エクサスケールに向けた可視化戦略

小野 謙二
理化学研究所　計算科学研究機構
/ 東京大学 生産技術研究所

TOC

- HPCI-FS
- 可視化技術
- 近未来のシナリオ
- ポスト処理の課題
- 大規模データの可視化の方針
- 京における可視化
  - VisItとLSV

HPCI-FS

- 「将来のHPCIシステムのあり方の調査研究 アプリケーション分野」
  - 研究振興局長の諮問会議「HPCI計画推進委員会」のもとに「今後のHPC技術の研究開発のあり方を検討するWG」が設置
  - 今後5年から10年において計算科学から貢献できる社会的・科学的課題を抽出
  - その実現に向けて必要なアプリケーションを整理し、必要システム性能を精査
  - 各システム設計研究チームの提案する計算機システムを正当評価
  - http://hpci-aplfs.aics.riken.jp

- これまでに4回のミーティング
  - ロードマップの再構築を実施中
  - 分野間連携が必要な社会的・科学的課題を抽出

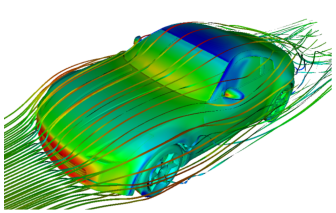
次世代機まわりの技術動向

- 2017年頃　数百PFLOPSのマシン
  - アーキは？
- 大規模シミュレーションの本格化
  - 利用技術の蓄積が進む
- データ集約科学へのシフト
  - 大規模なデータを効率よく処理し、有用なデータを提示する方法の研究
- ビッグデータの利用
  - 多様・大規模・リアルタイムのビッグデータを現実世界で活用する技術

可視化の歴史と分類

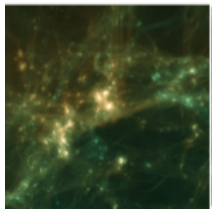
- Visualization in Scientific Computing Report
  - 1987 NSF
  - 空間的な構造が自明であるデータを対象
- 1990 IEEE Visualization
- 1995 Information Visualization
  - 空間的な構造を持たないデータ
- 2006 Visual Analytic
  - ヒトの視覚的特性を考慮した分析技術

可視化の目的と有用性

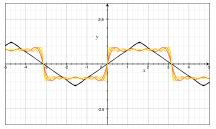


シミュレーション

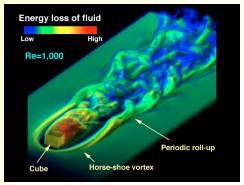
実験



観測



イメージ化
データ分析



発見

理解

判断

アウトリーチ

サイエンス/エンジニアリングを加速

- 発見・理解・判断のためのツール
  - イメージ生成
  - 分析ツール
  - 自動処理
  - データ探査（インタラクティブ性）
- 対象データは大規模分散型
  - 並列データマネジメントを共通化・効率化
  - In-Core Map Reduce

枠組みの研究と構築

近未来のシナリオ

- 単なる可視化＝イメージ生成だけではない
- データ処理との融合
- オンデマンド的な利用
- AR技術の援用
  - Google AR glass https://plus.google.com/+projectglass/posts

- “Physics of the future” - Michio Kaku
  - 2030年頃までにエキスパートシステム、人工知能は100年先
  - ハードは進化するが、ソフトはヒトがコーディング、技術革新には飛躍的なインクリメントが必要

大規模データ可視化処理の課題

- データの大規模性
  - 空間規模
  - 非定常データ
  - 多変数

- インタラクティブ性
  - ユーザの操作に対するレスポンス
  - データ量を減らす
    - プリコンピューティング、圧縮、特徴利用、並列処理
  - 効率の良いコード
    - SIMD, アルゴリズム

可視化処理技術　1

- 並列レンダリング処理
  - 並列画像生成
    - レイトレーサー、ボリュームレンダラーの効率
    - OpenGL, GLSL実装
  - 画像重畳
    - 大規模ノード間通信

可視化処理技術　2

- InSitu可視化
  - 計算と同時に可視化処理を行う
  - ファイルIOを抑制し、スケールアウトさせる
  - シミュレータ側からどのように利用するか
    - RVSLIB (NEC)

可視化処理技術　3

- データ削減
  - 圧縮技術
    - 符号圧縮技術、fpzip...
    - POD
- 特徴抽出
  - 分野毎に異なる手法・知識を利用
    - MD, CFD...

可視化処理技術　4

- データ処理との融合
  - Visual Analytic
- データベースとの融合
  - クエリー処理
- プロセスマイニング
  - 可視化プロセスの効率化

ポスト処理の課題

- データファイルハンドリング
  - シミュレータファイルの管理
  - ポスト処理からもシームレスに利用

- ワークフロー
  - 定型処理の自動化
  - ポータビリティ
  - 記述性、汎用性

可視化技術の難しさ

- 幅広い対象分野
- 共通技術
- 対象個別の技術

- ソフトウェア蓄積の必要性
  - 再利用
  - 統合化

可視化技術は計算科学と計算機科学の融合領域

可視化技術研究の方向

- 個別要素技術の研究開発

- 要素技術の統合化
  - 要素技術のコンポーネント化
  - コンポーネントを利用するフレームワーク
  - フレームワークを利用した可視化のカスタマイズ

All Japanでの取り組みが必要

京における可視化

- Issues
  - 多分野のアプリケーション、データ構造
  - 京はある種特殊な環境

- 2つのツールを提供
  - 既存の可視化ツール　(VisIt)
  - 京の上で動作する可視化ツール (LSV)

PCクラスタを利用した可視化環境

- Intel SB クラスタ
  - 32+1ノード、2CPU/ノード、8コア/CPU >> 512コア
  - メモリ　64GB/ノード >> 2TB
  - IB QDR

- 京では動かないアプリケーションを利用
  - VisIt, ParaView, FieldViewなどの汎用可視化ツール
  - その他、汎用データ処理アプリ

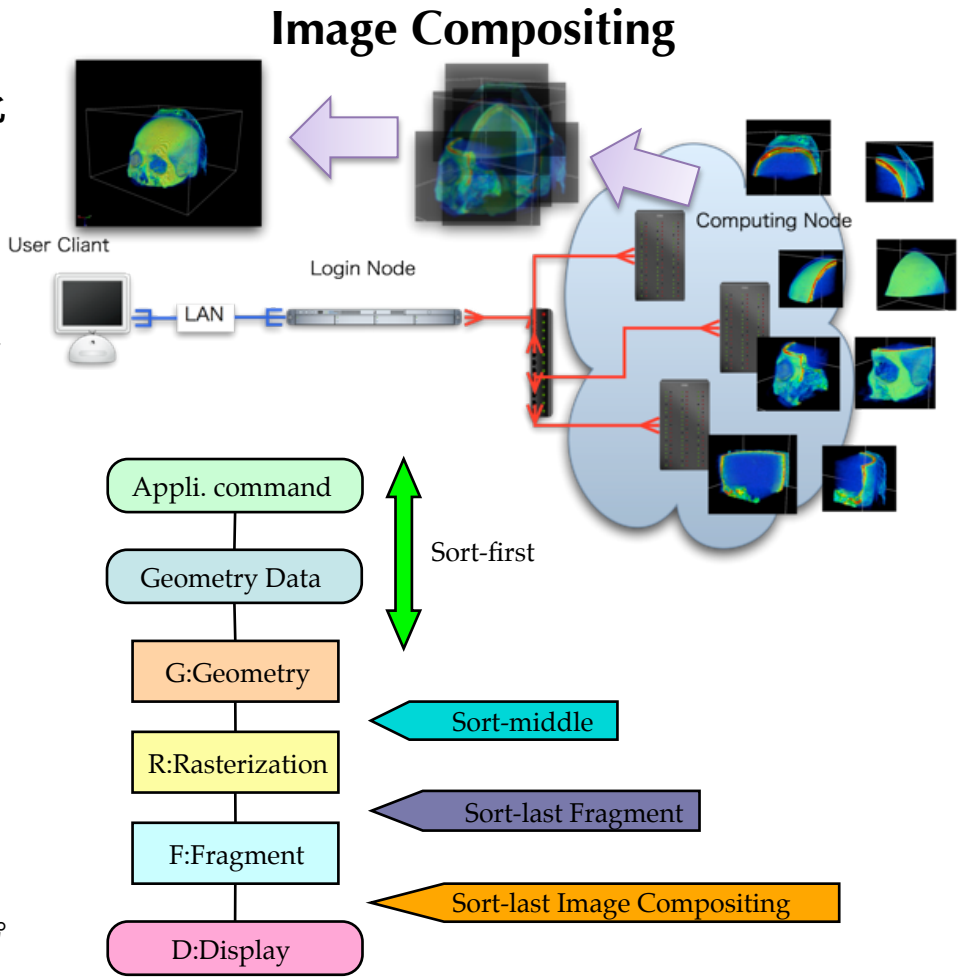
VisIt

- US-DOEファンドでLLNLが主導で開発、ParaViewと双璧をなすオープンソース可視化ツール
- VTK（Visualization Tool Kit）を用いて構築
- 多様な可視化方法を実装
- 12000プロセス並列での可視化実績

- 現時点では、京では動作せず

LSV : Large Scale Visualization

- LSV ?
  - 大規模データ可視化のための、可視化フレームワーク
- 機能:
  - リモート、バッチ、インタラクティブ可視化など代表的なユーザシナリオに対応
  - 構造格子、非構造格子、点群など基本的なデータ構造
  - 図種は基本的なコンター、等値面、ベクトル、VRなど
  - 分散データ対応、ソートラスト型のイメージ生成
  - モジュール化された構造
  - マルチプラットホーム・クライアント
- キーテクノロジ:
  - ソートラスト型の分散データ可視化
  - スケーラブルなイメージ重畳アルゴリズム
- 京の上でも動作する大規模並列可視化アプリとして開発中

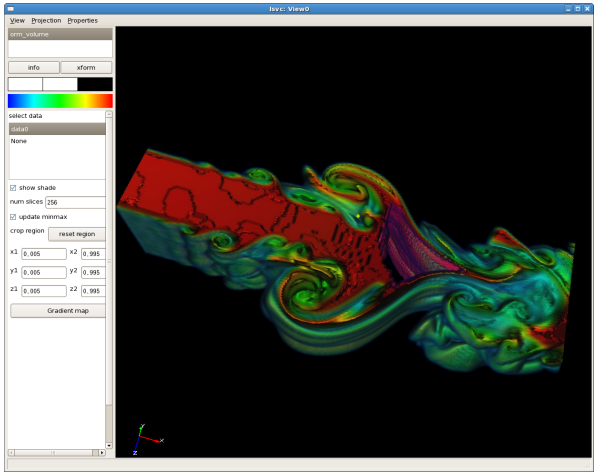


M. Eldridge, Parallel Graphics: Scalability and Communication, Course Note 37, SIGGRAPH 2001.

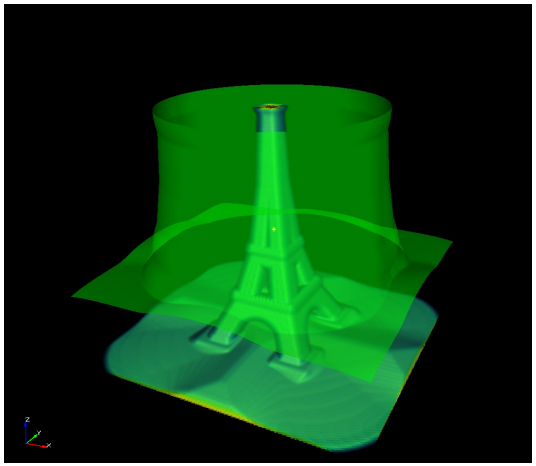
Sorting Taxonomy

LSVでできること

- 並列可視化
  - バッチ動画生成
  - インタラクティブ可視化
  - HW Rendering 100GPU (RICC)
  - SW Rendering 2048 (RICC)
- 対応データ
  - 直交格子、非構造格子
- 図種
  - コンター、塗りつぶし、ベクトル
  - ボリュームレンダリング
  - 半透明オブジェクト重ね合わせ
  - 勾配マップ



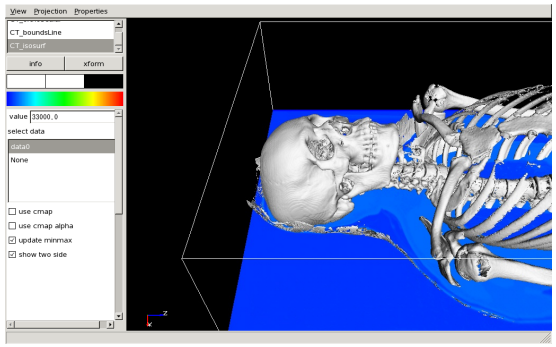
ボリュームレンダリングイメージ

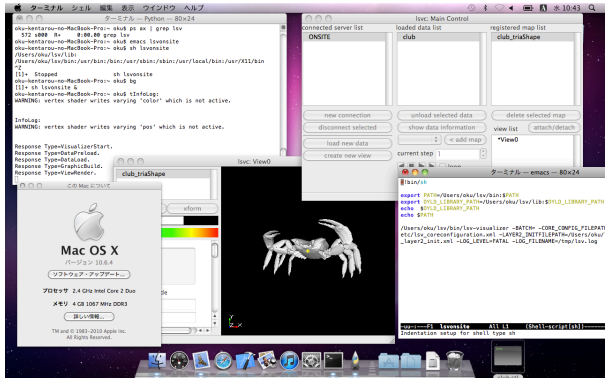


半透明オブジェクトの重ね合わせ

LSV開発概要

- 2007
  - 基盤チームにて可視化システム開発開始
  - 可視化ヒアリング（関連PJや大規模計算ユーザ）
  - システム設計、 I/Oライブラリプロトタイプ
- 2008
  - サーバー/クライアント開発
- 2009
  - 開発方針変更
- 2010
  - RICCにて100GPUを利用した並列可視化デモ
  - 京へ向けた稼働テスト開始
- 2011
  - 京でのテスト
- 2012
  - 京向けの可視化対応中





LSV開発経緯

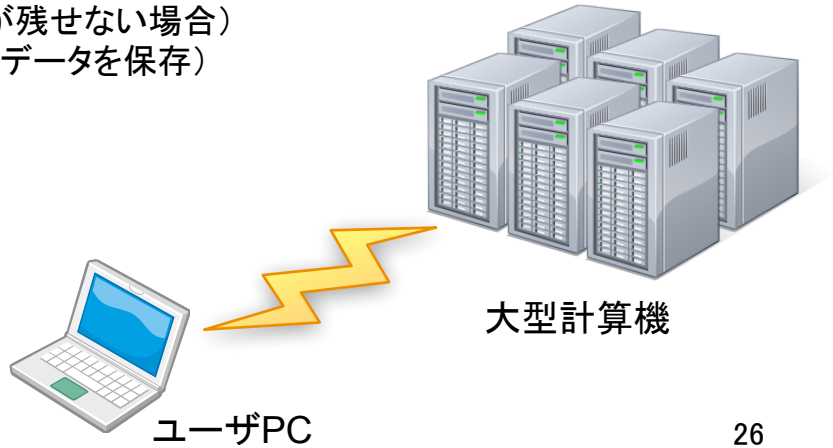
- 2007 当初計画
  - 多様な利用シナリオを想定した多機能可視化
    - インタラクティブ性 >> GPUクラスタ
    - 「京」での可視化対応 >> CPU描画+ 多並列処理
- 2009 開発プライオリティ変更
  - 行政刷新会議をうけ、予算削減
    - 京の可視化クラスタ導入中止へ
  - インタラクティブ可視化対応のプライオリティを下げる
    - 京でのSWレンダリング対応を開始
- 2012
  - レンダリングAPIの変更 OpenGL -> GLSL/GLESへ

大規模データ可視化システム LSV 利用シナリオの分類 ― 4つの軸

- サーバープログラムの実行モード

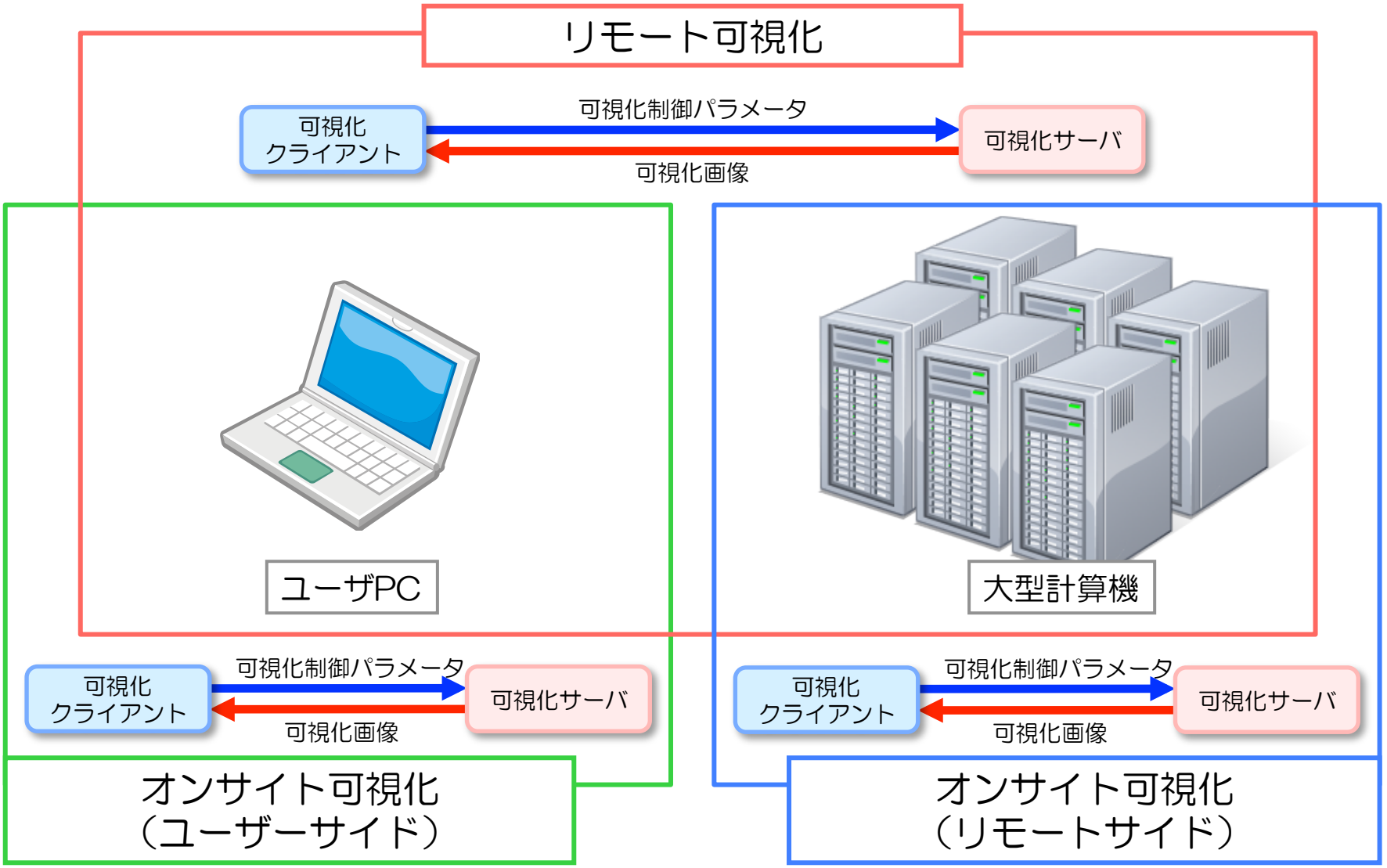
モード	クライアント	可視化プロセス
オンサイト	ユーザサイド	ユーザPC
	リモートサイド	サーバー
リモート	ユーザーサイド	サーバー

- ユーザインタラクション
  - インタラクティブ : ユーザの操作に応じて可視化
  - バッチ : 定義されたシナリオに基づき可視化
- 実行タイミング
  - リアルタイム : 計算途中に可視化（データが残せない場合）
  - ポスト : 終了後に可視化（ディスクにデータを保存）
- レンダラー
  - ハードウェア : GPUを利用した高速描画
  - ソフトウェア : CPUで描画可能，高品質

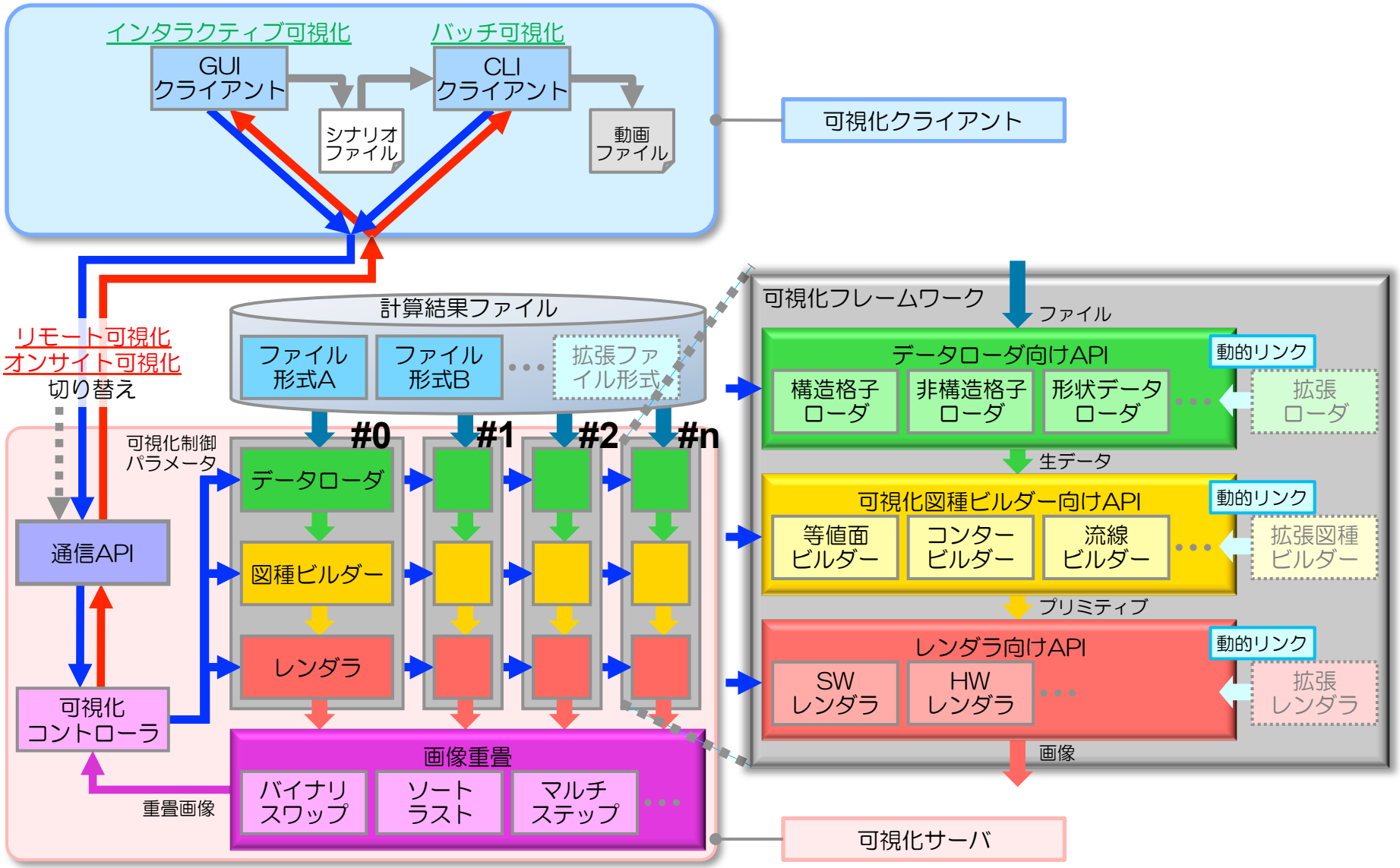


多様なユーザ利用シナリオに対応できる設計
プライオリティを付けて，開発

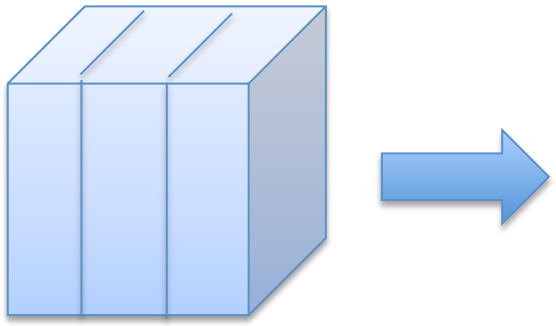
可視化の実行場所


リモート可視化
可視化制御パラメータ
可視化
クライアント
可視化サーバ
可視化画像
ユーザPC
大型計算機
可視化制御パラメータ
可視化
クライアント
可視化サーバ
可視化画像
オンサイト可視化
（ユーザーサイド）
可視化制御パラメータ
可視化
クライアント
可視化サーバ
可視化画像
オンサイト可視化
（リモートサイド）

システム構成イメージ

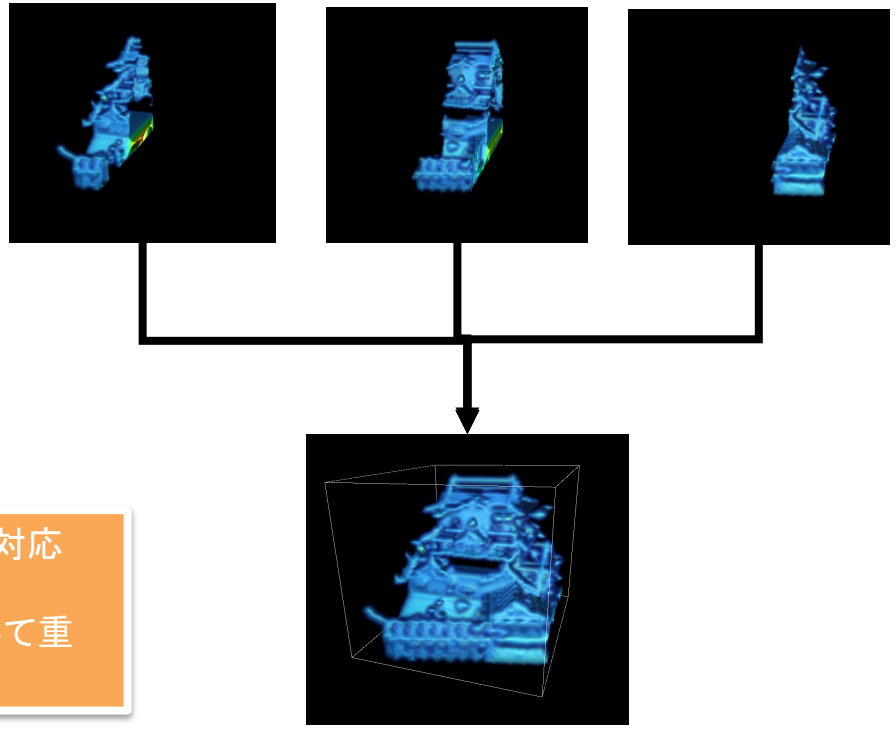

インタラクティブ可視化
バッチ可視化
GUI
クライアント
CLI
クライアント
シナリオ
ファイル
動画
ファイル
可視化クライアント
リモート可視化
オンサイト可視化
切り替え
計算結果ファイル
ファイル
形式A
ファイル
形式B
拡張ファ
イル形式
#0
#1
#2
#n
可視化制御
パラメータ
通信API
可視化
コントローラ
データローダ
図種ビルダー
レンダラ
画像重畳
バイナリ
スワップ
ソート
ラスト
マルチ
ステップ
重畳画像
可視化フレームワーク
ファイル
データローダ向けAPI
動的リンク
構造格子
ローダ
非構造格子
ローダ
形状データ
ローダ
拡張
ローダ
生データ
可視化図種ビルダー向けAPI
動的リンク
等値面
ビルダー
コンター
ビルダー
流線
ビルダー
拡張図種
ビルダー
プリミティブ
レンダラ向けAPI
動的リンク
SW
レンダラ
HW
レンダラ
拡張
レンダラ
画像
可視化サーバ

並列可視化



計算領域を分割して，並列に計算

部分領域の担当データのみを用いて可視化し，画像データを作成

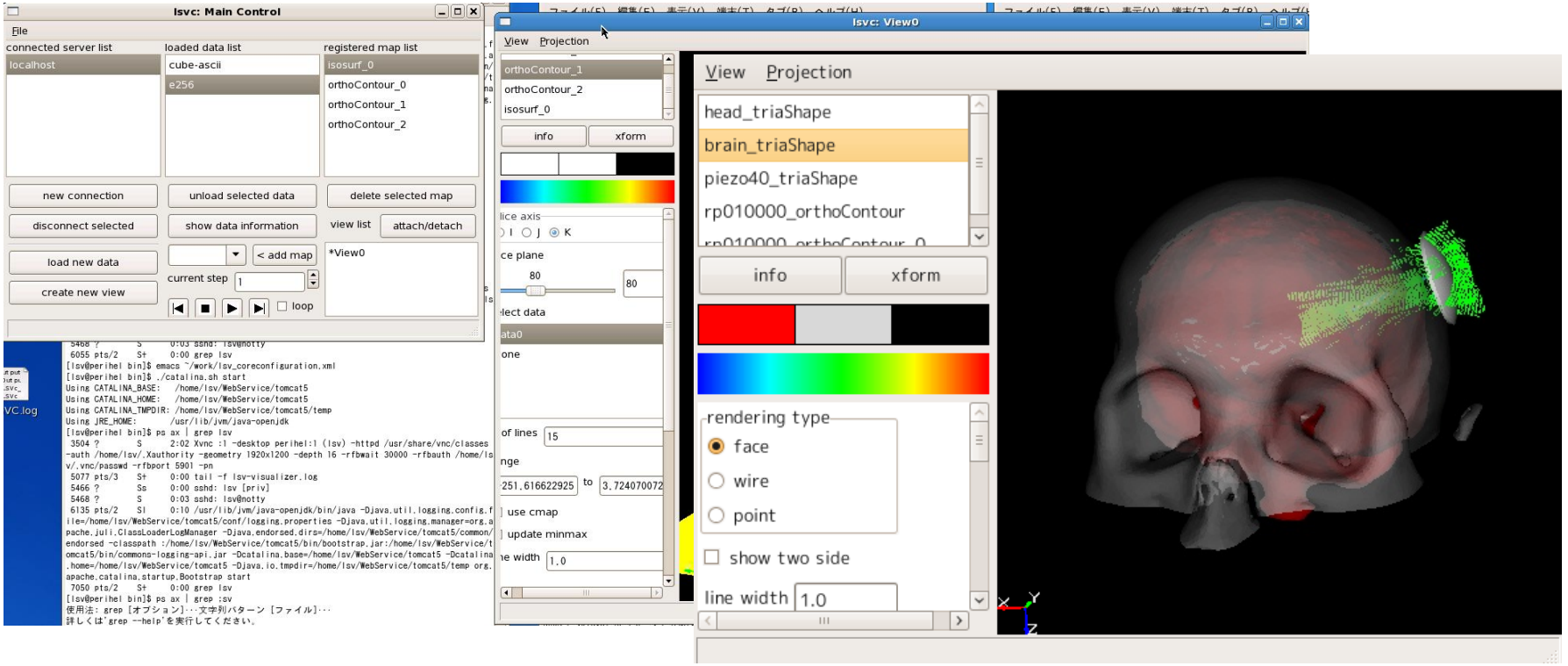


- 並列領域分割型のシミュレーションに対応
- 部分領域ごとに画像を作成
- 画像データを集めて，視線情報を用いて重ね合わせるソートラスト方式

全ての部分領域の画像データを集め，視線情報と部分領域の配置情報から画像を重ね合わせる

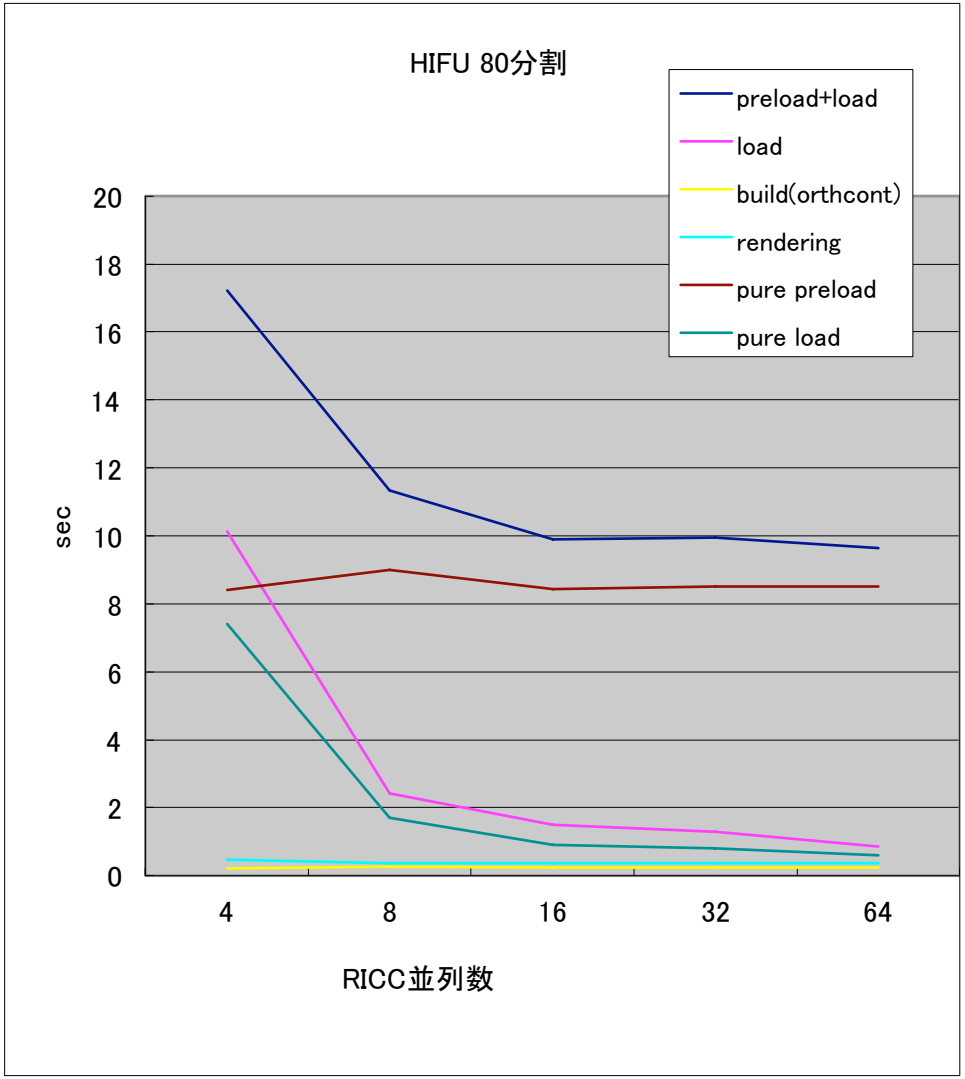
RICCでの並列可視化テスト結果
April 2010

- HIFU解析結果データ(600x800x600, 200step)
- 最大100並列（リモートインタラクティブモード）
- 視点変更・可視化パラメータの変更操作では、概ねインタラクティブ
- 現時点では，データロードがボトルネック

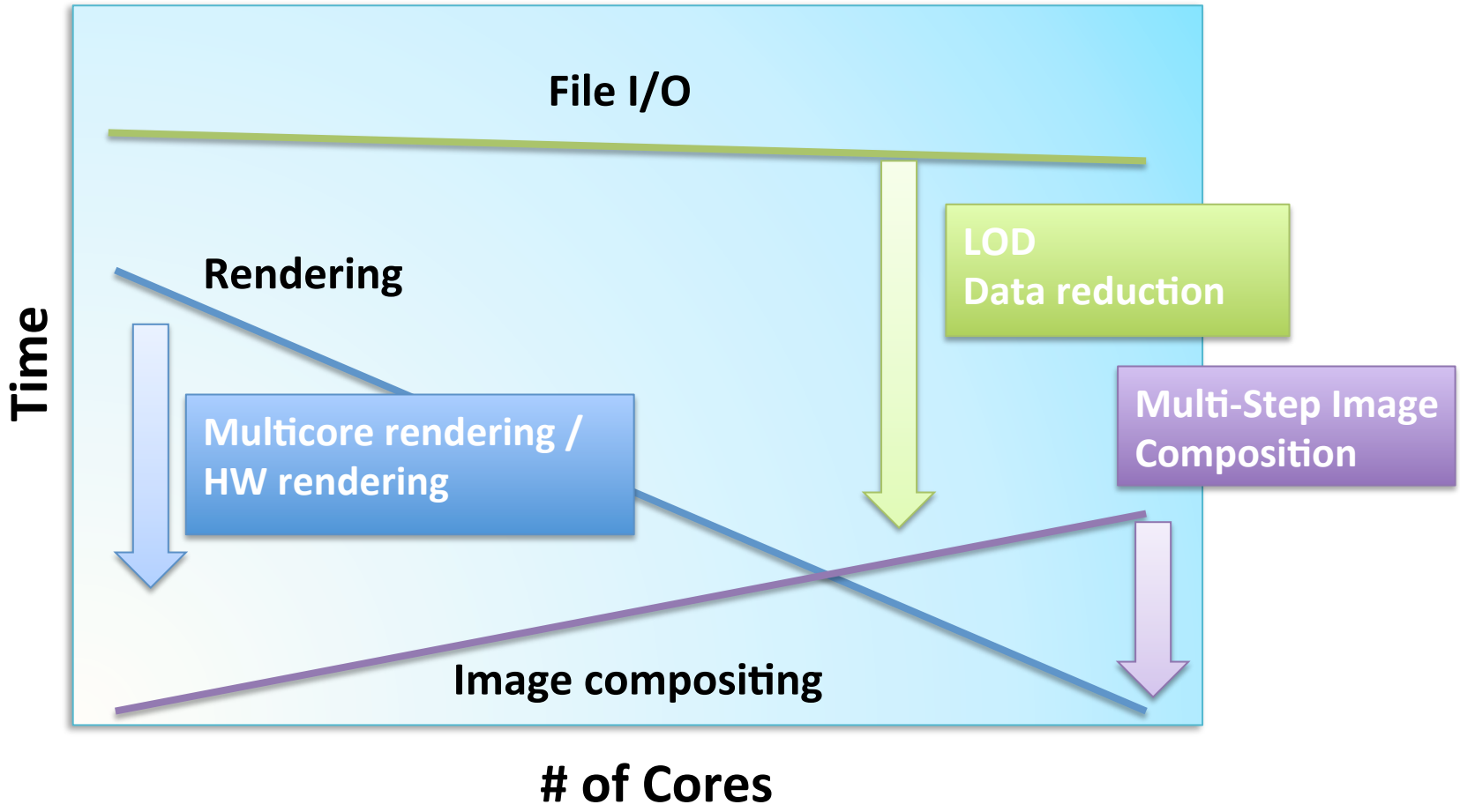


測定結果

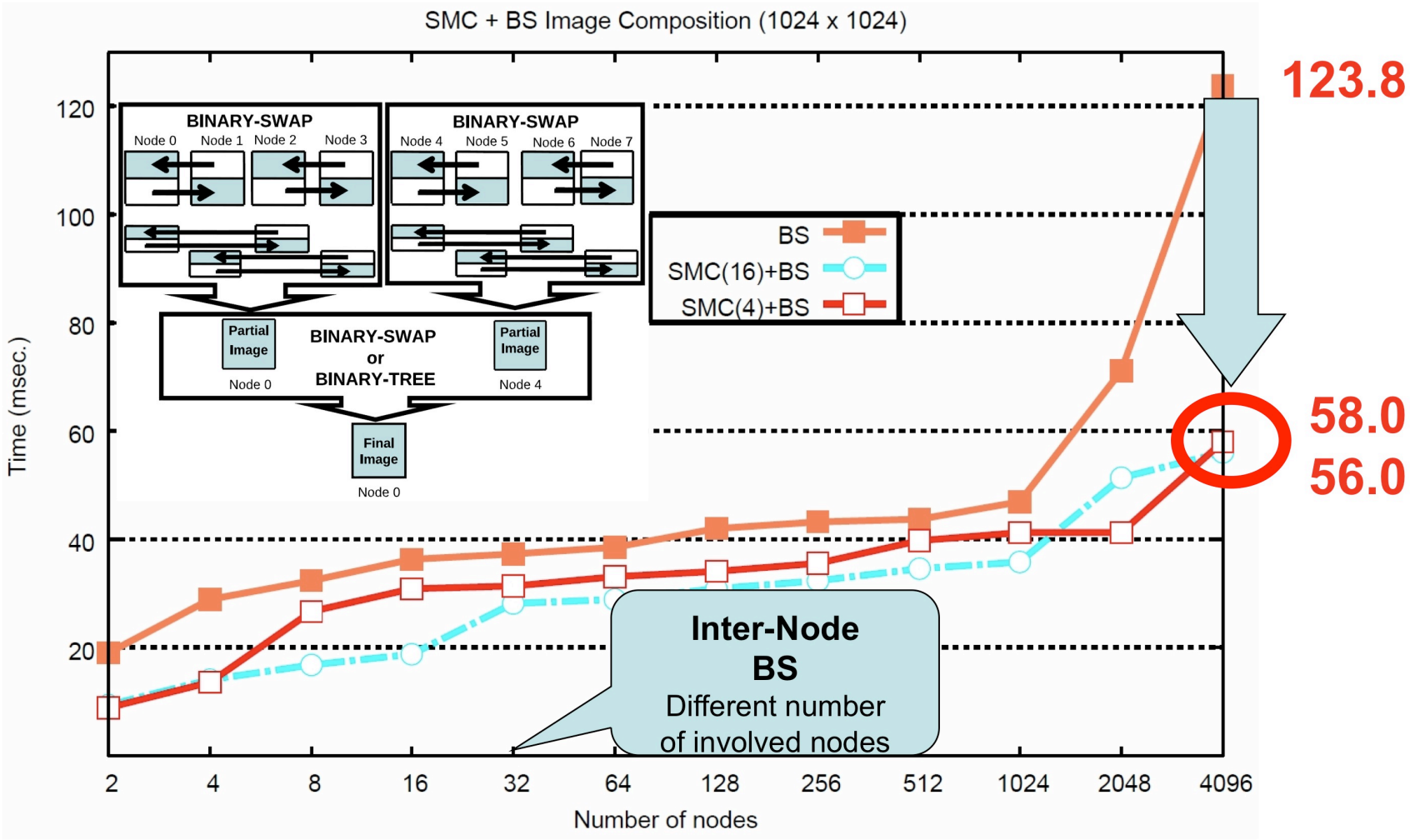
- HIFUデータを80領域に分割
- 読み込み時間と作画時間を計測
- 作画はコンターライン
- 16並列まではスケール
- 並列数を上げると読み込み時間よりも作画時間が支配的



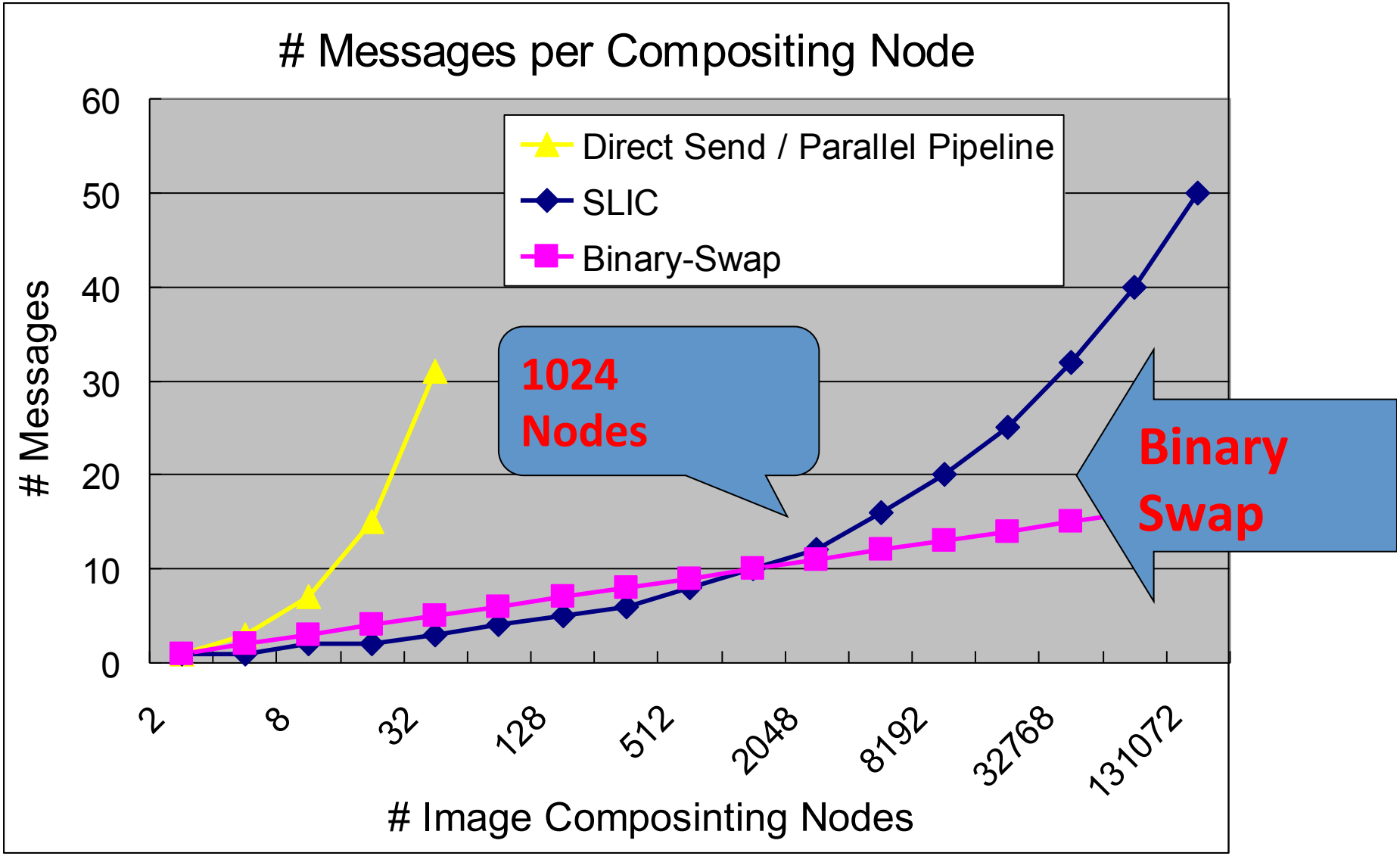
Strategy:
Interactivity and Scalability



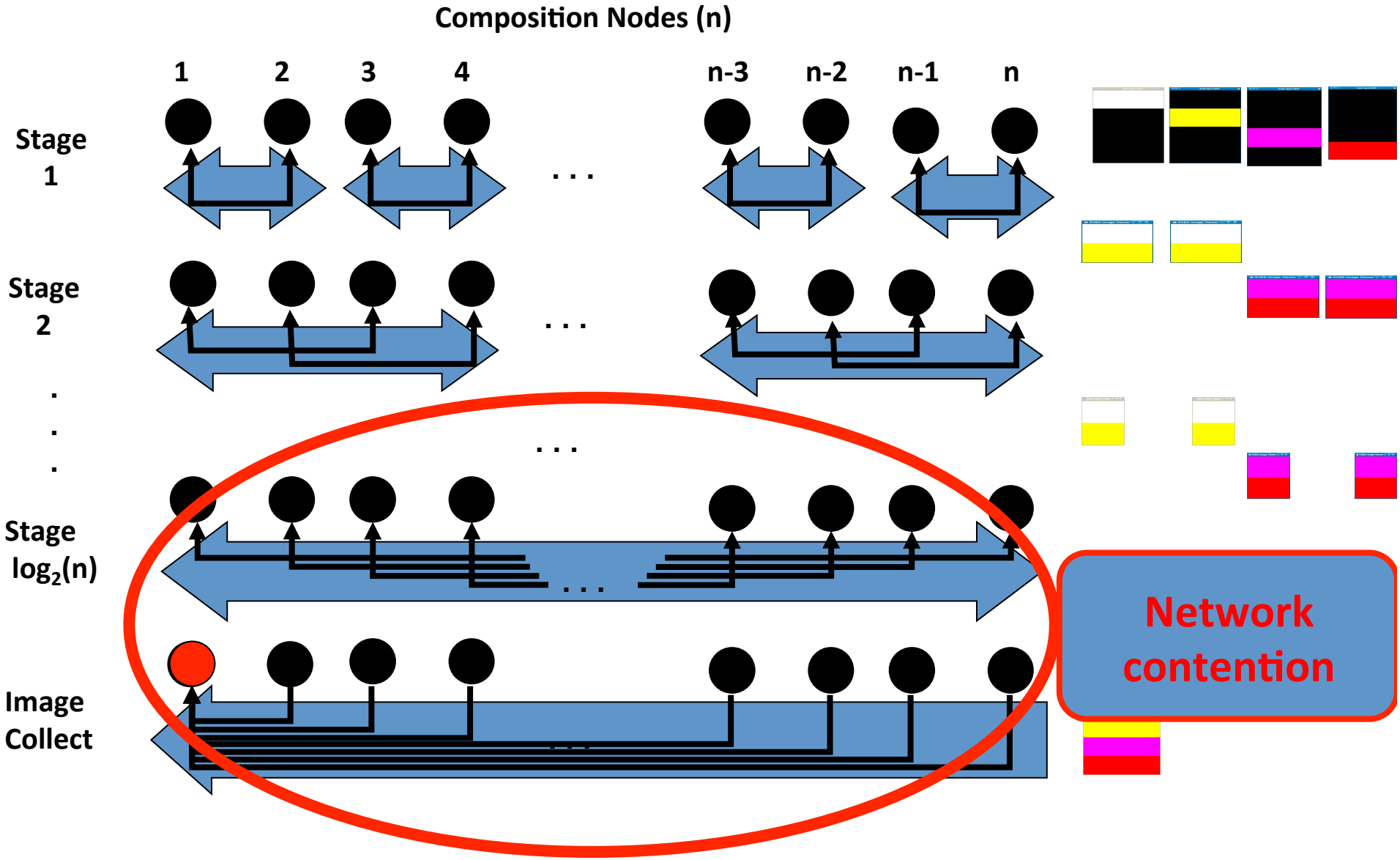
High speed Image Compositing



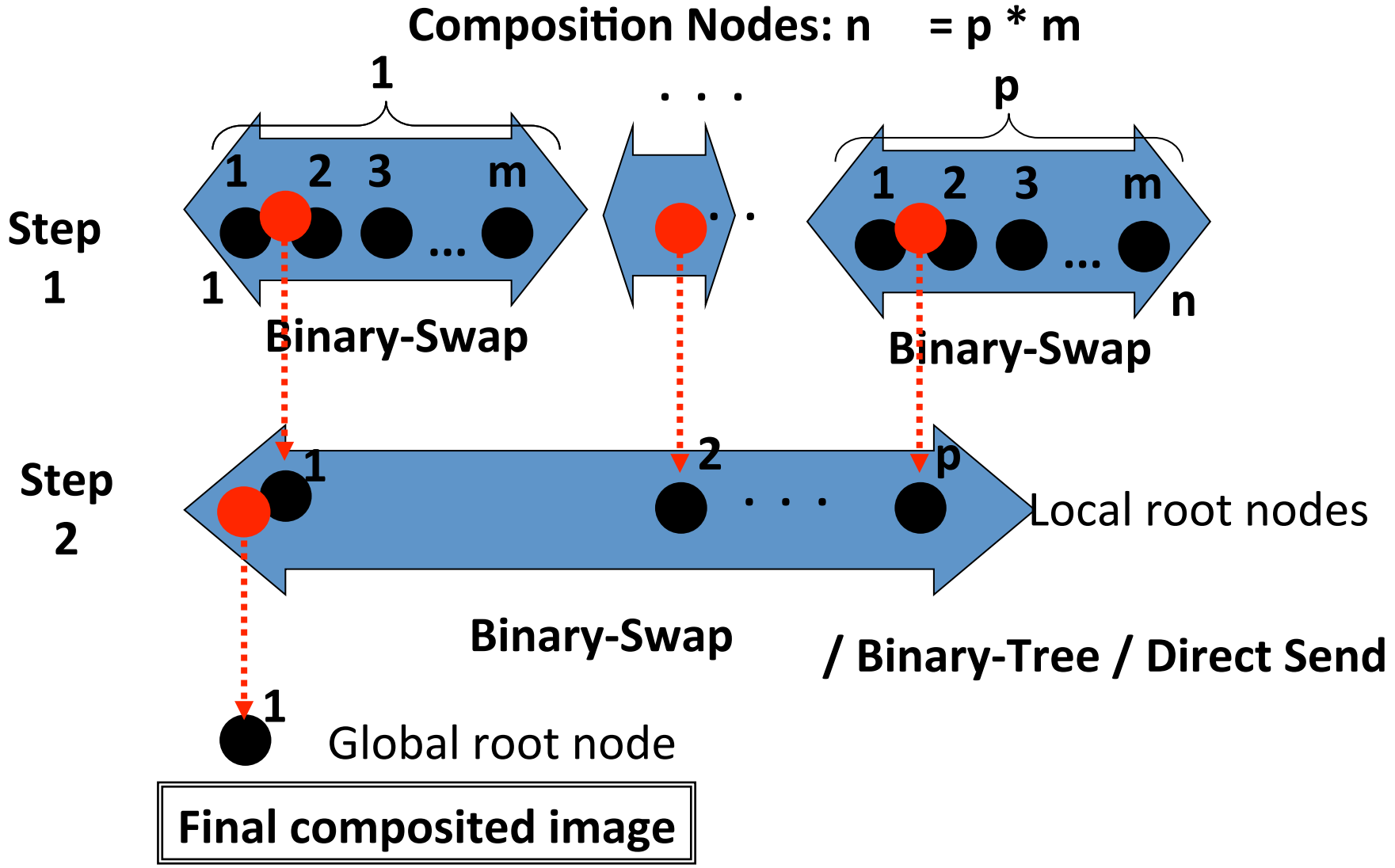
Large-Scale Image Compositing



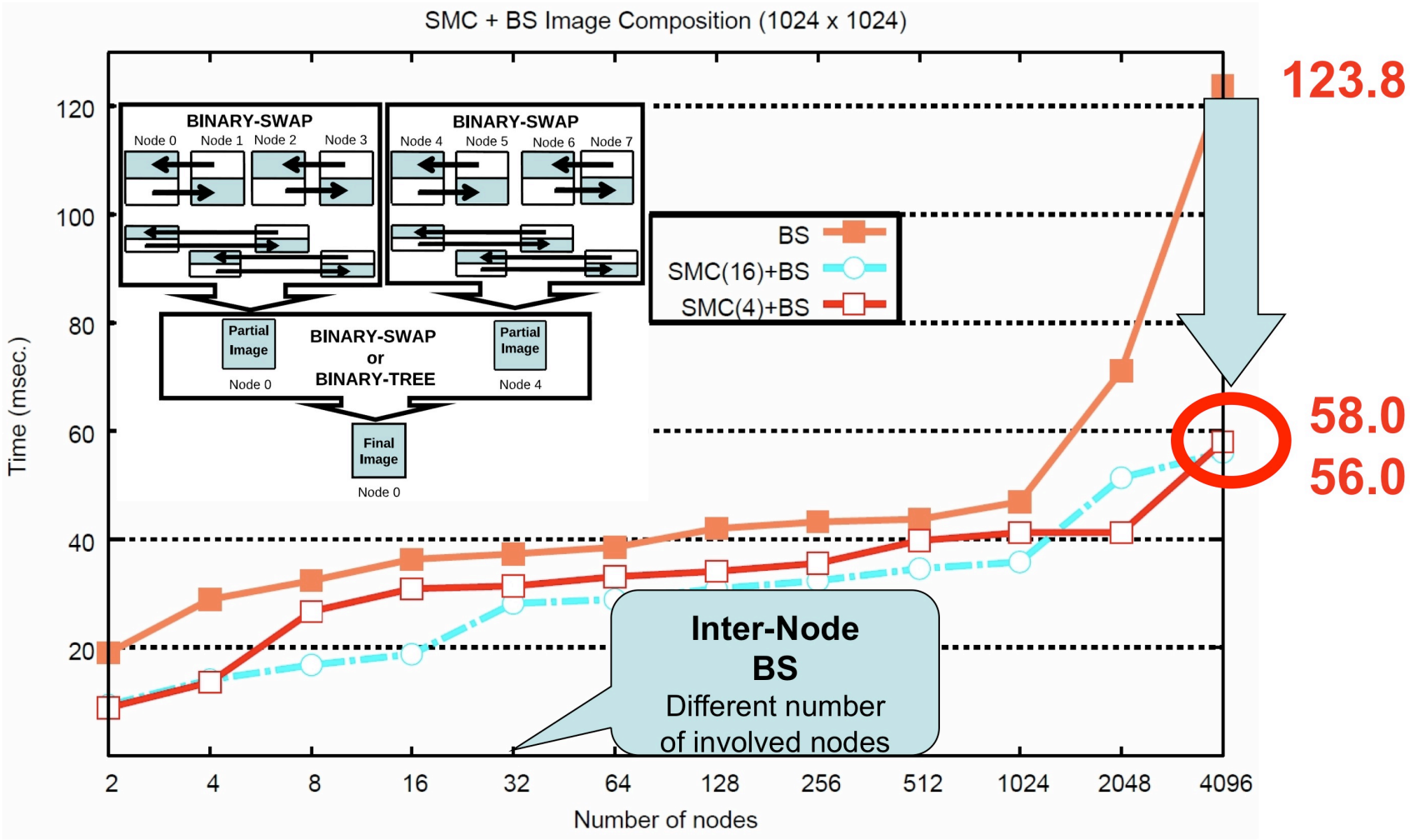
Binary-Swap Image Composition



Multi-Step Image Composition

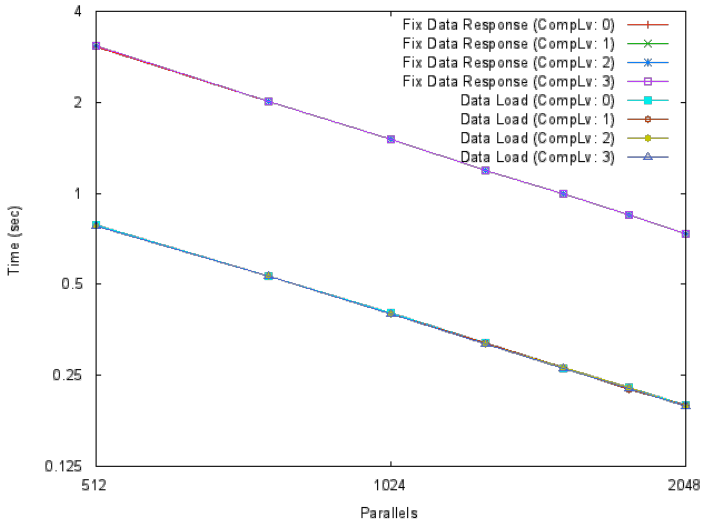


画像重畳の高速化


SMC + BS Image Composition (1024 x 1024)
Time (msec.)
Number of nodes
120
100
80
60
40
20
2
4
8
16
32
64
128
256
512
1024
2048
4096
BINARY-SWAP
Node 0
Node 1
Node 2
Node 3
BINARY-SWAP
Node 4
Node 5
Node 6
Node 7
Partial Image
Node 0
BINARY-SWAP
or
BINARY-TREE
Partial Image
Node 4
Final Image
Node 0
BS
SMC(16)+BS
SMC(4)+BS
123.8
58.0
56.0
Inter-Node
BS
Different number
of involved nodes

進行中の開発

- 京の上で稼働させる
  - OpenGLライブラリの代わりに、GLESライブラリを独自実装
  - AICS 可視化技術研究チームで継続開発
- MPIの課題
  - 3792ノード並列以上でMPI_GatherVでエラー
  - メモリ使用量、バッファ
- データロード性能改善
  - スケーラビリティ向上
- 適用範囲の拡大
  - 分子可視化
  - 10^{12}個オーダーの粒子可視化
- InSitu可視化開発
  - 京での直接可視化の技術開発



大規模ファイルのハンドリング

- 並列計算
  - ファイル入出力性能を高めるためには、並列入出力が必要
    - MPI-IO、高レベルIOライブラリ（netCDF, HDF5, ADIOS）
    - ステージングなども考慮せねば…
    - アーキが複雑になっても対応
    - 低レベルはCSのライブラリに任せて…
- アプリケーション寄りのライブラリ
  - 多数のファイルを管理
  - シミュレーションの利用方法に合わせた機能
  - できるだけ、プリミティブに

主要な機能

- 分散ファイル管理
  - 各プロセスが個別に入出力ファイル
  - 複数のファイルを管理するメタ情報
  - 複数ファイルをディレクトリで管理（各ステップ毎など）
  - エンディアンフリー
- Ｍ x Nデータロード
  - Mプロセスの出力ファイルをNプロセスにロードする
- リファインメントデータロード
  - 粗い格子で計算した結果を1段細かい格子（1:2）にマッピングする
- ステージング対応機能
  - スクリプトで記述できない場合の汎用的な対応
- 将来的にはストリーミング対応フォーマットへの拡張
  - インタラクティブ可視化用途
  - LOD or プログレッシブなデータ処理対応

ファイルの種類

- メタファイルの記述はTextParser形式，データはバイナリ
- 入力ファイル
  - hoge.tp
- ポリゴンファイル
  - ascii/binaryのSTLファイル
- 格子ファイル
  - ffvcが出力，binary
- メタファイル
  - メタ情報記述 index.dfi, proc.dfi

メタファイル概要

- メタ情報を記述したファイルは、以下の2種類のファイルから構成
  - インデクスファイル [index.dfi]
    - 速度や圧力など、変数毎に異なるメタ情報
    - FileInfo, FilePath情報
    - procファイルパス
    - 時系列情報ファイル
      - ✓ 出力ステップ，時刻の基本情報と最小値/最大値などの付加情報
      - ✓ Unit, TimeSlice
  - プロセス情報ファイル [proc.dfi]
    - 計算領域や計算時の並列数や計算ノードの情報を記述したファイル
    - Domain, MPI, Process
    - 共通

index.dfi

```
// ファイル情報
FileInfo {
  DirectoryPath = “./”       // ファイルの存在するディレクトリ
                                (DFFファイルからの相対パス)
  Prefix        = "vel"      // ベースファイル名       ※1
  FileFormat    = "bov"      // ファイルタイプ、拡張子  ※1
  GuideCell     = 0
  DataType      = "Float32”  // データタイプ           ※2
  Endian        = "little"   // データのエンディアン    ※3
  ArrayShape    = "ijkn"     // 配列形状               ※4
  Component     = 3          // 成分数(スカラーは1)    ※4
}

// ファイルパス情報
FilePath {
  Process      = “proc.dfi”
}
```

```
※1 ファイル名
[Prefix]_[ステップ番号:10桁]_id[RankID:6桁].[Extension]

※2 Int8, UInt8, Int16, UInt16, Int32, Uint32, Int64,
UInt64, Float32, Float64

※3 little, big, 省略時:実行プラットフォームと同じ

※4 ijkn, nijk
    ijkn:(imax,jmax,kmax,Component)
    nijk:(Component,imax,jmax,kmax)
```

```
// 単位系 必要に応じて追加
Unit {
  Length   = ”M"   //(NonDimensional, m, cm, mm)
  L0       = 1.0   // 規格化に用いた長さスケール
  Velocity = ”m/s" //(NonDimensional, m/s)
  V0       = 3.4   // 代表速度 (m/s)
  Pressure = “Pa”  // (NonDimensional, Pa)
  P0       = 0.0   // 基準圧力(Pa)
  DiffPrs  = 510.0 // 圧力差(Pa)
  Temperatur = “C” // (NonDimensional, C, K)
  BaseTemp = 10.0  // 指定単位
  DiffTemp = 35.0  // 指定単位
}
```

index.dfi (contd.)

```
// 時系列情報
TimeSlice {
  Slice[@] { // ファイル出力回数分
    Step = 0
    Time = 0.0
    MinMax[@] {          // Component個
      Min  = -1.56e-2
      Max  = 8.2e-01
    }
    ... 任意のアノテーションが追加可能
 }
  ...
  Slice[@] {
    Step = 1000
    Time = 10.0
    MinMax[@] {          // Component個
      Min  = -8.5e-1
      Max  = 9.1e+01
    }
    ... 任意のアノテーションが追加可能
 }
}
```

proc.dfi

```
// ドメイン情報
Domain {
 GlobalOrigin   = (-3.00, -3.00, -3.00) // 計算空間の起点座標
 GlobalRegion   = ( 6.00,  6.00,  6.00) // 計算空間の各軸方向の長さ
 GlobalVoxel    = (64, 64, 64)          // 計算領域全体のボクセル数
 GlobalDivision = (1, 1, 1)             // 計算領域の部分領域の分割数
}

// 並列情報
MPI {
  NumberOfRank  = 128  // プロセス数
  NumberOfGroup = 1    // グループ数
}

// 各プロセスの情報
Process {
  Rank[@] {
    ID         = 0                  // ファイル出力時のランク番号
    Hostname   = “Iridium.local”    // 計算ノードのホスト名
    VoxelSize = (64, 64, 64)        // 各領域のボクセルサイズ
    HeadIndex = (1, 1, 1)           // 各領域の開始インデクス
    TailIndex = (64, 64, 64)        // 各領域の終了インデクス
  }
  ...
}
```

ホスト名は必須では無い

ファイルフォーマット

```
?Formatting?
  “ijkn”
  “nijk”
 n; 任意
```

- VisItのBOV(Brick of Volume)形式
- 配列を並び順で出力
- ベクトルやテンソルは，スカラーx3, x9
- ファイル名 :　prs_0000002000_id000005.bov

prefix　*time stamp*　*rank*

```
/* Example C code */
float data[NZ][NY][NX][component];
FILE *fp = fopen(“bov.values”, “wb”);
fwrite((void *)data, sizeof(float), NX*NY*NZ*component, fp);
fclose(fp);
```

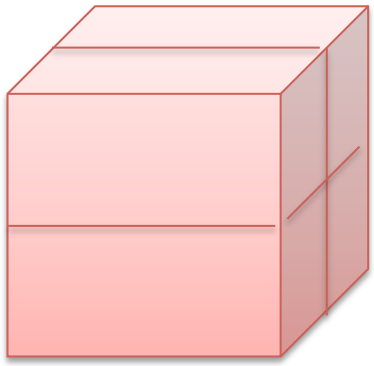
メタデータ

```
MPI {
  NumberOfRank  = 1
  NumberOfGroup = 1
  MyRankID  = 0
  MyGroupID = 0
}

FileInfo {
  DirectoryPath = “./”
  Prefix = "prs”
  GlobalVoxel = (64, 64, 64)
  GlobalDivision = (1, 1, 1)
  FileFormat = ”bov”
  GuideCell = 0
}

NodeInfo {
  Node[@] {
    RankID = 0
    HostName = "Strontium.local"
    VoxelSize = (64, 64, 64)
    HeadIndex = (1, 1, 1)
    TailIndex = (64, 64, 64)
  }
}
```

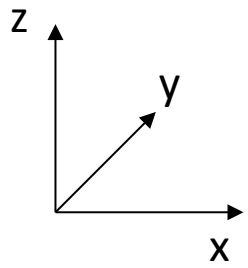
```
TimeSlice {
  Slice[@] {
    Step = 0
    Min = -1.56e-2
    Max = 8.2e-01
  }

...

  Slice[@] {
    Step = 100
    Min = 2.24
    Max = 6.3e02
  }
}
```

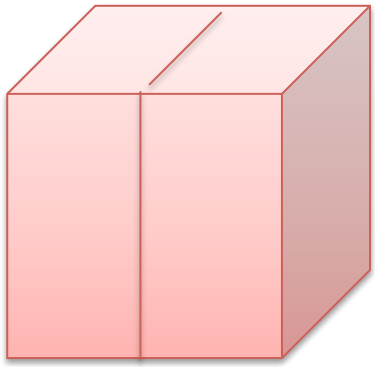
Prefix, Step, Rank情報からファイル名を特定

M x N Loading

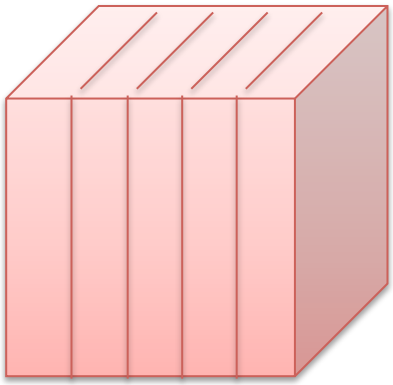


4並列実行時の領域分割
（前計算時の分割）

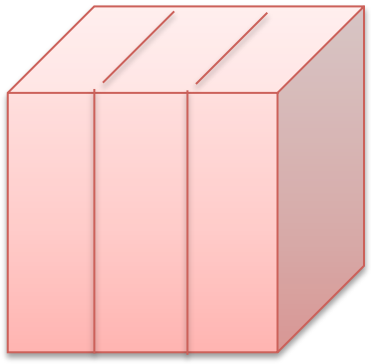




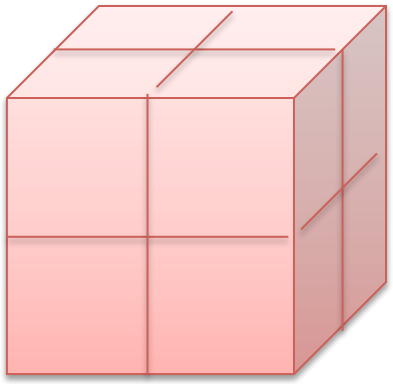
2並列リスタート時の領域分割



5並列リスタート時の領域分割



3並列リスタート時の領域分割



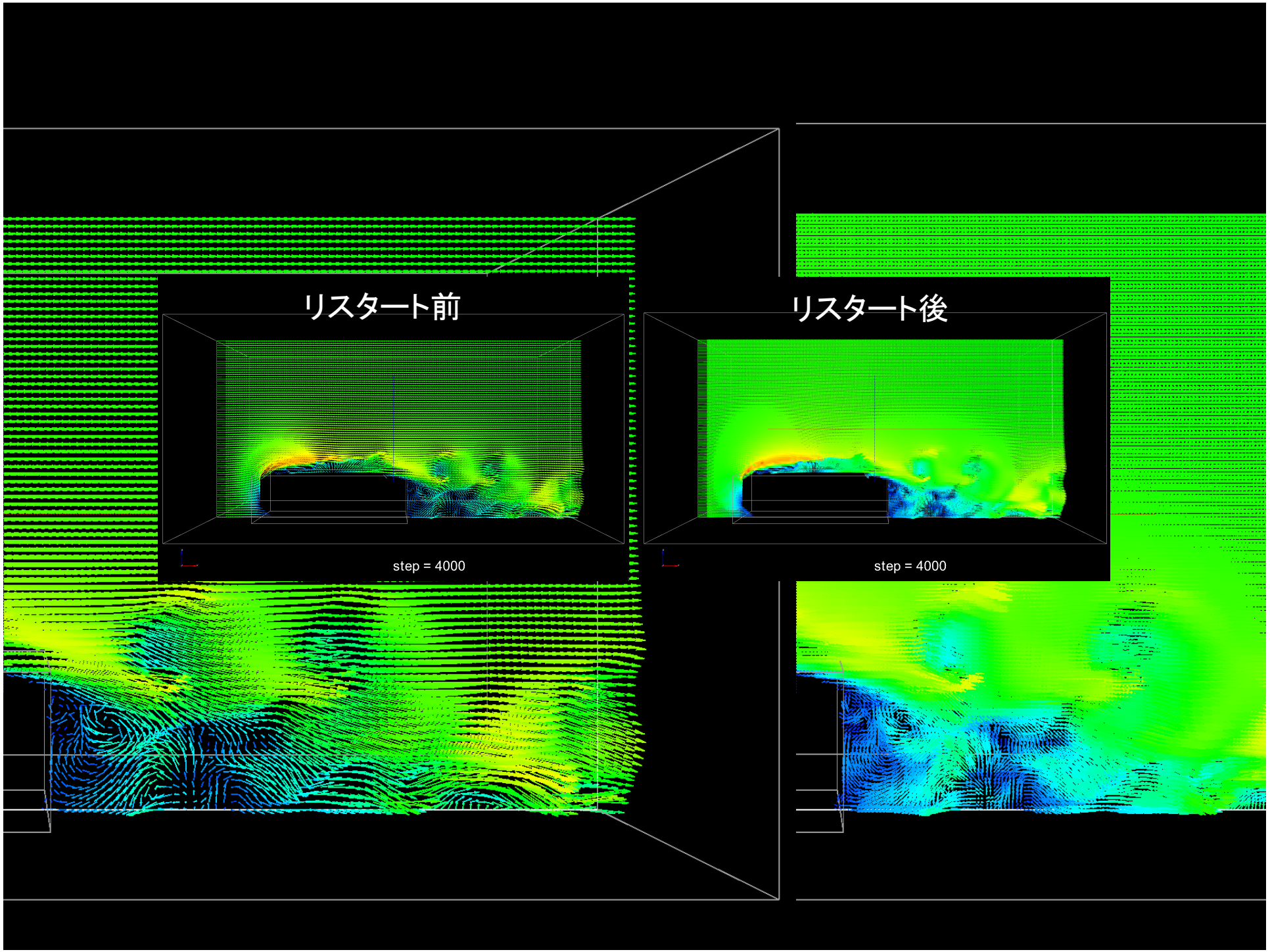
8並列リスタート時の領域分割

Refinement Loading

- 短い計算時間で目的の評価を実施するために利用する．初期値依存性の高いものは適用外．

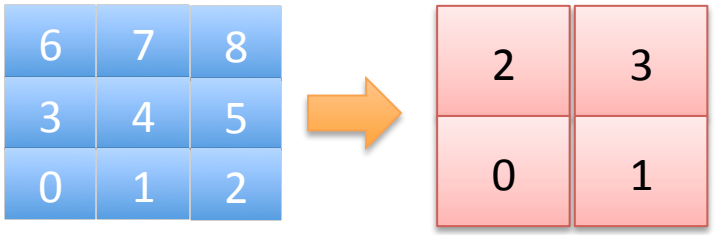
- 粗い格子で計算した結果を，リスタート時に細かい格子にマッピングする
- 粗い格子で大きなタイムステップで時刻をすすめ，現象を発達させる
- 準定常的になったら，細かい格子に移して詳細な計算を実施する

- 多様な分割パターンに対応した内挿処理


リスタート前
step = 4000
リスタート後
step = 4000

ステージング

- Mプロセスの計算
  - M個の出力ファイルをステージアウト
- Nプロセスで計算再開
  - どのランクにどのファイルが必要か？

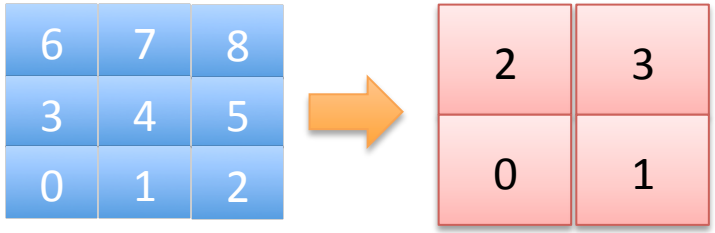


M　　N

N{0} << M{0, 1, 3, 4}

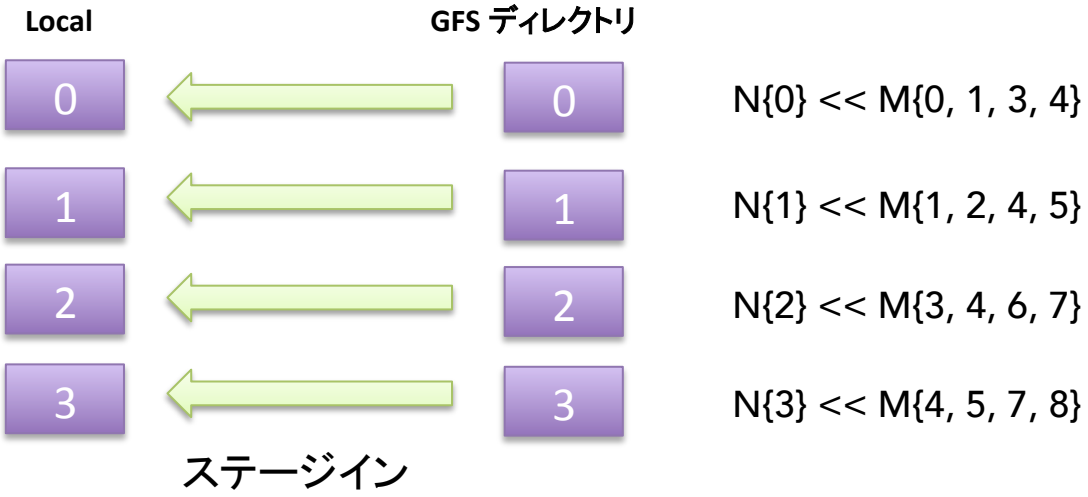
簡単なものはジョブスクリプトで書ける？
複雑なパターンは書けない

予め，Nの分割パターンを決めておいて，必要なファイルをロードするディレクトリに配置する



M　　N

事前に分割するアルゴリズムと実行時に自動分割するアルゴリズムは同じであること



ステージイン

```
#PJM --stgin  "rank=*  ./pm_1.tp  %r:./"
```

ステージングスクリプトは環境依存
どこでも実行できる仕組みが必要

今日話したこと

- 大規模なデータの可視化について、技術的な課題と研究開発方針についての話
- ポスト処理＝可視化+データ処理
- 計算科学と計算機科学の融合分野
- 京での可視化について
- ファイルハンドリングについて