Ruby における科学データ処理基盤: NArrayとPwrake

田中昌宏
筑波大学 計算科学研究センター

内容

- 多次元数値配列 NArray
  - リリース版の概要
  - 開発版の新機能
- 並列分散ワークフローシステム Pwrake

- 去年の7月に東大平木研で話した内容

Ruby/NArray

- Numerical N-dimensional Array
  - オブジェクト: 多次元数値配列
  - メソッド: 配列操作・演算
- ウェブサイト
  - http://narray.rubyforge.org
  - http://github.com/masa16/narray

NArrayバージョン

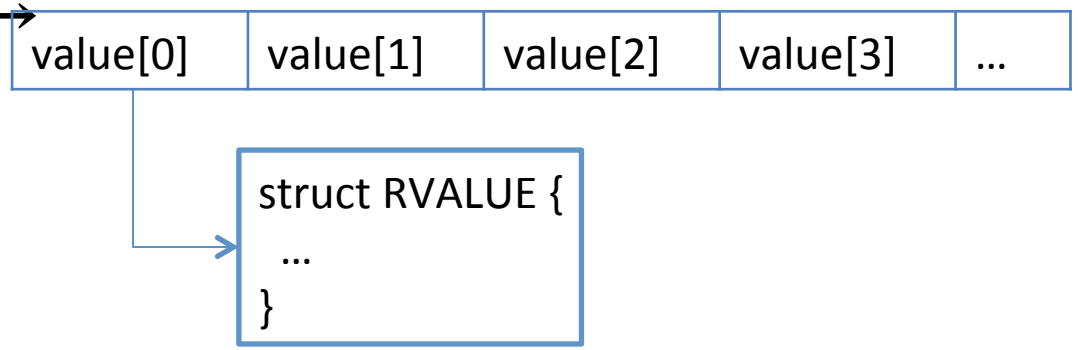
- 0.3.0 – 1999
- 0.5.0 – 2000
- 0.6.0 – 2011（リリース版）
  - 単に 0.5.9.9 の次
- 0.7.1 – 2007（開発版1）
- 0.9? – 2011（開発版2）

NArray オブジェクト (基本部分)

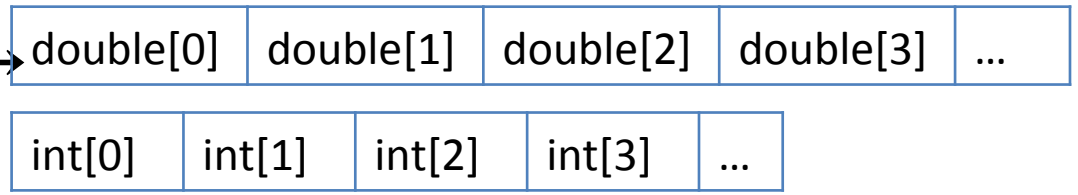
- rank
  - 次元数
- shape
  - 配列の「形」
- type
  - 要素の型
- memory block
  - データを格納するメモリーブロック

メモリーブロック

- Ruby Array :
  - VALUE *ptr;



- NArray
  - void *ptr;
  - or



NArray（リリース版）のデータ型（ビルトイン）

- 整数
  - BYTE (8bit)
  - SINT (16bit)
  - INT (32bit)
- 浮動小数点数
  - SFLOAT (32bit)
  - DFLOAT (64bit)
- 複素数
  - SCOMPLEX (64bit)
  - DCOMPLEX (128bit)
- Rubyオブジェクト
  - ROBJECT

配列の形：shape

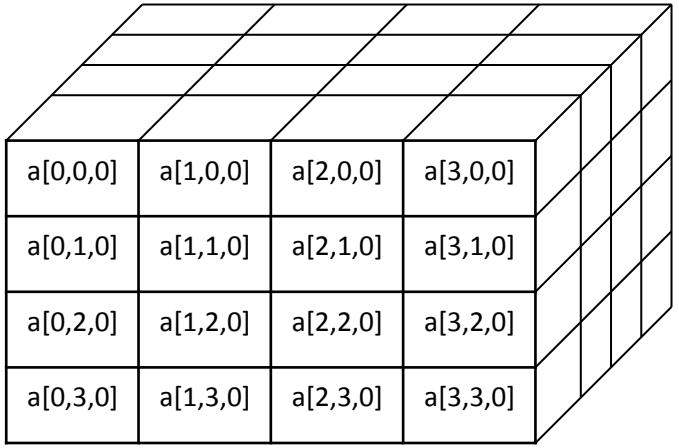
- shape = [4]
  - 1次元配列

a[0]	a[1]	a[2]	a[3]

- shape = [4,4]
  - 2次元配列

a[0,0]	a[1,0]	a[2,0]	a[3,0]
a[0,1]	a[1,1]	a[2,1]	a[3,1]
a[0,2]	a[1,2]	a[2,2]	a[3,2]
a[0,3]	a[1,3]	a[2,3]	a[3,3]

- shape = [4,4,4]
  - 3次元配列



多次元データの順序

- a[i,j]

- 左の次元(i)が早く回る
  - Fortran-order 、または
  - Row-major order

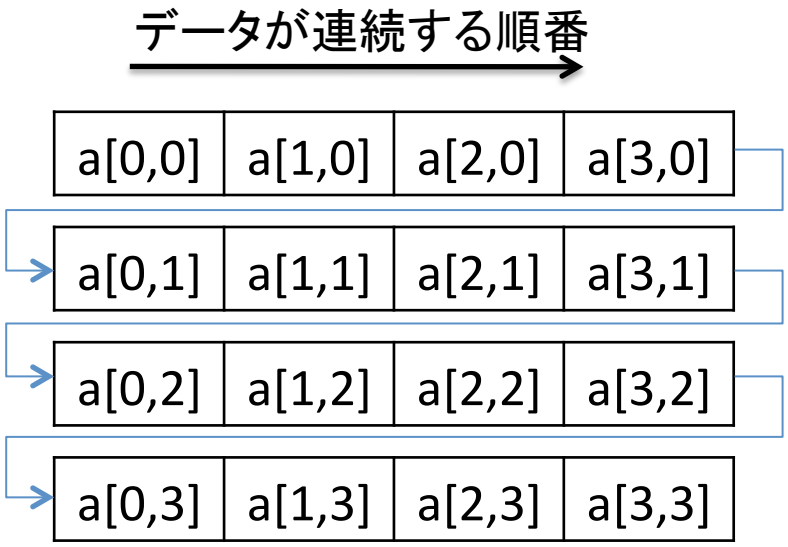
- 多次元配列を1次元でアクセス

shape=[m,n]のとき、

a[i,j] == a[i+j*m]

- Rubyの配列:

  a[j][i]



NArrayの生成

NArray.float(3,2)

=> NArray.float(3,2):

```
[ [ 0.0, 0.0, 0.0 ],
  [ 0.0, 0.0, 0.0 ] ]
```

NArray.float(3,2).indgen!

=> NArray.float(3,2):

```
[ [ 0.0, 1.0, 2.0 ],
  [ 3.0, 4.0, 5.0 ] ]
```

NArray[[1,2,3],[4,5,6]]

=> NArray.int(3,2):

```
[ [ 1, 2, 3 ],
  [ 4, 5, 6 ] ]
```

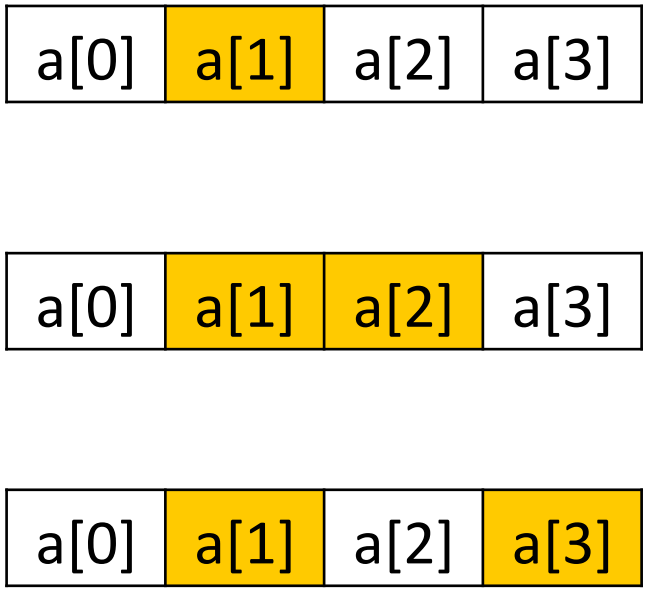
NArray[1..10]

=> NArray.int(10):

```
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

配列のスライス

- a[1]
  - 要素参照
- a[1..2]
  - 範囲参照
- a[[1,3]]
  - インデックス配列参照



NArrayの演算

- 演算は要素ごと (element-wise)

- 演算

a * b =

a[0]*b[0]	a[1]*b[1]	a[2]*b[2]	a[3]*b[3]

  - 四則演算
    - +, -, *, /, %, **
  - 統計・ソート
    - min, max, sum, mean, stddev, rms, rmsdev, median, sort
  - 数学関数演算
    - NMath module

サイズ1の繰り返しルール

- ２つの配列が絡む場合に適用される
  - 演算やスライス代入
- ある次元のサイズが、n対１の場合、
- サイズ１の次元で、データがn回繰り返す
- 例：　a = [1,2,3,4]
  - a[0..2] = 1　　→　　a[0..2] = [1,1,1]
  - a * 2　　→　　a * [2,2,2,2]

繰り返しルールの応用:クロス演算

- `x = [[1,2,3]]  (shape=[n,1])`
- `y = [[4],[5]]  (shape=[1,m])`
- `x * y = [[4,8,12],[5,10,15]]`

	x[0,0]	x[1,0]	x[2,0]
y[0,0]	x[0,0]*y[0,0]	x[1,0]*y[0,0]	x[2,0]*y[0,0]
y[0,1]	x[0,0]*y[0,1]	x[1,0]*y[0,1]	x[2,0]*y[0,1]

- 例: 原点からの距離のグリッドデータを作る場合
- `NMath.sqrt( x**2 + y**2 )`

↑ ここの'＋'で2次元に展開

条件演算

- 0 or 1 を含む バイト型NArray を返す
  a.eq(b) # == は使用できない
  ne, gt(>), lt(<), ge(>=), le(<=)
- whereメソッド: 非ゼロのインデックスを返す

  (a.gt b).**where**

  – 多次元の場合は、1次元のインデックス
- ゼロ未満の部分に0を代入
  – a[(a.lt 0).where] = 0
  – a[a.lt 0] = 0

NArray開発状況

- リリース版:
  - 0.6.0.1　　最新版
- 開発版
  - 0.7.0.1　　非互換の変更（2007年）
  - 0.9？　　0.7 からさらに仕様変更

開発版の変更点

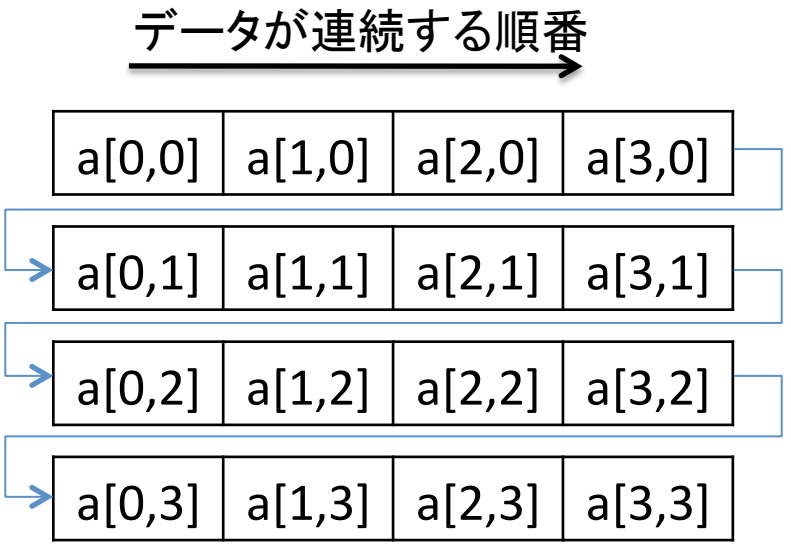
- 次元順序の変更
- 64bit対応
- データ型が追加可能
- 配列スライスのとき、コピーを作らない
- NMatrix, NVector は廃止
  - 行列積の記法以外にメリットがない
  - クラス変換とかcoerceで複雑になる
  - dot メソッドで代用

開発版の新機能

- inplace 演算
- ビット配列
- 構造体データ型
- mmap

次元の順序

- リリース版：
  - a[i,j]
  - 左の次元（i）がはやく回る
  - 画像や科学データはこちら
- Rubyのネスト配列と逆：
  - a[j][i]



- 開発版：**Rubyと同じ順序に変更**
  - a[j,i]
  - 説明が省けて混乱しにくい
  - フラグによって、オブジェクトごとに次元順序が変えられる

64bit対応

- リリース版
  - データ位置のデータ型:　int32_t
  - 配列サイズ: 最大2GB
- 開発版
  - NArrayに　Int64 型を追加
  - データ位置のデータ型 :　sizt_t
  - where が返すデータ型 :
    - 配列サイズによって、Int32 または Int64

データ型

- リリース版
  - NArrayクラスの内部属性
  - NArray#typecode で型コードを返す
  - NArray::DFLOAT　: 型コード定数
  - データ型は固定で、追加不可能
- 開発版
  - NArrayクラスのサブクラス
  - クラス名 :　NArray::DFloat　など
  - データ型の追加が可能

データ型一覧

- 既存の型
  - NArray::Int8
  - NArray::Int16
  - NArray::Int32
  - NArray::DFloat
  - NArray::SFloat
  - NArray::DComplex
  - NArray::SComplex
  - NArray::RObject
- 新規の型
  - NArray::Int64
  - NArray::UInt8
  - NArray::UInt16
  - NArray::UInt32
  - NArray::UInt64
  - NArray::Bit
  - NArray::NStruct

型変換

- 演算によって、どの型に変換するかが決まる
  - float * int　　=>　float * float
  - float ** int　=>　float ** int
- coerce は、演算ごとに設定できない
- NArrayでは、UPCASTを導入

UPCAST

- 例

  a = NArray::Int32[1,2,3]

  b = NArray::DFloat[4,5,6]

  c = a + b

- 演算の擬似コード

  tp = NArray::UPCAST[a.class][b.class]

    - tp は DFloat

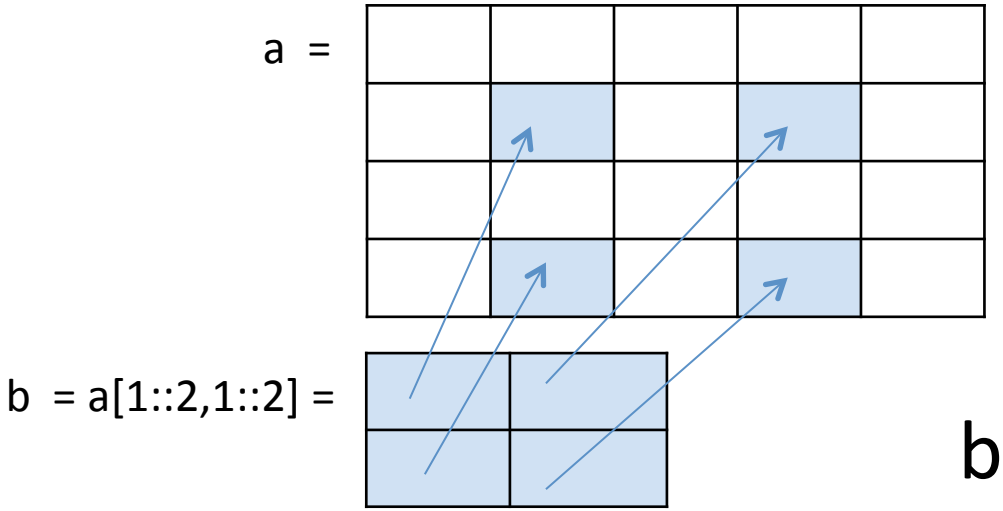
  c = tp.add(a,b)

    - a が DFloat にキャストされる

- ハッシュ UPCAST は、追加・変更可能

NumPy配列スライス

- 「元の配列の一部分を指す配列」が作られる
- データがコピーされない
- スライス配列の値を変えると、元の配列も変わる

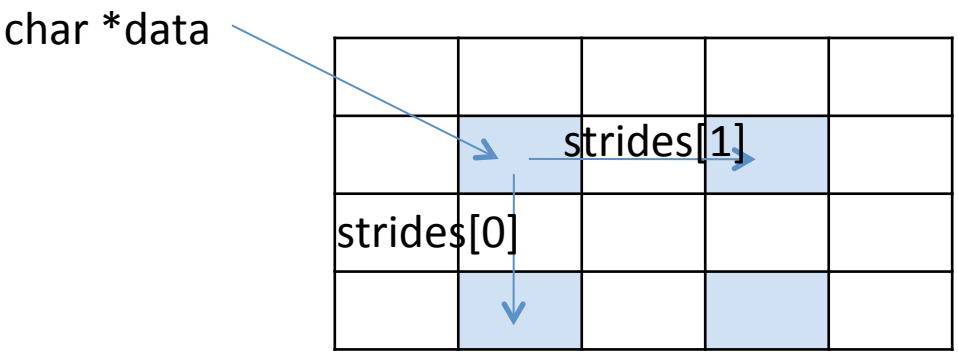


b [0,0] = 99

a[1,1] #=> 99

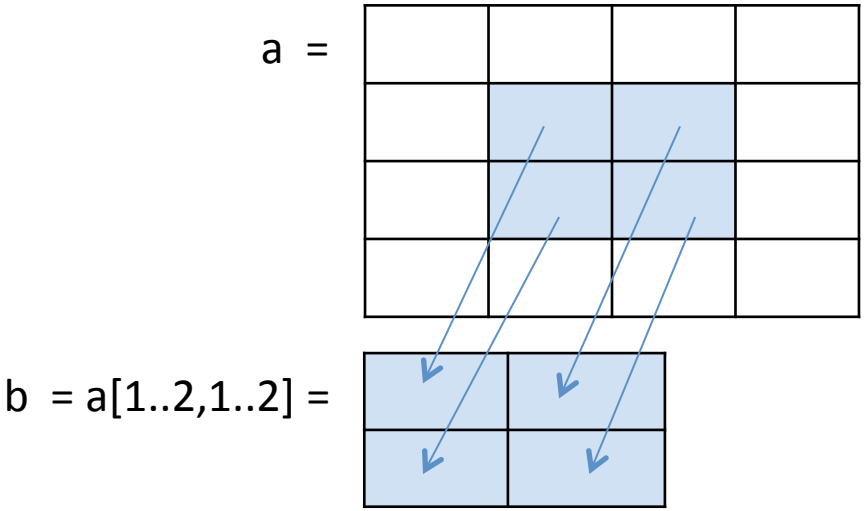
NumPy オブジェクト

```
typedef struct PyArrayObject {
    char *data;
    int nd;
    npy_intp *dimensions;   /* shapeと同じ */
    npy_intp *strides;
    …
```



リリース版NArray配列スライス

- 値をコピーした新しい配列が作られる
- スライス配列を書き換えても、元の配列の内容は変更されない



b [0,0] = 99

a[1,1] #=> 0

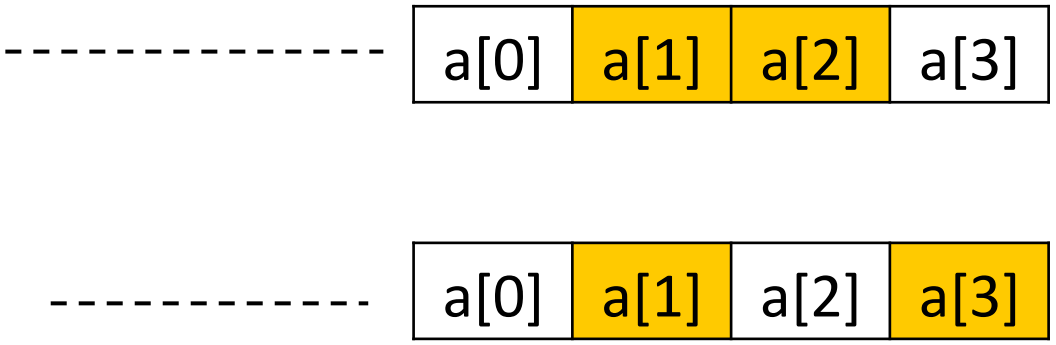
配列スライス

- NumPy式
  - スライス配列を書き換える
  - スライスだけなら高速
  - オブジェクトの構造が複雑
- NArray式
  - オブジェクトの構造が簡単

開発版NArrayの配列スライス

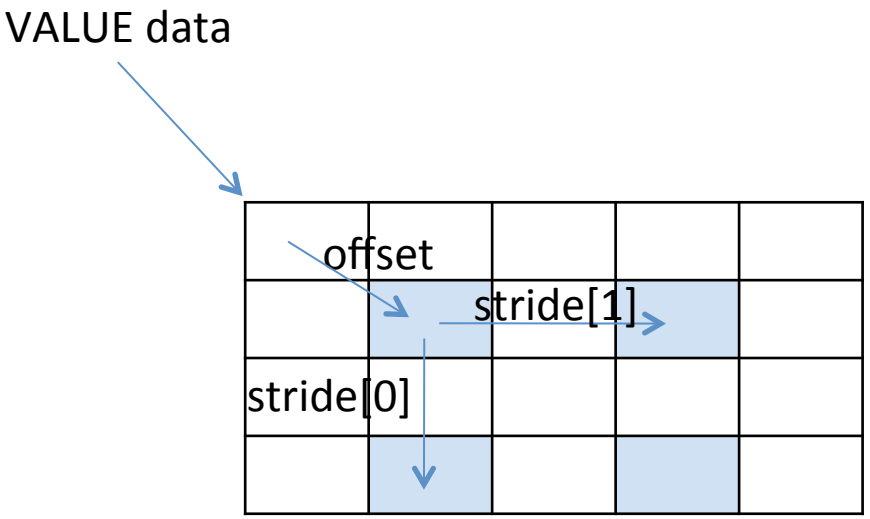
- NumPyの参照方式に変更
- NArrayでは、インデックスも指定可能

- a[1..2]
  - 範囲参照
- a[[1,3]]
  - インデックス配列参照



開発版NArray構造体（抜粋）

```
{
  unsigned char ndim;
  size_t size;
  size_t *shape;
  VALUE data;
  size_t offset;
  stridx_t *stridx;
  …
```



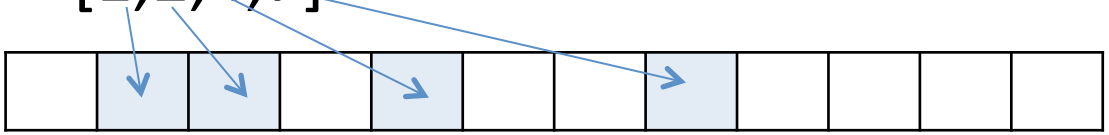
stridx

- 最下位ビットが1:
  - 1ビット下位にシフトした数を stride とみなす

  stride = stridx>>1 = 4

- 最下位ビットが0:
  - ポインタとみなし、index配列を指す

  *stridx = index = [1,2,4,7]



inplace

- 元の配列に結果を上書き
  - メモリーを確保しなくて済むので、高速
- リリース版
  - a.mul!(b).add!(c)
  - 演算子が使えない
  - メソッドを作る必要がある

inplace

- 開発版の機能
  - NArray オブジェクトに、inplace フラグ を導入
  - inplace属性のオブジェクトは、演算結果を自身に上書き
  - Numpy にはない機能

inplace 使い方

- x = a.inplace
  - 配列 a を指すviewオブジェクトを生成し、inplace 属性を付けて返す。(aのフラグは変えない)
- a.inplace + b
  - a+b の結果を、配列aに上書き
    - 元の a オブジェクトのフラグは変わっていないので、フラグを元に戻す必要はない
- a.inplace! または a.inplace = true
  - a のフラグが変わる

inplace (contd.)

- フラグの伝播

  a.inplace * b + c　　# すべて a に上書き

- 左側でも上書き

  a + b * c.inplace　　# すべて c に上書き

- 両側とも inplace のときは、左側に上書き

  a.inplace * b + c.inplace * d

（クロス演算のときはinplace不可）

inplace (contd.)

- NMath でも上書き

  NMath.sin(a.inplace)

- スライス配列でも上書き

  a[1..2,1..2].inplace + 1

  => a[1..2,1..2] += 1 と同じだが、テンポラリ配列は作らない

Matz さんにも受ける

ビット配列

- 0 or 1 の配列
- 1要素が 1ビット
  - バイト配列と比べて、サイズが1/8になる
- 条件演算の結果をビット配列で返す
  - and / or が高速
- 多次元スライスのルールも共通

構造体データ型

```
Foo = NArray::NStruct.new() do
  int8 :byte
  float64 :float, [2,2]
  dcomplex :compl
end
z = Foo.new([2,3])
z.fill([2,[[1,2],[3,4]],0.123])
z.field(:float)
z[1,1..2].field(:float)
```

mmap

- ファイルをメモリーにマップして、ポインターでファイルにアクセスできる仕組み
  - メモリーに全データをロードできない場合に需要あり
- 環境の違いを吸収
  - UNIX系 → mmap
  - Windows → MapViewOfFile

- NArrayでメモリー管理

今後の課題

- ネイティブ計算コード
- 疎行列
- 分散配列

ネイティブ計算コード

- NArray で a + b + c と書くと
  - for (i=0; i<n; i++){x[i]=a[i]+b[i];}
    for (i=0; i<n; i++){y[i]=x[i]+c[i];}
  - ループが2回
- ベターなのは
  - for (i=0; i<n; i++) {y[i]=a[i]+b[i]+c[i];}
  - ループが1回
  - キャッシュを有効利用できる
  - Rubyレベルでは現状不可能

ネイティブ計算コード

- ループ内部をネイティブコードに
  - ステンシル計算
  - GPU 対応
  - 自動チューニング
- 手法？
  - 拡張ライブラリ
  - Inline-C
  - Rubyコンパイラ

疎行列

- 配列のスライスの逆バージョン
- 需要は高い
- 格納方法が多数： CRS, CCS, ….

- 普通のNArrayとの相互変換を可能に
- ラップできるライブラリは？

分散配列

- 並列分散計算の時代
- 配列データを各ノードに分散

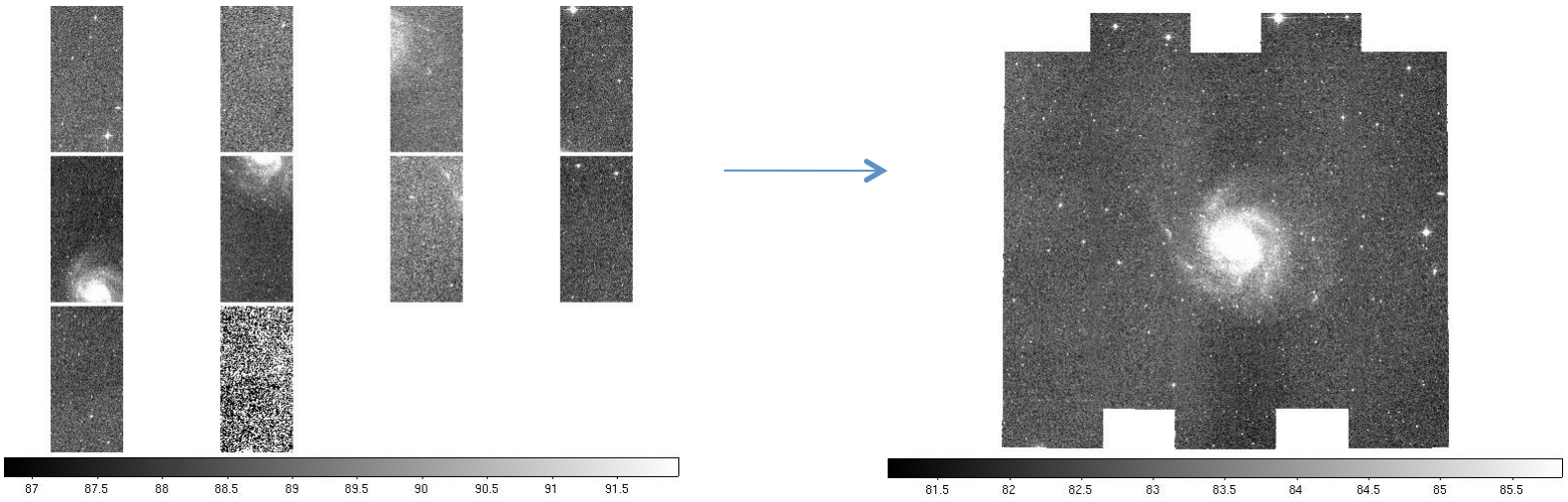
- Rubyレベルでは、容易に扱いたい
- 内部では、賢く扱ってほしい

並列分散ワークフローシステム Pwrake（プレイク）

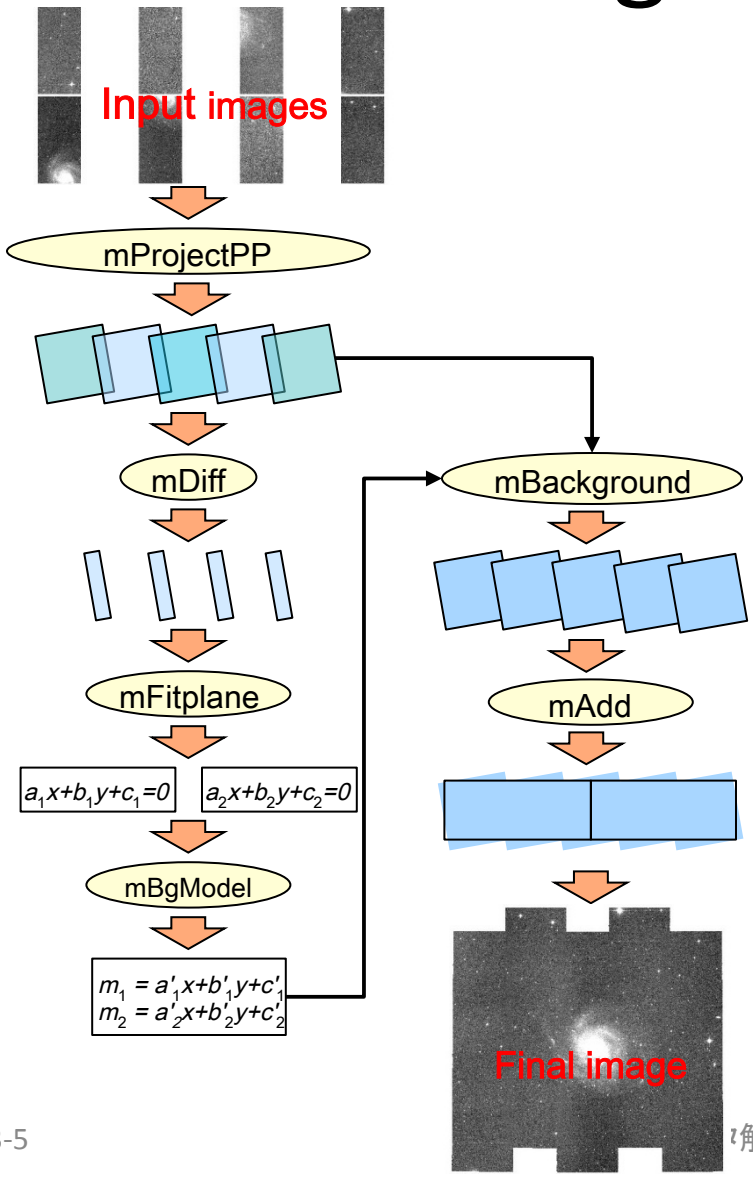
http://github.com/masa16/pwrake

天文データ処理の例

- Montage
  - 複数ショットの天文画像を、1つの画像に結合
  - http://montage.ipac.caltech.edu/



Montage Workflow



- 処理内容:
  - 座標変換
  - 明るさ補正
  - 足し合わせ
- 1 image : 1 process

クラスタを用いて並列処理すれば高速化が可能

ワークフロー実行ツールの例

- GXP
  - 東大田浦さんらが開発したツール
  - Python で記述
  - クラスターの各ノードで、一斉にコマンド実行
- GXP make : GNU make の機能
  - 始めは GXP make を使用
  - ローカリティした機能がない
  - 拡張しようにも、Pythonを考慮なのでよくわからない
- Rakeを拡張しよう

Rakeの拡張

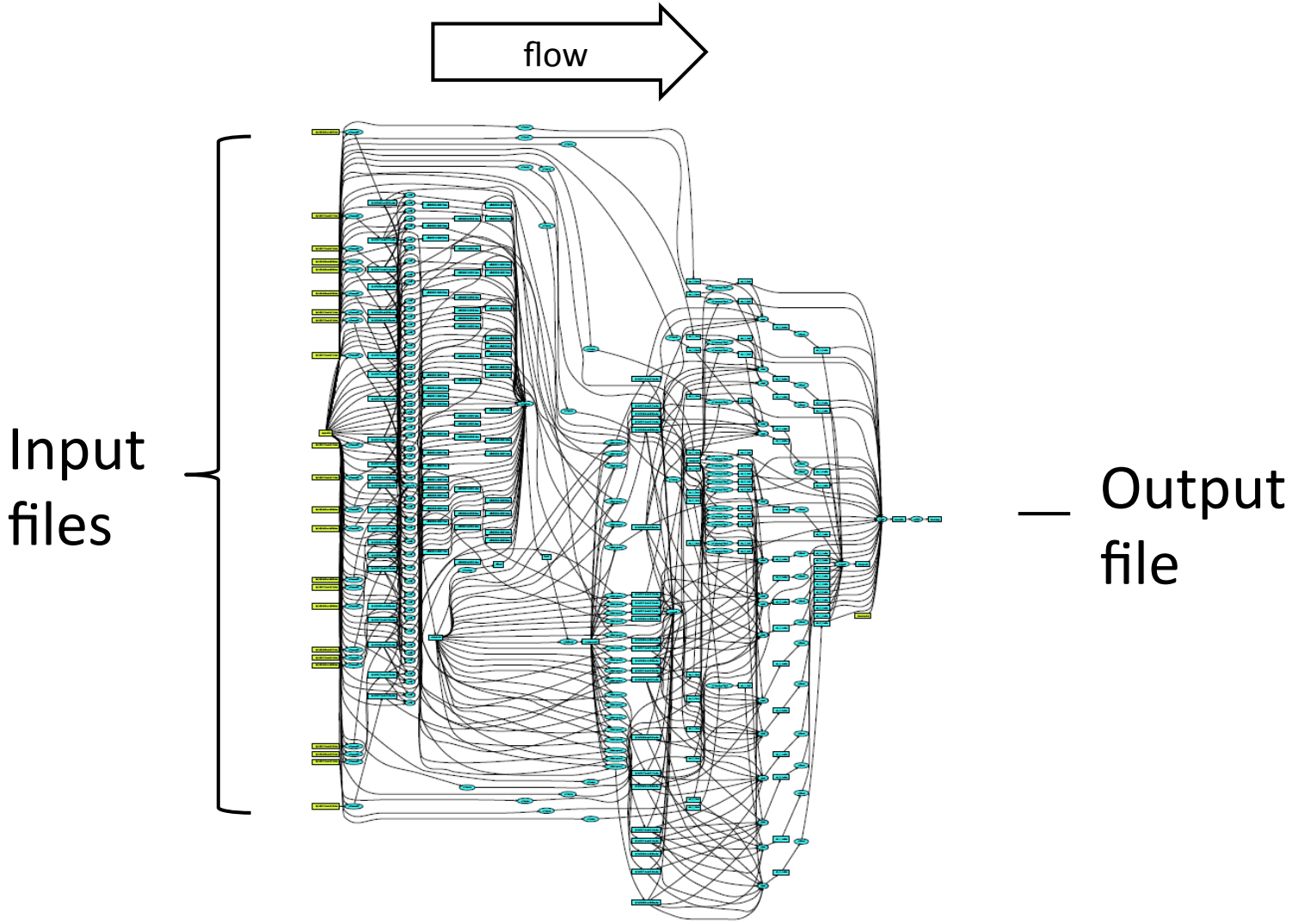
- Rake の並列実行

  multitask “out” => [“in1”, “in2”, “in3”]
  - in1..in3 についてスレッドを起動して、タスクを並列実行
  - タスクの数だけスレッドを立ち上げる
  - 並列度を制御できない
- 機能拡張が必要:
  - 並列度制御、リモート実行機能

ワークフロー記述言語としてのRakefile

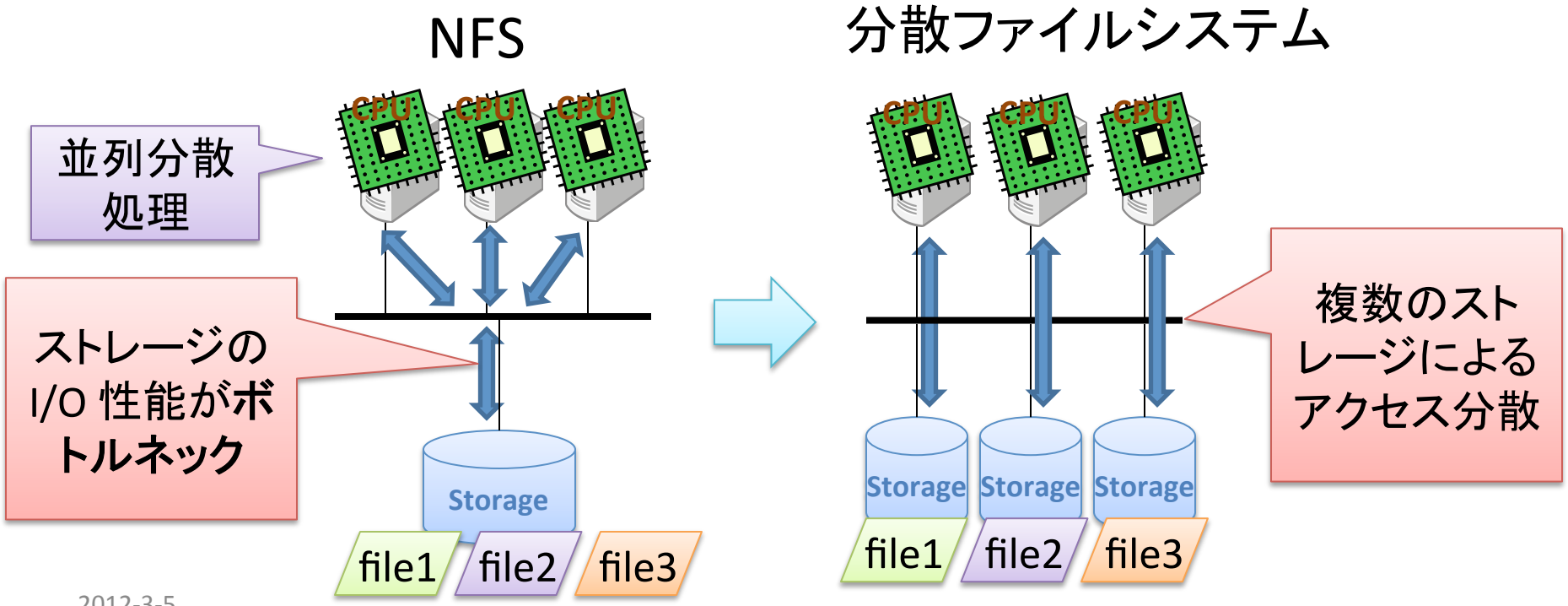
- 科学ワークフローは、Makefileのルールに収まらないことが多い
- Rakefileは、Rubyの内部DSLなので、プログラムが書ける

Montageワークフローのグラフ



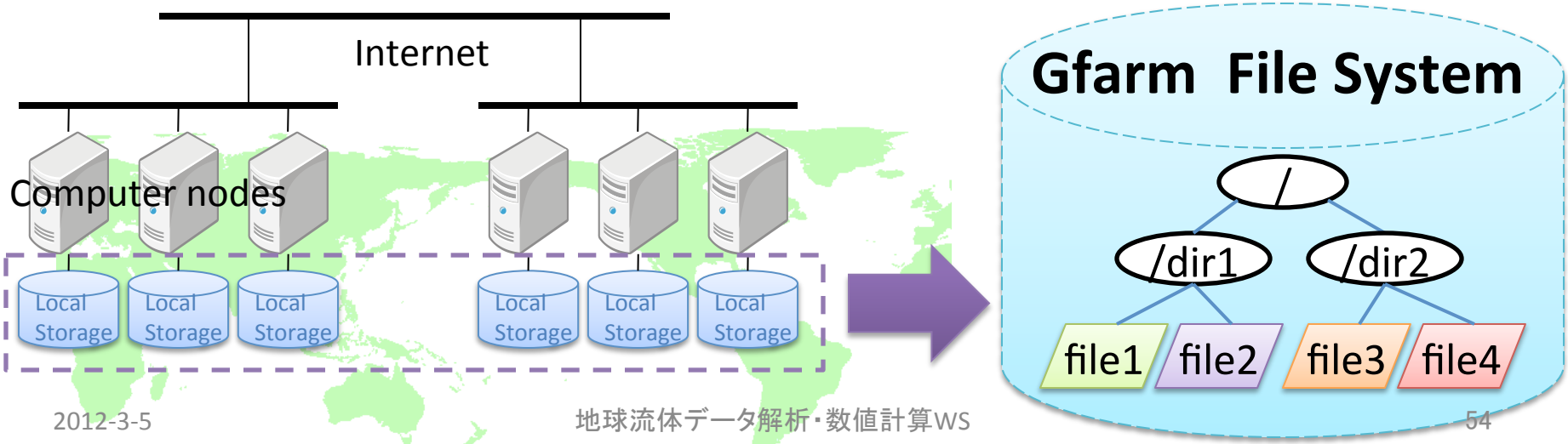
分散ファイルシステムの必要性

- データ処理では、ファイルI/O性能が重要
- NFSで並列処理すると、ストレージI/Oがボトルネック
- 並列分散処理では、分散ファイルシステムが必要



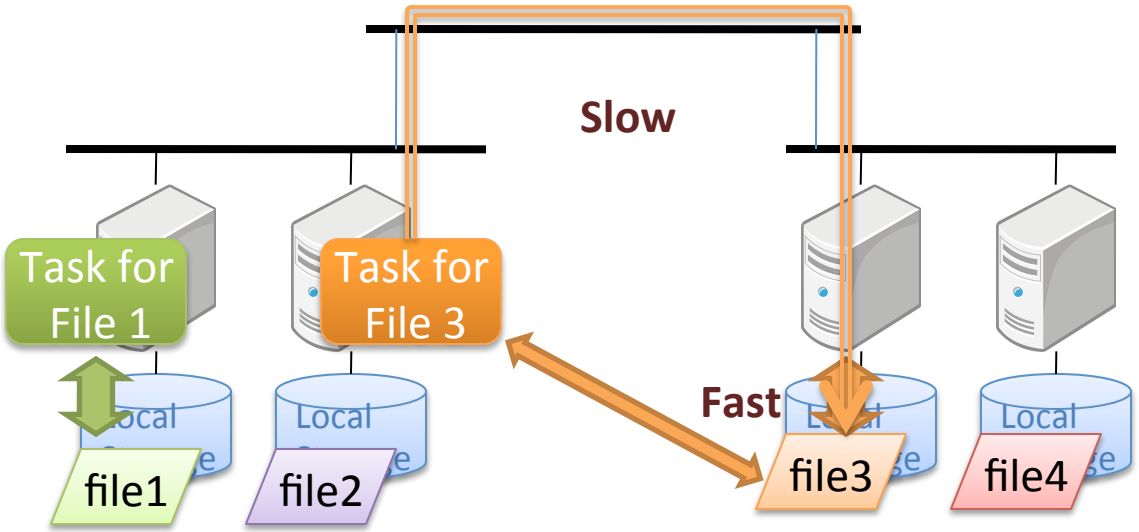
Gfarm 広域分散ファイルシステム

- 各ノードのストレージを統合
- 統一したディレクトリ空間
- 広域でファイルを共有
- http://datafarm.apgrid.org/

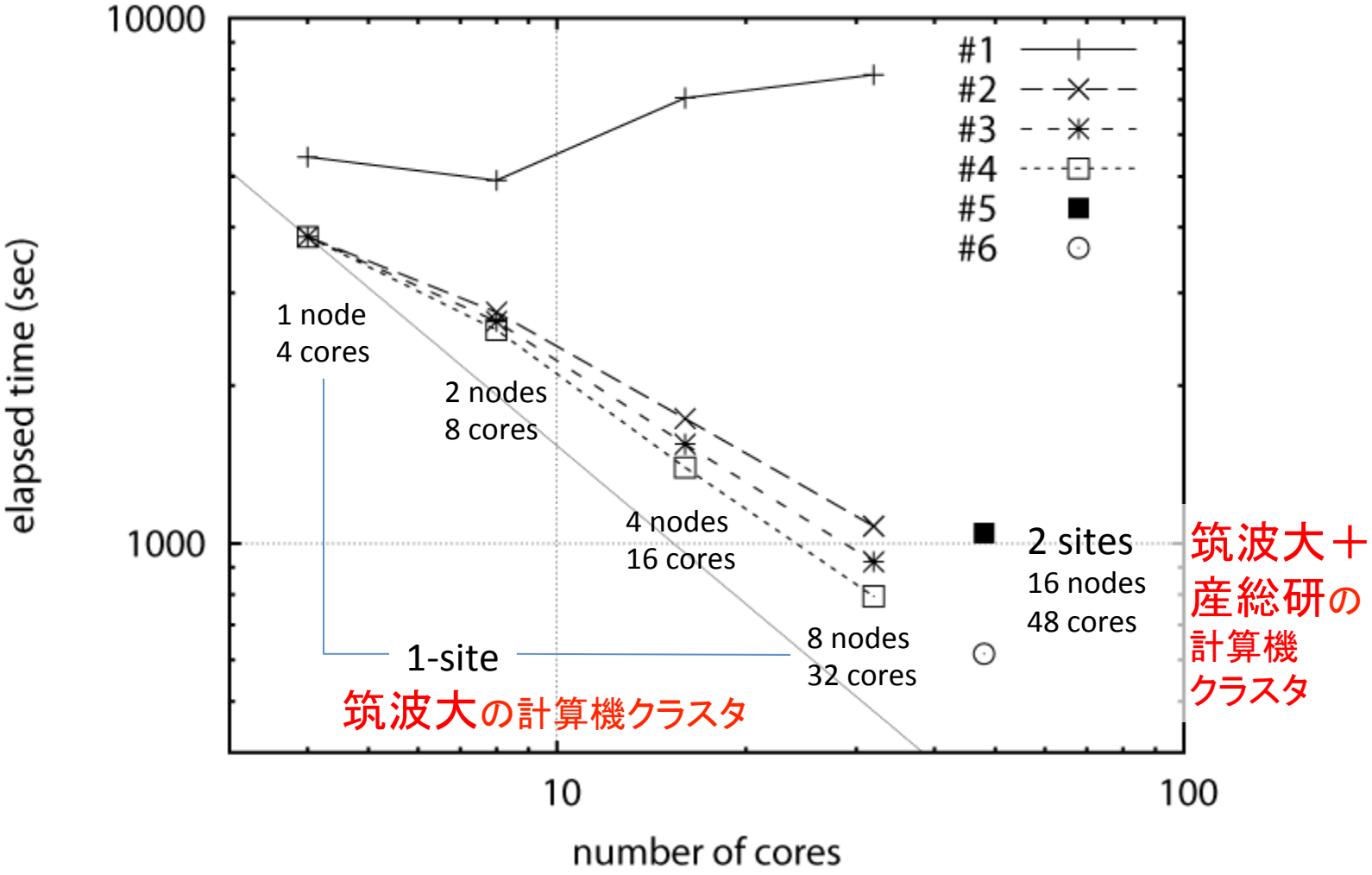


ローカリティを考慮したタスク実行

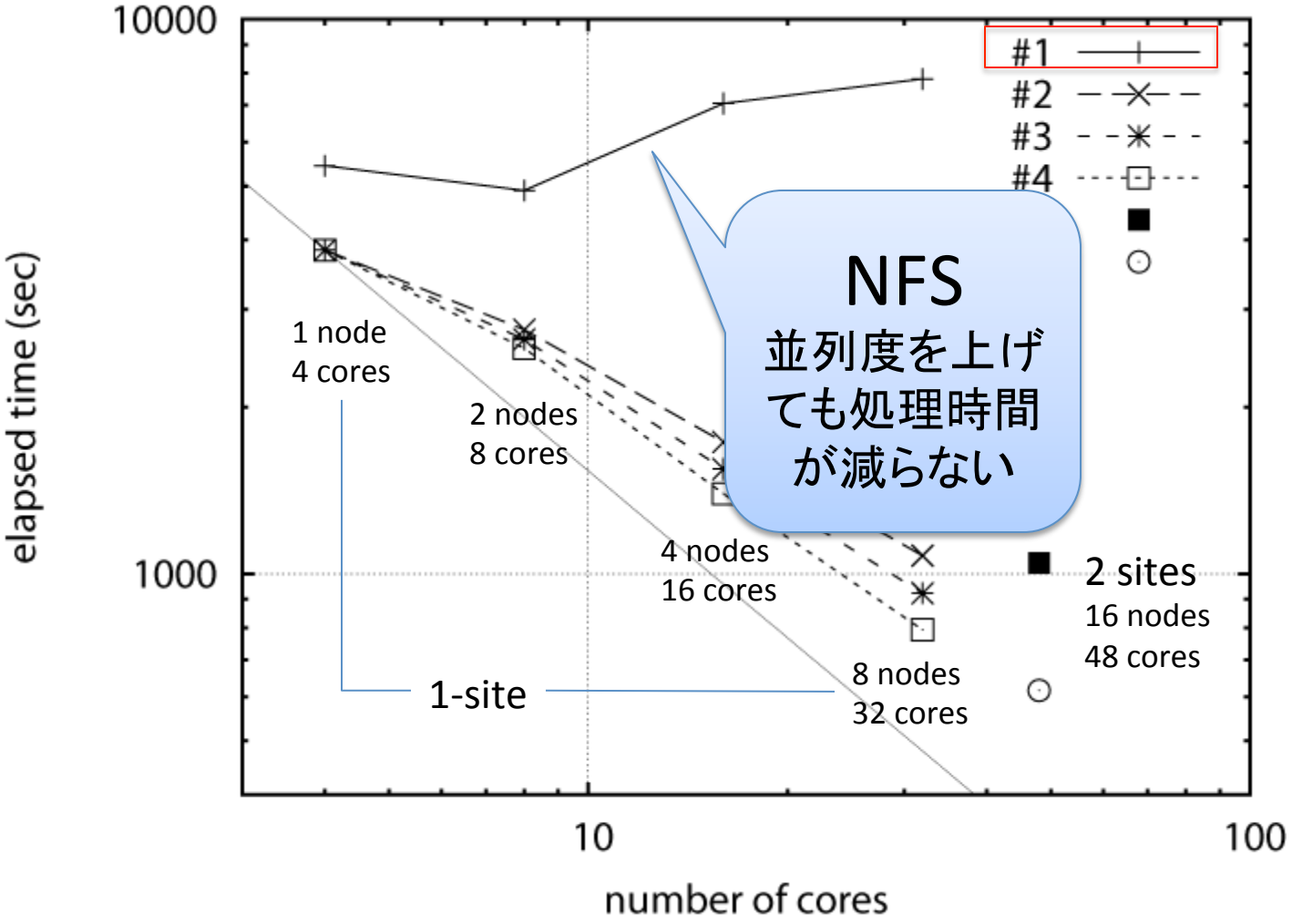
- Gfarmでは、計算ノードがとストレージノードを兼ねる
- ファイルが存在するノードで処理すれば、性能が向上
- タスク実行は、ワークフローシステムが行う



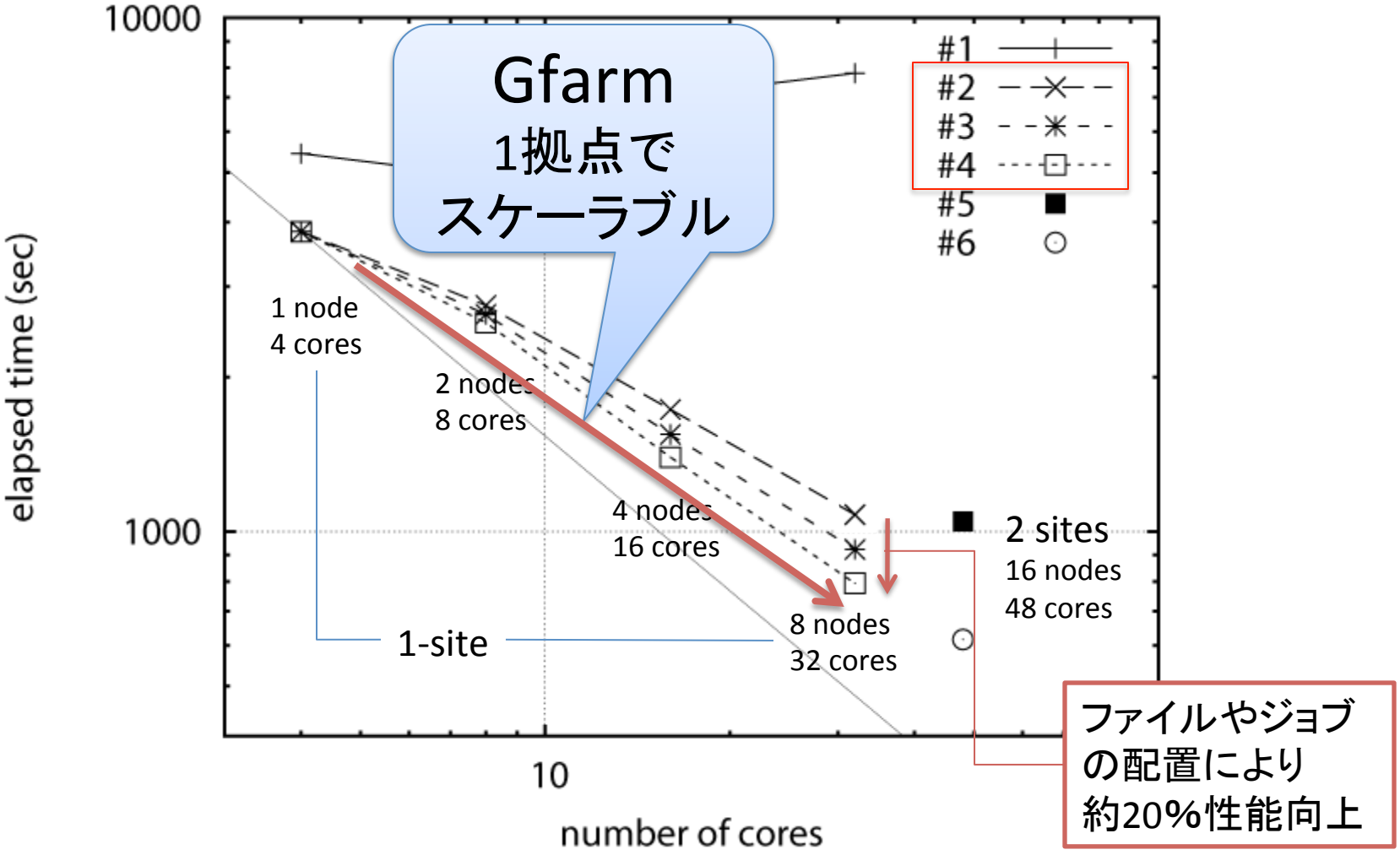
Montage Workflow の実行時間



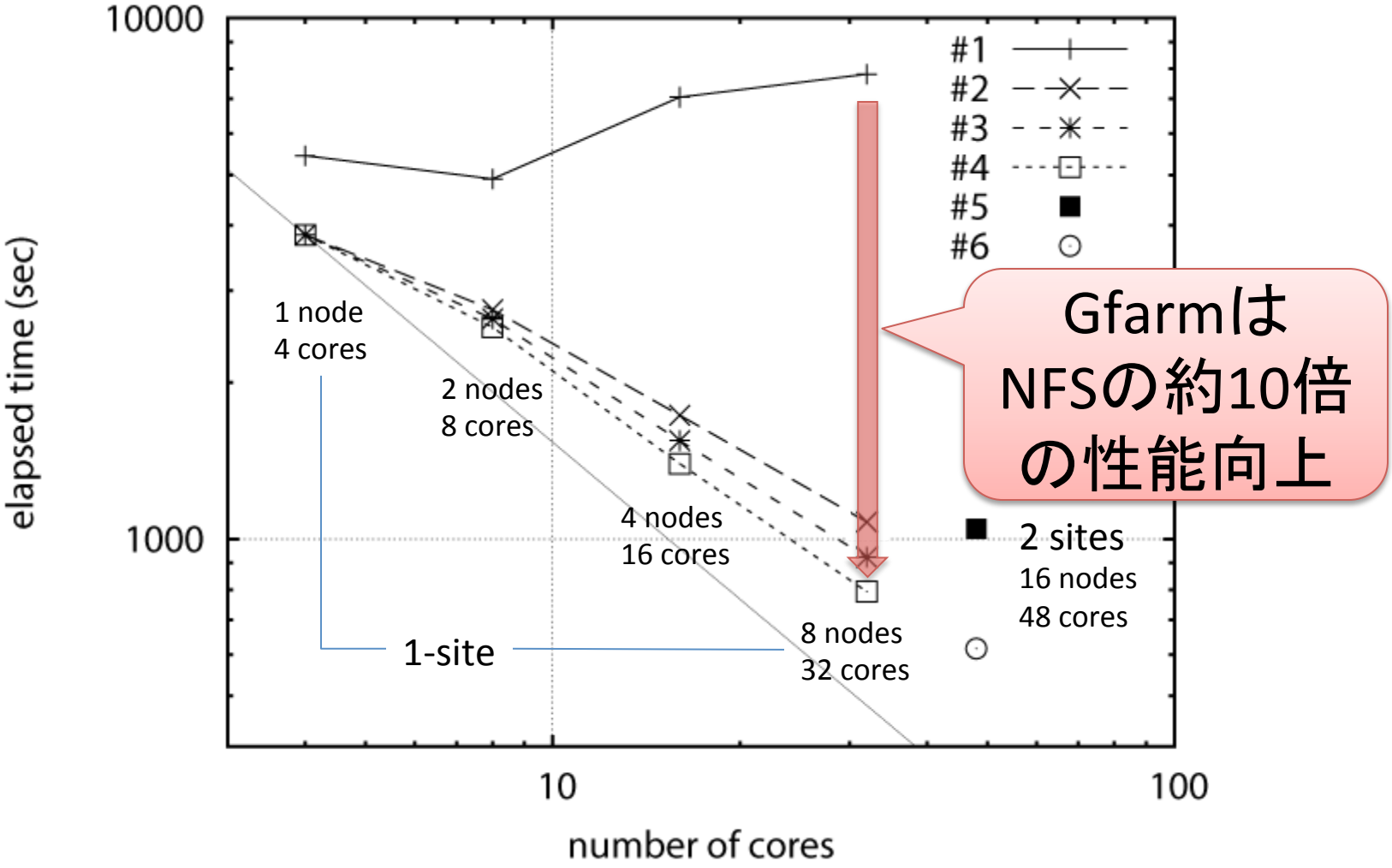
Montage Workflow の実行時間



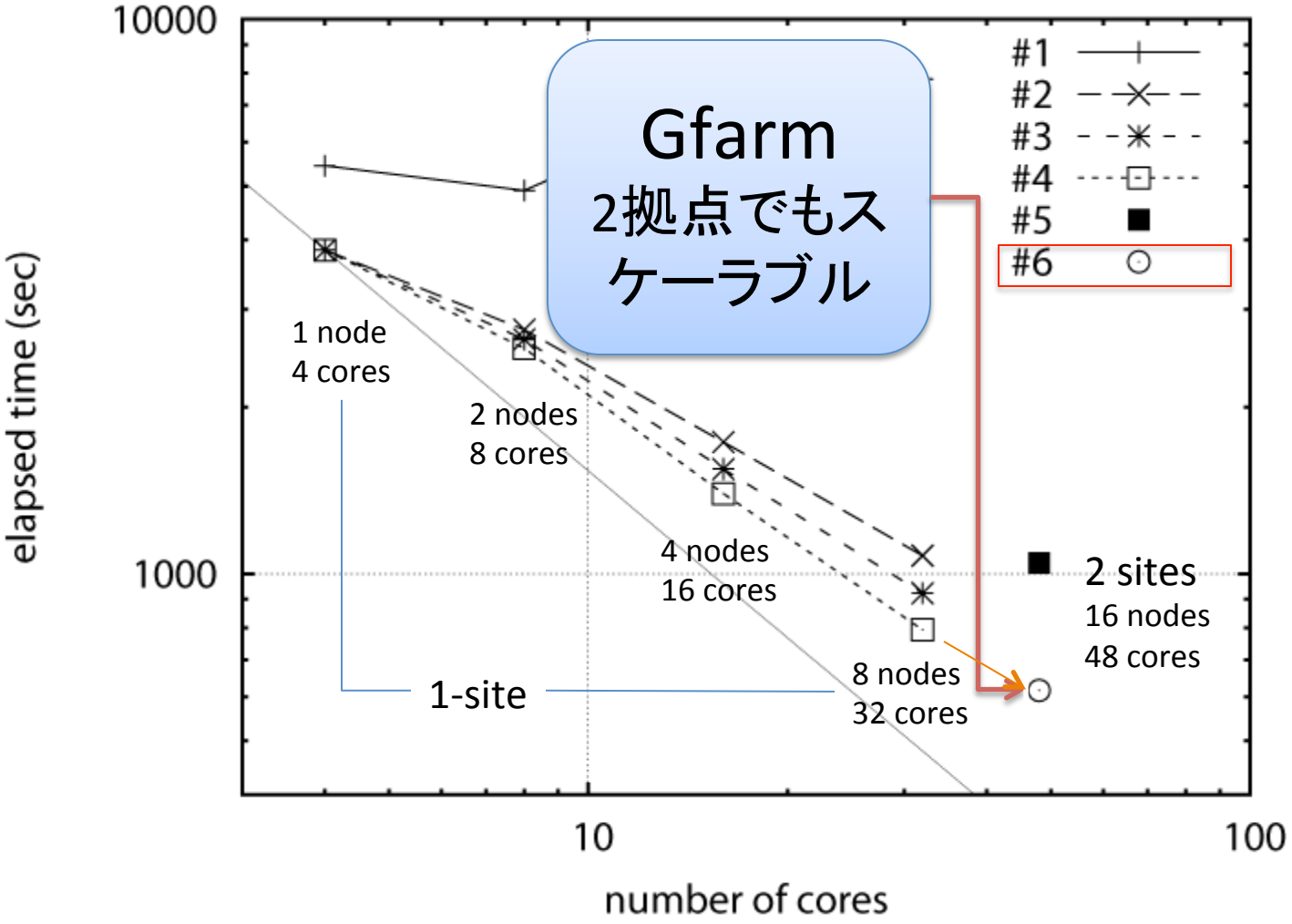
Montage Workflow の実行時間



Montage Workflow の実行時間

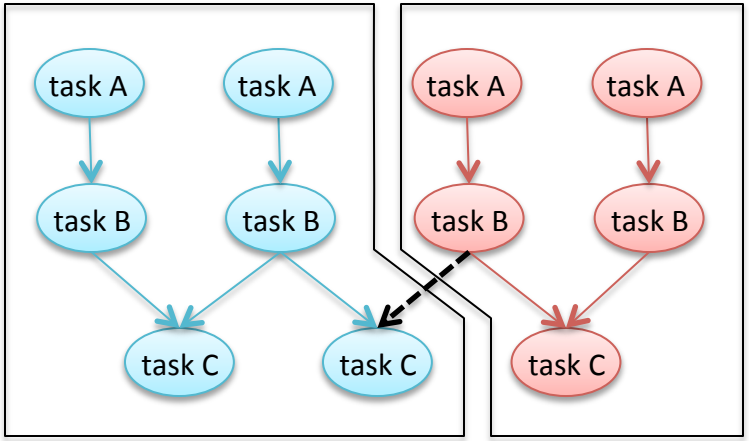


Montage Workflow の実行時間



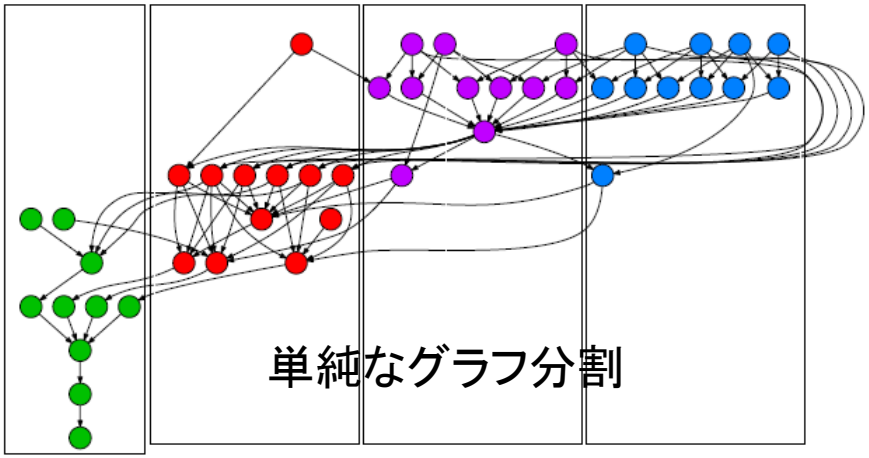
データ移動を最小化するタスク配置

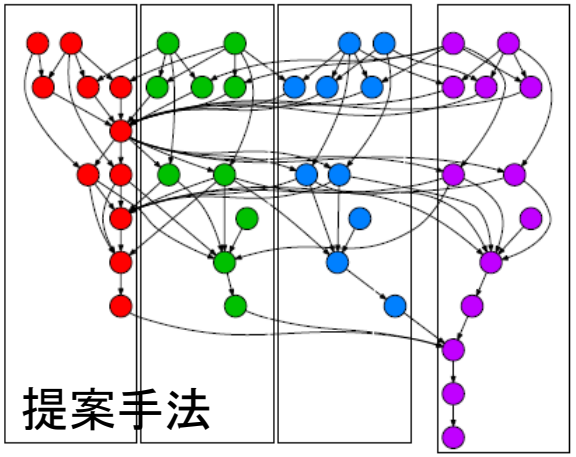
- タスクをノードに配置する際、ノード間のデータの移動を少なくする
- グループをまたぐエッジの数が最小となるように、頂点をグループ化する問題
- グラフ分割問題 と同等



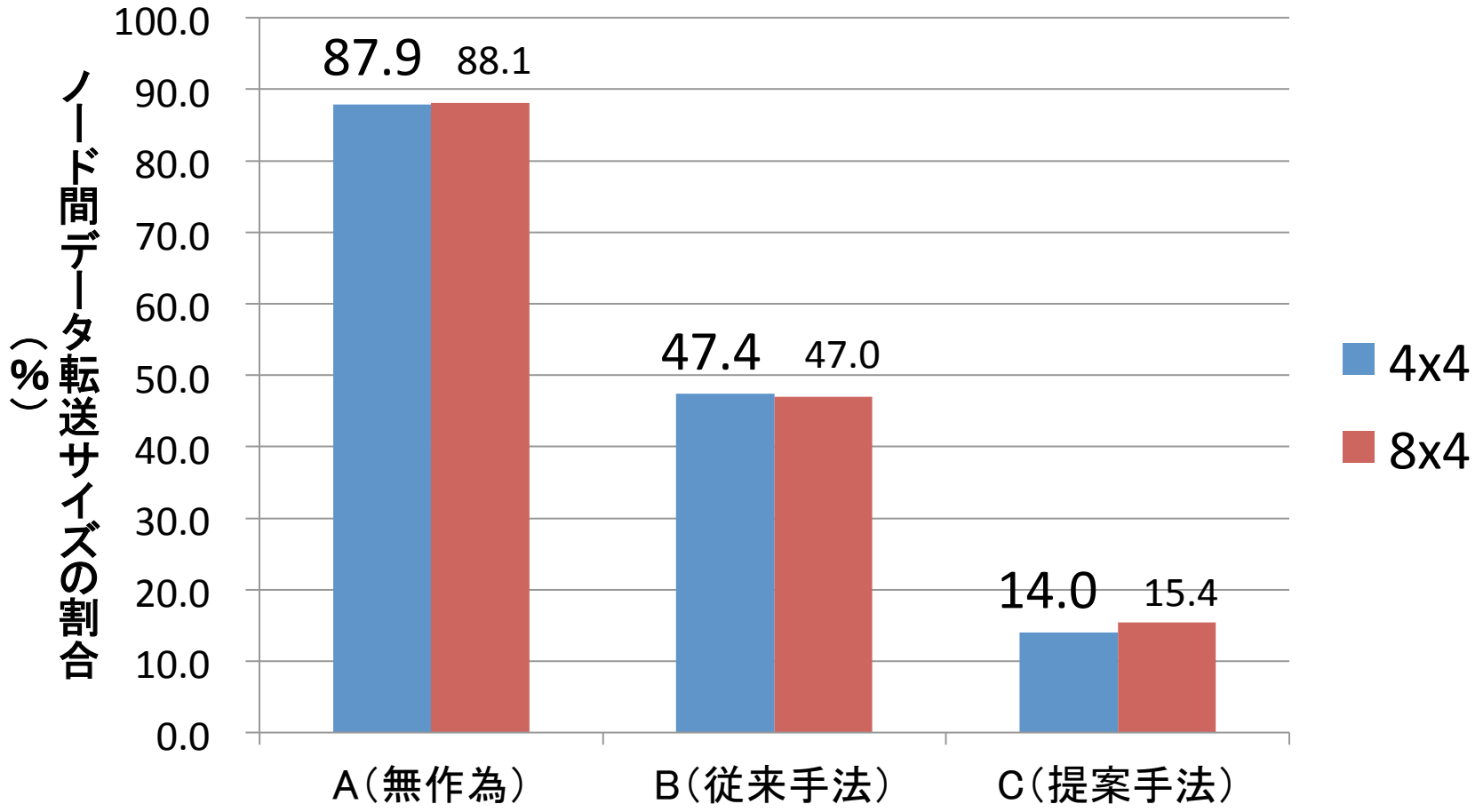
グラフ分割を用いた手法の提案

- グラフ分割問題は、タスクの依存関係や並列性を考慮していない
- 「多制約グラフ分割」を用いる手法を提案
- 国際会議 CCGrid 2012 で発表

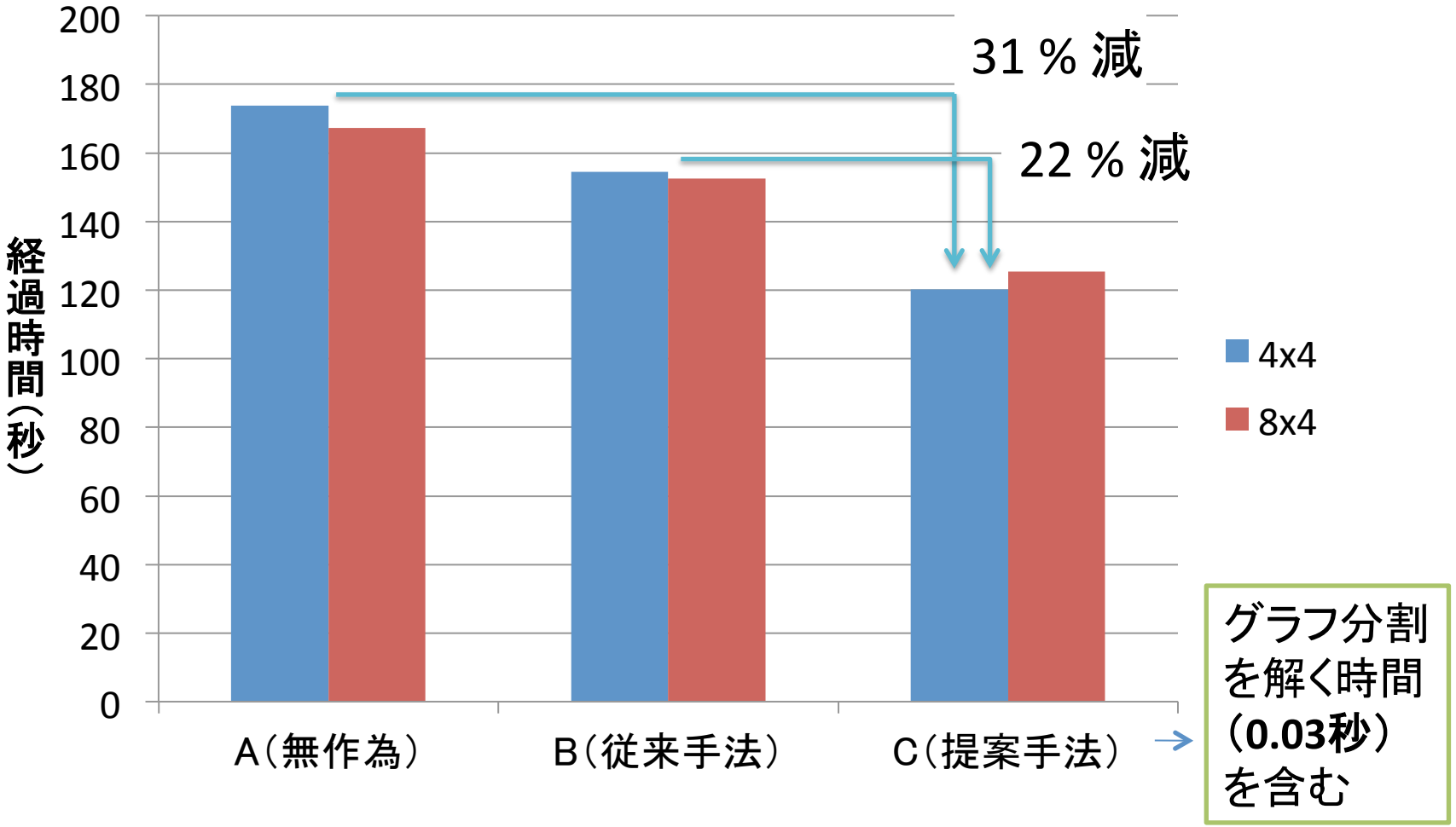




ノード間のデータ転送量



ワークフロー実行時間

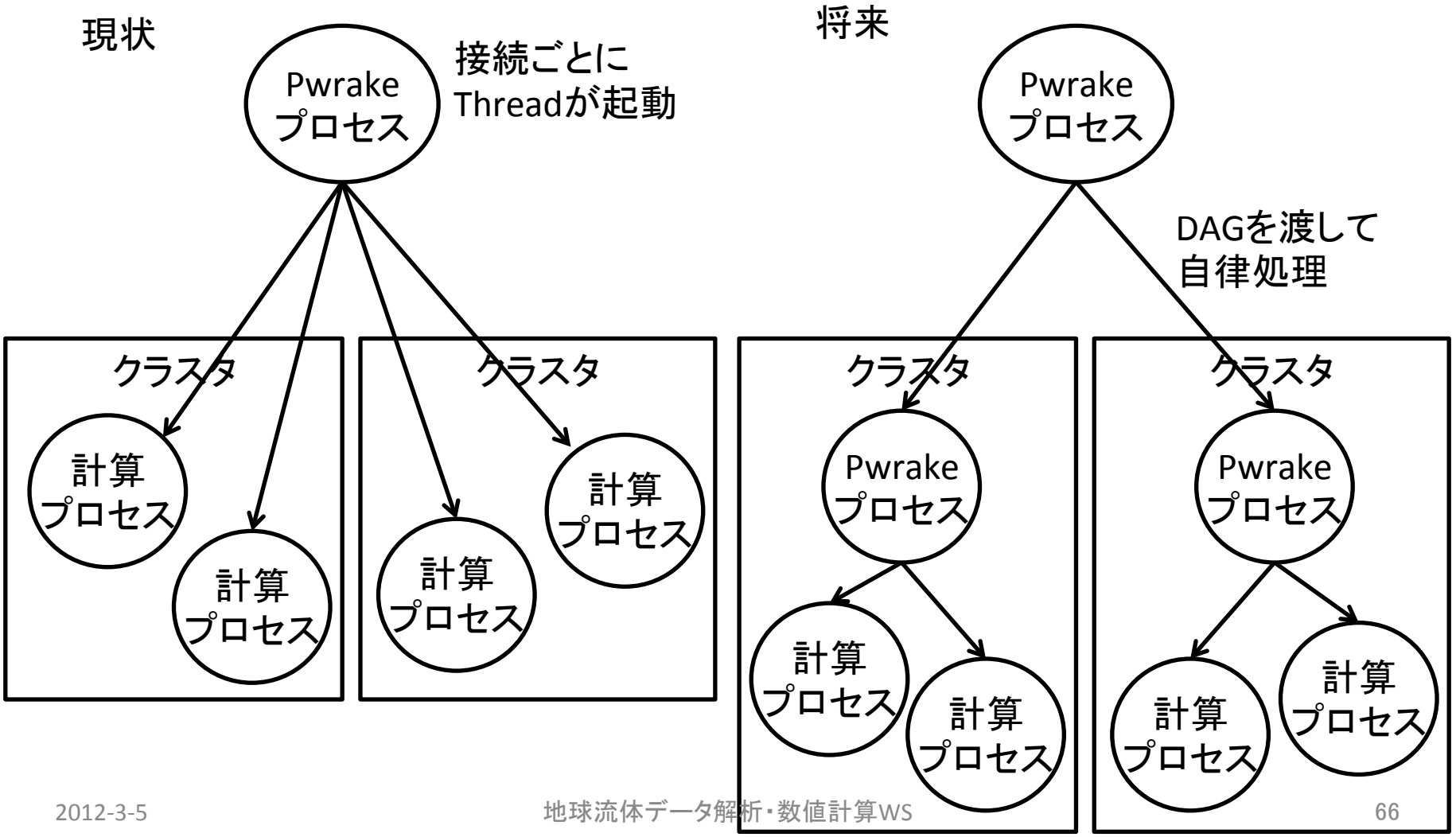

200
180
160
140
120
100
80
60
40
20
0
経過時間（秒）
31 % 減
22 % 減
4x4
8x4
A（無作為）
B（従来手法）
C（提案手法）
グラフ分割を解く時間（0.03秒）を含む

現在の Pwrake 研究開発

- **研究支援**
  - **CREST 「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」(研究総括:米澤明憲)**
  - ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア(研究代表:建部修見)

- 目標:大規模なワークフローの実行
  - 100万コアで実行可能なwrake

階層的自律分散システム



Pwrakeまとめ

- ワークフロー記述言語として、Rakeを採用
- 並列分散ワークフロー処理システムPwarkeを開発
  - URL: http://github.com/masa16/pwrake
- 分散ファイルシステムGfarmを用いることにより、スケーラブルな性能向上を達成