シミュレーションが 未来をひらく

CMSI計算科学技術 特論B

第2回
アプリケーションの性能最適化1
（高並列性能性能最適化）

2014年4月17日

独立行政法人理化学研究所
計算科学研究機構 運用技術部門
ソフトウェア技術チーム チームヘッド

南 一生
minami_kaz@riken.jp

講義の概要

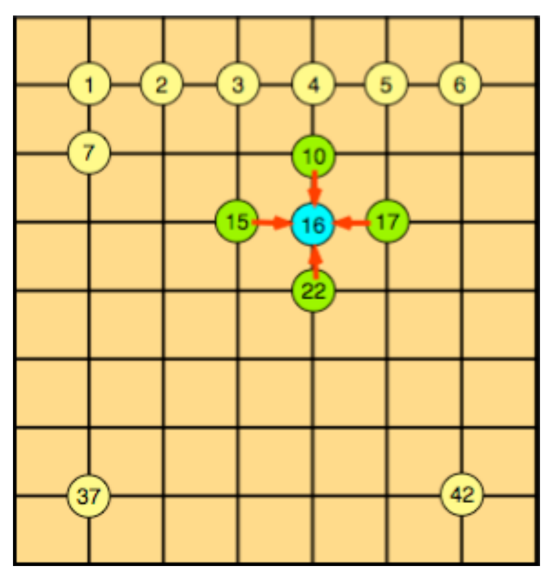
- スーパーコンピュータとアプリケーションの性能
- アプリケーションの性能最適化1（高並列性能最適化）
- アプリケーションの性能最適化2（CPU単体性能最適化）
- アプリケーションの性能最適化の実例1
- アプリケーションの性能最適化の実例2

内容

- 反復法
- アプリケーションの性能最適化
- 簡単な実例
- 並列性能のボトルネック

反復法

差分法と連立一次方程式



- 左図のような2次元の領域で微分方程式を解くとする.
- 5点差分で差分化すると自分自身の差分点と周りの4つの差分点の式ができる.

$$C_{1,3}u_1 + C_{1,4}u_2 + C_{1,5}u_7 = \alpha_1$$ ①の差分点について

$$C_{2,2}u_1 + C_{2,3}u_2 + C_{2,4}u_3 + C_{2,5}u_8 = \alpha_2$$ ②の差分点について

. . . .

$$C_{16,1}u_{10} + C_{16,2}u_{15} + C_{16,3}u_{16} + C_{16,4}u_{17} + C_{16,5}u_{22} = \alpha_{16}$$ ⑯の差分点について

. . . .

- 全部で42点の差分点が存在するので42元の連立方程式となる.
- したがって元の微分方程式は, 差分化により以下のような連立一次方程式を解く事に帰着させる事ができる.

$$\mathbf{Ax} = \mathbf{b}$$

反復法の一般的な議論

- 左のような連立一次方程式を考える.　$\mathbf{Ax} = \mathbf{b}$　・・・①
- Aを右のように書き直す. I は単位行列.　$\mathbf{A} = \mathbf{I} - \mathbf{B}$　・・・②
- ②を①に代入し整理する.　$(\mathbf{I} - \mathbf{B})\mathbf{x} = \mathbf{b}$

$$\mathbf{x} = \mathbf{Bx} + \mathbf{b} \quad \cdots ③$$

- ③を満たすベクトルの列：$\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \mathbf{x}^3 \cdots$ を考える.

$$\mathbf{x}^{(m+1)} = \mathbf{Bx}^{(m)} + \mathbf{b} \quad \cdots ④$$

- ここで④の両辺の極限を取れば⑤のようになる

$$\lim_{m\to\infty} \mathbf{x}^{(m+1)} = \mathbf{B} \lim_{m\to\infty} \mathbf{x}^{(m)} + \mathbf{b} \quad \cdots ⑤$$

- 極限ベクトルをxとおけば⑤は⑥の様になる　$\mathbf{x} = \mathbf{Bx} + \mathbf{b}$　・・・⑥
- ⑥は③と全く同一となる. つまり連立一次方程式：①を解く事は④を満たすベクトルの列：　$\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \mathbf{x}^3 \cdots$　を見つける事に帰着する.

各反復法

- ④を解くための反復法について説明する　$\mathbf{x}^{(m+1)} = \mathbf{B}\mathbf{x}^{(m)} + \mathbf{b}$ ・・・④
- Aは正則，Aの対角要素は全て0でない複素数であると仮定する．
- 解くべき方程式　$\mathbf{A}\mathbf{x} = \mathbf{k}$ ・・・⑦
- Aを⑧のように変形．DはAの対角成分のみの行列．EはAの狭義下三角行列の符号を変えた行列．FはAの狭義上三角行列の符号を変えた行列．

$$\mathbf{A} = \mathbf{D} - \mathbf{E} - \mathbf{F} \quad \cdots ⑧$$

$$\mathbf{D}\mathbf{x} = (\mathbf{E} + \mathbf{F})\mathbf{x} + \mathbf{k} \quad \cdots ⑨$$

- ⑧を⑦に代入し⑨を得る

$$\mathbf{x}^{(m+1)} = \mathbf{D}^{-1}(\mathbf{E} + \mathbf{F})\mathbf{x}^{(m)} + \mathbf{D}^{-1}k \quad \cdots ⑩$$

- ⑨を④の反復法にすると⑩を得る．
- この反復法を**点ヤコビ法**という．
- 具体的には⑪の計算法となる．
- 注目すべきは（m+1）世代の解を得るために（m）世代の解しか使用しない点である．

$$\left.\begin{aligned} a_{ii}x_i^{(m+1)} &= -\sum_{\substack{j=1\\ j\neq i}}^{n} a_{ij}x_j^{(m)} + k_j \\ x_i^{(m+1)} &= -\sum_{\substack{j=1\\ j\neq i}}^{n} \left(\frac{a_{ij}}{a_{ii}}\right) x_j^{(m)} + \frac{k_i}{a_{ii}} \end{aligned}\right\} \quad \cdots ⑪$$

K computer

各反復法

- ⑨を⑫のように変形する.

$$(\mathbf{D}-\mathbf{E})\mathbf{x}=\mathbf{F}\mathbf{x}+\mathbf{k} \quad \cdots ⑫$$

- ⑫を④の反復法にすると⑬を得る.

$$(\mathbf{D}-\mathbf{E})\mathbf{x}^{(m+1)}=\mathbf{F}\mathbf{x}^{(m)}+\mathbf{k} \quad \cdots ⑬$$

- この反復法を**点ガウス・ザイデル法**という.

$$a_{ii}x_i^{(m+1)}=-\sum_{j=1}^{i-1}a_{ij}x_j^{(m+1)}-\sum_{j=1+1}^{n}a_{ij}x_j^{(m)}+k_j \quad \cdots ⑭$$

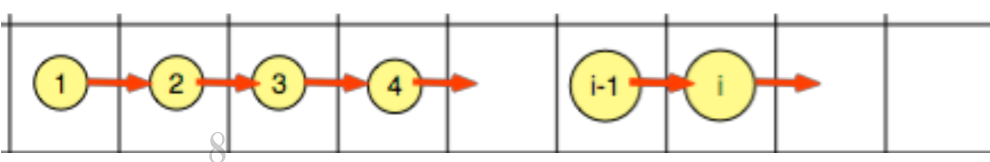
- 具体的には⑭の計算法となる.
- 注目すべきは (m+1) 世代の解を得るために (m+1) 世代の下三角行列を使用している点である **(リカレンスの発生)** .

- 右図のように, iを定義するのにi-1の値を参照するコーディングは下図のような参照関係となり,リカレンス (回帰参照) が発生している.

```
do i=1,n
```

$$x(i)=a(i)*x(i-1)+b(i)$$

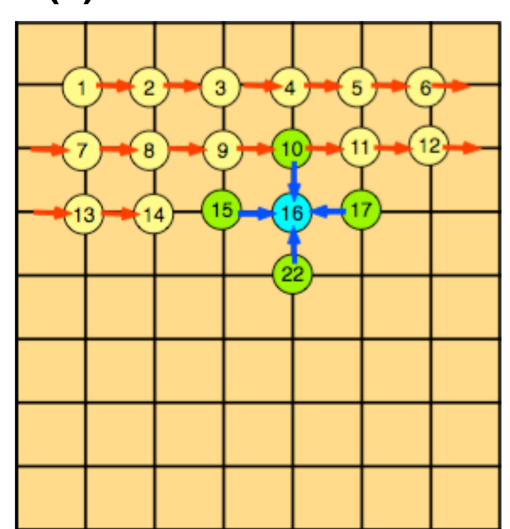
- **リカレンスがあると, iについては依存関係があり並列処理ができない.**



各反復法

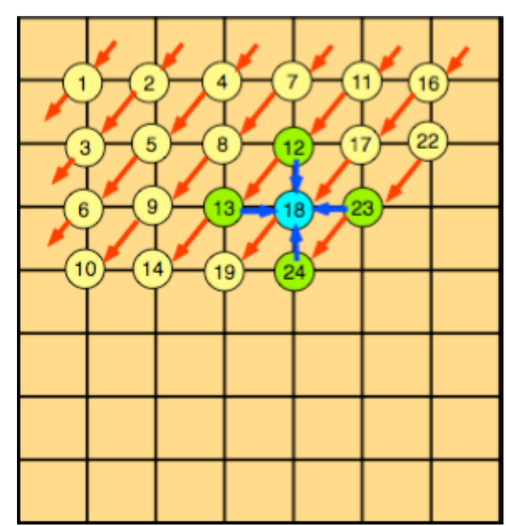
- ヤコビ法やガウス・ザイデル法はメッシュの**計算順序を変更しても収束する**事が数学的に証明されている.
- ガウス・ザイデル法オリジナルの計算順序は前ページのようにリカレンスのために並列化できない.
- そこで (b) のハイパープレーン法や (c) のRED-BLACK法を用いループ内のリカレンスによる依存関係をなくす。
- 依存関係をなくす事で前回講義のようにハードウェアの並列計算機構を使い高速に計算可能となる. また並列化そのものにも使用可能.

(a)オリジナルスイープ



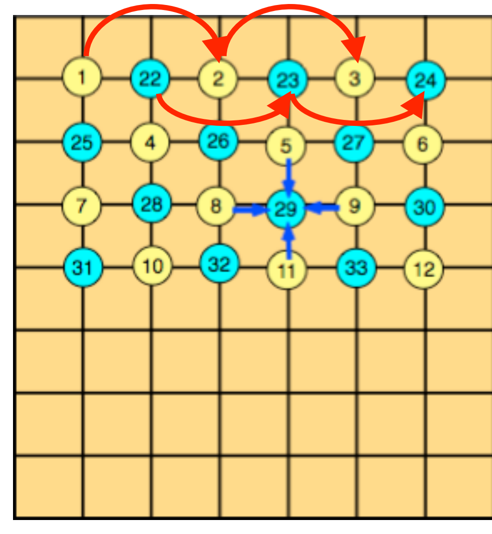
参照関係
計算順序(スイープ順序)

(b)ハイパープレーンスイープ



- 斜めのハイパープレーン内はリカレンスがない.
- ハイパープレーン間のみにリカレンスが発生.

(c)RED-BLACKスイープ



- 黄と青の2つのループに再構成する.
- それぞれのループ内はリカレンスがない.

共役勾配法 (CG法)

- ①の2次の関数を考える.
- この関数の最小値問題を解くためにf (x) の微分を取り0とおく.
- ②より①の最小値問題を解く事がax=bの解を得る事と同等であることが分かる.

$$f(x) = \frac{1}{2}ax^2 - bx \quad \cdots ①$$

$$f'(x) = ax - b = 0 \quad \cdots ②$$

- 上記と同様な事を多次元のベクトルで行う.
- ①と同等な関数として③を考える.
- ③を成分で標記すると④のようになる.

$$f(\mathbf{x}) = \frac{1}{2}(\mathbf{x}, \mathbf{A}\mathbf{x}) - (\mathbf{b}, \mathbf{x}) \quad \cdots ③$$

$$f(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_i x_j - \sum_{i=1}^{n} b_i x_i \quad \cdots ④$$

- ④をxで偏微分すると⑤になる.
- 行列Aの対称性を使い0とおくことにより⑥が得られる.
- ⑥は連立一次方程式⑦の成分表示である.

$$\frac{\partial f(\mathbf{x})}{\partial x_k} = \frac{1}{2}\sum_{i=1}^{n} a_{ik}x_i + \frac{1}{2}\sum_{j=1}^{n} a_{kj}x_j - b_k \quad \cdots ⑤$$

$$\frac{\partial f(\mathbf{x})}{\partial x_k} = \sum_{i=1}^{n} a_{ki}x_i - b_k = 0 \quad \cdots ⑥$$

- したがって③の最小値問題を解く事が連立一次方程式⑦を解くこととなる.

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \cdots ⑦$$

共役勾配法（CG法）

$$r_0 = b - A x_0$$

$$p_0 = r_0$$

for $i = 0, 1, 2, \cdots$

$$\alpha_i = \frac{(r_i, r_i)}{(p_i, A p_i)}$$

$$x_{i+1} = x_i + \alpha_i p_i$$

$$r_{i+1} = r_i - \alpha_i A p_i$$

$$\beta_i = \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)}$$

$$p_{i+1} = r_{i+1} + \beta_i p_i$$

- CG法アルゴリズムの基本形を左図に示す.
- この他に色々なアルゴリズムの派生形がある.
- ③式のf(x)が最小値が得られるようにxiの列を求めて行く.
- ri は残差ベクトルであり互いに直交し一次独立であることが証明されている.
- したがって残差ベクトルはN元の連立一次方程式にはN個しか存在しない.
- CG法を使うとN元の連立一次方程式は高々N回の反復で収束する.
- またAの固有値が縮重しているか密集していれば収束が速くなる性質を持つ.

K computer

前処理付き共役勾配法(CG法)

- Aに近い正値対称行列:Mを考えコレスキー分解する(①式).
- ここで②のようなxの置き換えを行う.
- これを使ってもとのAx=bを変形する(③④式)
- ここで⑤で置き換えたAの性質をみてみる.
- Mは⑥を満たすので①より⑦を満たす.
- ⑤に⑦を適用すると⑧をみたし**置き換えたAの性質は単位行列に近い**ことが分かる.
- したがって⑨によって**置き換えた行列Aの固有値は1の周りに密集している**ものと考えられる
- 置き換えた行列とベクトルに対する連立一次方程式:**⑩の収束は早いものと期待**できる.

$$\mathbf{M} = \mathbf{U}^T\mathbf{U} \quad \cdot\cdot\cdot ①$$

$$\tilde{\mathbf{x}} = \mathbf{U}\mathbf{x} \quad \cdot\cdot\cdot ②$$

$$\mathbf{A}\mathbf{U}^{-1}\tilde{\mathbf{x}} = \mathbf{b} \quad \cdot\cdot\cdot ③$$

$$\mathbf{U}^{-T}\mathbf{A}\mathbf{U}^{-1}\tilde{\mathbf{x}} = \mathbf{U}^{-T}\mathbf{b} = \tilde{\mathbf{b}} \quad \cdot\cdot ④$$

$$\mathbf{U}^{-T}\mathbf{A}\mathbf{U}^{-1} = \tilde{\mathbf{A}} \quad \cdot\cdot ⑤$$

$$\mathbf{A} \approx \mathbf{M} \quad \cdot\cdot\cdot ⑥$$

$$\mathbf{A} \approx \mathbf{U}^T\mathbf{U} \quad \cdot\cdot\cdot ⑦$$

$$\mathbf{U}^{-T}\mathbf{A}\mathbf{U}^{-1} \approx \mathbf{U}^{-T}(\mathbf{U}^T\mathbf{U})\mathbf{U}^{-1} = \mathbf{I} \quad \cdot ⑧$$

$$\tilde{\mathbf{A}} = \mathbf{U}^{-T}\mathbf{A}\mathbf{U}^{-1} \quad \cdot\cdot ⑨$$

$$\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}} \quad \cdot\cdot\cdot ⑩$$

前処理付き共役勾配法 (CG法)

- 不完全コレスキー分解による前処理CG法アルゴリズムの基本形を下図に示す.
- 不完全コレスキー分解は元の行列要素の位置のみを計算する.
- 0要素に対するフィルインは計算しない.
- 計算は行列ベクトル積・ベクトルスカラ積・ベクトルの和・内積・除算で構成される.
- 赤線で示した前処理部分は前進代入・後退代入で計算される.

ステップ1: $$\alpha^k = (r_i^k \bullet \underline{(LL^T)^{-1} r_i^k})/(Ap_i^k \bullet p_i^k)$$

ステップ2: $$x_i^{k+1} = x_i^k + \alpha^k p_i^k$$

ステップ3: $$r_i^{k+1} = r_i^k - \alpha^k A p_i^k$$

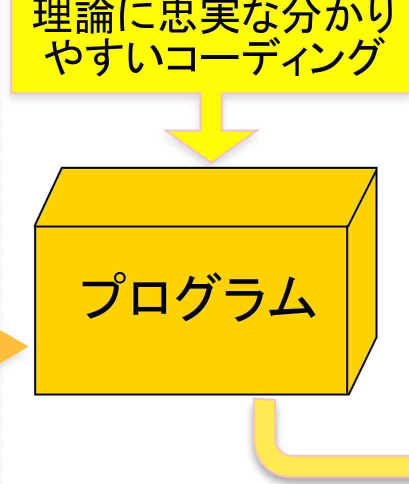
ステップ4: $$\beta^k = (r_i^{k+1} \bullet \underline{(LL^T)^{-1} r_i^{k+1}}) / (r_i^k \bullet \underline{(LL^T)^{-1} r_i^k})$$

ステップ5: $$p_i^{k+1} = \underline{(LL^T)^{-1} r_i^{k+1}} + \beta^k p_i^k$$

アプリケーションの性能最適化

計算科学　　計算機科学

- 問題設定
- 理論構築
- 定式化
- 離散化
- プログラム作成
- コンパイル
- デバッグ
- プロダクションRUN
- グラスィックス処理
- 結果の解釈
- 論文作成
- 評価



現代のプログラミング（スパコンを使う場合）

- アプリケーションの**超並列性**を引き出す
- プロセッサの**単体性能**を引き出す

プログラム

高並列化・
高性能コーディング

- 言語
- コンパイラ
- 開発・実行環境
- ハードウェア
- システム運用

計算科学

計算機科学

問題設定
理論構築
定式化
離散化
プログラム作成
コンパイル
デバッグ
プロダクションRUN
グラスィックス処理
結果の解釈
論文作成
評価

理論に忠実な分かりやすいコーディング

10万行以上のプログラム

現代のプログラミング（スパコンを使う場合）

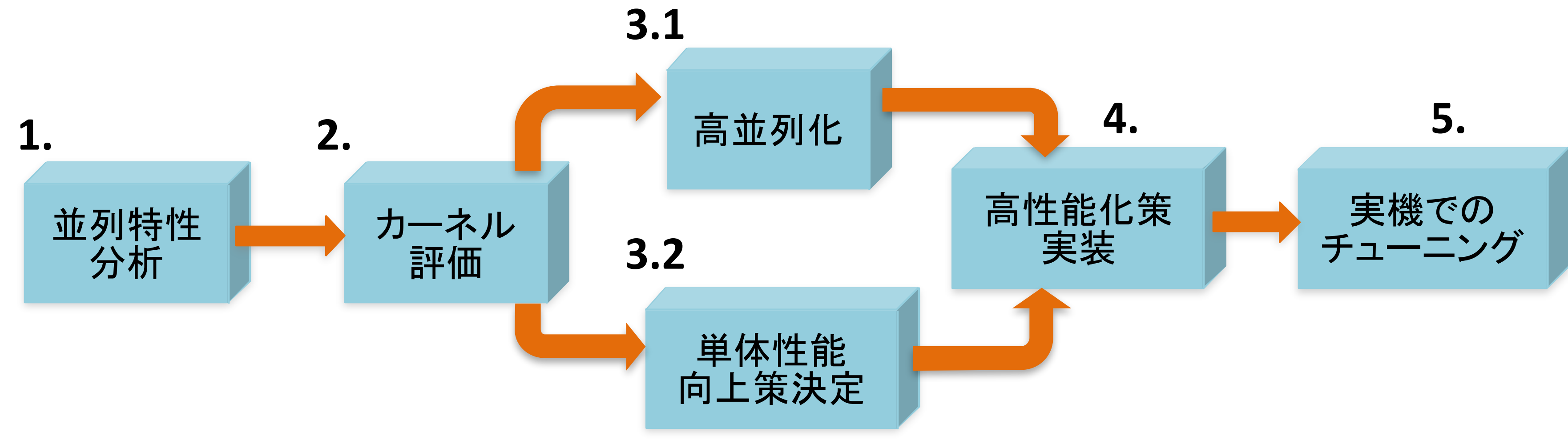
- アプリケーションの**超並列性**を引き出す
- プロセッサの**単体性能**を引き出す

プログラム

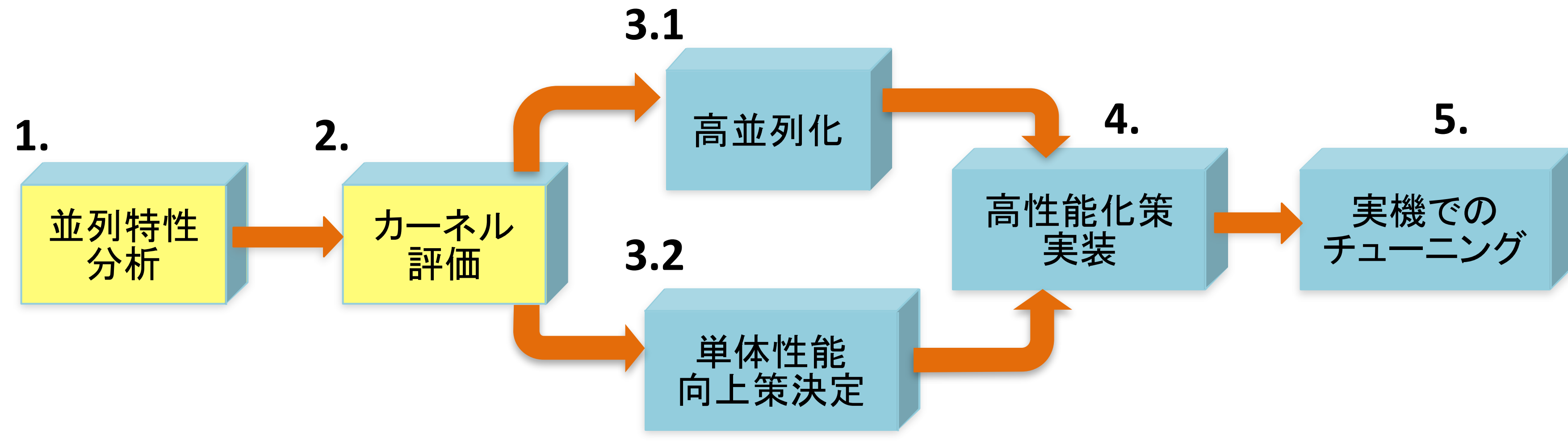
高並列化・高性能コーディング

言語
コンパイラ
開発・実行環境
ハードウェア
システム運用

アプリケーションの性能最適化のステップ



並列特性分析・カーネル評価



1. 並列特性分析
(処理構造分析・ブロック特性分析)

(1)コードの構造を分析し物理に沿った処理ブロック(計算/通信)に分割
(2)コードの実行時間の実測
(3)プログラムソースコードの調査
(4)処理ブロックの物理的処理内容を把握
(5)計算ブロック毎の計算特性把握(非並列/完全並列/部分並列、Nに比例/N**2に比例等)
(6)通信ブロック毎の通信特性把握 (グローバル通信・隣接通信、隣接面に比例&隣接通信/体積に比例、等)

	実行時間・スケーラビリティ	物理的処理内容	演算・通信特性	演算・通信見積り	カーネル
ブロック1(計算)			部分並列	Nに比例	
ブロック2(計算)			完全並列	N**3に比例	○
(通信)			隣接通信	隣接面に比例	○
ブロック3					

2. カーネル評価

(1)計算・通信ブロックについて物理的処理内容・コーディングの評価を行い同種の計算・通信ブロックを評価し異なる種類の計算・通信ブロックをカーネルの候補として洗い出す
(2)並列特性分析の結果から得た問題規模に対する依存性の情報を元にターゲット問題実行時に、また高並列実行時にカーネルとなる計算・通信カーネルを洗い出す

ブロック毎の実行時間とスケーラビリティ評価(例)

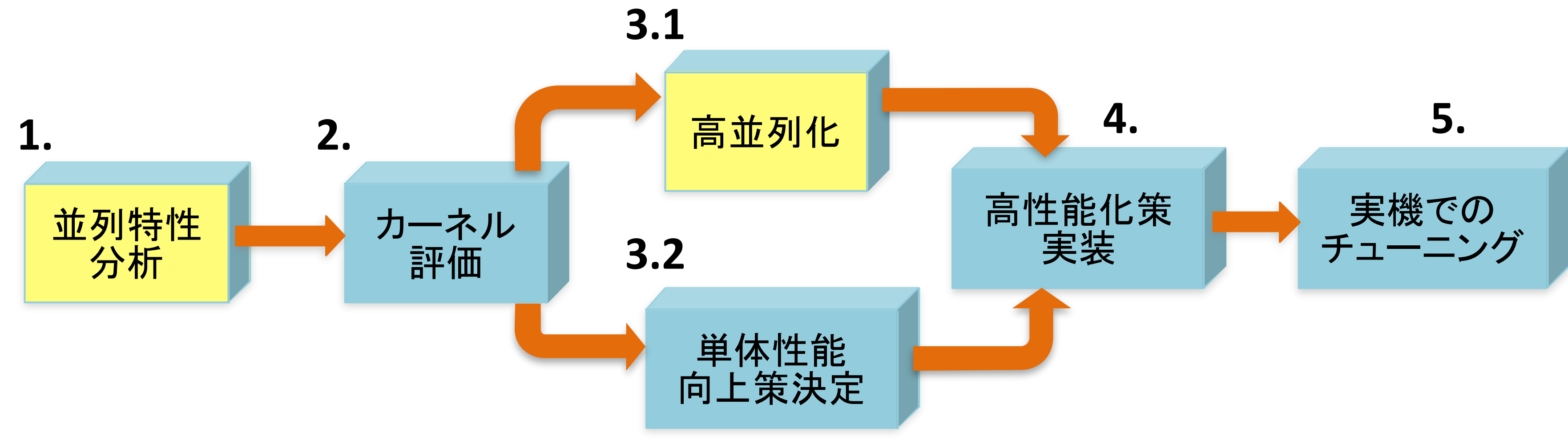
→従来の評価はサブルーチン毎・関数毎等の評価が多い
→サブルーチン・関数は色々な場所で呼ばれるため正しい評価ができない

全体
処理ブロック1
処理ブロック2
処理ブロック3
処理ブロック4
処理ブロック5
処理ブロック6
m_FFT_WF

10.0
1.0
スケーラビリティ

128
256
512
1024
並列数

並列特性分析・高並列化



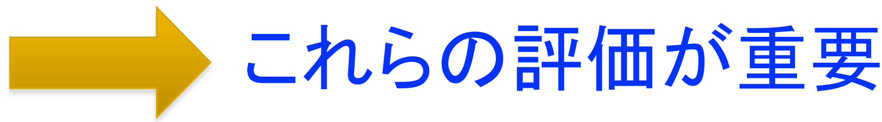
アプリケーションを高並列性を評価するためには？

- ✓ 十分な並列度を得る並列化手法を採用する
- ✓ 非並列部分を最小化する
- ✓ 通信時間を最小化する
- ✓ ロードインバランスを出さない

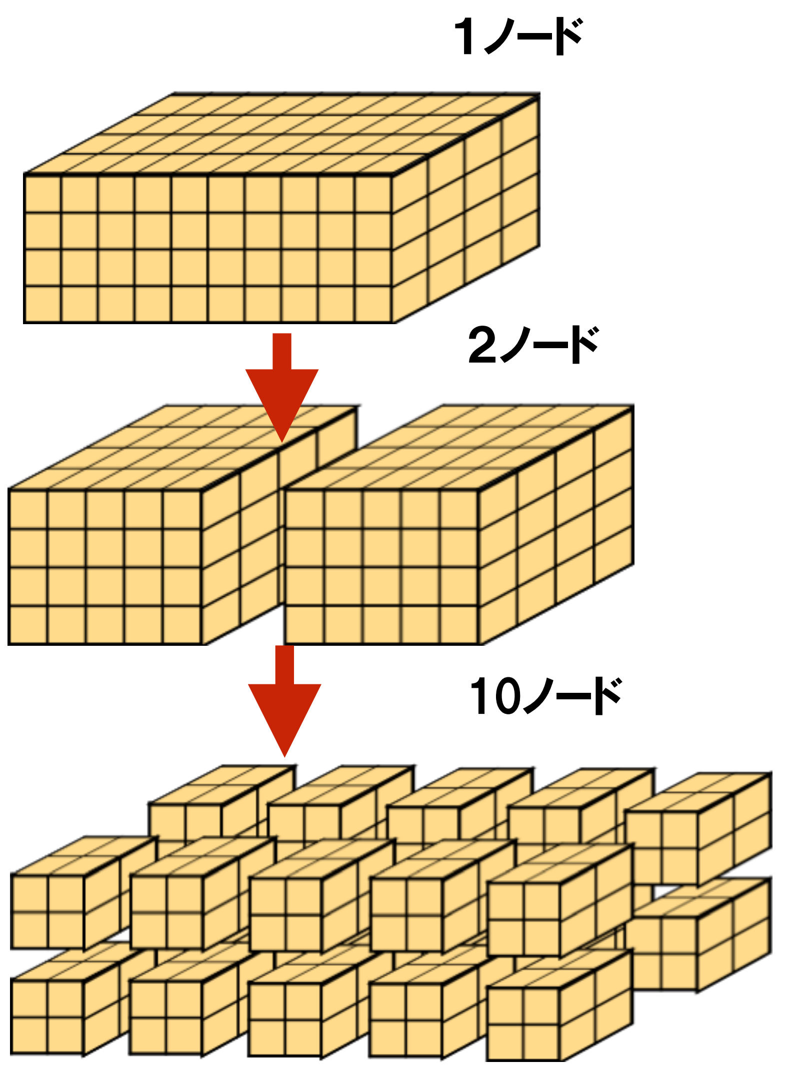
**最初の並列特性分析のフェーズ
および高並列化のステップで
アプリケーションの高並列を阻害する
要因を洗い出す事が重要**

超高並列を目指した場合の留意点−ブロック毎に以下を評価する

- 非並列部が残っていないか？残っている場合に問題ないか？
- 隣接通信時間が超高並列時にどれくらいの割合を占めるか？
- 大域通信時間が超高並列時にどれくらい増大するか？
- ロードインバランスが超高並列時に悪化しないか？

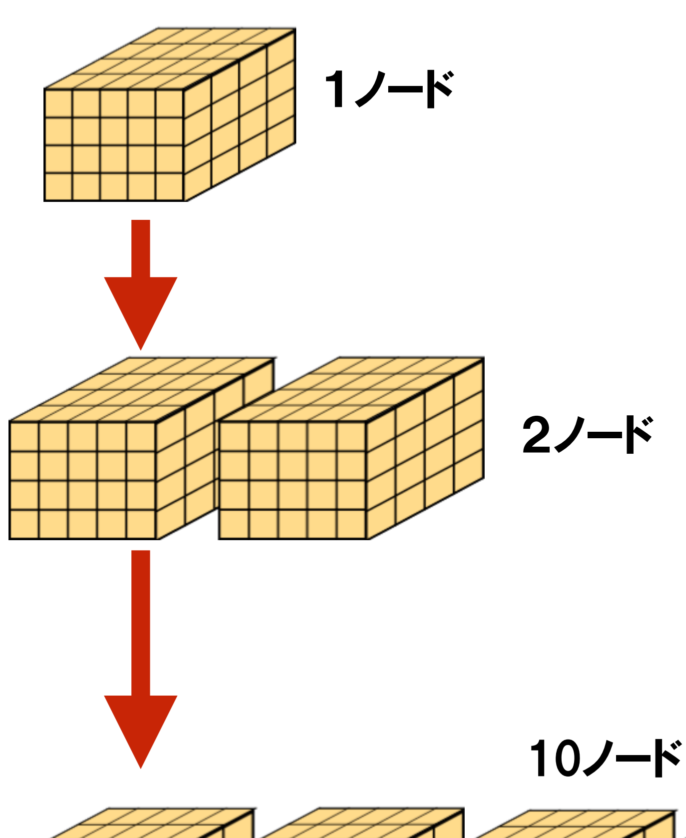


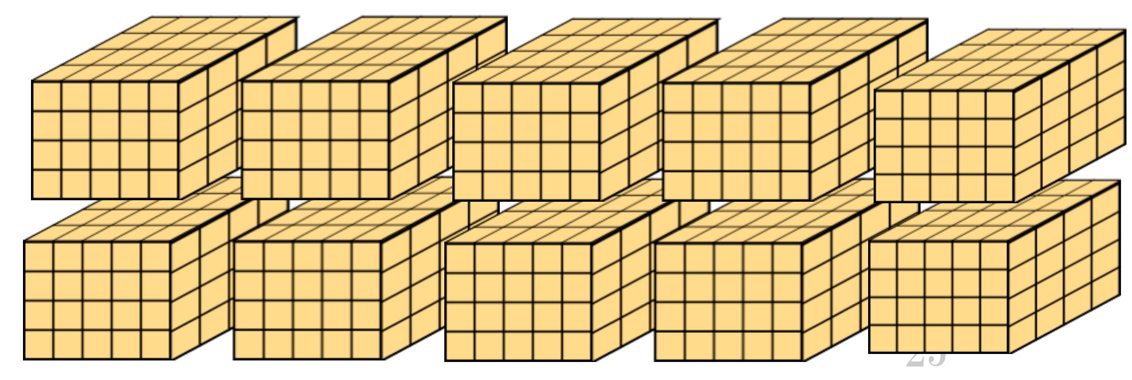
ストロングスケール測定



- 全体の問題規模を一定にして測定する方法.
- ここでは4×4×10が全体の問題規模.
- 2ノード分割では1ノードあたりの問題規模は4×4×5となる.
- 10ノード分割では1ノードあたりの問題規模は2×2×2となる.
- 問題を1種類作れば良いので測定は楽である.
- 並列時の挙動は見えにくい.

ウィークスケーリング測定

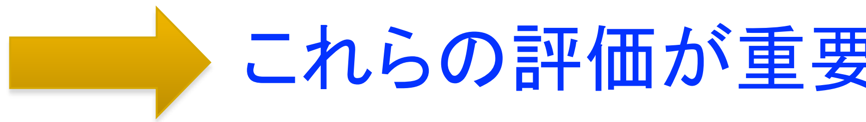




- 1プロセッサあたりの問題規模を一定にして測定する方法.
- ここでは4×4×5が1プロセッサあたりの問題規模.
- 2ノード分割では全体の問題規模は4×4×10となる.
- 10ノード分割では全体の問題規模は8×4×25となる.
- 問題を複数作る必要が測定は煩雑である.
- ただし並列時の挙動が見え易い.

超高並列を目指した場合の留意点–ブロック毎に以下を評価する

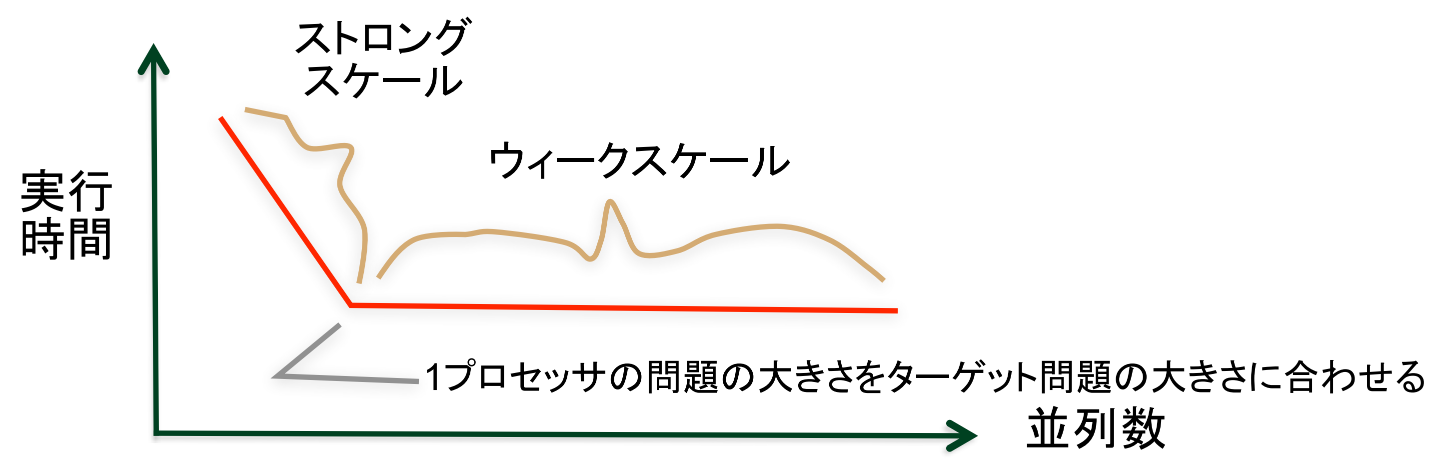
- 非並列部が残っていないか？残っている場合に問題ないか？
- 隣接通信時間が超高並列時にどれくらいの割合を占めるか？
- 大域通信時間が超高並列時にどれくらい増大するか？
- ロードインバランスが超高並列時に悪化しないか？



そのために

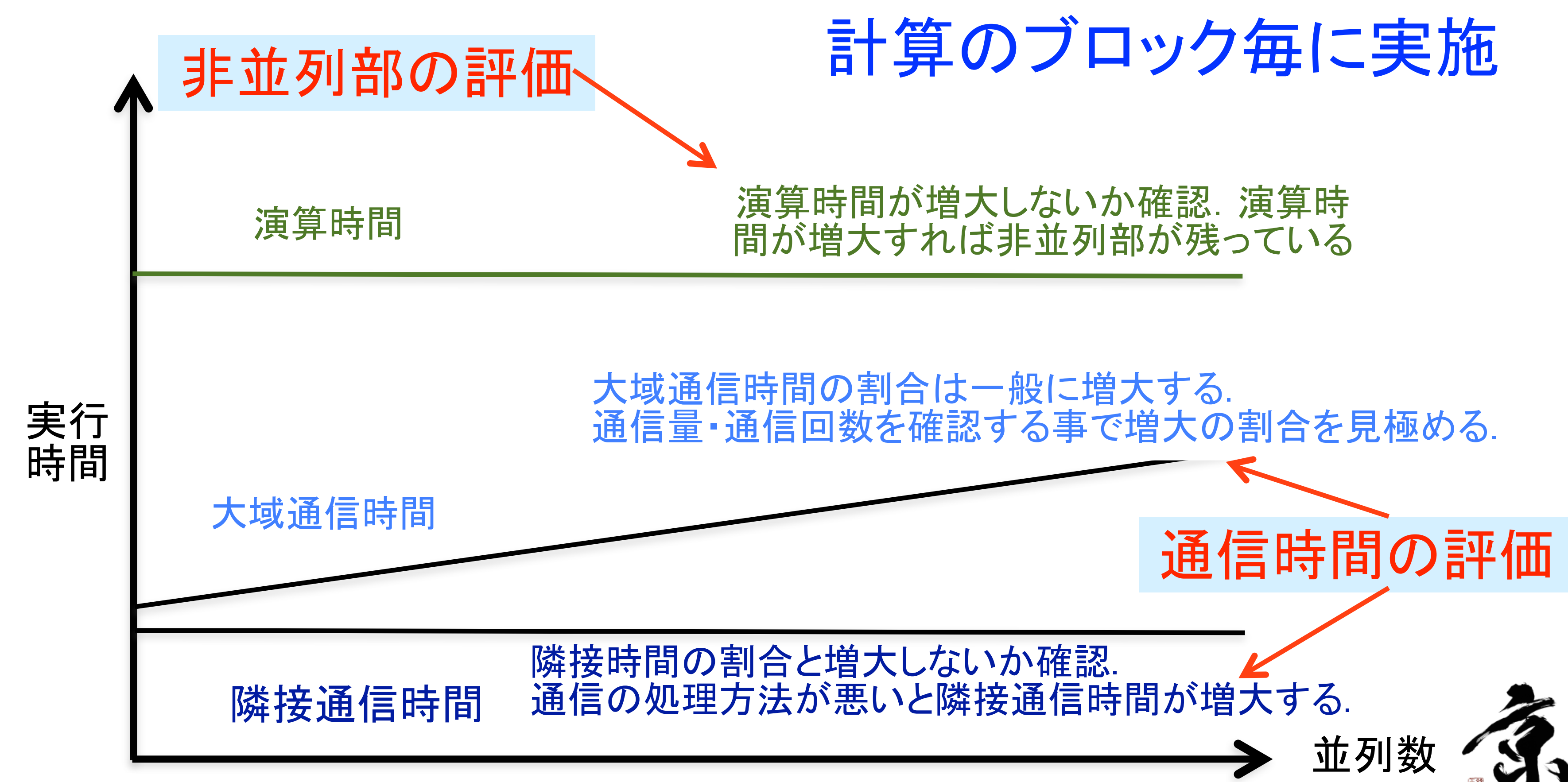
ストロングスケールとウィークスケーリング測定を使う

- ターゲット問題を決める
- 1プロセッサの問題規模がターゲット問題と同程度となるまでは、ストロングスケーリングで実行時間・ロードインバランス・隣接通信時間・大域通信時間を測定・評価する(100から数百並列まで)
- 上記もウィークスケーリングでできればなお良い
- 上記の測定・評価で問題が有れば解決する
- 問題なければ並列度を上げてウィークスケーリングで大規模並列の挙動を測定する



大規模並列のウィークスケーリング評価

- 現状使用可能な実行環境を使用し100程度/1000程度/数千程度と段階を追ってできるだけ高い並列度で並列性能を確認する(ウィークスケーリング測定).
- ウィークスケーリングが難しいものもあるが出来るだけ測定したい. 難しい場合は, 演算時間をモデル化して実測とモデルとの一致を評価する.



ストロングスケーリングだけどスケールをモデル化した例

(a)

Time per GS (sec.)

400.0
300.0
200.0
100.0
0.0

0
20,000
40,000
60,000
80,000

Number of cores

theoretical computation
computation
global/space
global/orbital
wait/orbital

(b)

Time per CG (sec.)

160.0
120.0
80.0
40.0
0.0

0
20000
40000
60000
80000

Number of cores

theoretical computation
computation
adjacent/space
global/space
global/orbital

(c)

Time per MatE/SD (sec.)

200.0
150.0
100.0
50.0
0.0

0
20,000
40,000
60,000
80,000

Number of cores

theoretical computation
computation
adjacent/space
global/space
global/orbital

(d)

Time per RotV/SD(sec.)

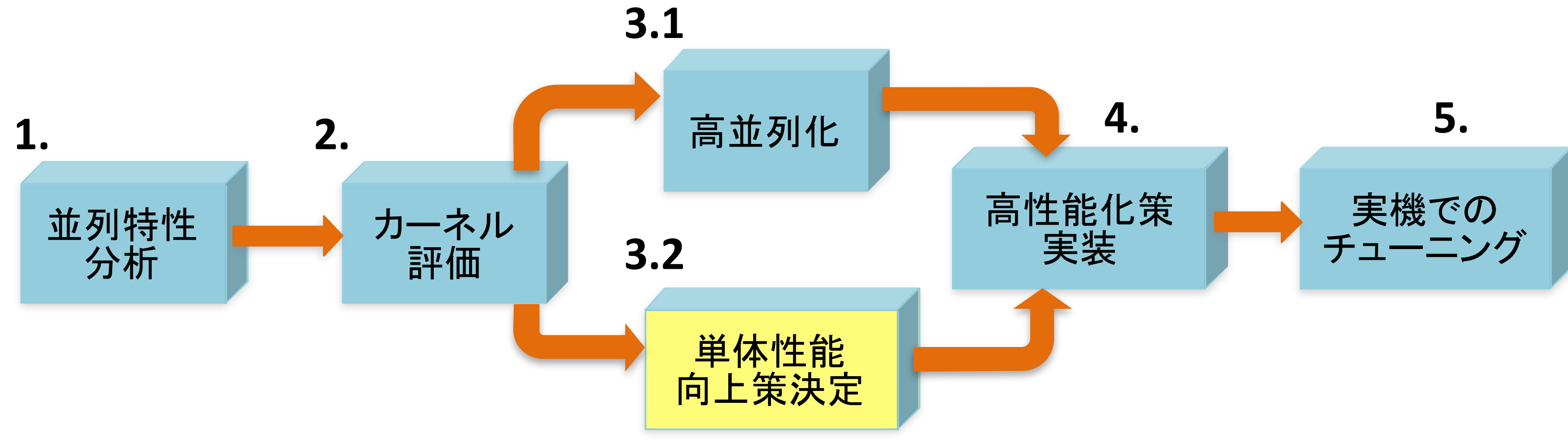
300.0
200.0
100.0
0.0

0
20,000
40,000
60,000
80,000

Number of cores

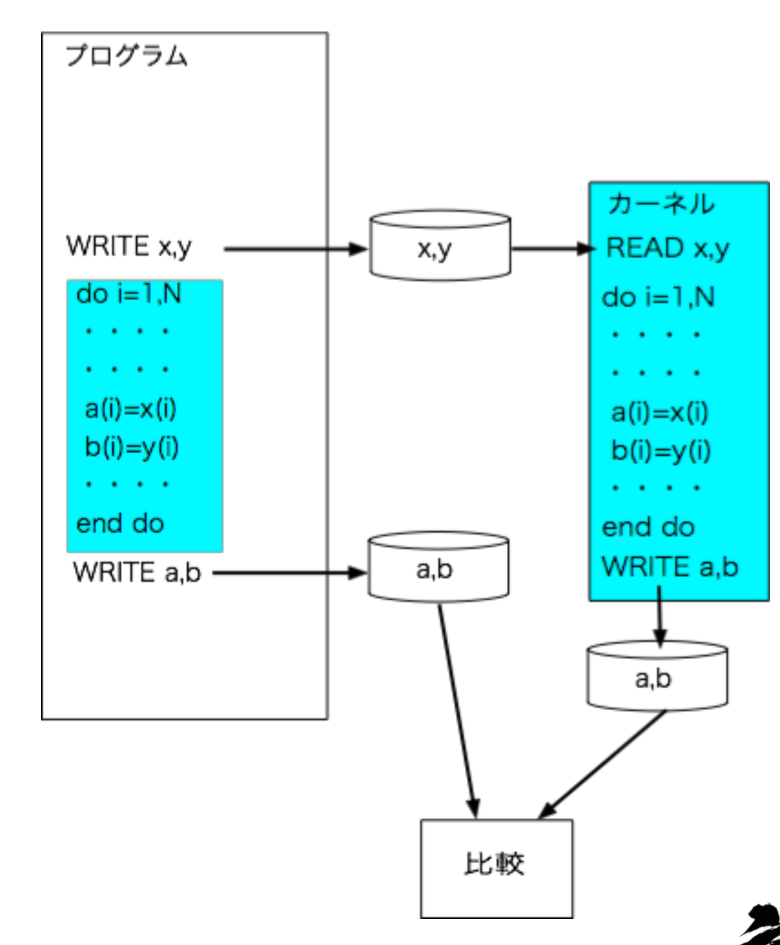
theoretical computation
computation
adjacent/space
global/space
global/orbital

単体性能向上策決定



カーネルの切り出しと性能向上策の試行

- 計算カーネルの切り出し→計算カーネルを独立なテストプログラムとして切り出す.
- 性能向上策の試行→切り出したテスト環境を使用し様々な性能向上策を試行する.
- 性能向上策の評価・決定→試行結果を評価し性能向上策の案を策定する.
- 作業量見積り→性能向上策を実施した場合のコード全体に影響する作業を洗い出す.
- それら作業を実施した場合の作業量を見積る.
- それらを評価し最終的に採用する案を決定する.



京の場合のCPU性能高速化技術

SPARC64™ VIIで実装した
技術を拡張

VISIMPACT

CPU

SPARC64™ VIIIfx
新規技術

HPC-ACE

科学技術計算を高速で処理
するための拡張命令セット

コア

演算器

レジスタ

ハードウェア
バリア同期

SIMD

浮動小数点
レジスタ拡張

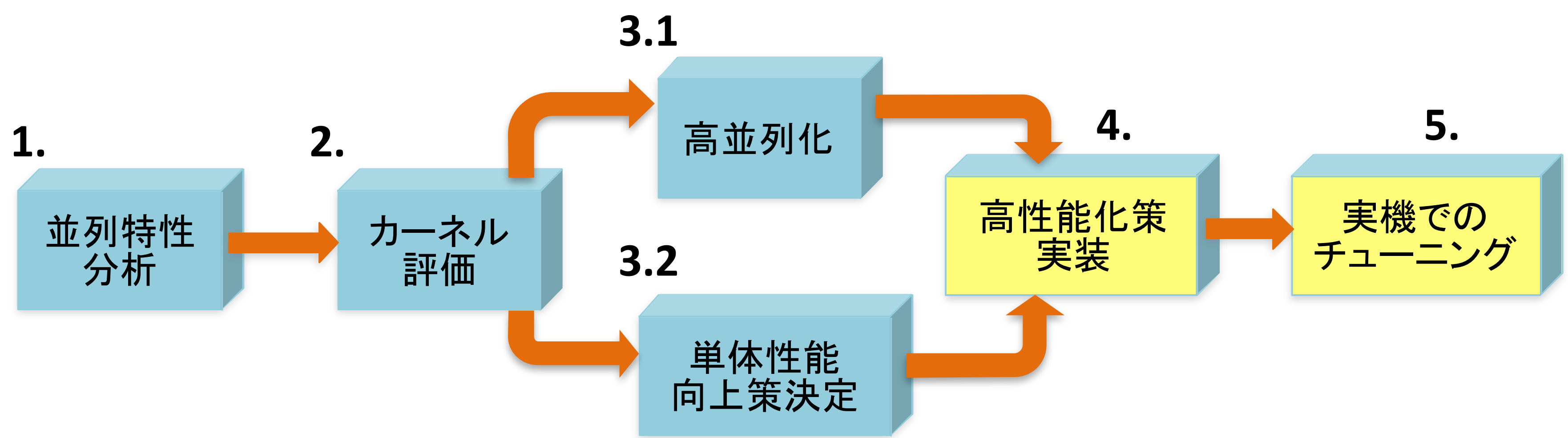
共有キャッシュ

セクタ
キャッシュ

メモリ

→コンパイラ/アプリケーションで、いかに
活用できるか/するかが高速化のキーポイント

高性能化策実装・実機でのチューニング



高性能化策実装

高並列化の対策を実施したコードに対して策定した単体性能向上策を実装する

設計・プログラミング・デバッグ作業であり作業工数的には大きな作業となる

実機でのチューニング

4まで実施されたコードをさらに実機に載せて全体としてチューニングする

(1)測定

・実機上で並列性能/単体性能を測定する.

(2)問題点洗い出し

・並列性能/単体性能のそれぞれについて問題点を洗い出す.

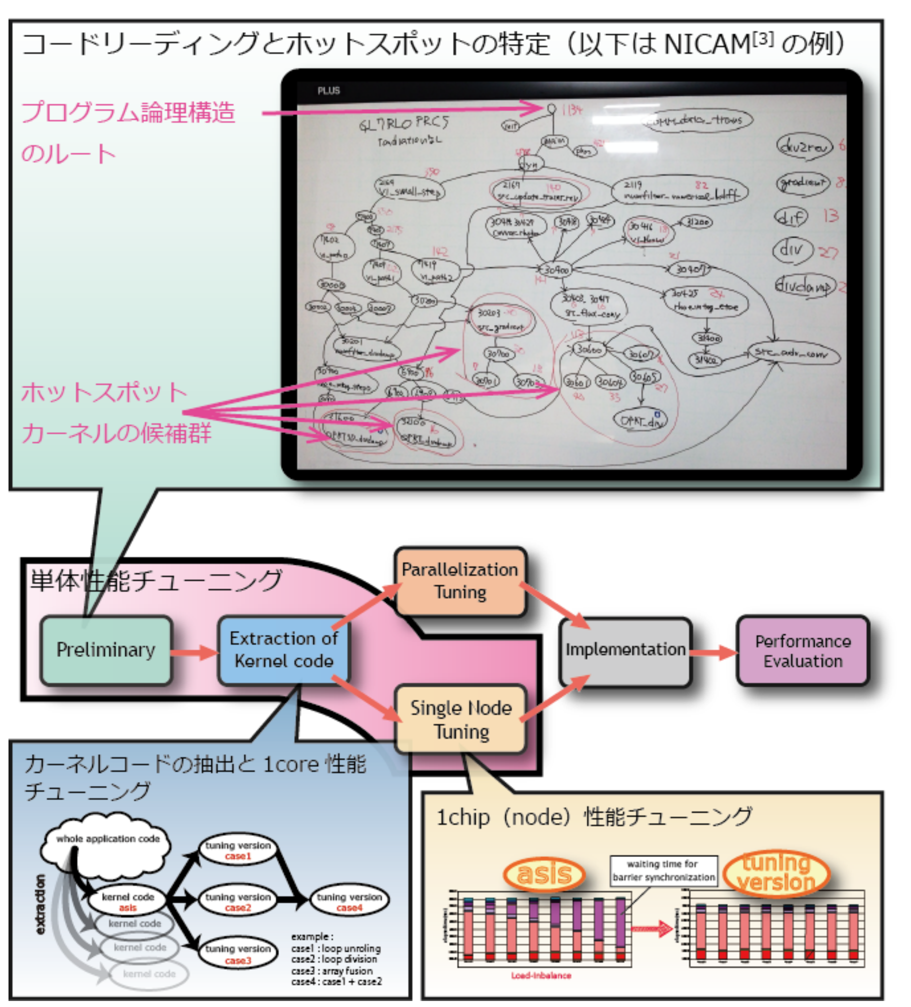
(3)問題点解決

・洗い出された問題点の解決策策定

・解決策の実装

簡単な実例

K-scope



- プログラム構造を俯瞰的にみて理解を助けるツール.
- Fortran90及びFortran77が対象.
- プロシージャ呼び出し,ループ,分岐に代表されるプログラムの構造の可視化を行う.
- 「京」プロファイルデータに対応した動的解析情報も表示可能.
- プログラム構造解析支援ツール.

NPB MGを使用した解析例

NPB MG (マルチグリッド法により連立一次方程式を解くベンチマークテストプログラム)

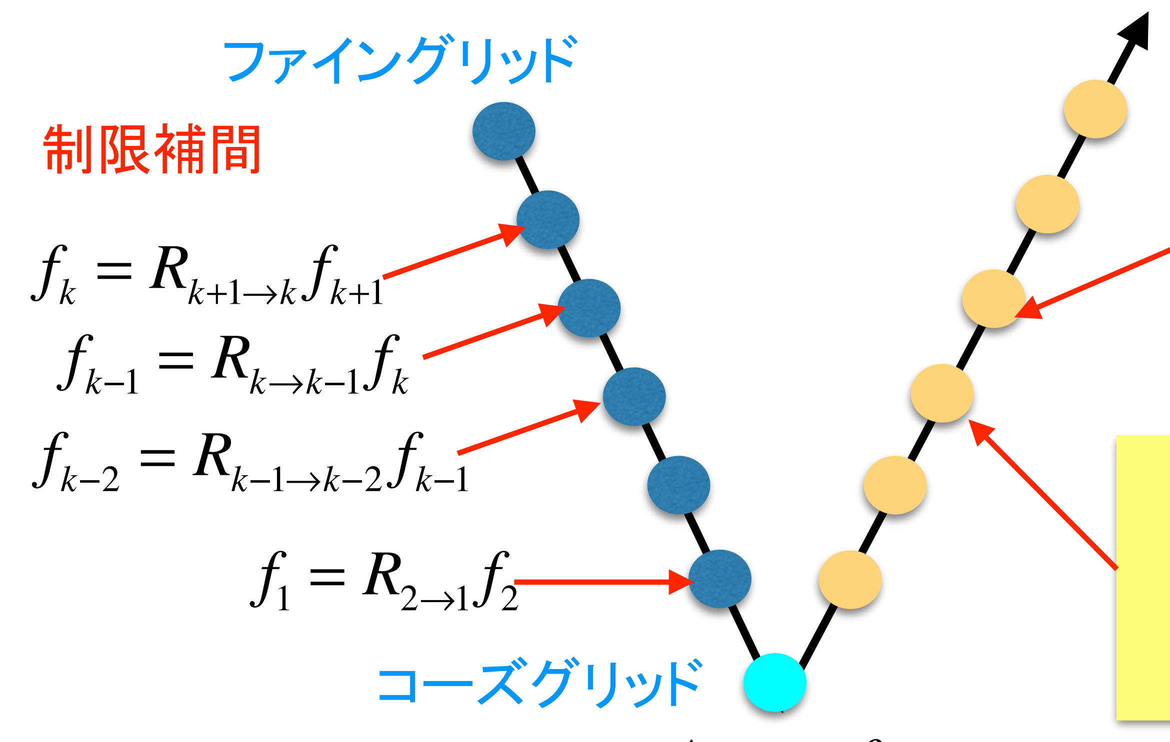
Vサイクルマルチグリッド

ファイングリッド

コーズグリッド

NPB MGを使用した解析例

NPB MG (マルチグリッド法により連立一次方程式を解くベンチマークテストプログラム)



Vサイクルマルチグリッド

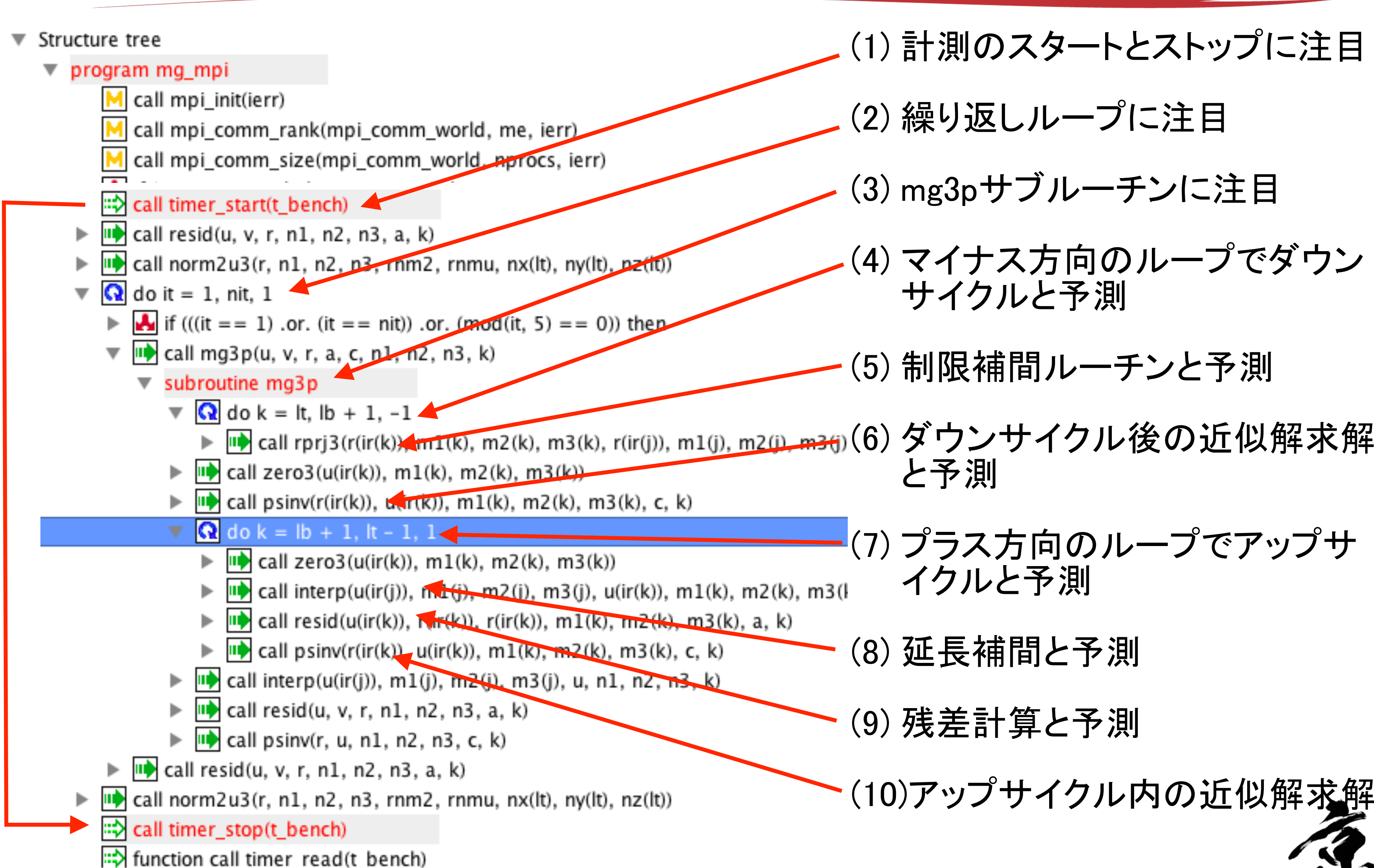
$$A_{k+1} u_{k+1} = r_{k+1}$$
$$r_{k+1} = f_{k+1} - A_{k+1} u_{k+1}$$
$$u_{k+1} = R_{k \to k+1} u_k$$

$A_k u_k = r_k$ 近似解求解

$r_k = f_k - A_k u_k$ 残差計算

$u_k = R_{k-1 \to k} u_{k-1}$ 延長補間

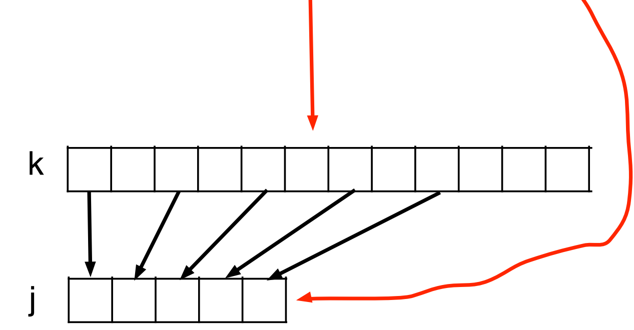
アプリケーションの構造の調査（概要）



アプリケーションの構造の調査（確認）

call rprj3(r(ir(k)),m1(k),m2(k),m3(k),r(ir(j)),m1(j),m2(j),m3(j),k)

subroutine rprj3(r,m1k,m2k,m3k,s,m1j,m2j,m3j,k)



(1) rprj3を調査の結果制限補間処理と判明
(2) ファイングリッドからコーズグリッドへの補間処理を確認した
(3) m1j,m2j,m3jはm1,m2,m3に結合

(5) その他　interp(延長補間),resid(残差計算),psinv(近似解求解)と確認

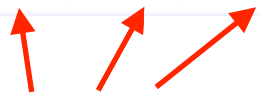
```
▼ subroutine rprj3
  ▶ if (timeron) then
  ▼ do j3 = 2, m3j - 1, 1
    ▼ do j2 = 2, m2j - 1, 1
        do j1 = 2, m1j, 1
        do j1 = 2, m1j - 1, 1
```

(6) rprj3のループ構造調査
(7) m3j,m2j,m1jの３重ループ構造

アプリケーションの構造の調査（確認）

(1) psinvのループ構造調査
(2) n3,n2,n1の３重ループ構造
(3) n1,n2,n3はm1,m2,m3に結合

call psinv(r(ir(k)), u(ir(k)), m1(k), m2(k), m3(k), c, k)



subroutine *psinv*(r,u,n1,n2,n3,c,k)

▼ subroutine psinv
▶ if (timeron) then
▼ do i3 = 2, n3 - 1, 1

do i1 = 1, n1, 1
do i1 = 2, n1 - 1, 1

(1) residのループ構造調査
(2) n3,n2,n1の３重ループ構造
(3) n1,n2,n3はm1,m2,m3に結合

call resid(u(ir(k)), r(ir(k)), r(ir(k)), m1(k), m2(k), m3(k), a, k)

subroutine *resid*(u,v,r,n1,n2,n3,a,k)

▼ subroutine resid
▶ if (timeron) then
▼ do i3 = 2, n3 - 1, 1
▼ do i2 = 2, n2 - 1, 1
do i1 = 1, n1, 1
do i1 = 2, n1 - 1, 1

アプリケーションの構造の調査（確認）

(1) interpのループ構造調査
(2) mm3,mm2,mm1の３重ループ構造
(3) mm1,mm2,mm3はm1,m2,m3に結合

```
call interp(u(ir(j)), m1(j), m2(j), m3(j), u(ir(k)), m1(k), m2(k), m3(k), k)
```

```
subroutine interp( z,mm1,mm2,mm3,u,n1,n2,n3,k )
```

```
▼ subroutine interp
  ▶ if (timeron) then
  ▼ if (((n1 /= 3) .and. (n2 /= 3)) .and.
    ▼ do i3 = 1, mm3 - 1, 1
      ▼ do i2 = 1, mm2 - 1, 1
          do i1 = 1, mm1, 1
          do i1 = 1, mm1 - 1, 1
          do i1 = 1, mm1 - 1, 1
          do i1 = 1, mm1 - 1, 1
          do i1 = 1, mm1 - 1, 1
```

(1) 主要4サブルーチンとも処理量は
m1,m2,m3の大きさに依存している.

(2) m1,m2,m3の分割はどうなっているか？

データ分割の調査

(1) m1の宣言・定義・参照について調べる

(1) setupサブルーチンで配列：mi　から代入されている

(1) setupの該当部にジャンプする

```
if( mi(1,k).eq.2 .or.
>       mi(2,k).eq.2 .or.
>       mi(3,k).eq.2      )then
   dead(k) = .true.
endif
m1(k) = mi(1,k)
m2(k) = mi(2,k)
m3(k) = mi(3,k)
```

データ分割の調査

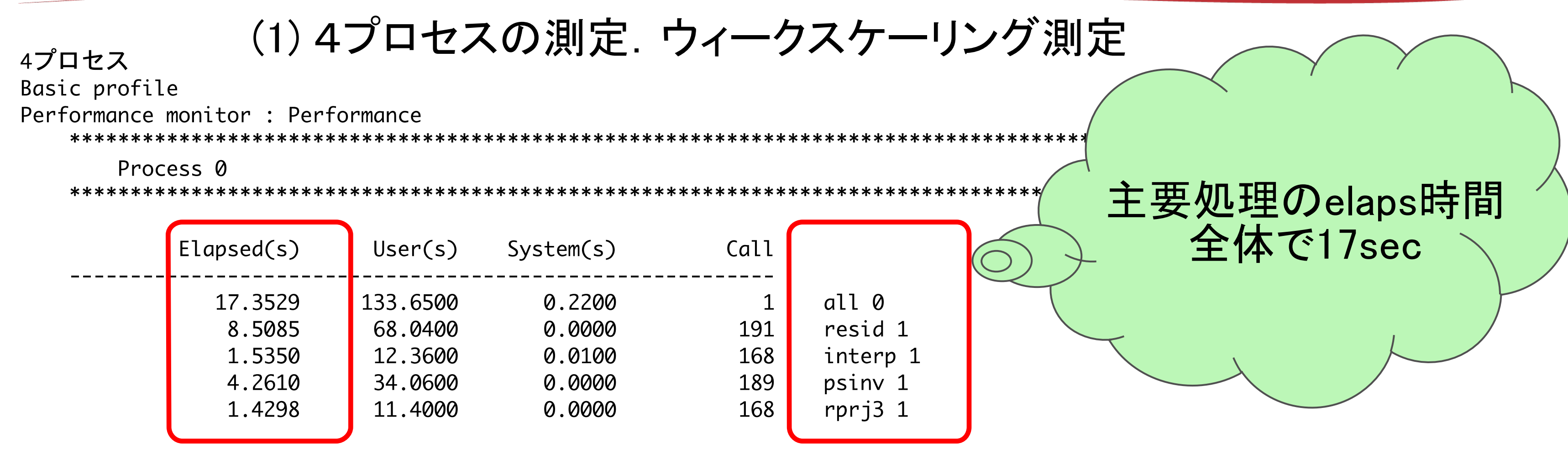
setup

```
      do  j=-1,1,1
         do  d=1,3
            msg_type(d,j) = 100*(j+2+10*d)
         enddo
      enddo

      ng(1,lt) = nx(lt)
      ng(2,lt) = ny(lt)
      ng(3,lt) = nz(lt)
      do  ax=1,3
         next(ax) = 1
         do  k=lt-1,1,-1
            ng(ax,k) = ng(ax,k+1)/2
         enddo
      enddo
 61   format(10i4)
      do  k=lt,1,-1
         nx(k) = ng(1,k)
         ny(k) = ng(2,k)
         nz(k) = ng(3,k)
      enddo

      log_p  = log(float(nprocs)+0.0001)/log(2.0)
      dx     = log_p/3
      pi(1)  = 2**dx
      idi(1) = mod(me,pi(1))

      dy     = (log_p-dx)/2
      pi(2)  = 2**dy
      idi(2) = mod((me/pi(1)),pi(2))

      pi(3)  = nprocs/(pi(1)*pi(2))
      idi(3) = me/(pi(1)*pi(2))

      do  k = lt,1,-1
         dead(k) = .false.
         do  ax = 1,3
            take_ex(ax,k) = .false.
            give_ex(ax,k) = .false.

            mi(ax,k) = 2 +
     >           ((idi(ax)+1)*ng(ax,k))/pi(ax) -
     >           ((idi(ax)+0)*ng(ax,k))/pi(ax)
            mip(ax,k) = 2 +
     >           ((next(ax)+idi(ax)+1)*ng(ax,k))/pi(ax) -
     >           ((next(ax)+idi(ax)+0)*ng(ax,k))/pi(ax)

            if(mip(ax,k).eq.2.or.mi(ax,k).eq.2)then
               next(ax) = 2*next(ax)
```

(1) setup内の配列：miを調べた結果，プロセス分割されている事が判明

スケーラビリティの調査

(1) 4プロセスの測定. ウィークスケーリング測定



```
4プロセス
Basic profile
Performance monitor : Performance
    ************************************************************************
        Process 0
    ************************************************************************

              Elapsed(s)       User(s)     System(s)          Call
    ---------------------------------------------------------------
                 17.3529      133.6500        0.2200             1    all 0
                  8.5085       68.0400        0.0000           191    resid 1
                  1.5350       12.3600        0.0100           168    interp 1
                  4.2610       34.0600        0.0000           189    psinv 1
                  1.4298       11.4000        0.0000           168    rprj3 1
```

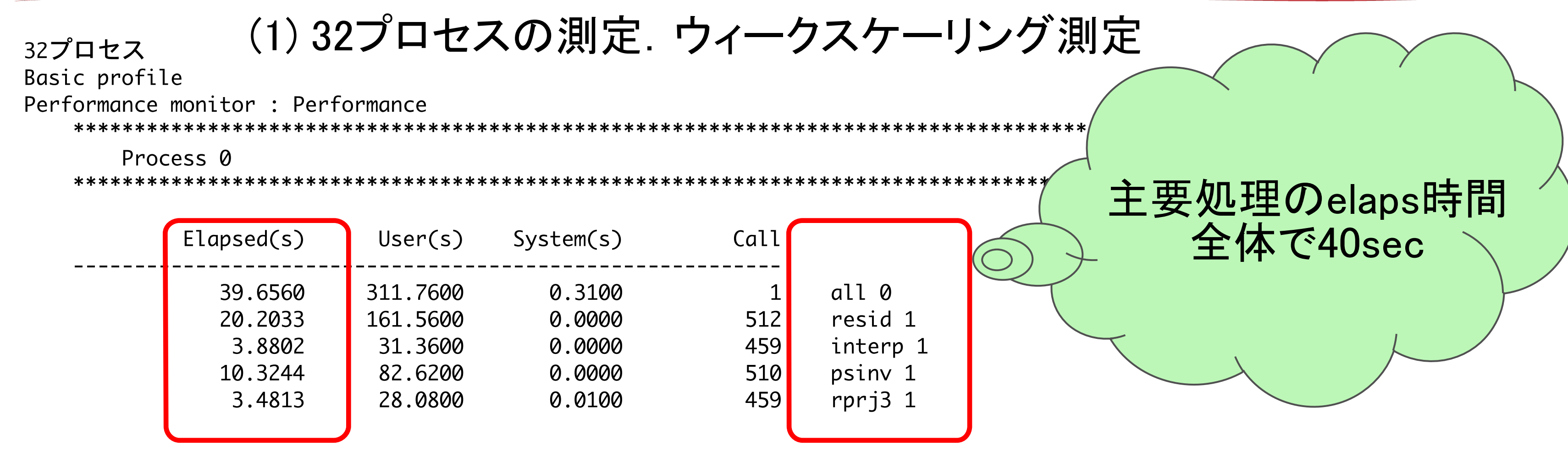


```
Performance monitor event
  ------------------------------------------------------------------------------------------

  ************************************************************************
      Application - performance monitors
  ************************************************************************

      Elapsed(s)        MFLOPS  MFLOPS/PEAK(%)          MIPS    MIPS/PEAK(%)
  ----------------------------------------------------------------------------
       16.6556      10361.0515          2.0236    18605.9895          7.2680    Application
  ----------------------------------------------------------------------------
       16.6556       2590.2629          2.0236     4651.6009          7.2681    Process      0
       16.6052       2598.1206          2.0298     4668.2023          7.2941    Process      2
       16.5407       2608.2634          2.0377     4683.4974          7.3180    Process      1
       16.5213       2611.3249          2.0401     4686.9329          7.3233    Process      3

                   Mem throughput   Mem throughput
      Elapsed(s)   _chip(GB/S)      /PEAK(%)             SIMD(%)
  ----------------------------------------------------------------
       16.6556          32.3398          12.6327         48.7947    Application
  ----------------------------------------------------------------
       16.6556           8.0885          12.6382         48.7936    Process      0
       16.6052           8.1070          12.6672         48.7676    Process      2
       16.5407           8.1357          12.7120         48.7981    Process      1
       16.5213           8.1551          12.7423         48.8196    Process      3
```

プロセスのロード
インバランスは
発生していない
ピーク性能比2%

スケーラビリティの調査

(1) 32プロセスの測定．ウィークスケーリング測定



```
32プロセス
Basic profile
Performance monitor : Performance
    ********************************************************************************
        Process 0
    ********************************************************************************

              Elapsed(s)      User(s)    System(s)          Call
    ----------------------------------------------------------
                 39.6560     311.7600       0.3100             1    all 0
                 20.2033     161.5600       0.0000           512    resid 1
                  3.8802      31.3600       0.0000           459    interp 1
                 10.3244      82.6200       0.0000           510    psinv 1
                  3.4813      28.0800       0.0100           459    rprj3 1
```



```
------------------------------------------------------------------------------------------------
Performance monitor event

 ***********************************************************************************
      Application - performance monitors
   ***********************************************************************************

      Elapsed(s)         MFLOPS  MFLOPS/PEAK(%)           MIPS    MIPS/PEAK(%)
   ----------------------------------------------------------------------
         39.3654     83044.5238          2.0275    149003.2766          7.2756    Application
   ----------------------------------------------------------------------
         39.3654      2595.1412          2.0275      4654.6126          7.2728    Process    0
         39.3575      2595.6627          2.0279      4650.9112          7.2670    Process    7
         39.3546      2595.8520          2.0280      4656.6897          7.2761    Process    8
                                             :
```

プロセスのロード
インバランスは
発生していない
ピーク性能比2%

```
                     Mem throughput   Mem throughput
      Elapsed(s)     _chip(GB/S)      /PEAK(%)             SIMD(%)
   --------------------------------------------------------
         39.3654        258.7562         12.6346           49.3192    Application
   --------------------------------------------------------
         39.3654          8.0773         12.6208           49.3303    Process     0
         39.3575          8.0573         12.5895           49.3647    Process     7
         39.3546          8.0544         12.5850           49.3071    Process     8
                                             :
```

スケーラビリティの調査

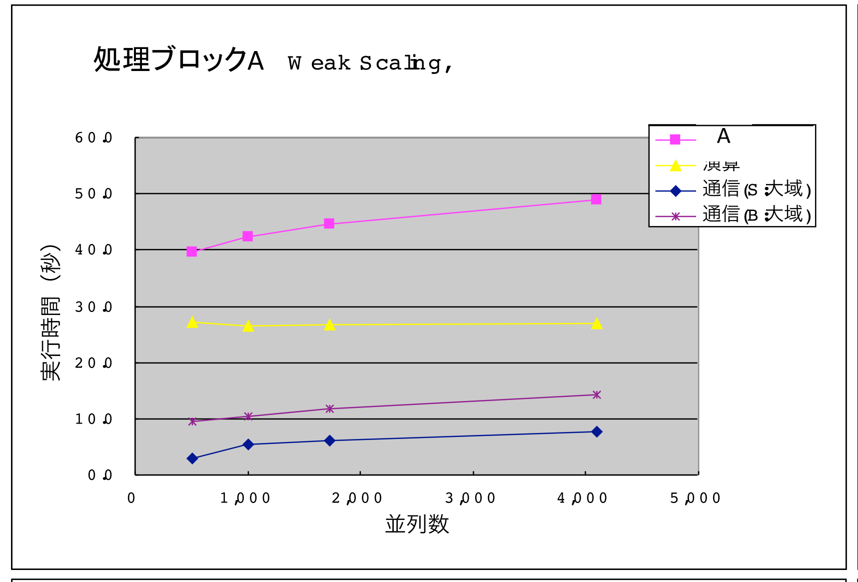
- ウィークスケーリングで4プロセスから32プロセスで全体で17secから40secへ実行時間が増大．約2.5倍の増大.
- 主要4サブルーチンも同様の傾向.
- しかしピーク性能比は、2%で変わらず.
- 1プロセスの演算量が同じなら実行時間が延びる事でピーク性能比も落ちるはず？
- 調査の結果ウィークスケール測定であったが32プロセスでは全体のイタレーション回数が2.5倍に設定されていた事が判明.
- スケーラビリティとしては良好である事が分かった.
- またロードインバランスも良好であることが分かった.
- 並列化の観点では問題なし.

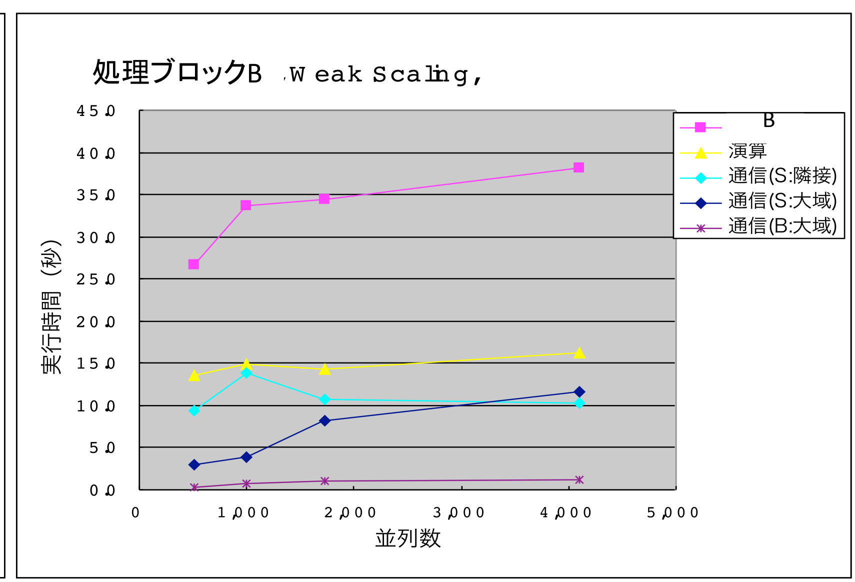
並列特性分析(結果)

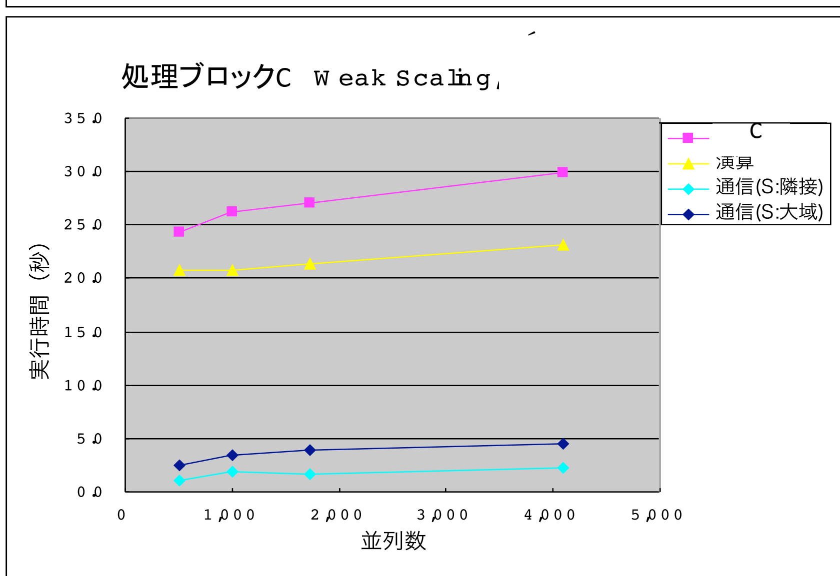
```
▼ subroutine mg3p
  ▼ do k = lt, lb + 1, -1
      ▶ call rprj3(r(ir(k)), m1(
  ▶ call zero3(u(ir(k)), m1(k),
  ▶ call psinv(r(ir(k)), u(ir(k)),
  ▼ do k = lb + 1, lt - 1, 1
      ▶ call zero3(u(ir(k)), m1
      ▶ call interp(u(ir(j)), m1
      ▶ call resid(u(ir(k)), r(ir(
      ▶ call psinv(r(ir(k)), u(ir(
```

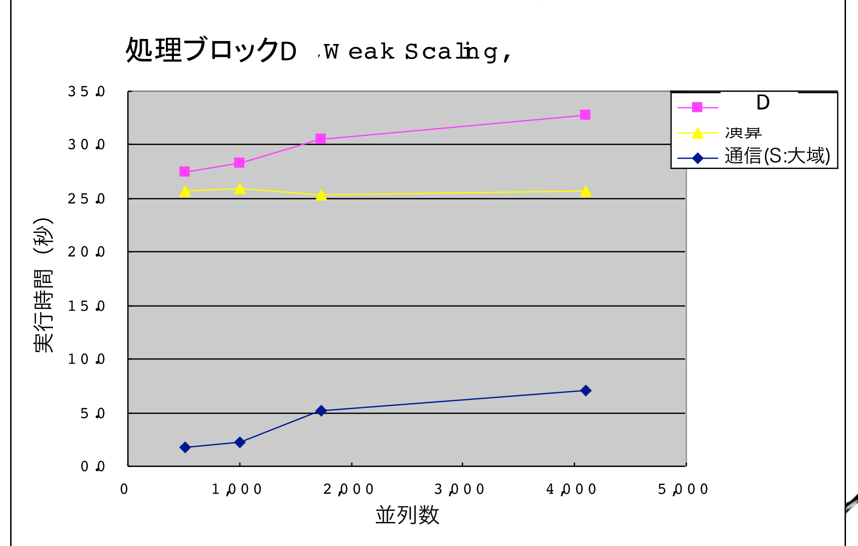
実行時間・スケーラビリティ	物理的処理内容	演算・通信特性	演算・通信見積り	カーネル
良好	制限補間	完全並列	Nに比例	○
良好	延長補間	完全並列	Nに比例	○
良好	残差計算	完全並列	Nに比例	○
良好	簡易求解	完全並列	Nに比例	○

Weak Scaling測定例

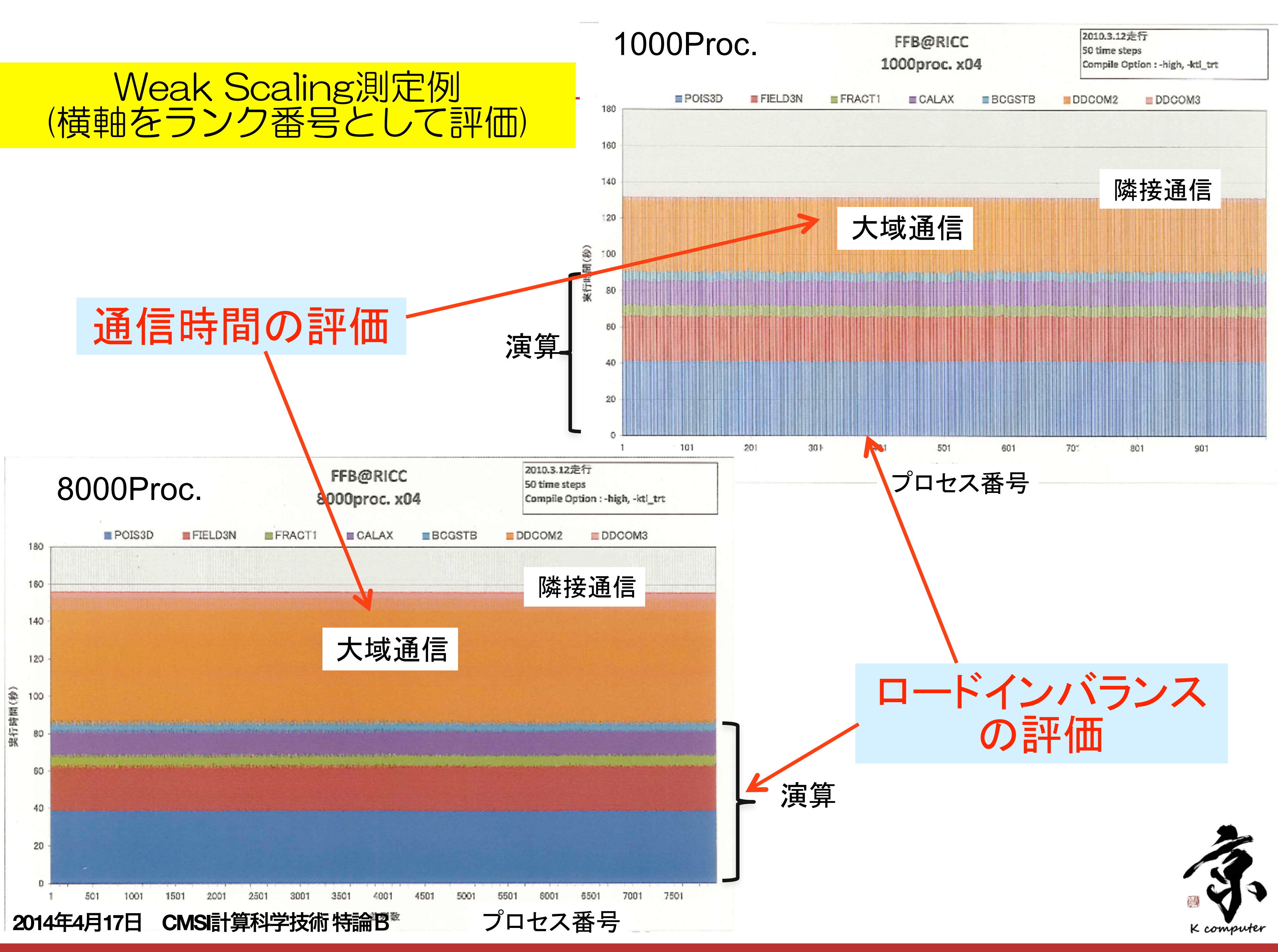








K computer


Weak Scaling測定例
(横軸をランク番号として評価)
1000Proc.
FFB@RICC
1000proc. x04
2010.3.12走行
50 time steps
Compile Option : -high, -ktl_trt
POIS3D
FIELD3N
FRACT1
CALAX
BCGSTB
DDCOM2
DDCOM3
隣接通信
大域通信
演算
通信時間の評価
プロセス番号
8000Proc.
FFB@RICC
8000proc. x04
2010.3.12走行
50 time steps
Compile Option : -high, -ktl_trt
隣接通信
大域通信
演算
ロードインバランス
の評価
プロセス番号
K computer

並列性能上の
ボトルネック

並列性能上のボトルネック

- 今まで示した調査を実施することにより処理ブロック毎に並列性能上の問題題がある事が発見される.
- それらを分析するとだいたい以下の６点に分類されると考える.

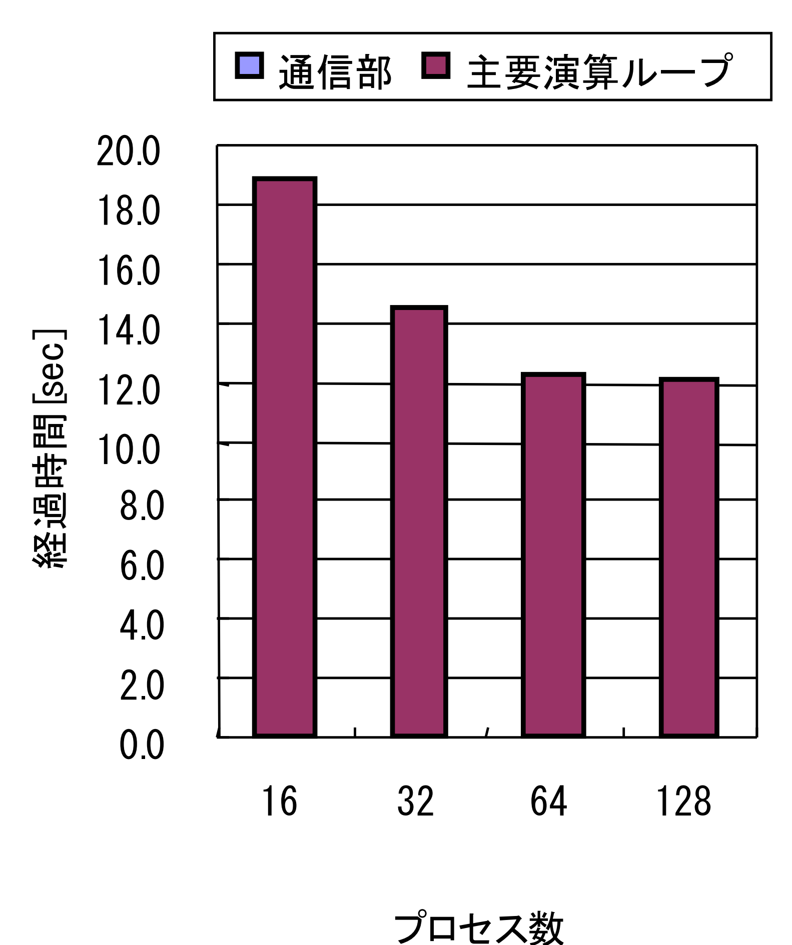
1	アプリケーションとハードウェアの並列度のミスマッチ （アプリケーションの並列度不足）
2	非並列部の残存
3	大域通信における大きな通信サイズ、通信回数の発生
4	フルノードにおける大域通信の発生
5	隣接通信における大きな通信サイズ、通信回数の発生
6	ロードインバランスの発生

アプリケーションとハードウェアのミスマッチ

- 波数展開を使う第一原理計算の場合を考える.
- 電子のエネルギーバンド並列のみ実装されていると仮定
- 原子一個に対し最外核の電子2個と考えるとエネルギーバンドの数は，おおよそ原子数：N×２となる.
- N=10000ならエネルギーバンド数は20000程度である.
- この場合，最大でも20000並列までしか到達しない.
- 通信の増大や計算の粒度を考えると数百並列で限界となる.
- 波数等その他の分割を組み合わせた並列化が必要となる.
- 詳細は4回目の講義で実例を示す.

非並列部の残存

- 第一原理アプリケーションの処理ブロック:ノンローカル項の計算部分を例に示す.
- 低並列でストロングスケールで測定.
- すでにこの並列度でスケールしていない.原因は次ページに示す.



[s]
14
12
10
8
6
4
2
0
16

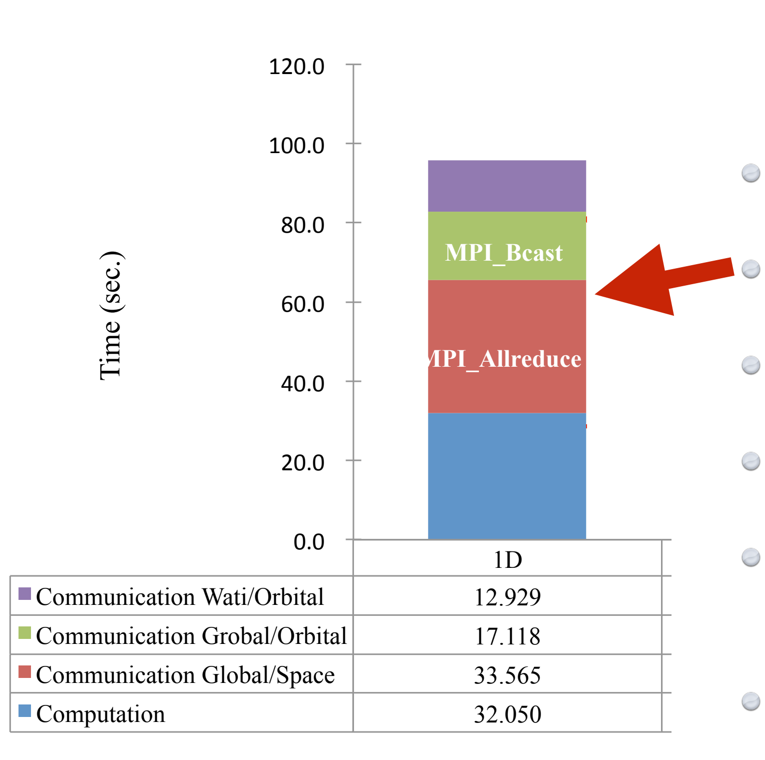
K computer

非並列部の残存

- 第一原理計算のアプリでエネルギーバンド並列を使用している場合の非並列部が残っている例.

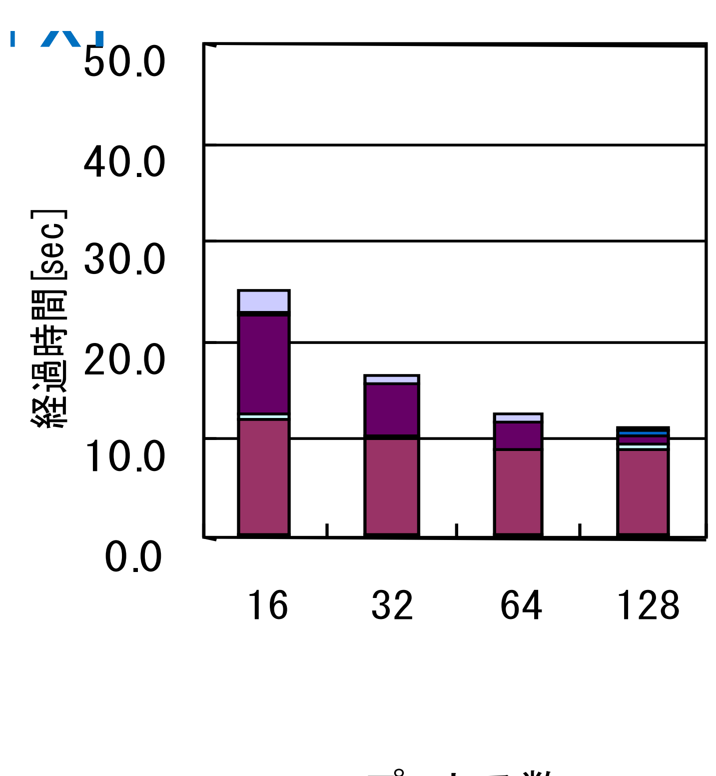
```
subroutine m_es_vnonlocal_w(ik,iksnl,ispin,switch_of_eko_part)
 |   +-call tstatc0_begin
 |   |   loop_ntyp: do it = 1, ntyp
 |   |   |   loop_natm : do ia = 1, natm ---------原子数のループ
 |   |   |    +-call calc_phase
 |   |   |    |   T-do lmt2 = 1, ilmt(it)
 |   |   |    |   +-call vnonlocal_w_part_sum_over_lmt1
 |   |   |    |   |   +-call add_vnlph_l_without_eko_part
 |   |   |    |   |   |   subroutine add_vnlph_l_without_eko_part()
 |   |   |    |   |   |   |   T-if(kimg == 1) then
 |   |   |    |   |   |   |   |   T-do ib = 1, np_e   ---------エネルギーバンド並列部
 |   |   |    |   |   |   |   |   |   T-do i = 1, iba(ik)
 |   |   |    |   |   |   |   |   |   V-end do
 |   |   |    |   |   |   |   |   V-end do
 |   |   |    |   |   |   |   +-else
 |   |   |    |   |   |   |   |   T-do ib = 1, np_e   ---------エネルギーバンド並列部
 |   |   |    |   |   |   |   |   |   T-do i = 1, iba(ik)
 |   |   |    |   |   |   |   |   |   V-end do
 |   |   |    |   |   |   |   |   V-enddo
 |   |   |    |   |   |   |   V-end if
 |   |   |    |   |   |   end subroutine add_vnlph_l_without_eko_part
 |   |   |    |   V-end do
 |   |   |    V-end do loop_natm
 |   |   V-end do loop_ntyp
end subroutine m_es_vnonlocal_w
```

大域通信における大きな通信サイズ・通信回数

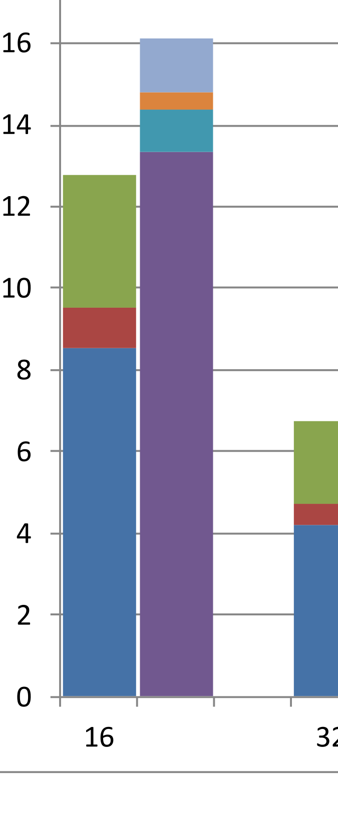


- 通信はレイテンシーが問題になる場合とバンド幅が問題になる場合がある.
- 各ノードでベクトルデータを総和しスカラー値を求め，スカラー値をノード間でallreduceすれば良いのに，全ノードで巨大なベクトルをallreduceしていませんか？
- この場合は，よけいなバンド幅を消費していることになる.
- 最適化された大域通信のライブラリを使っていますか？
- 最適なライブラリと最適化されていないライブラリでは数倍性能差が発生する場合がある.
- 小さなデータを頻繁に大域通信していませんか？
- まとめることが出来る通信はまとめて回数を減らす事でレイテンシーネックを解消することができます.
- 大域通信ライブラリで可能な通信と同様な事を隣接通信で自作しているような場合がある.
- 性能の良い大域通信ライブラリの使用を推奨します.

フルノードにおける大域通信の発生



- FFTをフルノードで実施する事は性能を著しく劣化させる.
- 第一原理アプリケーションの理ブロックの例を左図に示す
- 低並列でストロングスケール
- すでにこの並列度でスケール
- 並列軸の追加によりFFTをフく一部のノードに限ることがある.
- 詳細は4回目の講義で説明
- CG法の内積計算のようにフ域通信を実施する必要があすことができない.
- その場合は，高速なハードウ機能を使用する等の対処を

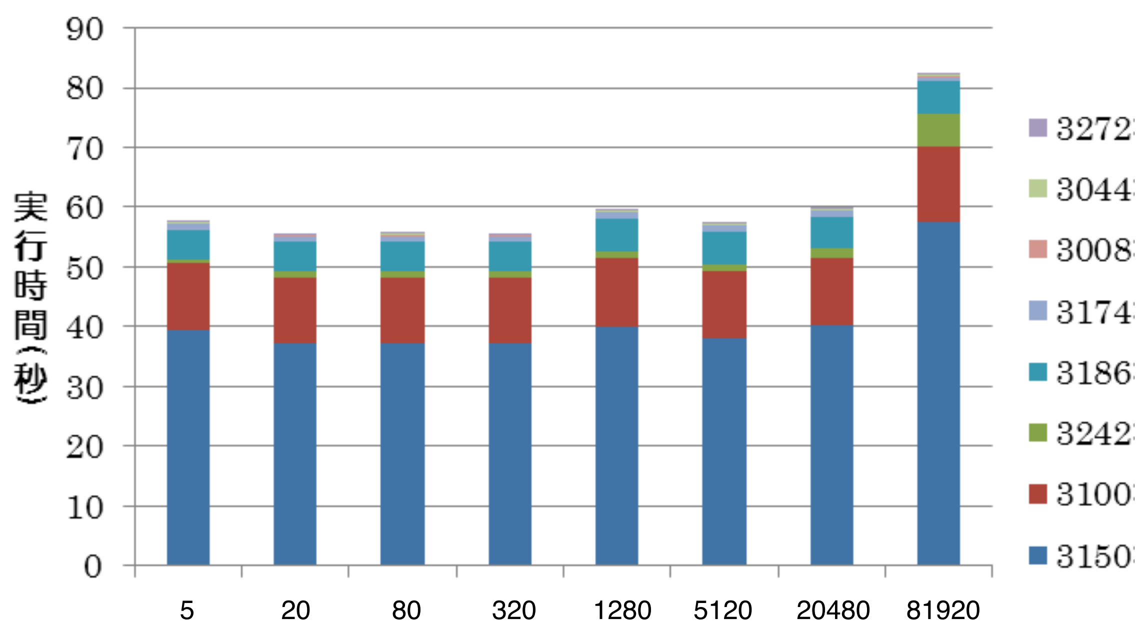


隣接通信における大きな通信サイズ・通信回数

- 隣接通信がネックになる場合，アルゴリズムを見直す事で隣接通信をやめてしまうことが出来る場合がある．
- 例えばCG法前処理のローカライズ化等の場合．
- 中途半端な並列化が通信性能の悪化を招く場合がある．
- これらの詳細は4回目の講義で説明する．

ロードインバランスの発生

- ロードインバランスの発生は色々な原因がある.
- 第一は粒子の計算で時間発展により粒子のバランスが変わるような場合.
- 動的負荷分散を取る方法等色々な手法が研究されている.
- その他システムの外乱が原因となる場合もある.
- 詳細は3回目以降に講義する.
- 以下の例は処理ブロック毎にスケーラビリティを測定した例.



まとめ

- 反復法
  - 次回以降に使用するため反復法について説明しました.
- アプリケーションの性能最適化
  - 性能最適化の流れと高並列化のための分析手法を説明しました.
- 簡単な実例
  - NPB MGを例に高並列化のための分析手法の実例を示しました.
- 並列性能のボトルネック
  - 並列性能のボトルネック要因について説明し例を示しました.