

ORB を多少変更

- 深さを 3 層までに制限
- 2分割以上も許す

例えば、 12 ノードなら $3 \times 2 \times 2$ とかいうふうに分ける。
もちろん、 2, 4, 8 ノードでは従来の ORB と同じ。

ORB 空間分割の復習

ORBも実装はいろいろ。良く知られているのは Dubinski の論文によるもの。粒子はすでに各プロセッサに適当に分配されているとする。

1. まず2分法で分割点を捜す。各プロセッサで粒子を数えた後、関係するプロセッサ全部での合計をとって新しい分割点を決める。
2. 各プロセッサで分割点の右と左に粒子を振り分ける
3. 各プロセッサは、 ORB ツリーの現在の階層で自分とペアになるプロセッサと粒子を交換する

というのを、階層分繰り返す。
この方法の問題点：計算量、通信量（回数）ともに大きい。

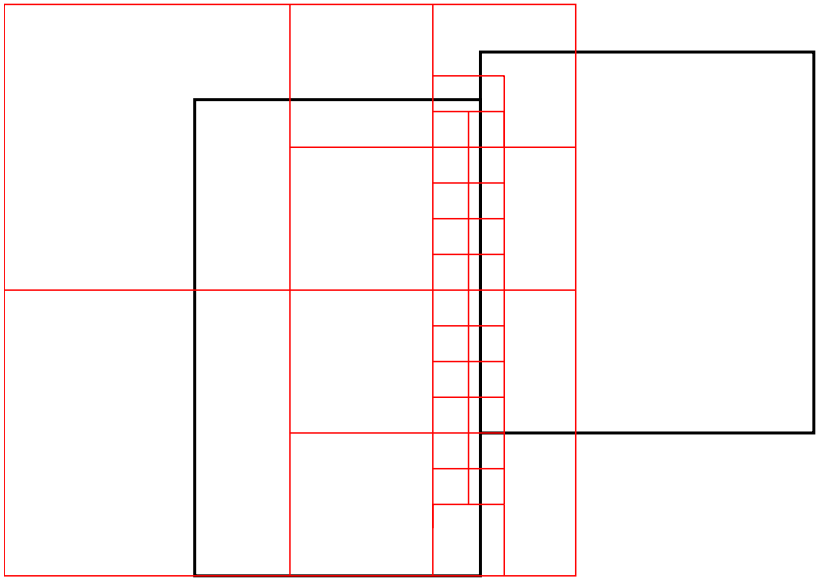
もうちょっと安易な方法

今回使った方法： Brackston & Suel によるものを踏襲

- 各プロセッサは適当に選んだサンプル粒子のセットをノード 0 に送る。
- ノード 0 は、集めたサンプルセットを使って分割を生成する。できた分割は全ノードに放送する。
- 各プロセッサは、もらった分割を使って粒子を振り分けて、担当するプロセッサに送る。

分割に多少の誤差が入るが、計算時間としては問題にならない程度。
計算・通信時間は要求する分割精度に依存

力の計算（復習）



他のプロセッサから、「必要ないところをカットした」ツリーをもらってくる (local essential tree, LET)

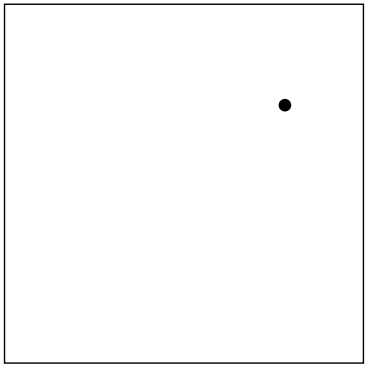
それらと自分のところを適当にまとめて全体ツリーを作る

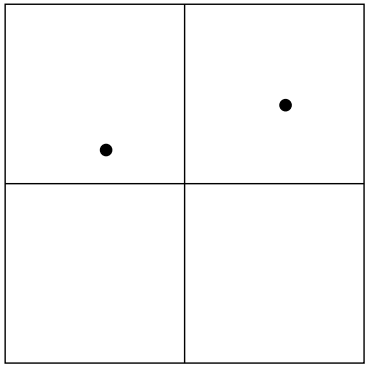
今回の実装

- LET ではなく、LET の最下層にある「粒子」だけを送る。
- 受けとった「粒子」を、自分のツリーに挿入する。

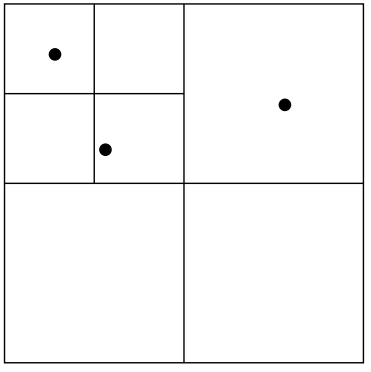
ツリーに粒子を挿入：もともとの Barnes のプログラムで使われている木の作り方。

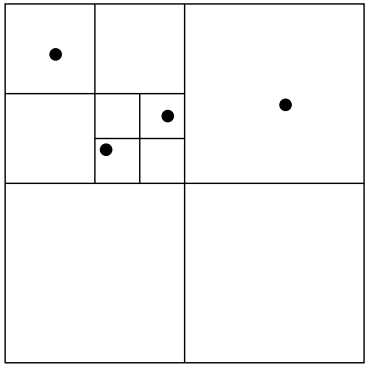
粒子挿入によるツリー構成





粒子がどのセルに入っているか決める。





既に子セルがあればそのどれかに入れる

既に粒子が入っていれば分割する。

粒子挿入による全体ツリー構成

- 実装は簡単（もともとのツリーを作るルーチンがそのまま使い回せる）
- 通信量は理論的に可能な最小量
- ツリー構成の計算量は多少増える
- 重力計算の計算量はシリアルコードと同じ
- 結果もシリアルコードと一致

別のアプローチ — 計算機を作る

速い計算機を買ってきて動かすというのもなかなか大変である。

- 10 年もたつと計算機アーキテクチャが変わる (かもしれない) ので昔頑張って作ったプログラムが水の泡になる (かもしれない)
- 最近考えないといけないことが増えた
  - 分散メモリでの並列化
  - キャッシュの有効利用による高速化
  - その他なんだか分からないテクニック

もうちょっと違う人生はないか?

一つの考え方：計算機を自分で作る

人が作った計算機を苦労して使おうと思うから面倒。
ハードウェアから好きなように作ればむしろ簡単にならないか？

なぜ専用計算機を考えるのか

基本的には、汎用計算機に比べて、「速く、安く」できる（かもしれない）から。
なぜ「速く、安い」か？

- 問題自体の特性
- 技術的な要因
- 歴史的、経済的な要因

問題自体の特性

粒子系シミュレーションの特徴:一つの粒子が多数の粒子と相互作用する

- 計算量が多い（メモリ必要量に比べて）
- 計算が比較的単純な繰り返しである
- 通信パターンが規則的である

（独立時間刻み、ツリー法では細かい考慮が必要にはなる）

対象システムの分類

連続系（流体等）：規則的、近傍通信、計算量小

粒子系：規則的（ $N \times N$ 通信)、計算量大

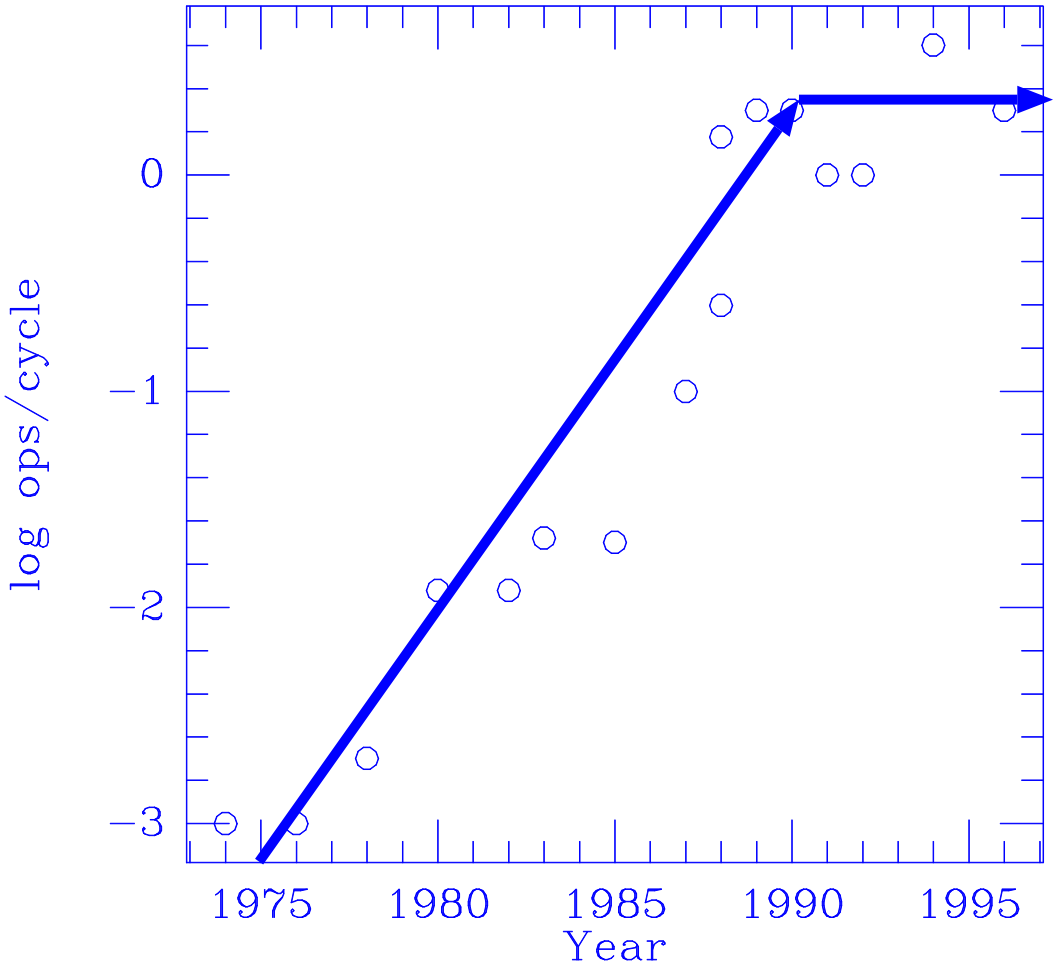
その他（離散不規則系）：回路シミュレーション等？

規則的で計算量が大きい $=$ 専用ハードウェアに適している

技術的な要因

- 半導体製造技術の進歩 $=$ 多数の演算回路を集積する大規模回路が実現可能
- 半導体設計技術の進歩 $=$ 半導体技術の「素人」でもモジュールジェネレータ等で設計ができる $(???)$
- 汎用計算機の設計技術の限界 $(?)$ $=$ トランジスタ利用効率の急速な低下

マイクロプロセッサの「進化」



代表的なマイクロプロセッサの、サイクル当たりの浮動小数点演算数

1 を超えてから、ほとんど止まっている ＝ 汎用の並列処理は難しい

マイクロプロセッサにおける並列処理

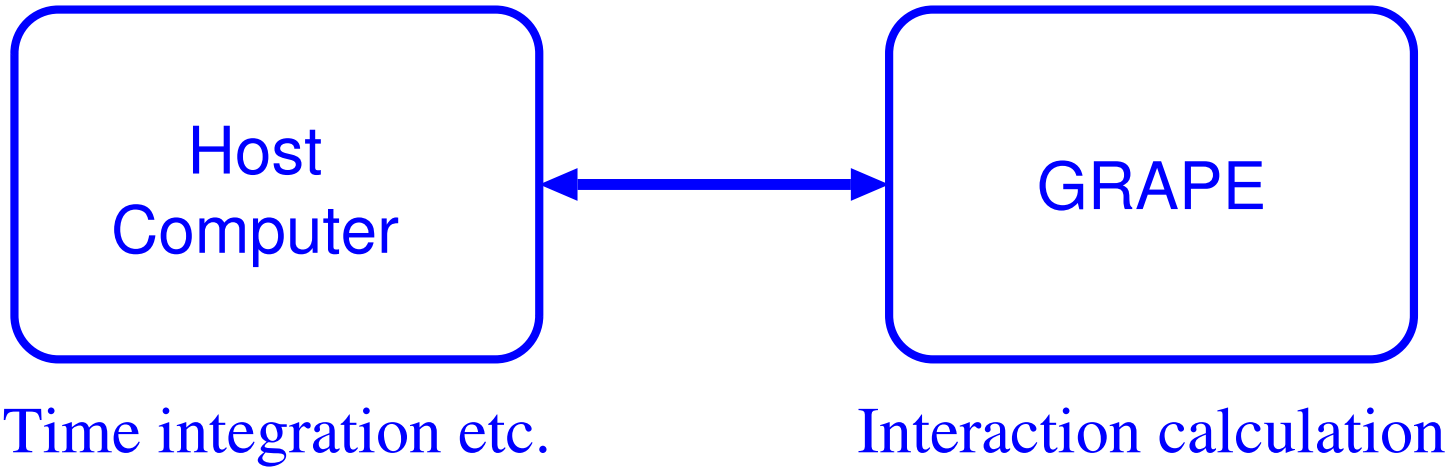
用途を制限すればある程度の並列化はできる

- ソニー PS2, 日立 SH4 : 3D 座標変換に特化したハードウェア
- Intel SSE, AMD 3DNow!, Motorola AltiVec: SIMD 並列処理。座標変換に重点

汎用の並列処理は難しい

- スーパースカラ
- VLIW

GRAPE の基本的考え



専用ハード: 相互作用の計算

汎用ホスト: 他のすべての計算

専用ハードウェア

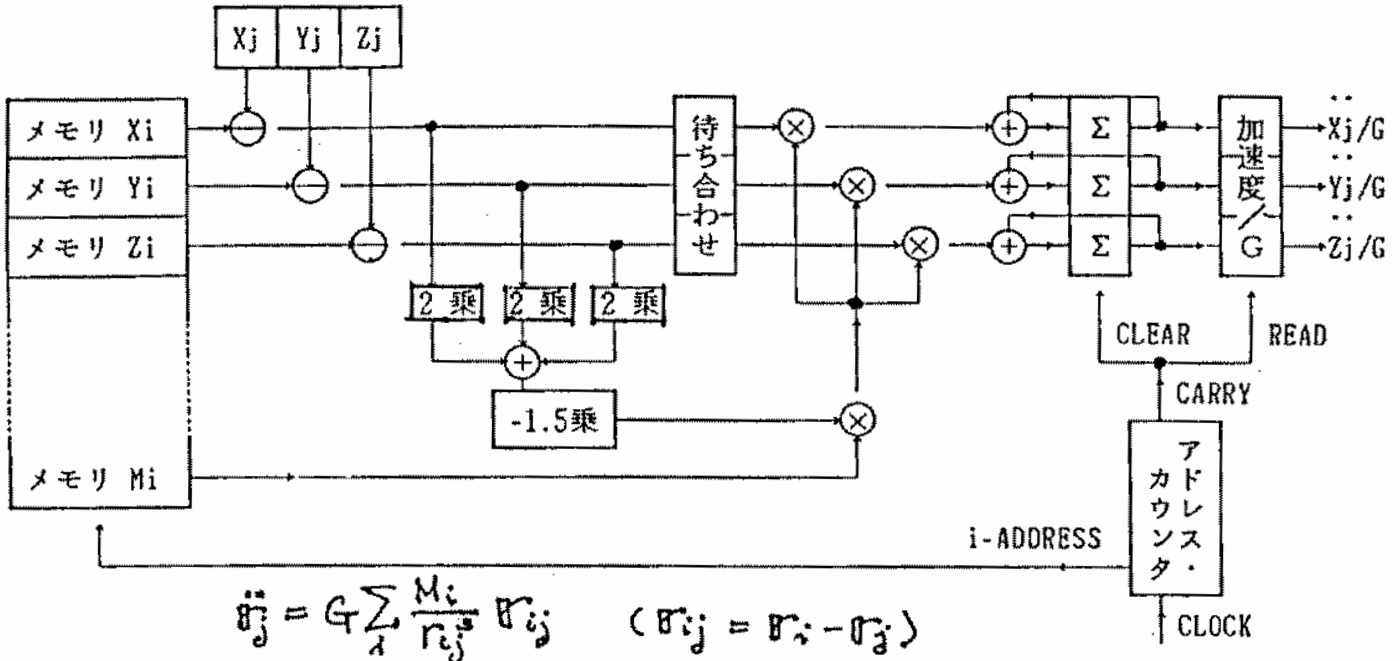
- 相互作用計算のための専用パイプラインプロセッサ
  - 多数の演算器を集積可能
  - すべての演算器が常時並列動作

  → 非常に高い性能
- すべてハードウェア → ソフトウェア不要

汎用ホスト計算機

- 高レベル言語 (Fortran, C, C++...)
- すでにあるプログラムが少しの変更で使える。
- 独立時間刻み、ツリー法等も使える

GRAPE パイプライン



$$\ddot{\vec{r}}_j = G\sum_i \frac{M_i}{r_{ij}^3}\vec{r}_{ij} \qquad (\vec{r}_{ij} = \vec{r}_i - \vec{r}_j)$$

+, −, ×, 2乗は1 operation, -1.5乗は多項式近似でやるとして10operation 位に相当する.
総計24operation.
各operation の後にはレジスタがあって, 全体がpipelineになっているものとする.
「待ち合わせ」は2乗してMと掛け算する間の時間ズレを補正するためのFIFO(First-In First-Out memory).
「Σ」は足し込み用のレジスタ. N回足した後結果を右のレジスタに転送する.

図2. N体問題のj-体に働く重力加速度を計算する回路の概念図.

(近田 1988)

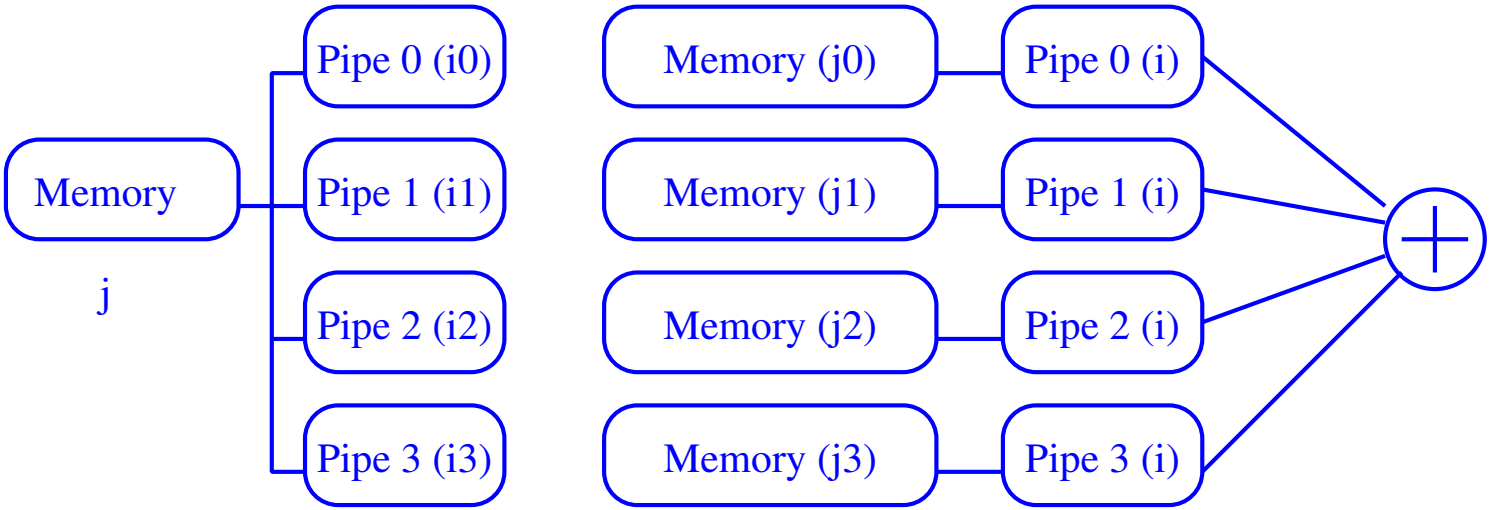
GRAPE における並列化

パイプライン1本が速くても、並列化できなければたいした意味はない。
ホストは1台として、 GRAPE の側の並列化を考える。
方針は 2 つ

- 複数の粒子への力を並列に計算する (i 並列)
- 複数の粒子からの力を並列に計算する (j 並列)

実現方法

i 並列：一つのメモリからのデータを複数パイプラインに配る。
j 並列：複数の（メモリ＋パイプライン）に同じ粒子への力を計算させて、終ってから合計する。



i-parallel　　*j-parallel*

並列パイプラインの実装

メモリを共有できる範囲（ボード／チップ）：
i 並列
それより上： j 並列
メモリデータの複製を許すなら i 並列も使える

GRAPE-4: ボード上 i 並列、ボード間 j 並列
GRAPE-6: チップ内 i 並列、ボード上 j 並列

GRAPE アーキテクチャの発展

1989 GRAPE-1 低精度、EPROM で演算

1990 GRAPE-2 高精度、浮動小数点演算 LSI

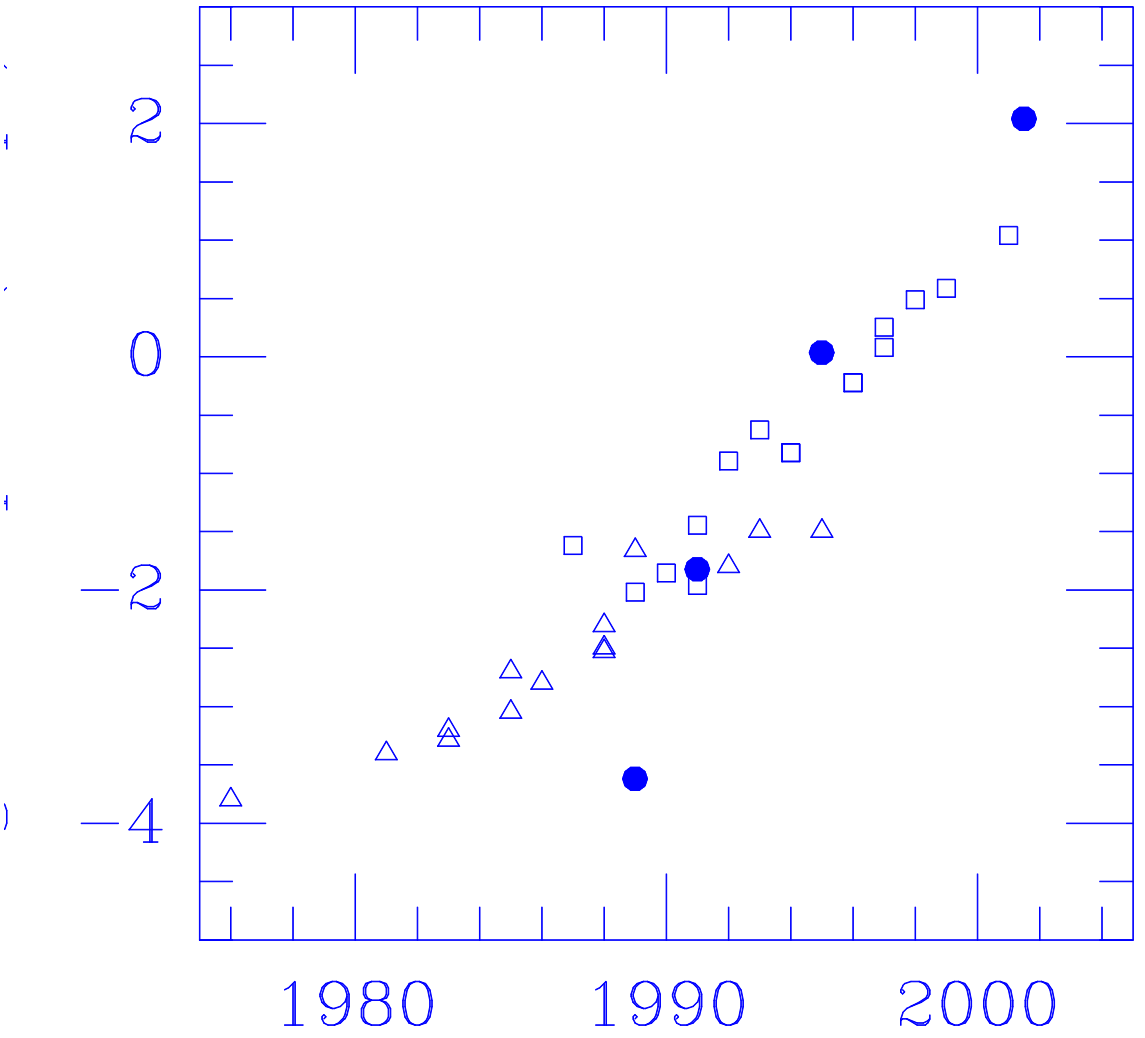
1991 GRAPE-3 低精度、カスタム LSI

1995 GRAPE-4 高精度、カスタム LSI 、超並列

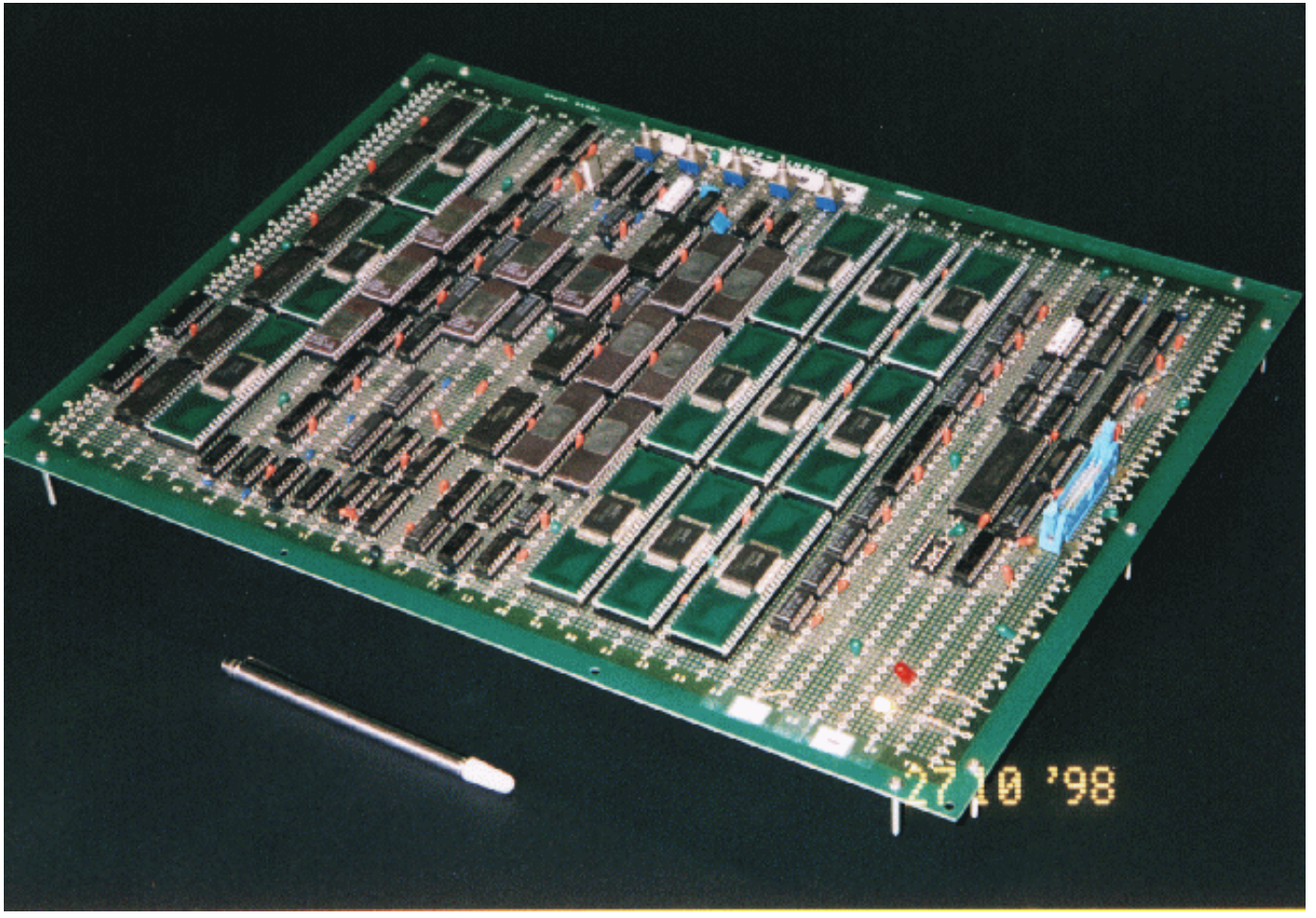
1998 GRAPE-5 低精度、複数パイプラインを集積

2001 GRAPE-6 高精度、複数パイプラインを集積

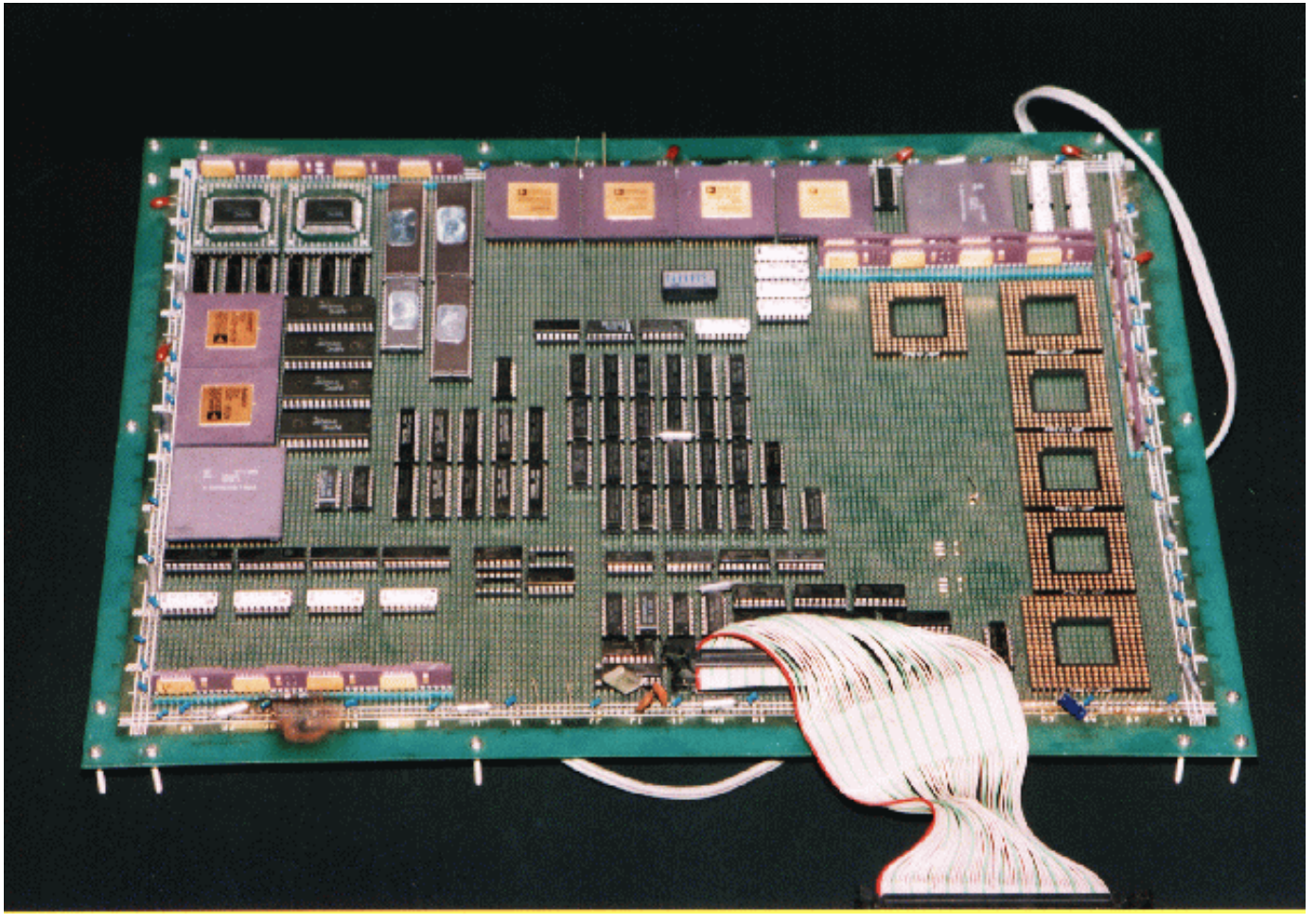
計算速度の発展



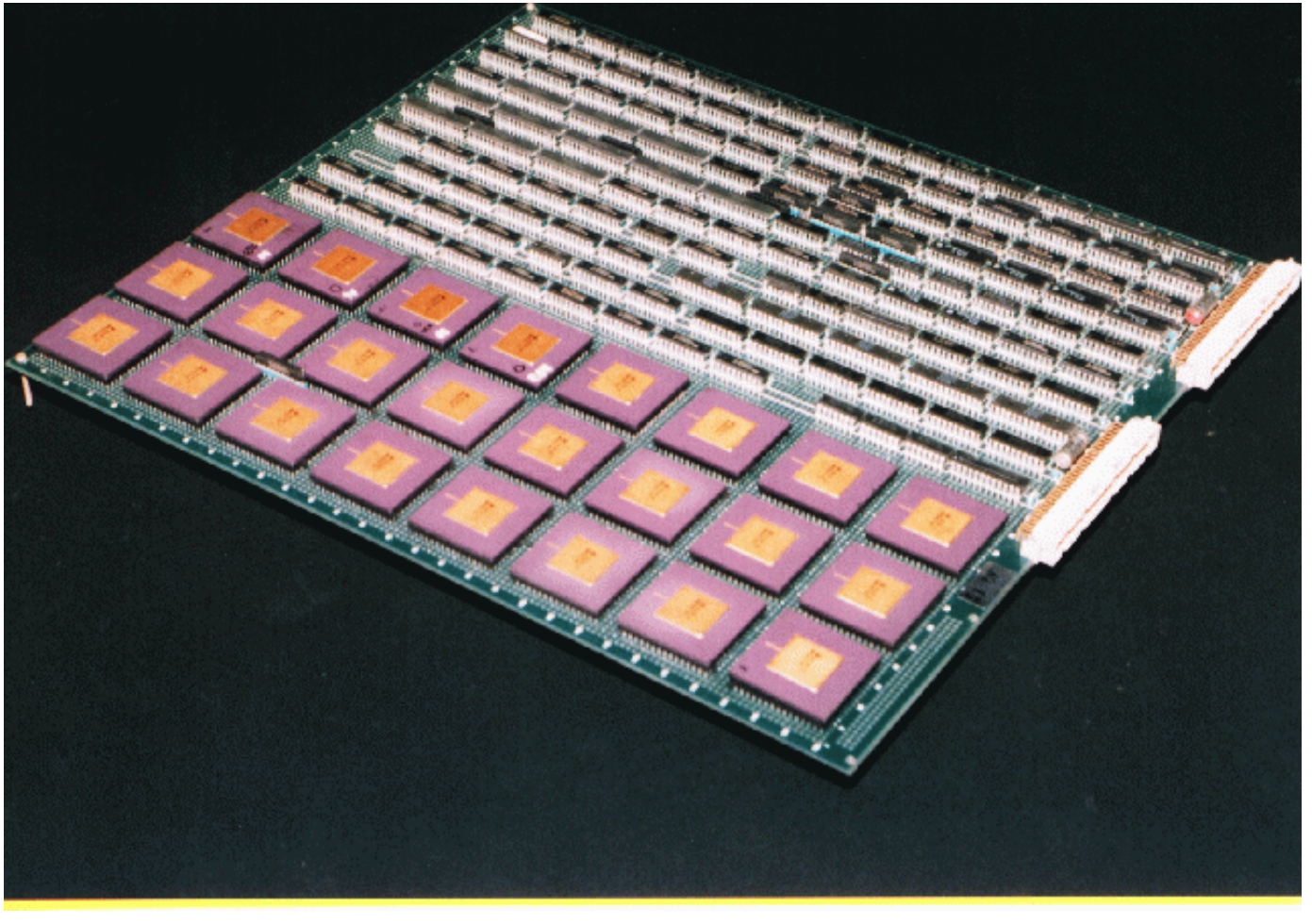
GRAPE-1



GRAPE-2

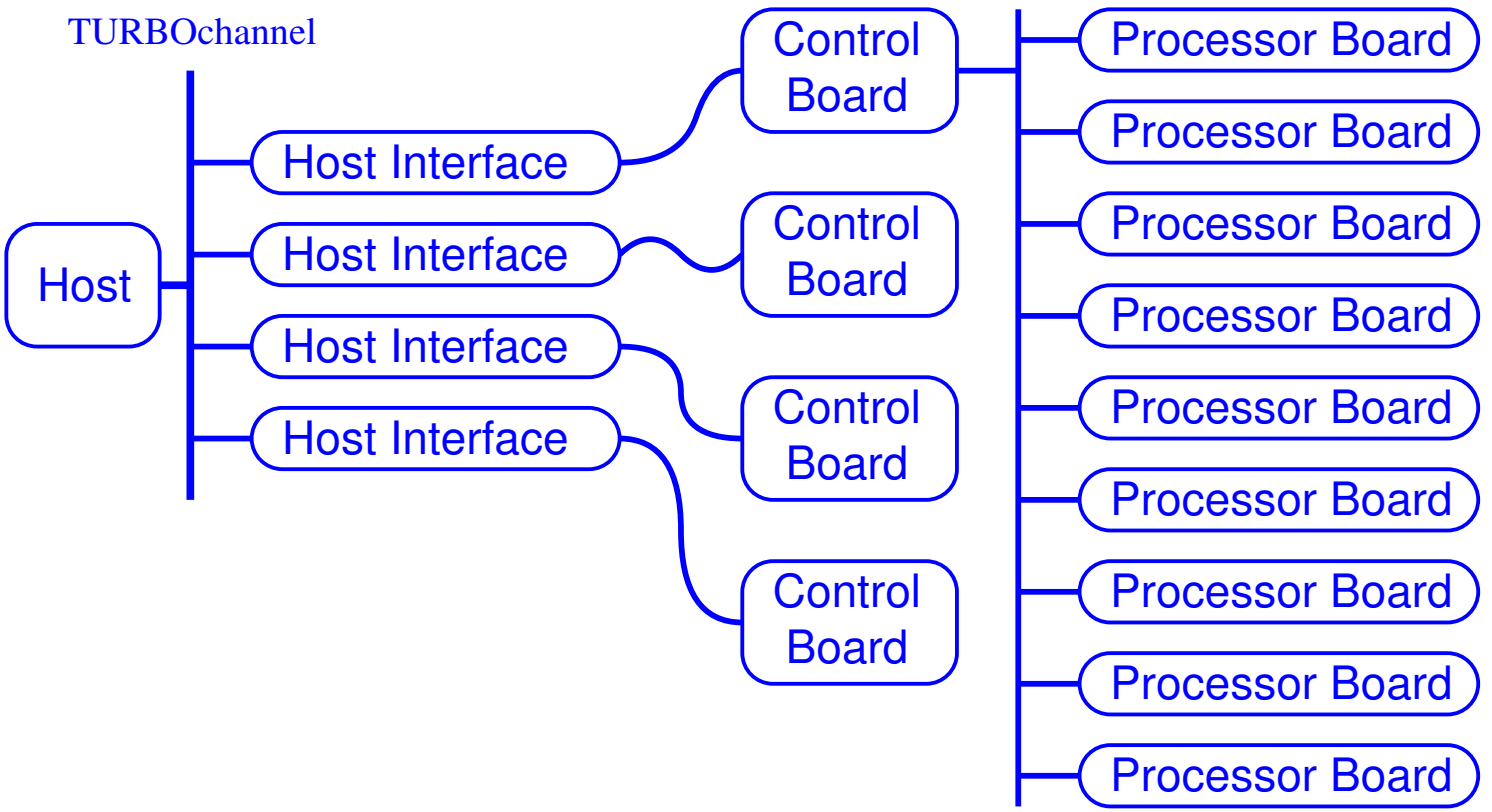


GRAPE-3

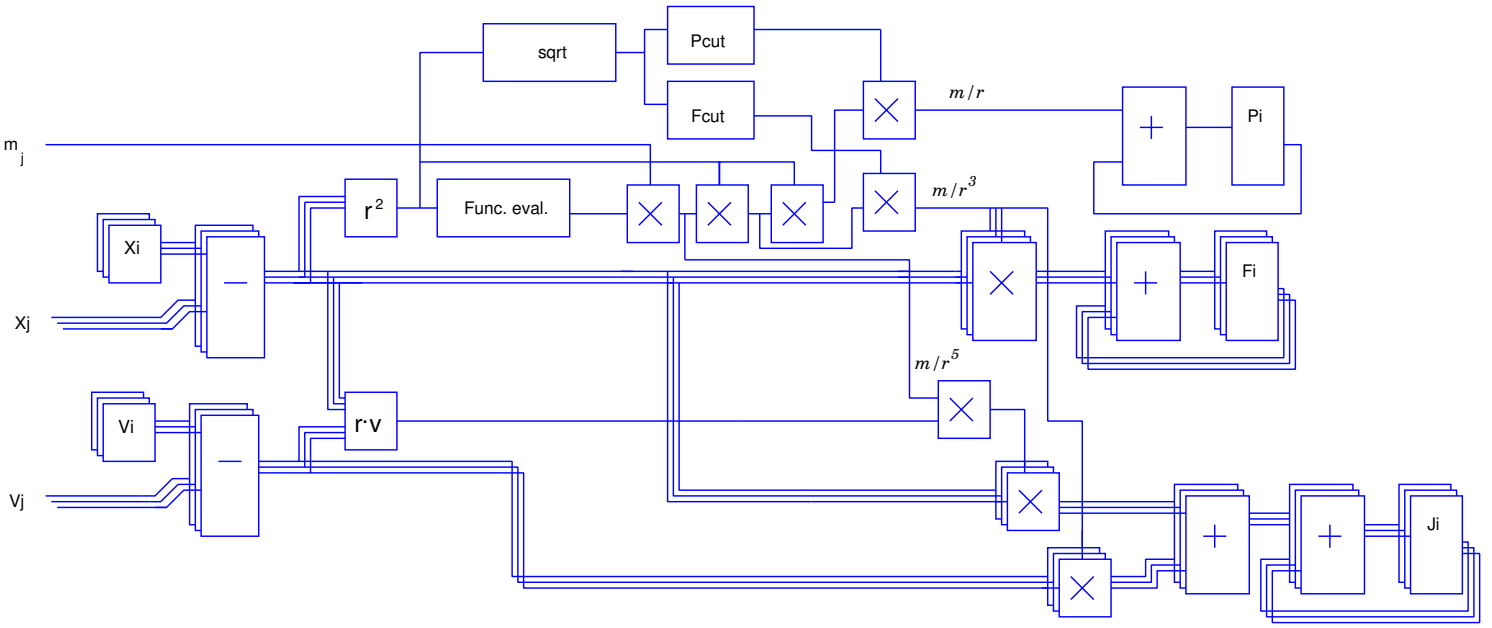


GRAPE-4

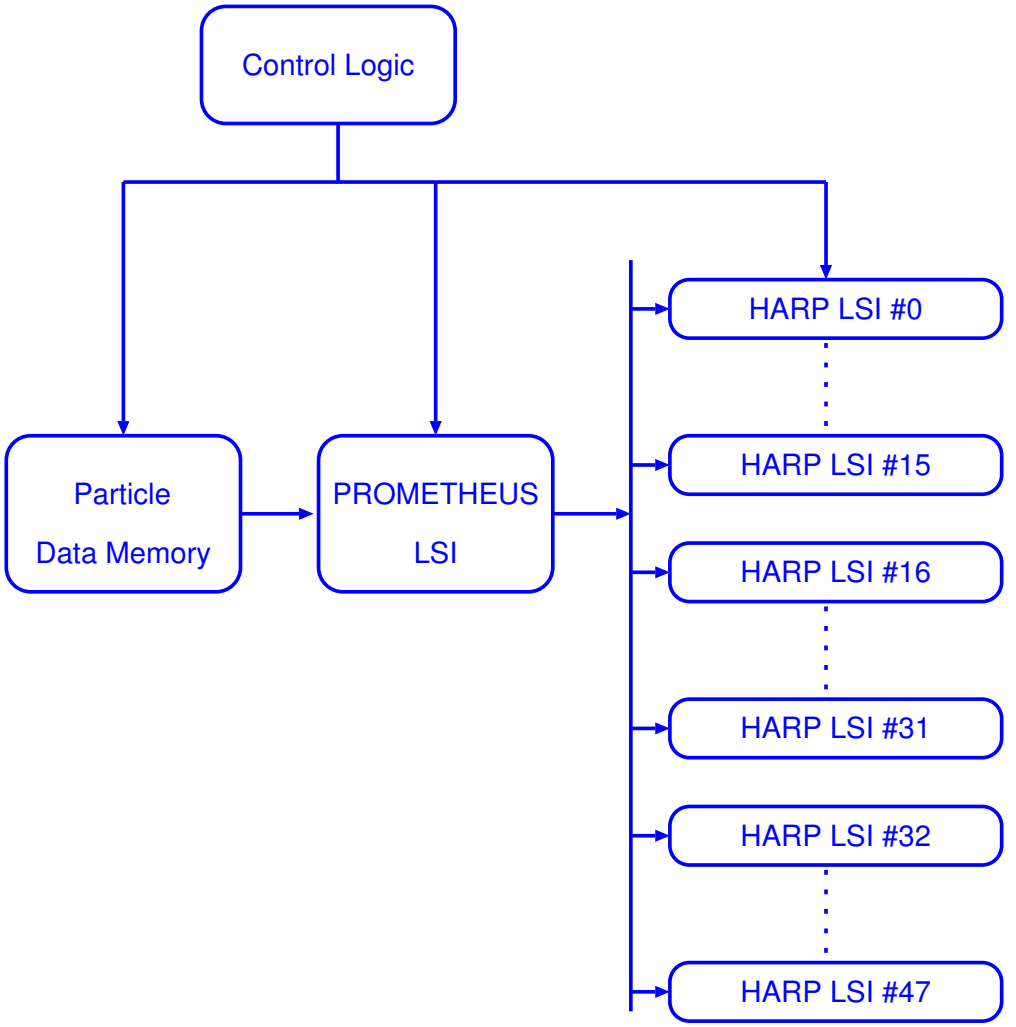
GRAPE-4 の構成


TURBOchannel
Host
Host Interface
Host Interface
Host Interface
Host Interface
Control Board
Control Board
Control Board
Control Board
Processor Board
Processor Board
Processor Board
Processor Board
Processor Board
Processor Board
Processor Board
Processor Board
Processor Board

GRAPE-4 パイプライン



GRAPE-4 演算ボード



多数の パイプライン LSI がメモリユニットを共有

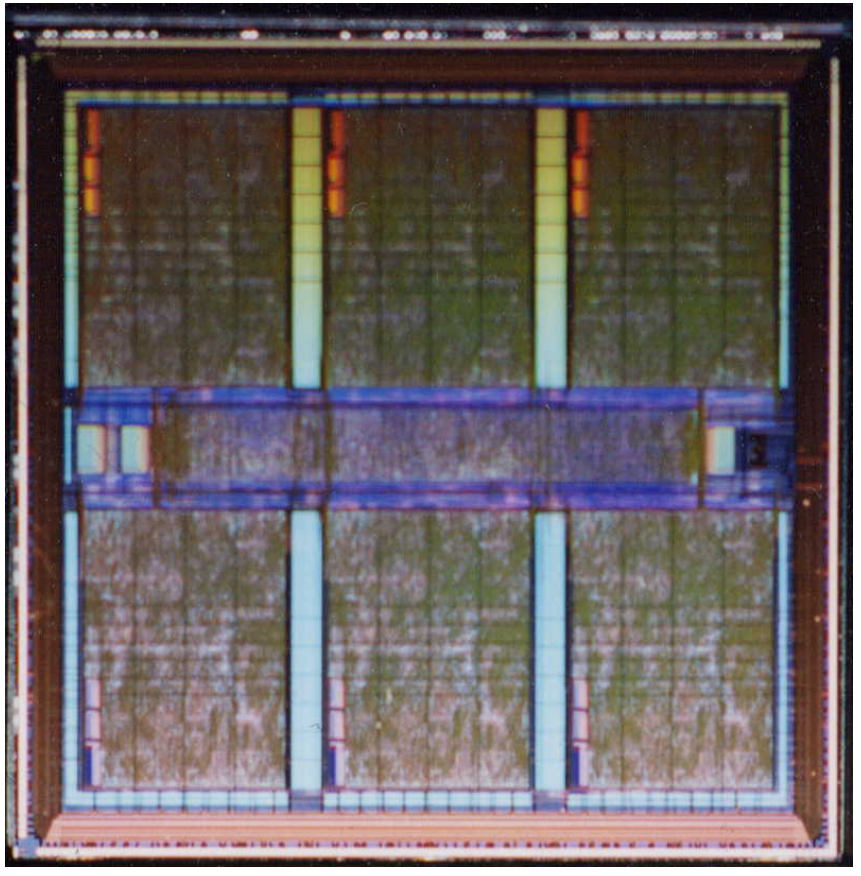
↓

ボードが単純になり、集積度をあげられる

GRAPE-6 について

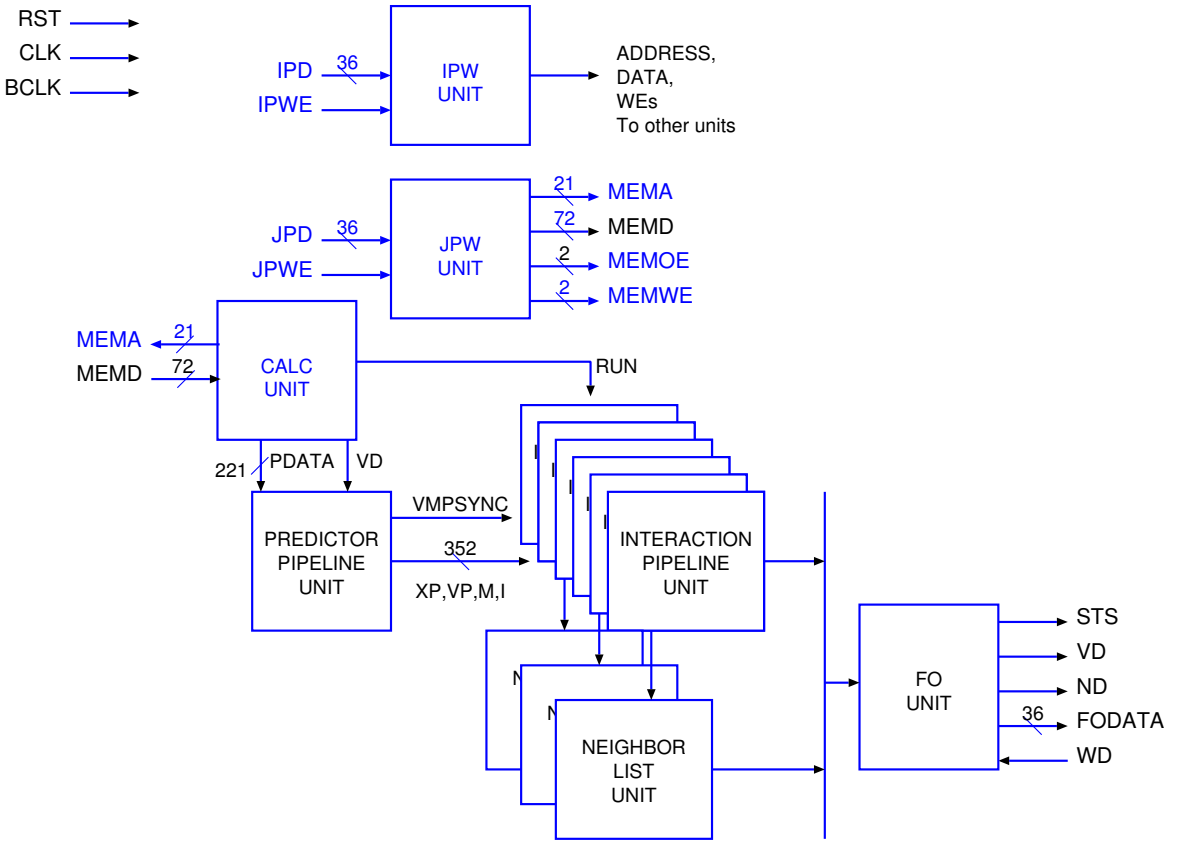
- プロセッサチップ
- プロセッサモジュール
- プロセッサボード
- ネットワークボード
- 開発の現状

パイプライン LSI



- 0.25 μm ルール
  (東芝　TC-240，　1.8M ゲート)
- 90 MHz 動作
- 6 パイプラインを集積
- チップあたり 31 Gflops

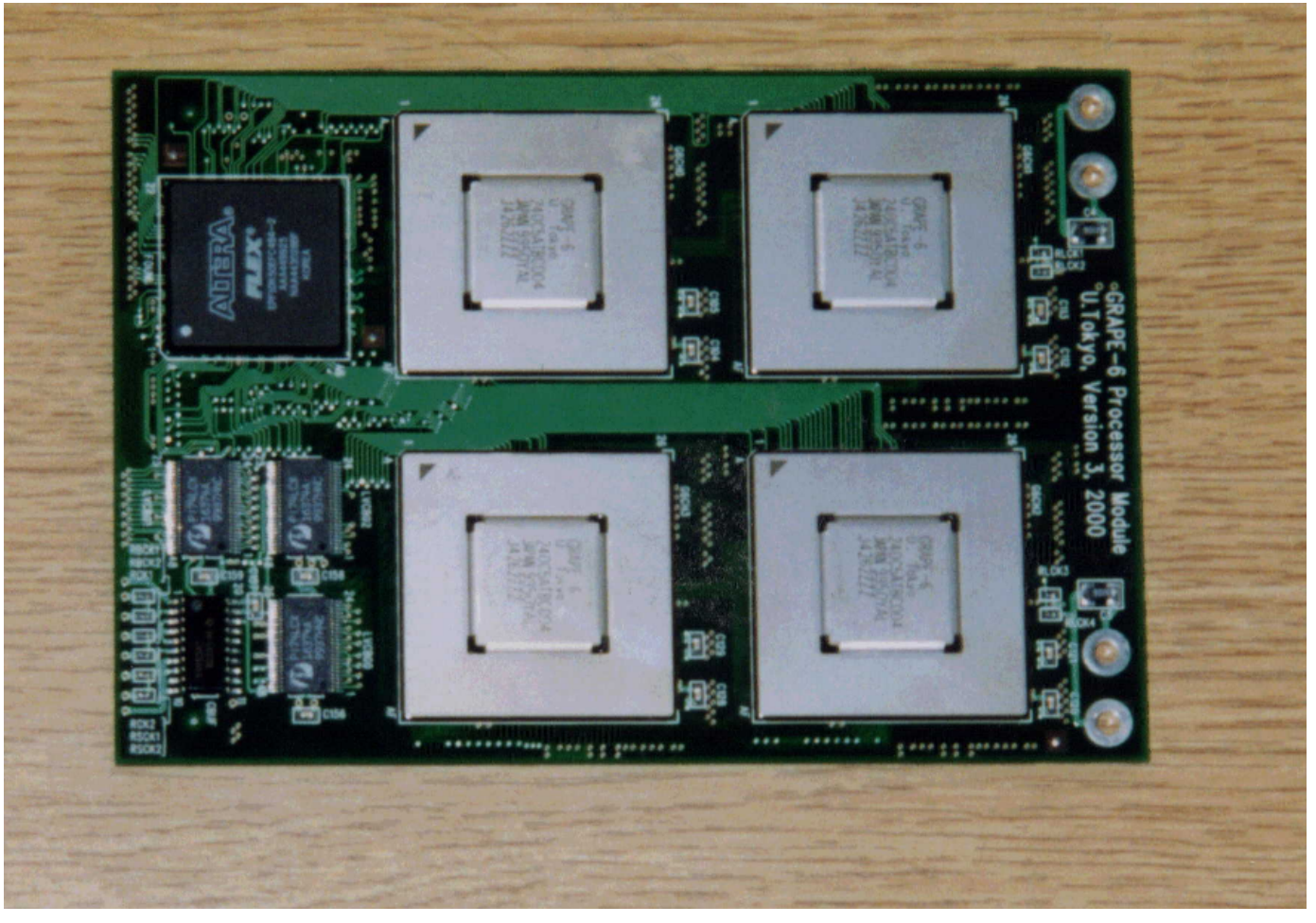
パイプライン LSI詳細



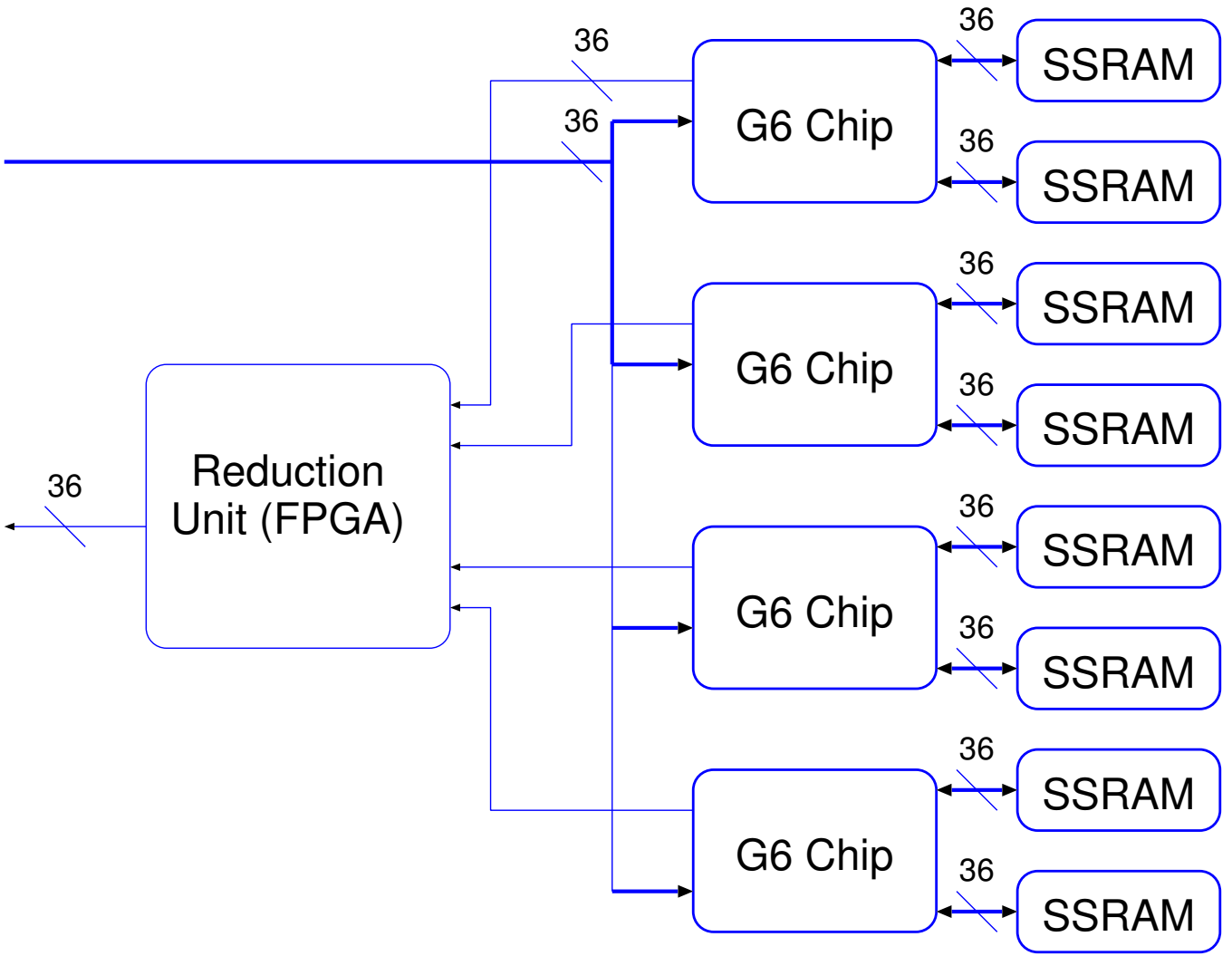
GRAPE-4 プロセッサボードの全機能を集積

- ホスト IF
- メモリ IF
- パイプライン本体
- 制御回路

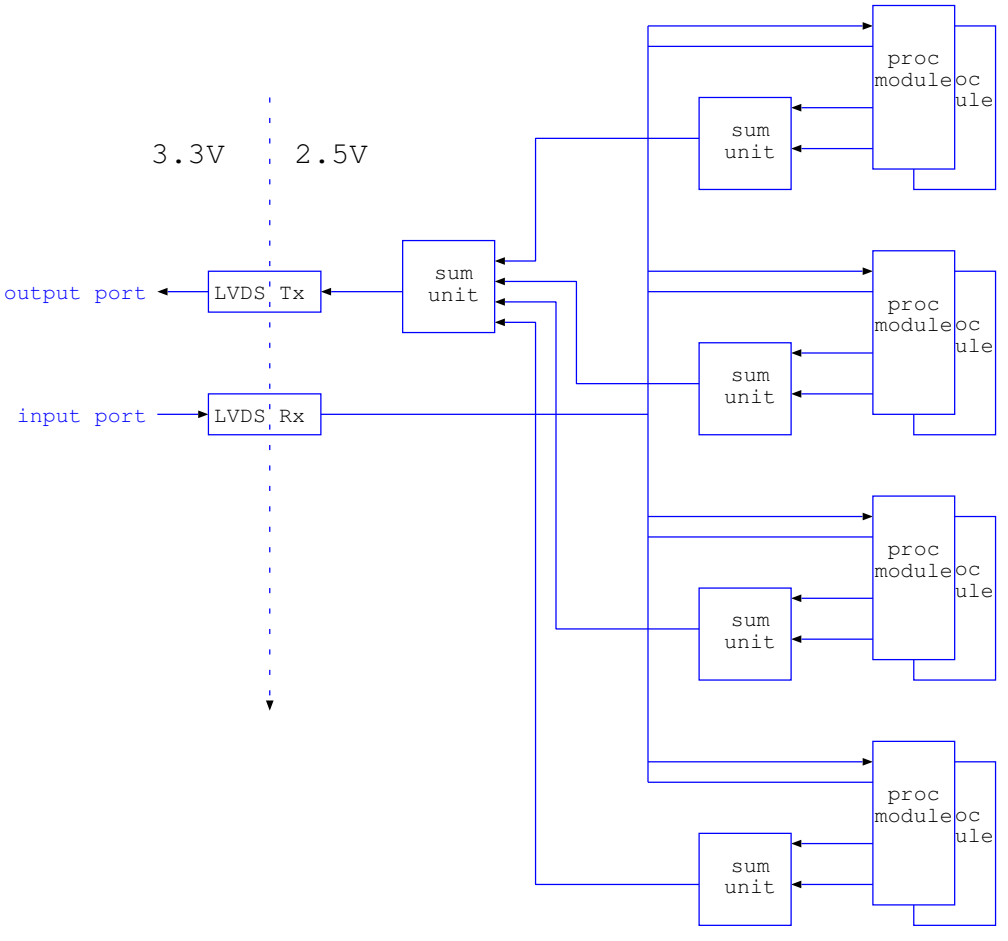
GRAPE-6 processor module



GRAPE-6 processor module

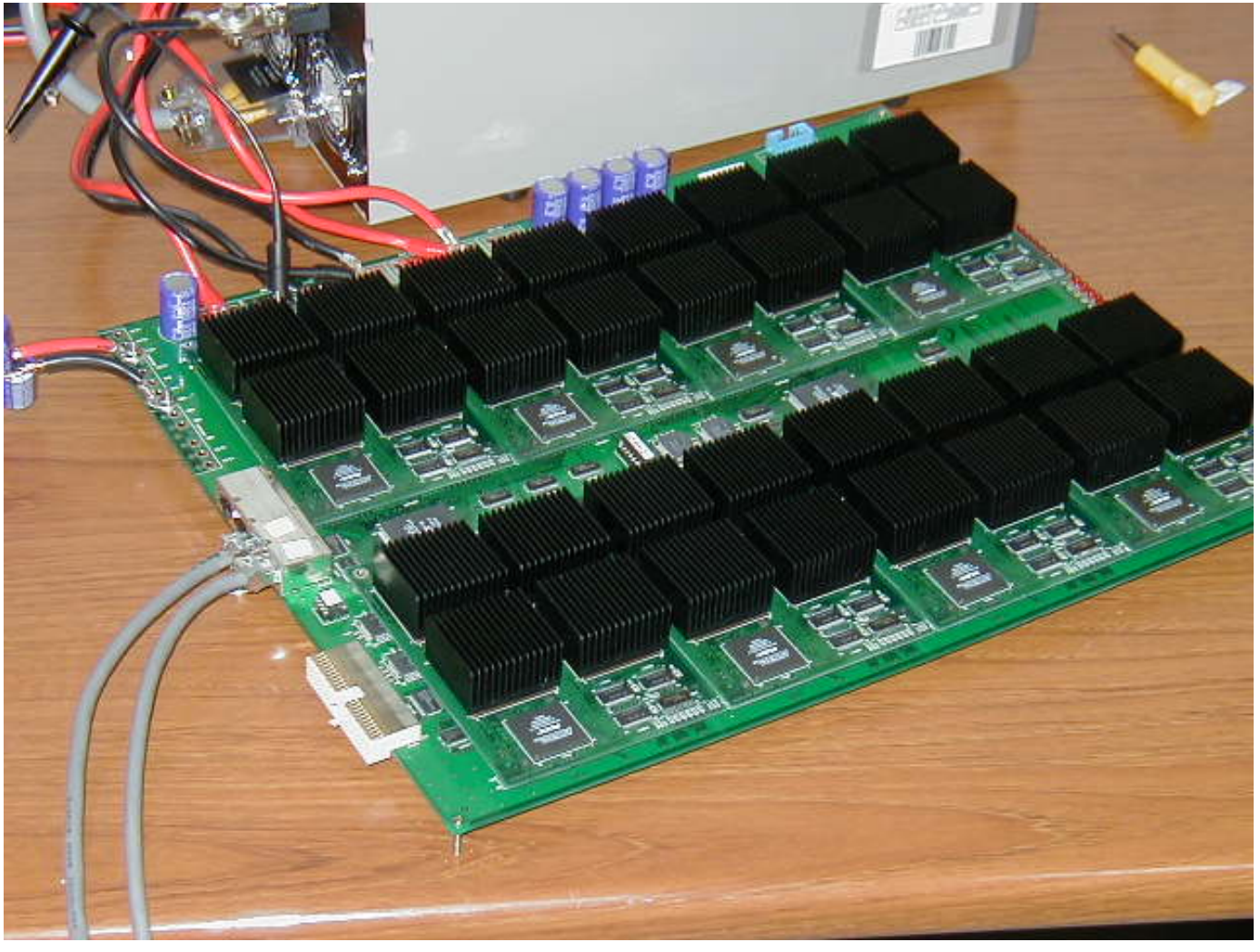


GRAPE-6 Processor board

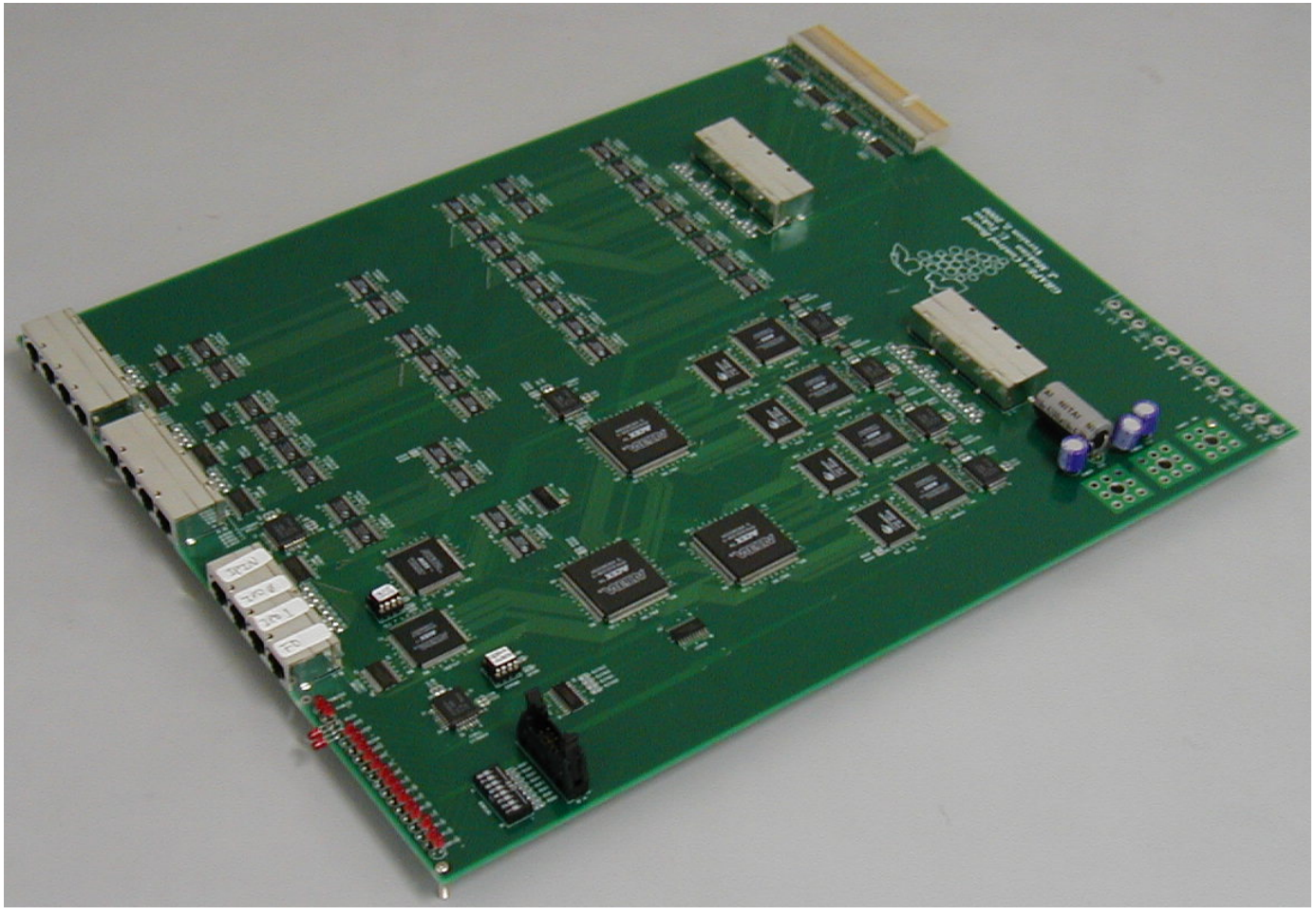


- ボード 1 枚に 32 チップ
- セミシリアル (LVDS) インターフェース (350MHz clock, 4 wires)

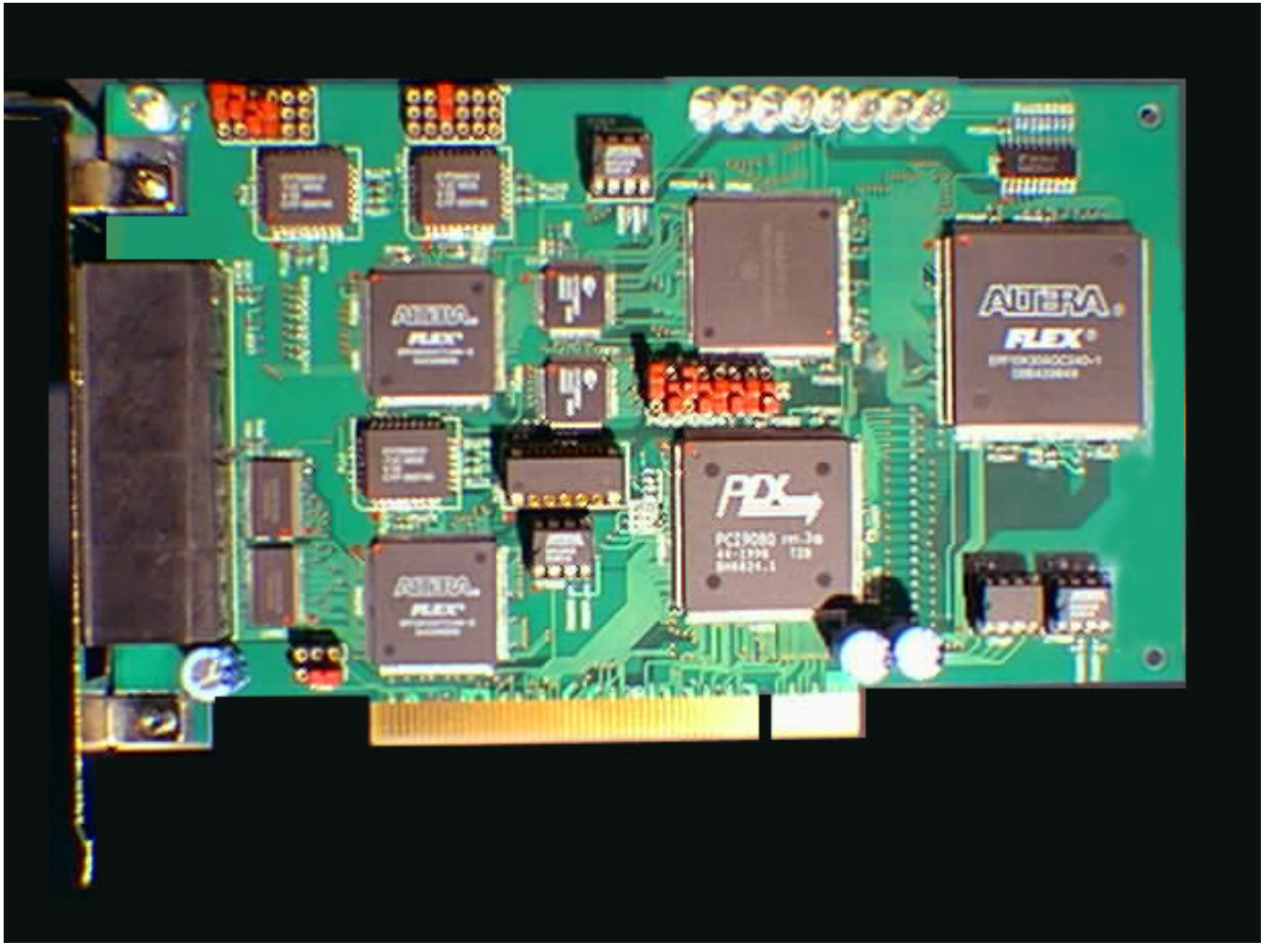
GRAPE-6 processor board



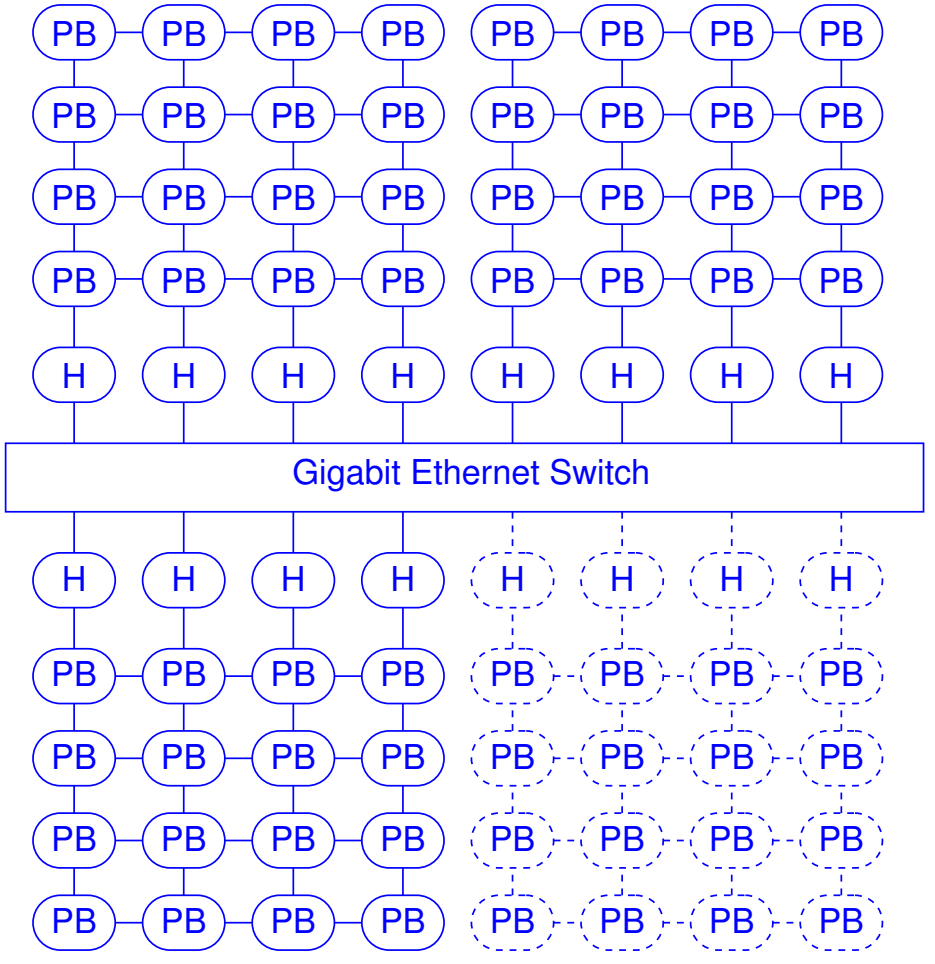
GRAPE-6 network board



GRAPE-6 host interface board



The full 64 Tflops GRAPE-6 system



- 4-host, 16-board "block" with dedicated network
- 4 (currently 3) "blocks" connected through GbE network

Combination of host network solution and dedicated network solution.

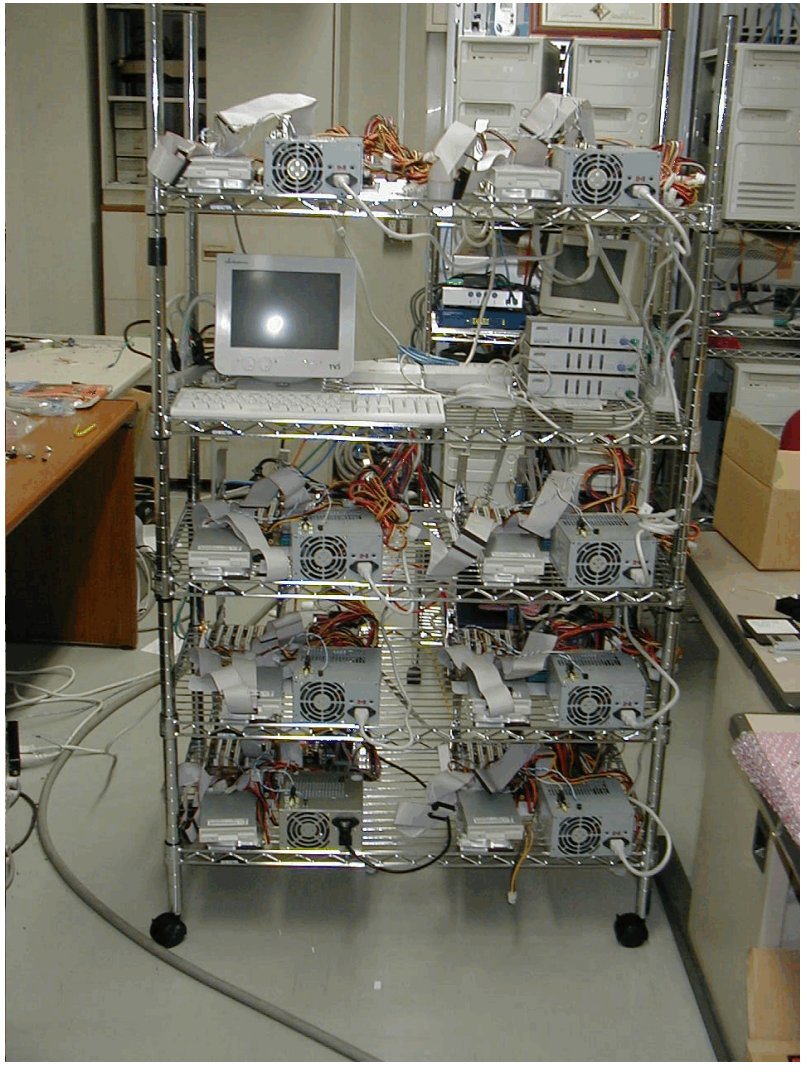
The 48-Tflops GRAPE-6 system

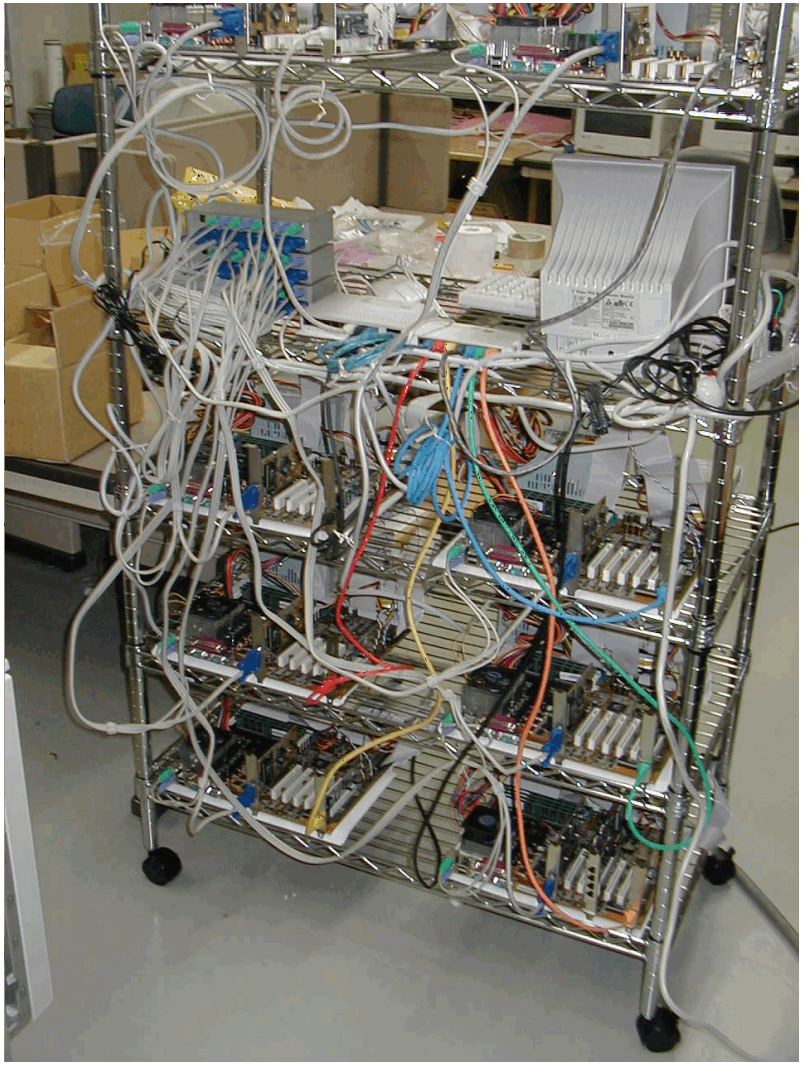
Present 48-Tflops system.

Three blocks with 16 host computers.

The fourth block will be operational by the end of this month.

The host "PC Cluster"





専用計算機開発の損得

13 年ばかりやってきてどうだったか？ということ。

もともとターゲットとした問題（天文の多体問題）に対しては、予想以上の成果をあげたといって嘘にはならないと思う。

GRAPE は 100 枚以上のコピーが世界中の 20 以上の研究機関で使われ、惑星形成や星団の進化のシミュレーションのほとんど、また銀河形成や宇宙論シミュレーションについてもかなりの割合が GRAPE を使って行なわれている。

というあたりが「公式見解」ということで。

個人的な感想

GRAPE のようなものを「うまく作る」のは結構大変である。

- 対象となる問題
- アルゴリズム・要求精度
- 計算機アーキテクチャ
- 論理設計
- 実装
- OS、デバイスドライバ等のソフトウェア

の全般についてまあまあの理解が統合されていないと全体設計が出来ない。

普通の計算機を作るのに比べると

普通の計算機では

- 対象となる問題、アルゴリズム、要求精度

「必要ない」（本当？）

- それ以外

「まあまあ」ではまあまあの計算機しか出来ない。

他の誰かと同じようなことをすれば済むという面もある（1番になれるか？という問題はある）

Case Study： GRAPE-6 の場合

演算方式、論理設計： GRAPE-4 のものを多少改良／変更。大きな問題はなかった。
実装：テクノロジ微細化、低電圧化のためにいろいろ問題が起きた。

- レイアウトが収束しない（タイミング要求を満たせない）。
- チップ内で電源電圧が下がる。
- 負荷変動に電源が対応できない。

レイアウトが収束しない

（最近はましになってきてるかもしれないけど）ディープサブミクロン設計でありがちな問題。
配線遅延 ＞＞ スイッチング遅延
→ レイアウトしないと遅延時間がわからない

従来の論理設計段階では仮配線長を仮定するやりかたでは駄目。

本来、レイアウトと論理設計を並行して進めるような手法が必要

GRAPE での配線遅延

GRAPE では

- 長い配線はパイプラインとインターフェース回路の間くらいしかない
- そこのタイミング要求はあんまり厳しくない（マルチサイクルでいいのが多い）

ので比較的楽で、大幅なレイアウト修正なしでなんとか誤魔化していた、、、

実際、ほぼ同じ時期に同じ会社が作っていた高速マイクロプロセッサに比べれば半分以下の期間でレイアウトが終った。

チップ内で電源電圧が下がる。

最初のチップでは

- パッケージの電源ピンからダイのパッドまでの配線抵抗
- ダイ上の電源供給ネットの抵抗

が大きかったために、内部で電圧が下がってまともに動かなかった。

計算中と非動作時では2倍程消費電力が違うので、電圧もその分変わる。

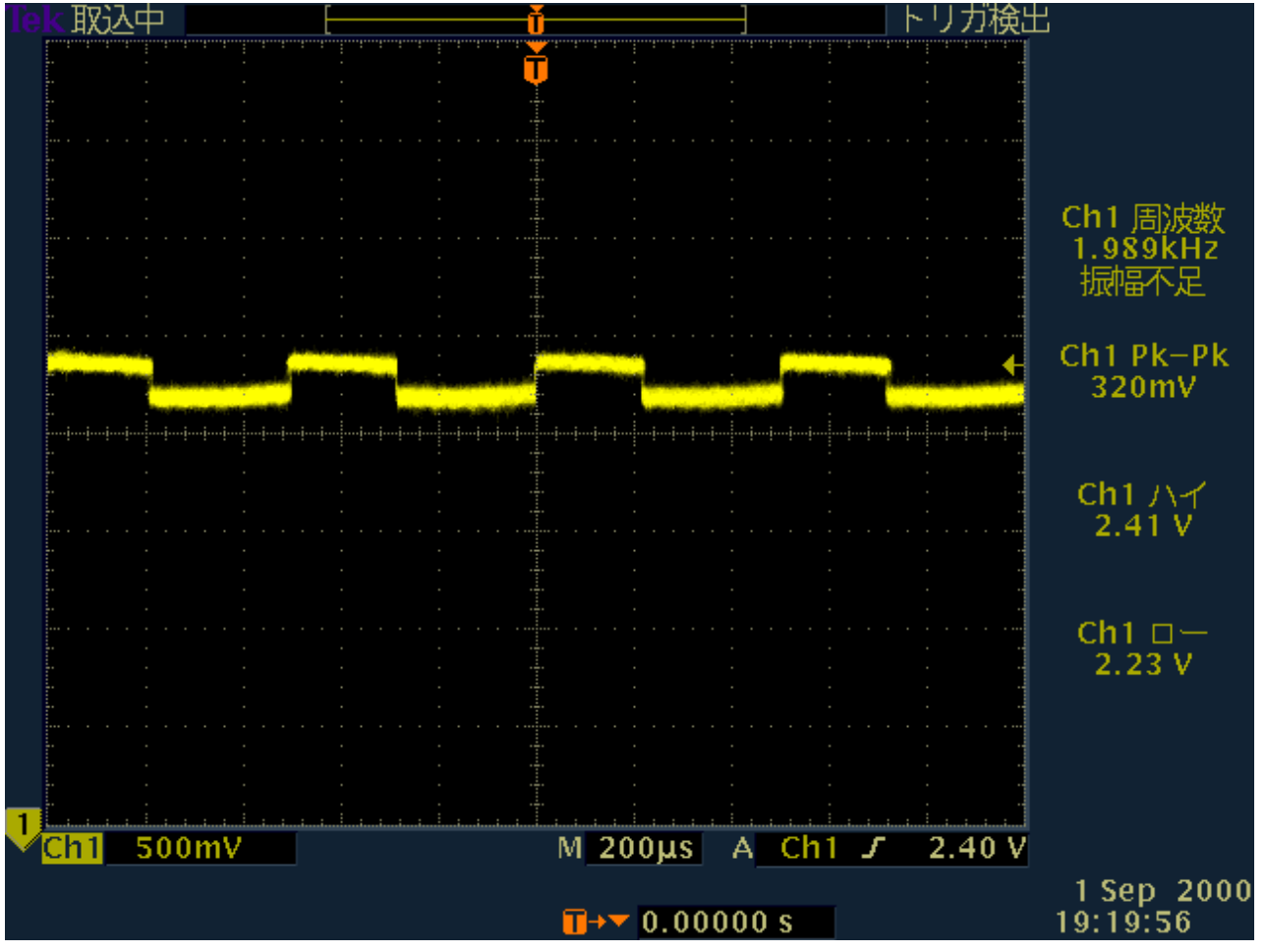
形式的な原因：チップ製造を担当した会社の設計ガイドラインに、コアロジックの消費電力から電源ピン数等への制約がなかった（らしい）

負荷変動に電源が対応できない。

チップが動作時と非動作時で2倍くらい消費電力が違う。動作時間は$10\mu s$ から $1\mathrm{ms}$ 程度。ボード1枚では $70\mathrm{A}$ と $140\mathrm{A}$ の間で変わる

- 普通のスイッチング電源では電流変化に応答できない
- 大容量の電解コンデンサをつけてもほとんど気休めにしかならない

電源波形


Tek 取込中
トリガ検出
Ch1 周波数
1.989kHz
振幅不足
Ch1 Pk-Pk
320mV
Ch1 ハイ
2.41 V
Ch1 ロー
2.23 V
Ch1 500mV
M 200µs A Ch1 2.40 V
0.00000 s
1 Sep 2000
19:19:56

負荷変動への対応

今から設計するなら

- スイッチング電源をフィードフォワード制御する
- 非動作時でも消費電力が下がらないように設計する

現在の対応

- ある程度応答の速い電源に変える
- 内部抵抗の小さい電解コンデンサを大量につける

Case Study からの教訓

結局は：
専用計算機も大きくなると大規模システムで起きるようなあらゆる問題が起きる。
というだけ。

あとは、、、

担当エンジニアのいうことは正しいとは限らない

↓

ボードやチップは専門家にまかせておけば出来るというものでもない

とはいえ、これはあまり専用計算機固有の問題というわけではない。

専用計算機は難しい？

「成功」といえるようなものはそんなに多くない。
多体系に話を限るとアプローチは2つ。どちらも結構いろいろある。

専用パイプライン

- DMDP (Delft)
- FASTRUN
- GRAPE
- MD-Engine
- MDM

並列計算機

- Digital Orrery
- Transputer-based projects...
- HaMM
- **他にも沢山**

あんまりうまくいかないのはなぜ？

どれが成功でどれが失敗かというのはさておき、、、
基本的な理由は 2 つ。

1. 出来たものが実は使えなかった。
2. 出来た時には速度が汎用計算機より速くなかった。

理由 (1) は専用計算機に固有だが、例は少ない（公表されていないのがあるのかもしれないが）
理由 (2) は普通の計算機と同じ。開発に時間がかかるとムーアの法則の分相対的に遅くなる。

開発期間の問題

大抵の人は（私を含めて）見積もりが甘いというのは確か。

本質的な問題：
専用計算機の場合、開発期間が多少（1-2年）延びたくらいで意味がなくなるようなプロジェクトは、そもそもあまり意味がない（出来てからのハードウェアの寿命がどうせ短い）。
大雑把には、計画時点で

- 価格性能比が 1000倍 — 問題なし
- 価格性能比が 100倍 — ちょっと苦しい
- 価格性能比が 10倍 — やめたほうがいい

別の方向

開発に時間がかからないなら話は別: Beowulf

安い DSP を使うのはやはりそれなりに時間がかかる (QCDSP)

Dreamcast, PS2 ???

価格性能比 100 倍は可能か？

あくまでも「計画時点」。とはいえ、、、
半導体技術は 1-2 年先を見込める: 1.5-2 倍得
大量生産されるマイクロプロセッサほどクロックはあげられない: 3-5 倍損
結局、

- 演算数が同じならチップの値段を 1/100 にする
- 値段が同じなら演算数を 100 倍にする

必要がある。前者は多分無理なので、演算数を増やす以外の方法は多分ない。

100個の演算器を1チップに

マイクロプロセッサはクロックあたり1演算程度しかしない
$\rightarrow$ 100個の演算器が 1 チップに入れば OK。

入れるだけならなんということはない。 10^7 トランジスタもあれば十分。

どうやって使うか

- チップ外への通信速度
- 演算器間の通信速度
- 内部メモリ、レジスタと演算器の間の通信速度

GRAPE の場合

- 実／仮想マルチパイプラインによるチップ外通信速度の削減
- ハードワイヤドパイプラインによる最小コストでの演算器間接続
- 同じくハードワイヤドのため内部メモリ不要

通常 on-chip multiprocessor で起きる問題をほぼ完全に回避している。
基本的にはそういうことが出来る問題だから。
逆にいえば、そういうことが出来る問題（アルゴリズム）でないと専用計算機はうまくいかない。

専用計算機の今後

GRAPE アーキテクチャ：汎用計算機に比べた相対的優位はだんだん大きくなる

ムーアの法則でトランジスタは増える

汎用計算機では増えたトランジスタの全部を演算器には使ってない
GRAPE は全部使える

ムーアの法則が成り立たなくなる頃 ???

FPGA はどうか

小規模な機械：短い開発期間で出来る（ソフトウェアを別にすれば）
が、カスタムに比べれば遅くて回路規模が小さい

大規模な機械：開発に時間がかかるという問題が復活する
FPGA を（数年に渡って）更新可能な設計をすれば、、、

汎用計算機の今後 ???

本当はこっちが問題。
GRAPE: 汎用計算機につける「アクセラレータ」
5 年後、10 年後の（科学技術計算用）汎用計算機はどんなものか？（そんなものがあるのか？）

専用計算機についてのまとめ

- GRAPE では、もっとも演算量の多い相互作用の計算だけを専用化し、他の計算は汎用計算機に回すことで高い性能と柔軟性を両立させている。
- この分担のためにハード開発は比較的容易である。
- このような分担をうまく機能させるためには、採用するアルゴリズムがそういうふうにできている必要がある。可能ならアルゴリズムの変更も考えるべき。
- 5-10 年先でも GRAPE の方法は多分有効
- 重力多体系以外にこういう方法が有効かどうかはいまだよくわからない。