汎用グラフ処理モデル GIM-V の複数GPUによる大規模計算とデータ転送の最適化

白幡 晃一[*1], 佐藤 仁[*1,*3], 鈴村 豊太郎[*1,*2,*3], 松岡 聡[*1,*3,*4]

[*1] 東京工業大学

[*2] IBM 東京基礎研究所

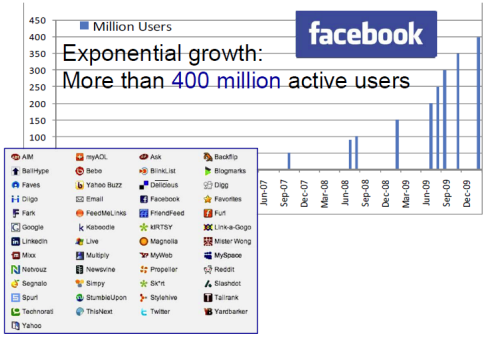
[*3] 科学技術振興機構

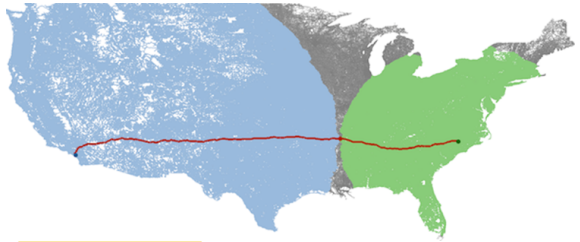
[*4] 国立情報学研究所

GPGPU を用いた大規模グラフ処理

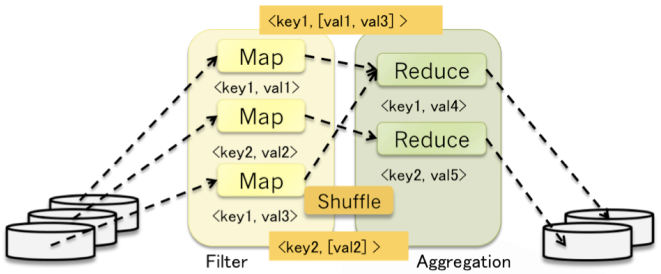
- 大規模グラフの出現
  - 幅広い応用例
    ソーシャルネットワークに対する解析，道路ネットワークの経路探索，スマートグリッド，創薬，遺伝子解析
  - 数百万頂点～数兆頂点，数億枝～数百兆枝からなる超大規模なグラフ解析
    - ex) Facebook ユーザ 8 億人以上，平均130人の友人
      → 8 億頂点，1000 億枝
    - ex2) 全米道路ネットワーク → 2400万頂点，5800万枝

  → 大規模グラフの高速処理が必要





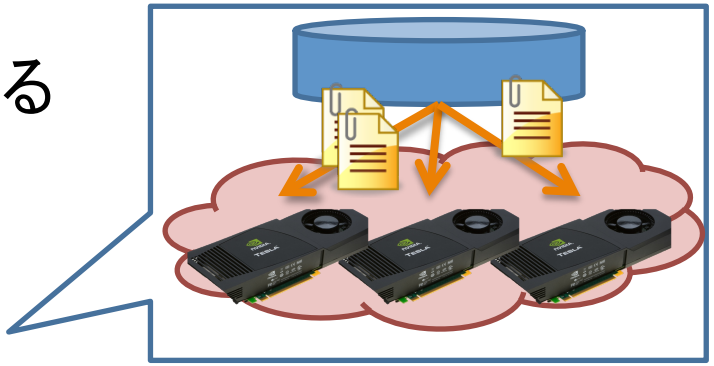
- 高速大規模グラフ処理手法
  - MapReduce
    - ペタバイト級の大規模データ処理
    - 自動的なメモリ階層の管理
    - ストリームによるリアルタイム処理
  - GPGPU
    - スーパーコンピュータやクラウドに搭載
    - コア数，メモリバンド幅の活用による高速処理

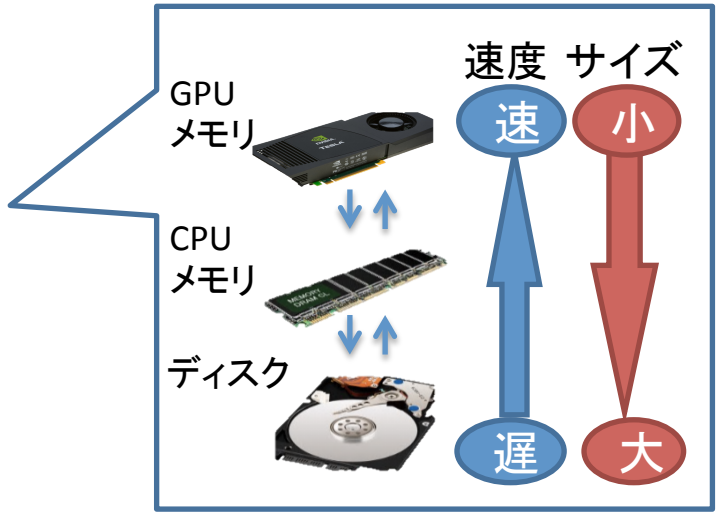


→ GPGPU を用いた MapReduce 型大規模グラフ 処理の高速化

GPGPU を用いた大規模グラフ処理における問題点

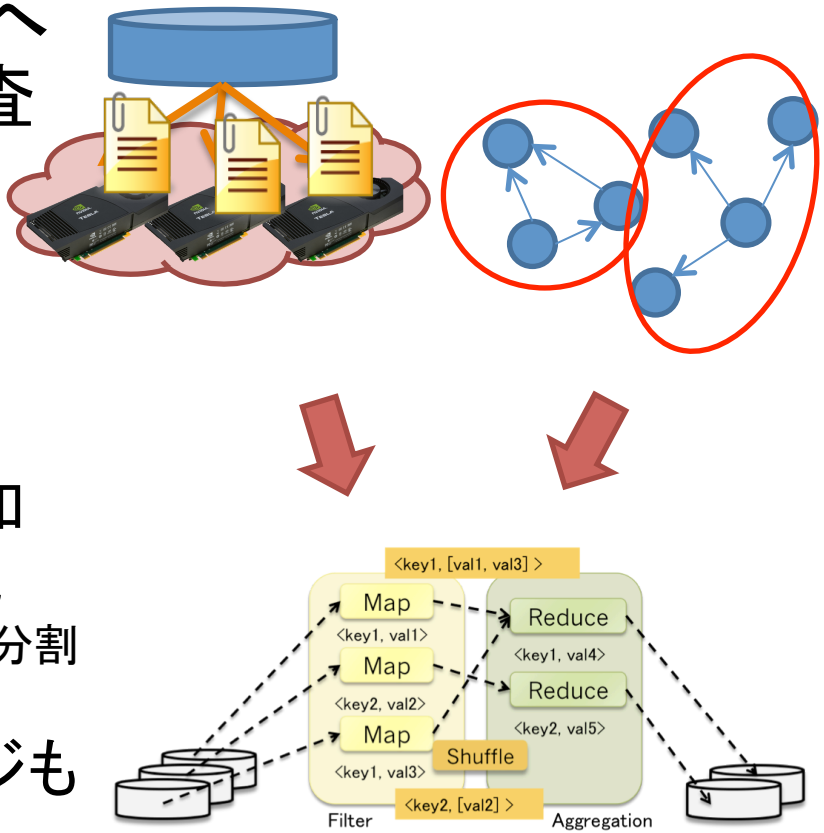
- MapReduce 型汎用グラフ処理モデルへの GPU の適用
  - GPU の使用によりどの程度高速化できるか明らかでない
    - データ転送オーバーヘッド等の要因
- 超大規模グラフへの対応
  - 複数 GPU 化
    - CPU-GPU 間，ノード間の通信による遅延
      - 通信量の削減が必要
  - メモリあふれ
    - GPU は CPU に比べメモリ容量が少ない
      - ex) TSUBAME2.0 (GPU 3GB, CPU 54GB)
      - CPU メモリやローカルストレージの活用
      - 効率的なメモリ階層の管理





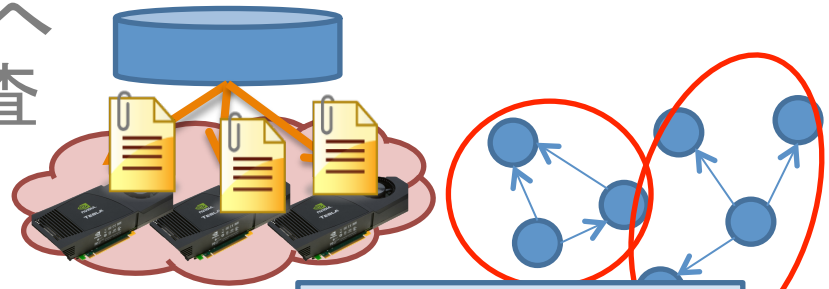
解決法: 複数GPUによる大規模化と，グラフ分割によるデータ転送量の削減

- MapReduce 型汎用処理モデルへの GPU の適用による効果を調査
  - 既存実装との比較
    - 既存の CPU 実装
    - MapReduce 型でない最適化実装
- 超大規模グラフへの対応方法
  - 複数 GPU 化によりメモリ量を増加
    - データの割り振りによる通信量を削減
      - 解法の一つとして，前処理としてグラフ分割
      - → Shuffle 処理での転送量を抑制
  - メモリではないローカルストレージも活用
    - データをファイルシステムから次々に読み GPU へデータを転送
    - 最適なデータ配置のスケジューリング

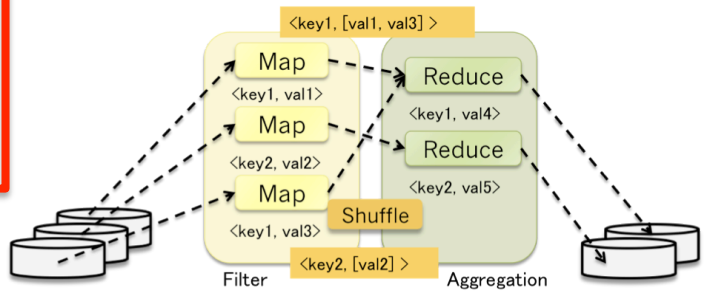


解決法: 複数GPUによる大規模化と，グラフ分割によるデータ転送量の削減

- MapReduce 型汎用処理モデルへの GPU の適用による効果を調査
  - 既存実装との比較
    - 既存の CPU 実装
    - MapReduce 型でない最適化実装
- **超大規模グラフへの対応方法**
  - **複数 GPU 化によりメモリ量を増加**
    - **データの割り振りによる通信量を削減**
      - **解法の一つとして，前処理としてグラフ分割**
      - **→ Shuffle 処理での転送量を抑制**
  - メモリではないローカルストレージも活用
    - データをファイルシステムから次々に読み GPU へデータを転送
    - 最適なデータ配置のスケジューリング



本発表



目的と成果

- 目的

  大規模グラフ処理への複数 GPU 化とデータ転送量削減の有効性の検証

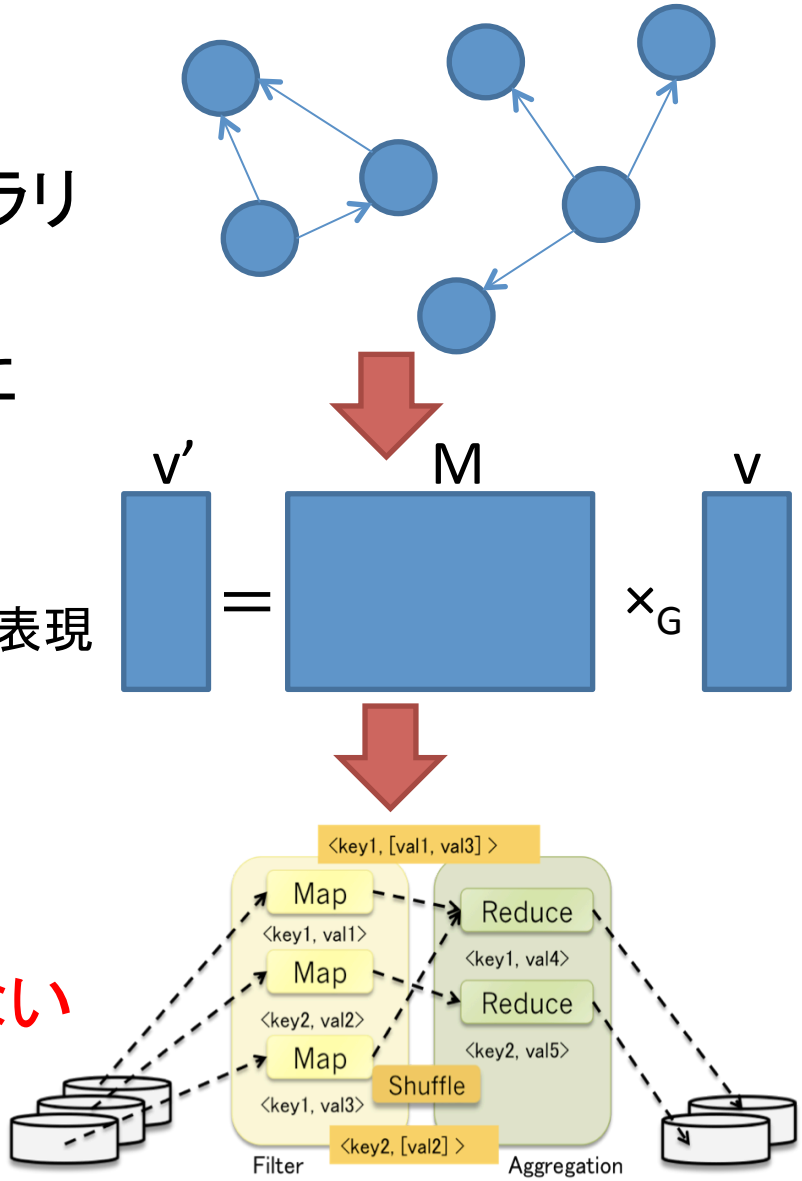
- 成果
  - MapReduce 型汎用グラフ処理モデルの複数 GPU による高速化
    - Map は GPU により 7.2 倍の高速化
    - Sort, Reduce は性能改善の余地あり
  - メモリ内にデータが収まる場合の，グラフ分割の有効性
    - グラフ分割を行った場合にデータ転送量を 54 % 削減
    - GPU 毎の負荷の不均衡によりグラフ処理の実行時間が増加

背景

既存の大規模グラフ処理システム

- PEGASUS
  - Hadoop ベースのグラフライブラリ
  - GIM-V (Generalized Iterated Matrix-Vector multiplication) による計算
    - 反復行列ベクトル積の一般化
      - 頂点をベクトル，枝を隣接行列で表現
    - 典型的なグラフ処理を記述可能
    - MapReduce により記述可能

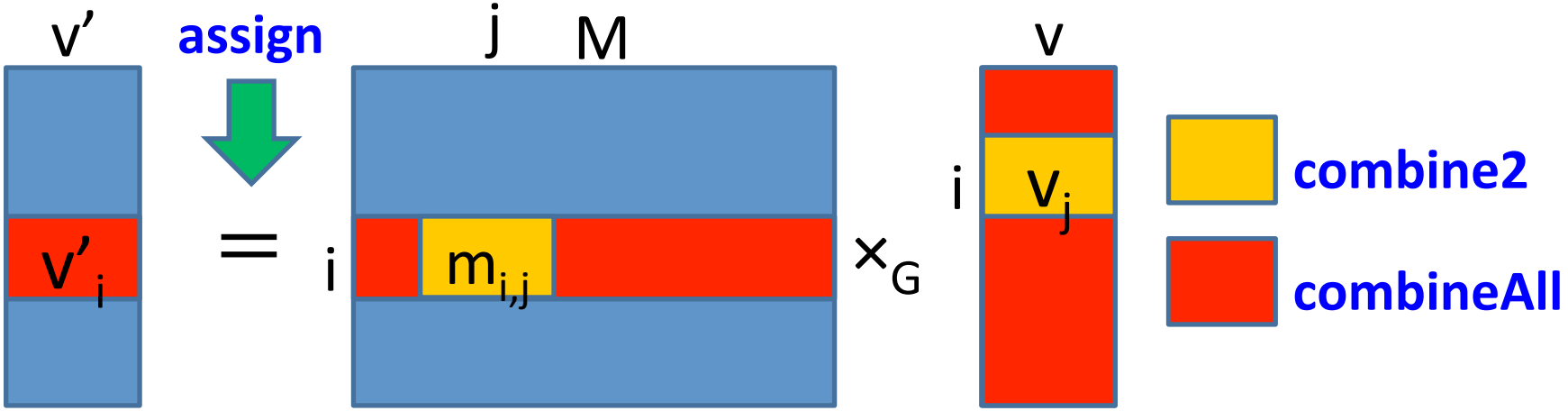
→ GPU の使用によりどれほど高速化できるかは定かではない



大規模グラフ処理モデル GIM-V

- Generalized Iterative Matrix-Vector multiplication[*1]
  - 反復行列ベクトル積の一般化
  - $v' = M \times_G v$ where
    $v'_i = \mathrm{assign}(v_j, \mathrm{combineAll}_j(\{x_j \mid j = 1..n, x_j = \mathrm{combine2}(m_{i,j}, v_j)\}))$
    $(i = 1..n)$
    - 上記3つの関数を実装することで，様々なグラフ処理を記述可能
  - 演算は収束するまで反復して実行
  - GIM-V を 2 ステージの MapReduce により記述可能

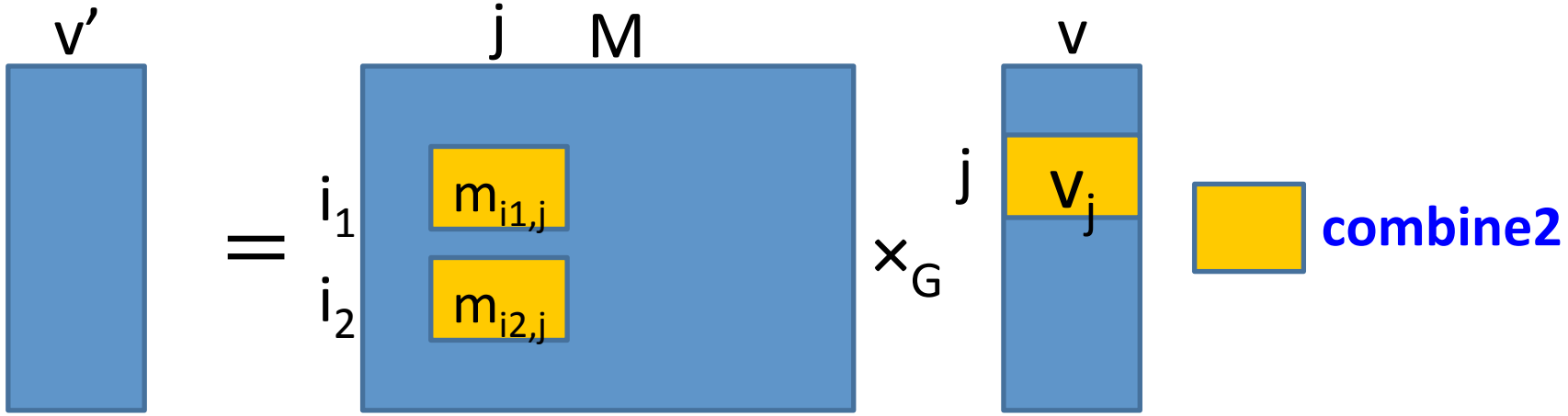
→ 既存の GPU 向け MapReduce フレームワーク(Mars)上に実装



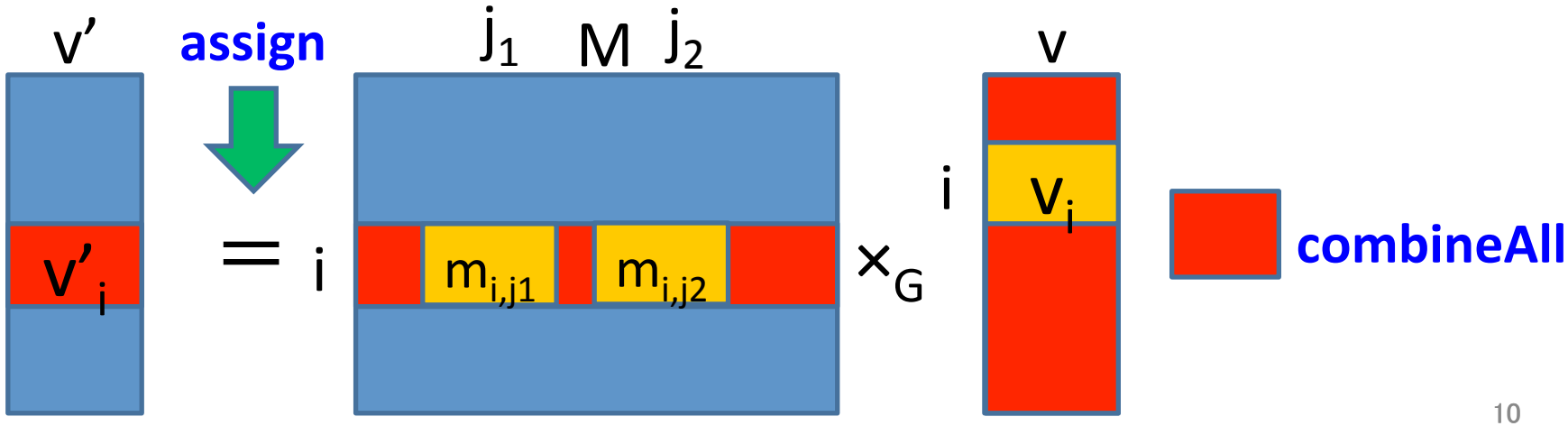
*1 : Kang, U. et al, “PEGASUS: A Peta-Scale Graph Mining System- Implementation and Observations”, IEEE INTERNATIONAL CONFERENCE ON DATA MINING 2009

MapReduce による GIM-V 計算

- MapReduce-1



- MapReduce-2



問題: ノード間データ転送量の増加

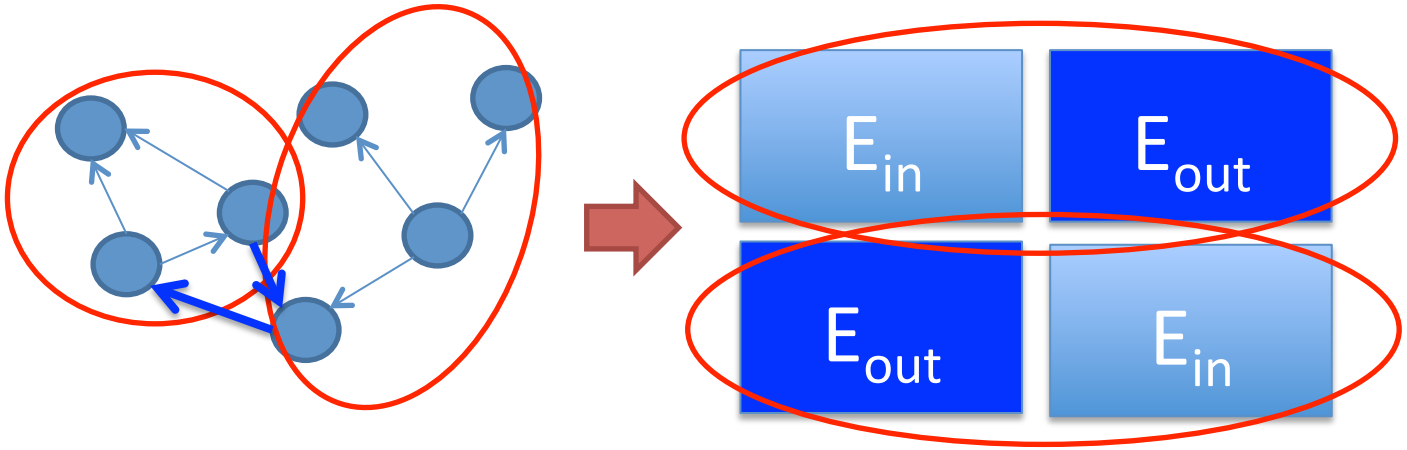
グラフが大規模化するほどノード間転送が増加

- 大規模グラフを大規模計算環境を用いて処理

→ 大規模になるほどデータ転送量が増加

- グラフ処理ではノード間をまたぐ枝の数 (エッジカット)の大きさに依存

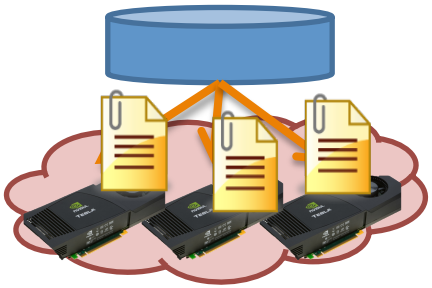
→ グラフ分割を行い，エッジカット削減による最適化

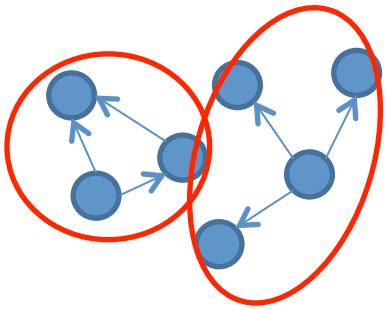


提案

提案: 複数 GPU による大規模化と，グラフ分割によるデータ転送量の削減

- 複数 GPU の使用によりメモリ量を増加
  - 既存の GPU 向け MapReduce 実装である Mars を MPI により複数 GPU 向けに改良
    - 複数 MapReduce ステージの連結
    - Shuffle における全対全通信の実装
    - 収束判定
  - 複数 GPU 向け GIM-V への CPU-GPU 間通信の実装
    - ノード間通信における CPU-GPU 通信の実装
    - 反復処理への CPU-GPU 間通信の実装
- グラフ分割による性能最適化
  - クラスタごとにグラフを分割し，エッジカットの最小化によりノード間通信量を削減
  - GIM-V の前処理として，GPU の数に分割
    OSS である METIS ライブラリを使用

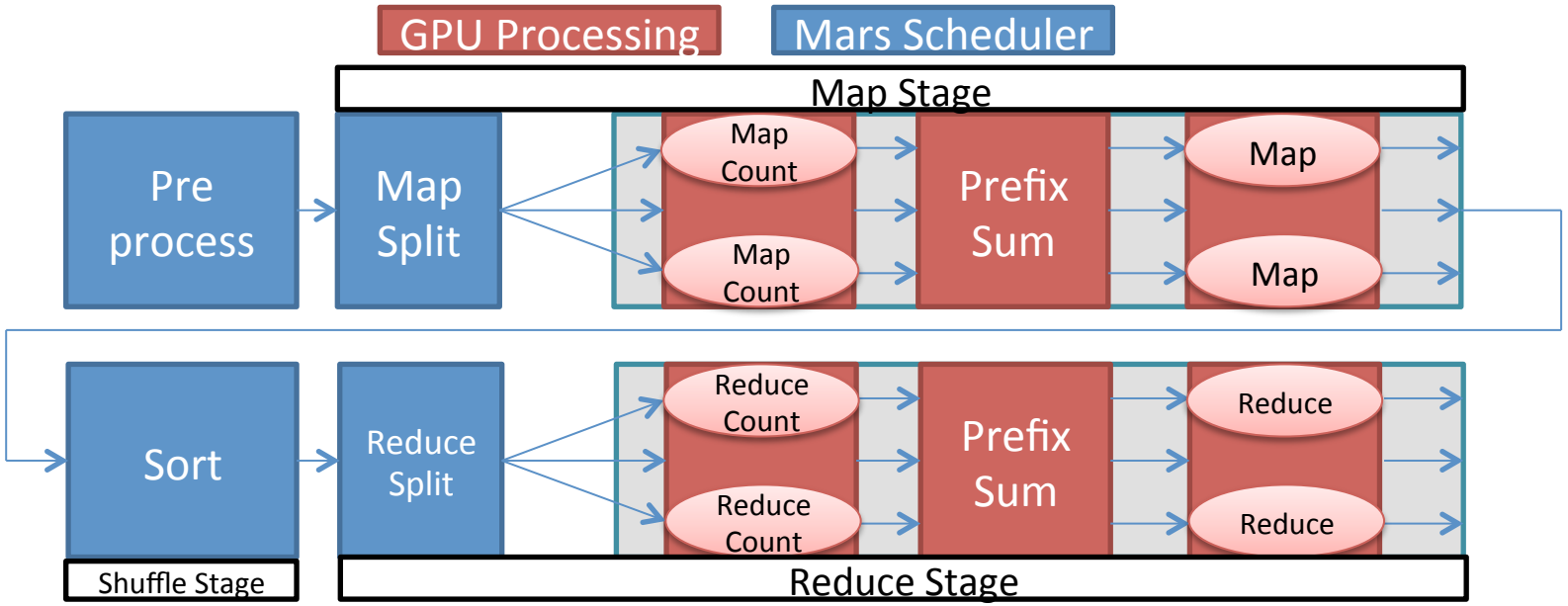




Mars の構造

- Mars*1 : 既存の GPU 用 MapReduce フレームワーク
  - Map, Reduce 関数を CUDA カーネルとして実装
    - GPU のスレッド単位で Mapper, Reducer を呼び出し
    - Map/Reduce Count → Prefix sum → Map/Reduce の順に実行
  - Shuffle フェーズは GPU ベースの Bitonic Sort を実行
  - Map 開始時に CPU-GPU 間の通信

→ Mars を複数 GPU による GIM-V 処理向けに改良

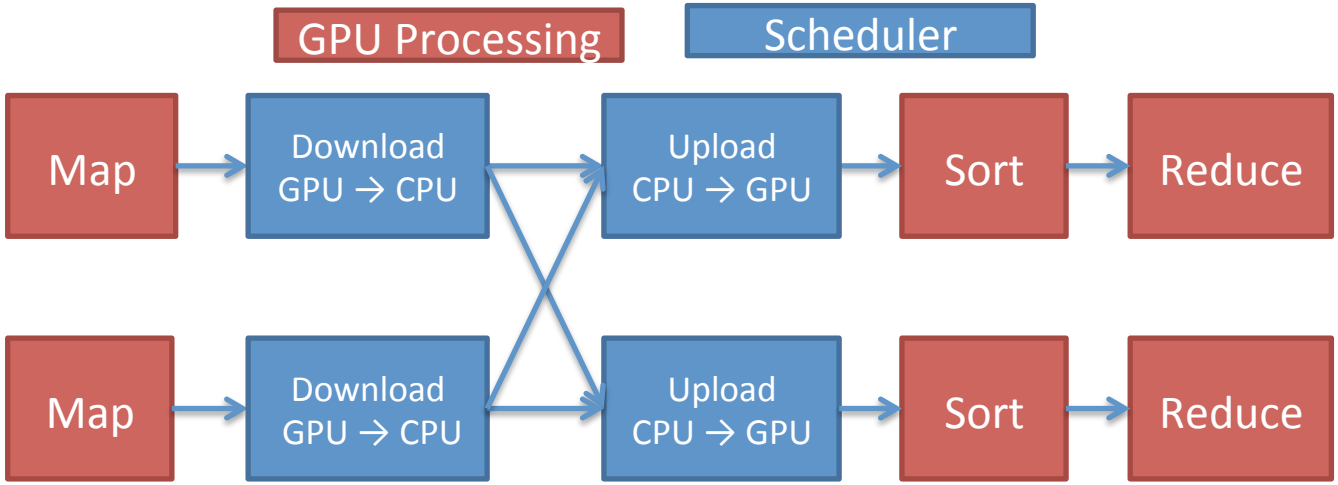


*1 : Fang W. et al, “Mars: Accelerating MapReduce with Graphics Processors”, Parallel and Distributed Systems, 2011

複数 GPU 上への GIM-V の実装

Mars 上への MPI による GIM-V のマルチノード実装

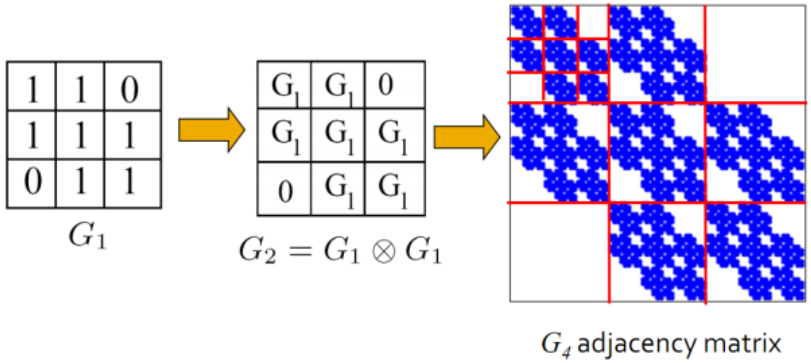
- 複数の MapReduce ステージを連続して実行
  - 反復の開始時と終了時に CPU-GPU 間の通信
  - 後処理で収束判定の計算
- Shuffle でノード間通信
  - MPI_Alltoallv 関数により全対全通信
    CPU-GPU 間の通信が発生
  - 各ノードがローカルソート
- 収束判定
  - 各ノードがローカルに収束した頂点数をカウント
  - MPI_Allreduce 関数により各ノードの収束した頂点数の合計を計算



グラフ分割による性能最適化

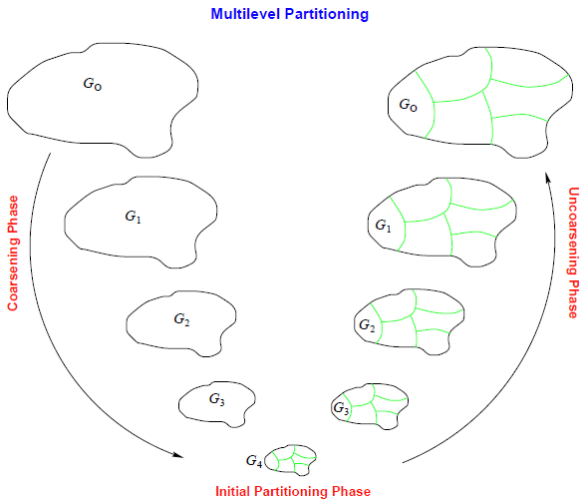
- 前処理としてグラフ分割
  - クロネッカーグラフ
    - SNS などの現実世界のグラフの代表的なモデル
    - スケールフリー性

      再帰的に大きなクラスタが現れる性質



  - クラスタごとにグラフを分割し，エッジカットの最小化によりノード間データ通信量を削減
  - グラフ分割アルゴリズム

    Multilevel k-way partitioning *1
    - G = (V, E) に対し，O(|E|) の時間計算量
    - メッセージパッシングを用いる並列アルゴリズムも存在

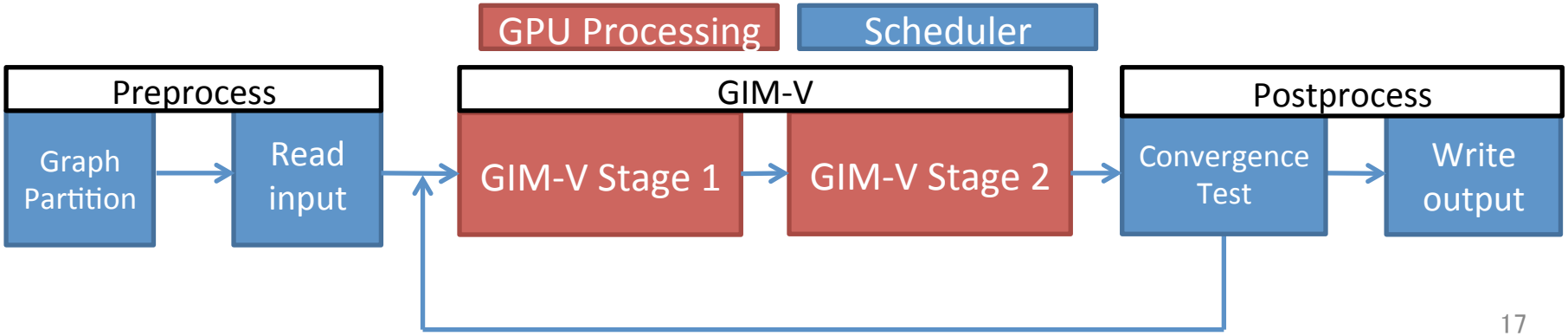


*1 : George Karypis and Vipin Kumar, “Multilevel k-way Partitioning Scheme for Irregular Graphs”, Journal of Parallel and Distributed Computing, 1998

グラフ分割の実装

- OSS である METIS を使用
  - GPU の数に一致させて分割
  - 各 GPU は自分が担当する頂点番号データも保持

    Shuffle 時の通信は，各 GPU が持つ頂点番号データに一致させて頂点と枝を割り振り
- 全ノードが並列で入力ファイルの読み込み
  - グラフ分割の結果を各 GPU が読み込み
  - （グラフ分割を行わない場合）MPI-IO による並列読み込み



評価

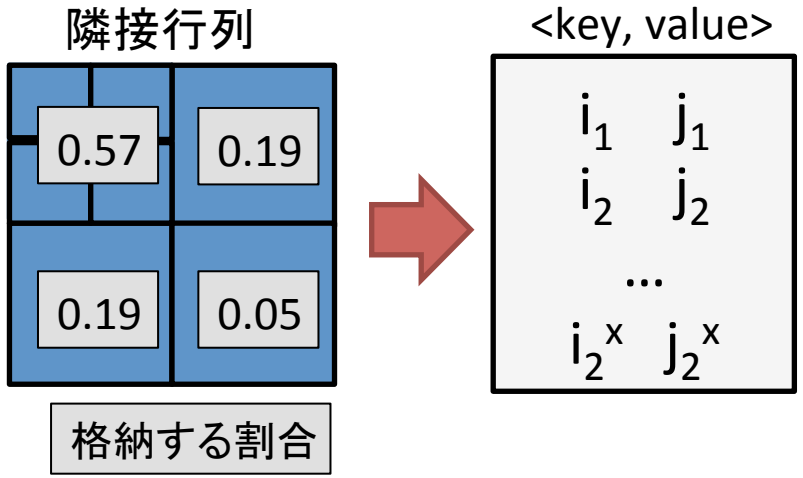
実験

- 検証事項
  - 複数 GPU による GIM-V 処理の高速化
  - グラフ分割によるデータ転送量削減の有効性
- 測定内容

  グラフ処理アプリの反復処理 1 ラウンドの平均時間を測定
  - 複数 GPU によるスケーラビリティの測定
  - CPU 版 Mars との比較
  - グラフ分割の有無による比較
- 方法
  - PageRank アプリケーション

    ウェブページの重要度を決定
  - 入力データ
    - クロネッカーグラフを人工的に生成

      Graph 500 の Generator により生成
    - パラメータ
      - SCALE: 頂点数の 2 の対数
      - 辺の数 = 頂点数 × 16



実験環境

- TSUBAME2.0
  - CPU 6 コア x 2 ソケット，24 スレッド (HyperThread オン)，GPU 3台
  - GPU
    - CUDA Driver Version: 4.0
    - CUDA Runtime Version: 4.0
    - Compute Capability: 2.0
    - 共有/L1 キャッシュサイズ: 64 KB

	CPU	GPU
種類	Intel® Xeon® X5670	Tesla M2050
物理コア数	12	448
周波数	2.93 GHz	1.15 GHz
メモリ	54 GB	2.7 GB (Global)
コンパイラ	gcc 4.3.4	nvcc 3.2

- Mars
  - Mars GPU
    - 1ノード当たり 1GPUで実行
    - スレッド数は異なる key の個数に等しい
      - スレッドブロック当たり 256 スレッド
  - Mars CPU
    - 1 ノード当たり 1コアで実行
    - CUDA カーネルの代わりに C 言語で実装
    - Sort はクイックソートを実装
- METIS
  - CPU 1 コアを使用して実行

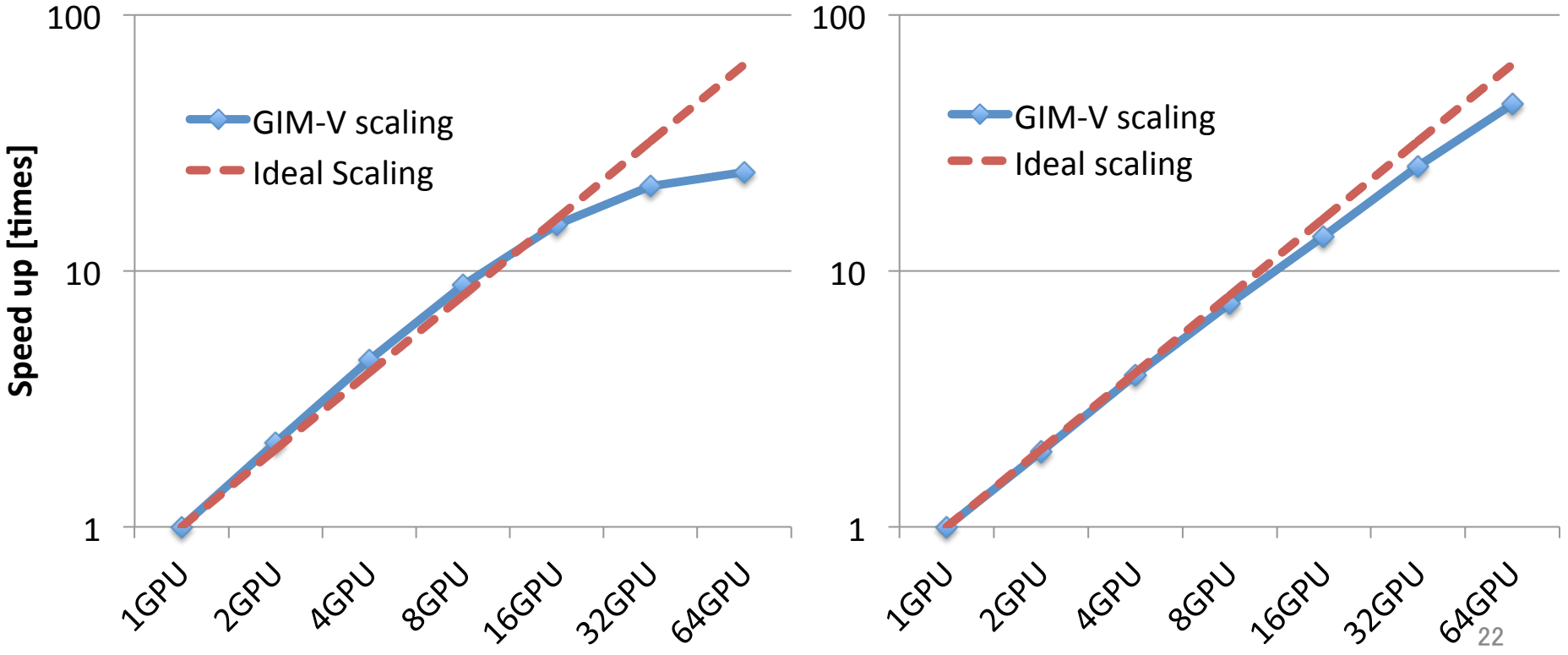
複数 GPU の使用による GIM-V 処理の高速化の検証

複数 GPU によるスケーラビリティ

- ストロングスケールを測定
  SCALE=19 に固定
- ウィークスケールを測定
  1 GPU 当たり SCALE=18

32 GPU 以降はスケールしていない
・データ転送時間の増加
・Reduce の実行時間が一定

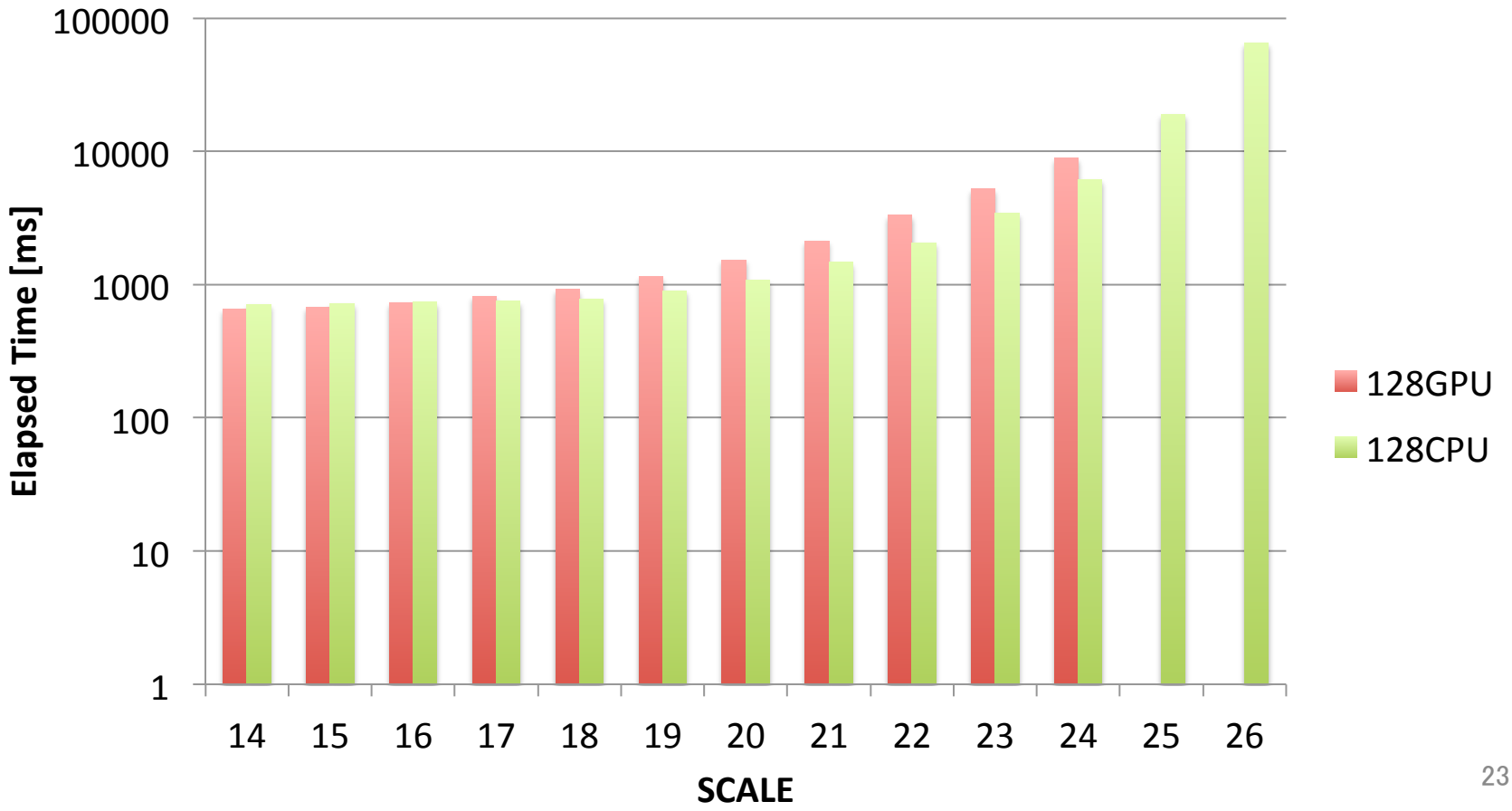
良好なスケール



Mars GPU と Mars CPU：実行時間の比較

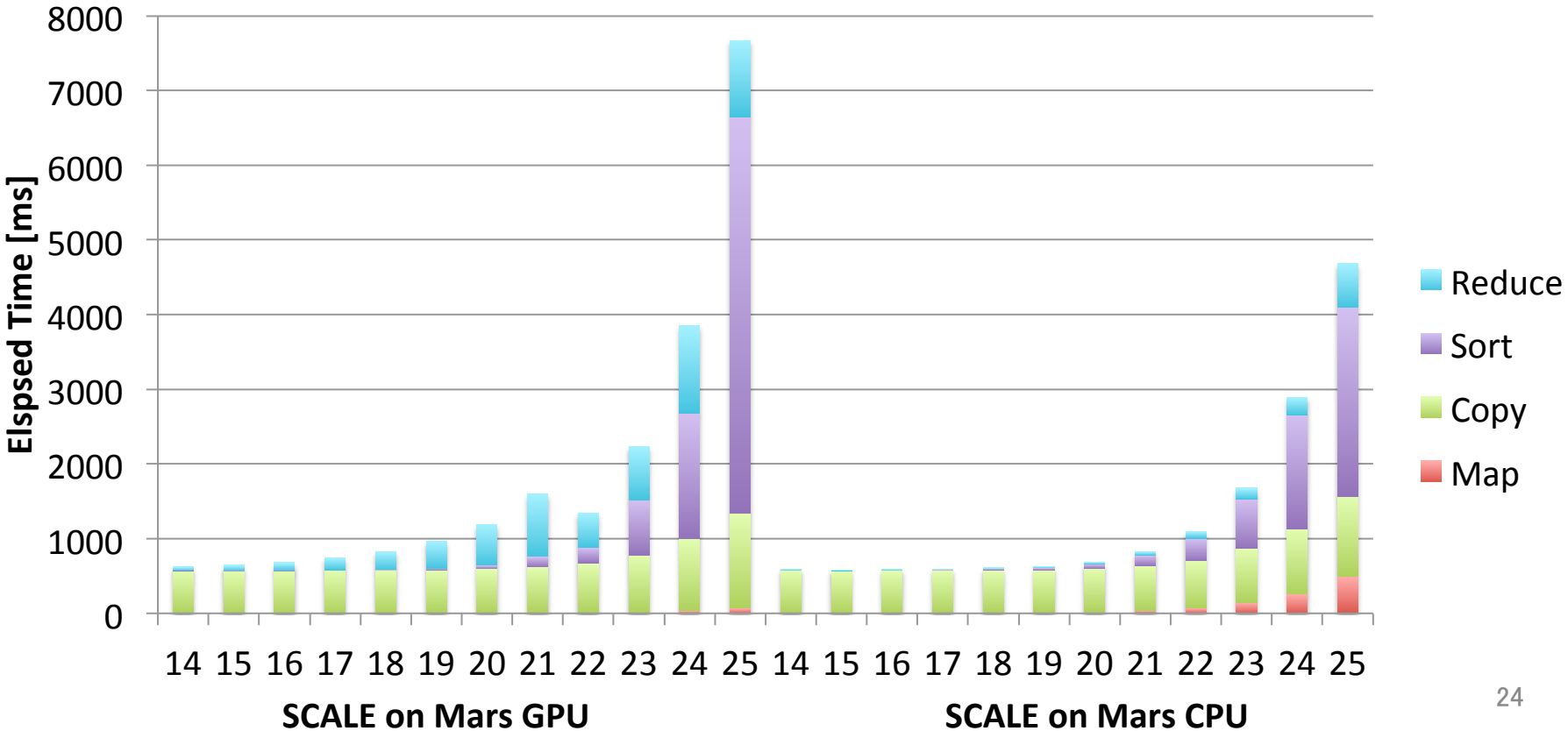
- Marsに対して，Map, Sort, Reduce を CPU で実行する場合を比較
  128 ノードを使用

GPU の使用による優位性は見えていない
→ Sort, Reduce が GPU を使用しても加速されていないため



Mars GPU と Mars CPU: 実行時間の内訳

- 高速化されている部分とされていない部分がある
  - SCALE=25 で Map フェーズは 7.2 倍高速
  - Sort, Reduce は高速化されていない
  - Map のみ GPU を使用した場合，SCALE=25 で 1.12 倍の高速化
- Sort の実装を改善する必要性
  - Mars の Sort 実装の改良
  - Radix Sort 等のアルゴリズムの使用

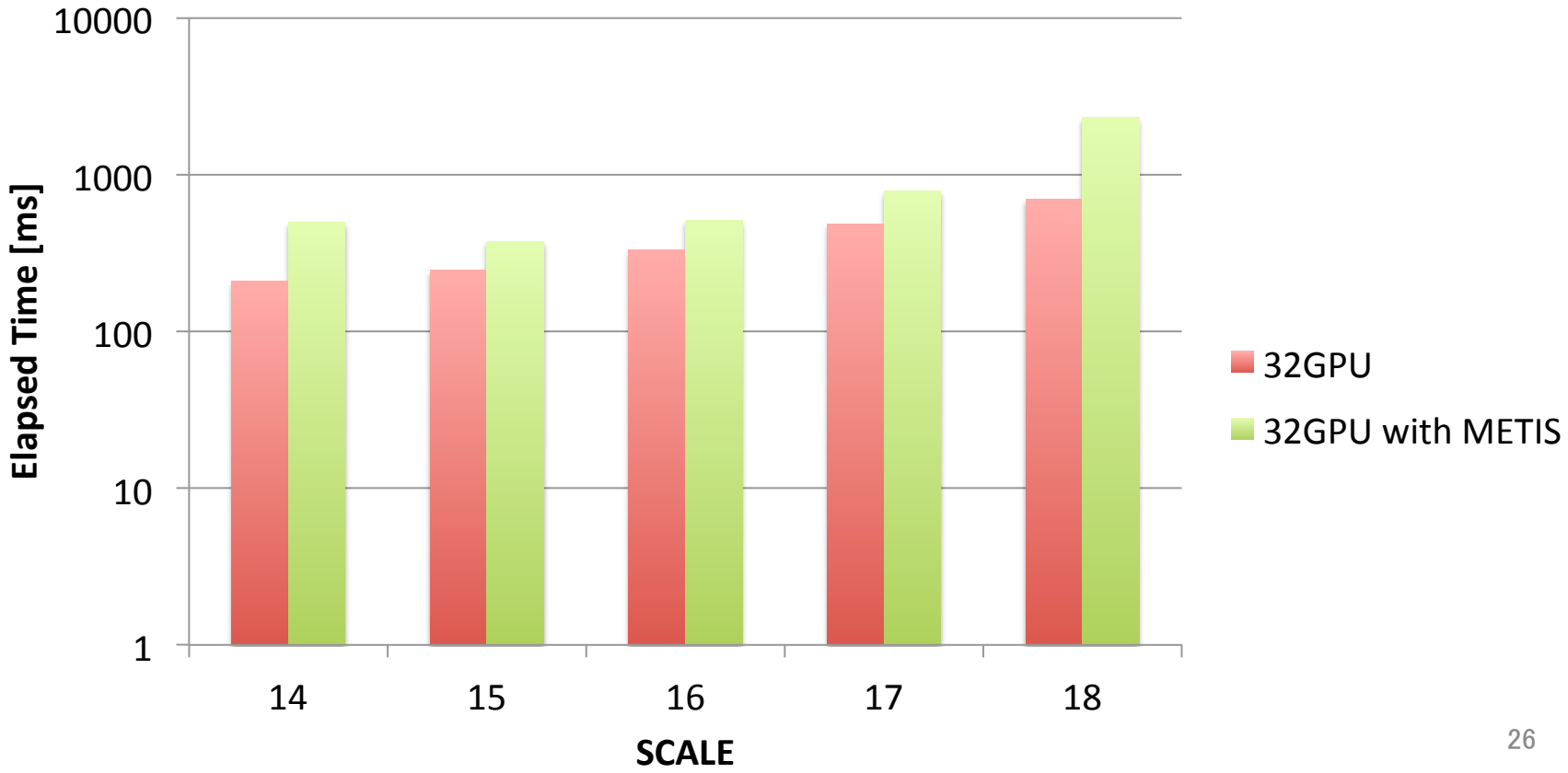


グラフ分割によるデータ転送量削減による効果の検証

グラフ分割の有無による GIM-V の比較

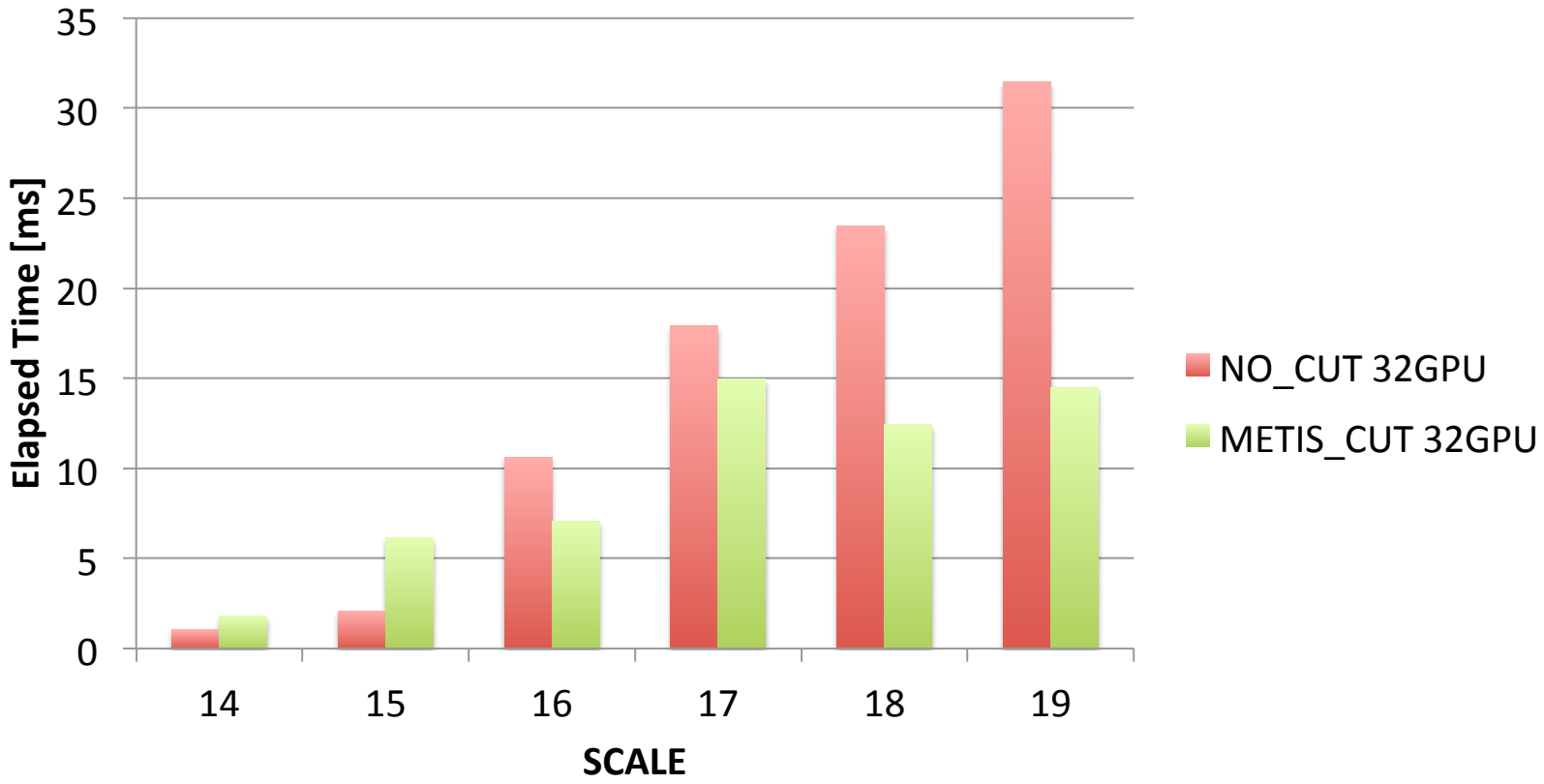
- 前処理としてグラフ分割を行う場合と行わない場合を比較
  - 32 GPU を使用

グラフ分割により実行時間が増加
→ GPU 毎の負荷が不均衡になり，同期を取る際に負荷が大きい GPU の実行時間に律速されるため



Shuffle における MPI によるデータ転送時間

- MPI_Alltoallv 関数によるデータ転送部分の時間を測定
  - グラフ分割により一定の効果
    最大 54 % 削減
  - データ転送時間の全体に占める割合が小さい為，全体の実行時間には影響を与えていない
    - TSUBAME2.0 ではネットワークバンド幅が大きいため
    - しかし，将来的にはネットワーク性能がボトルネックになる可能性あり



実験のまとめ

- GIM-V の GPU による高速化
  - 複数 GPU の使用によるスケーラビリティ
    - ストロングスケール
      - 32 GPU 以上の場合にデータ転送量増加，Reduce の実行時間が一定
    - ウィークスケール
      - 64 GPU まで良好なスケールを確認
  - GPU と CPU の性能比較
    - 現状では Map のみを GPU で実行する場合が最適
      - SCALE=25 で 1.12 倍の高速化
      - Sort はアルゴリズムの変更により改良される可能性
      - Reduce は高速化される場合とされない場合がある
- メモリ内にデータが収まる場合の，グラフ分割の有効性
  - グラフ分割により GPU 毎の負荷が不均衡になる現象
  - 分割によりデータ転送時間は 54% 削減

→ GPU, CPU の性能や通信量等を考慮した動的な自動チューニングを行うことを検討

関連研究

- 既存の大規模グラフ処理システム
  - Pregel*[1] : Master/Worker モデルによるC++の実装
    Worker 毎に頂点を分担して計算
  - Parallel BGL*[2] : MPI ベースの C++ 並列グラフ処理ライブラリ
- GPU，MapReduce を用いたグラフ処理
  - GPU を用いて最短路問題を解くアルゴリズム*[3]
    幅優先探索，単一始点最短路問題を高速に計算
  - Graph500 *[4] のリファレンス実装に MapReduce を用いた最短路問題が公開される予定
- マルチ GPU 上，マルチノード上での MapReduce 実装
  - GPMR*[5] : 既存のマルチGPU上での MapReduce 実装
  - MapReduce-MPI*[6] : MPI を使用した MapReduce ライブラリ

→ グラフの性質を考慮した，GPU による効率的な MapReduce 処理

*1 : Malewicz, G. et al, “Pregel: A System for Large-Scale Graph Processing”, SIGMOD 2010.
*2 : Gregor, D. et al, “The parallel BGL: A Generic Library for Distributed Graph Computations”, POOSC 2005.
*3 : Harish, P. et al, “Accelerating large graph algorithms on the GPU using CUDA”, HiPC 2007.
*4 : David A. Bader et al, “The Graph 500 List”
*5 : Stuart, J.A. et al, “Multi-GPU MapReduce on GPU Clusters”, IPDPS 2011.
*6 : Plimpton, S.J. et al, “MapReduce in MPI for Large-scale Graph Algorithms”, Parallel Computing 2011.

まとめと今後の課題

- まとめ
  - 複数 GPU 上での GIM-V 処理の高速化
    - Map フェーズは GPU により 7.2 倍の高速化
    - Sort, Reduce フェーズについては性能改善の余地あり
  - GPU メモリ内にデータが収まる場合の，グラフ分割の有効性
    - グラフ分割を行った場合に Shuffle でのノード間通信量を 54% 削減
    - GPU 毎の負荷の不均衡により GIM-V 処理の実行時間は増加
- 今後の課題
  - 実装の最適化
    - Sort, Reduce フェーズの性能改善
    - グラフ分割の最適化
    - ノード内複数 GPU 対応
  - GPU メモリに載り切らない入力データへの対応
    - メモリではないローカルのストレージも活用
    - 効率的なメモリ階層の管理
    - 動的な自動チューニング