❖ 再構成可能な高性能並列計算のための PGAS プログラミング処理系 ❖

田浦研 M2 原健太朗

2011.2.15

❖ 序論

背景

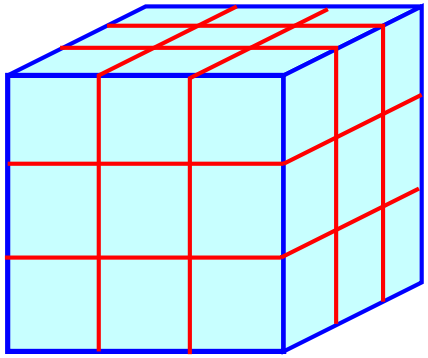
- アプリの多様化・高度化
  - → 気象予測
  - → Web グラフ解析
  - → 地震シミュレーション
  - → ...
- 並列分散環境の大規模化・コモディティ化
  - → スパコン
    - ♦ TSUBAME-2.0：1408 ノード，T2K：952 ノード
  - → クラウド
    - ♦ Amazon EC2：0.085 ドル/時間/OS
- 並列分散プログラミングはますます身近に
  - → 並列言語処理系への要請が多様化

並列言語処理系に対する要請

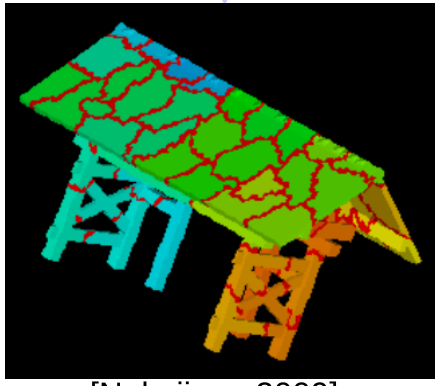
- 当然の要請
  - → 性能
  - → プログラマビリティ
- 多様な要請
  - → 耐故障
  - → 複雑なネットワーク構成のサポート
  - → 非定型な並列計算のサポート
  - → 再構成可能な並列計算のサポート
  - → …

要請 I：非定型な並列計算のサポート

定型的な領域分割
(各種ベンチマーク)



非定型な領域分割
(現実のアプリ)

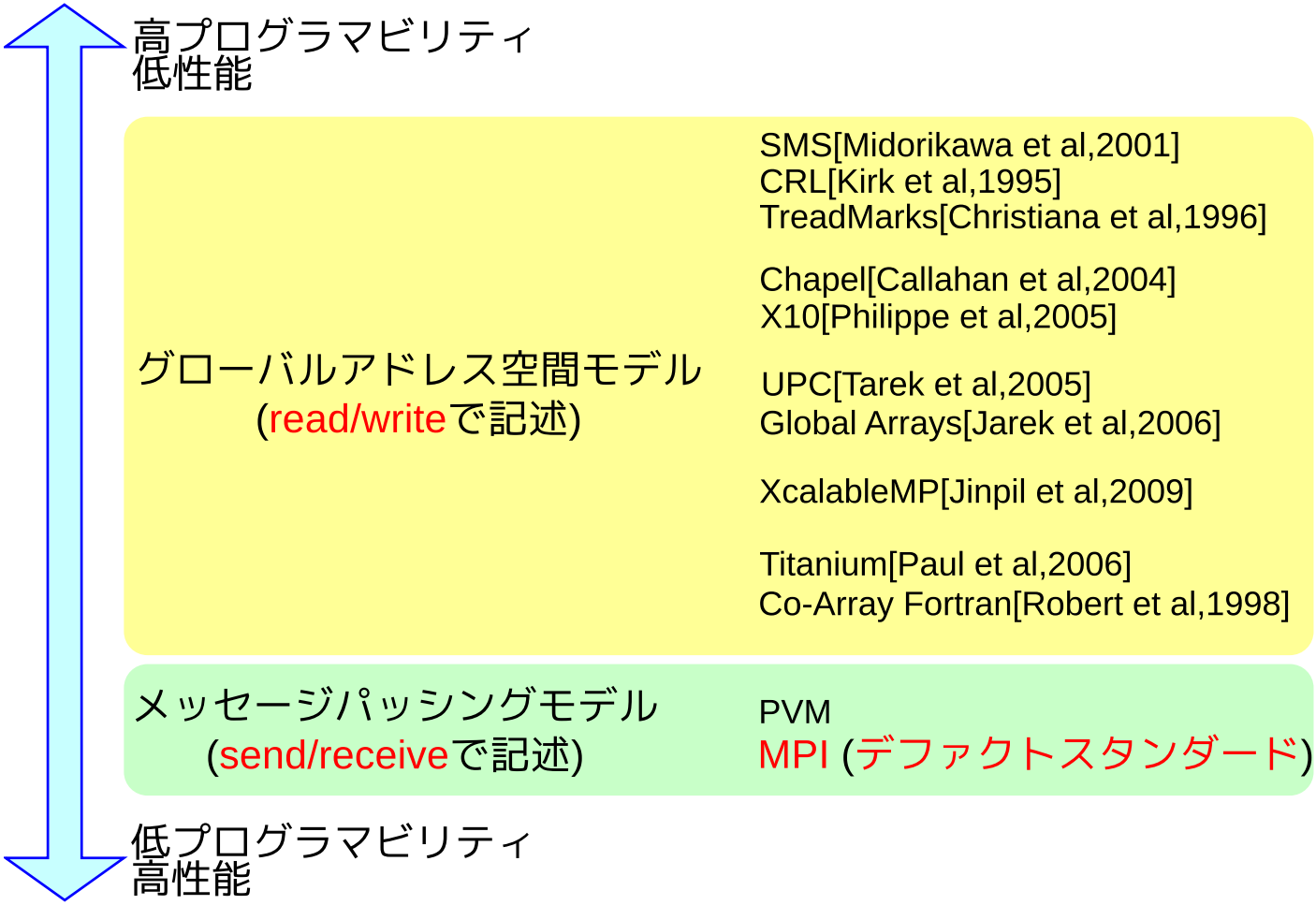


[Nakajima, 2009]

- **定型的な並列計算：**
  - → **多次元配列などを使えば，「簡単に書ける」かつ「性能良く実行できる」**
  - → **多くの言語処理系がサポート：**Co-Array Fortran，Titanium，Chapel，XcalableMP，Global Arrays...
- **非定型な並列計算：**
  - → **どういう API があれば，「簡単に書ける」かつ「性能良く実行できる」かは難しい**
  - → **十分な API を備えた言語処理系はない**

並列言語処理系の現状

- 問：非定型な高性能並列計算を**より良く**サポートするためには，どんな並列言語処理系が求められているのか?



- **性能とプログラマビリティのトレードオフが鍵**

どんな並列言語処理系が求められているのか?(1)

- 並列プログラム開発の流れ：
  (1)「プログラミング」：とりあえず正しく動くものを書く
  (2)「性能最適化」：速くする
  (3)「実行」：結果を得る
- ベストな言語処理系 $\neq$「実行」の時間を最短化できる言語処理系
- ベストな言語処理系 $=$「プログラミング」を開始してから結果を得るまでの「全体」の時間を最短化できる言語処理系

「プログラミングに3時間
実行に6時間」
な言語処理系

>

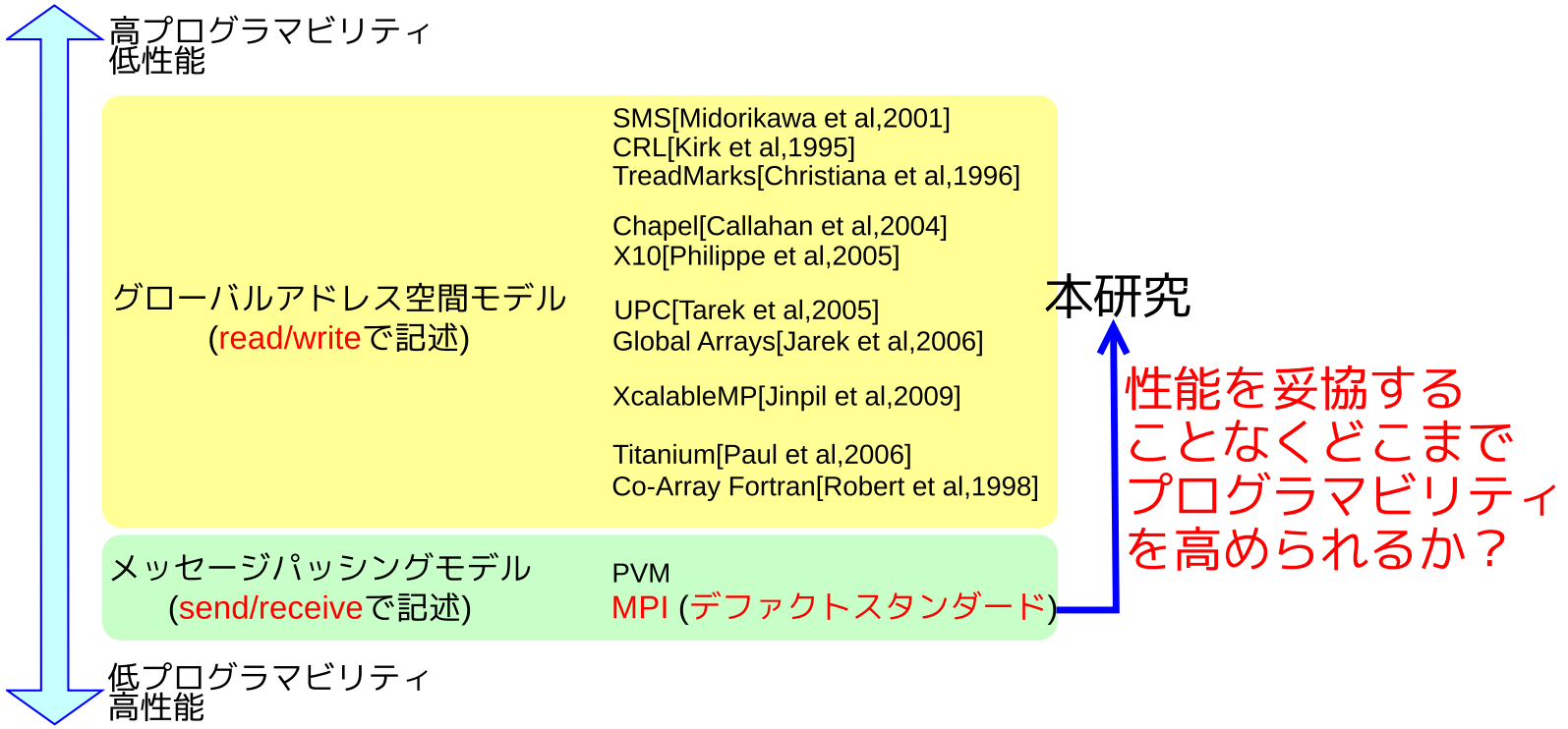
「プログラミングに10時間
実行に1分」
な言語処理系

どんな並列言語処理系が求められているのか?(2)

- ただし「並列」言語処理系では事情がやや特殊
  - → 観察 1：並列プログラムを「使い捨て」目的で書く人はまずいない
    - (1)「プログラミング」：1 回
    - (2)「性能最適化」：1 回
    - (3)「実行」：n 回
  - → 観察 2：「実行」時間を問題にしているからこそ (逐次ではなく) 並列で書いている
    - ♦ 逐次より遅かったら意味がない
- 並列言語処理系にとっては，(プログラマビリティよりも) 「性能最適化」の強力さと「実行」時性能こそが第一義的に重要

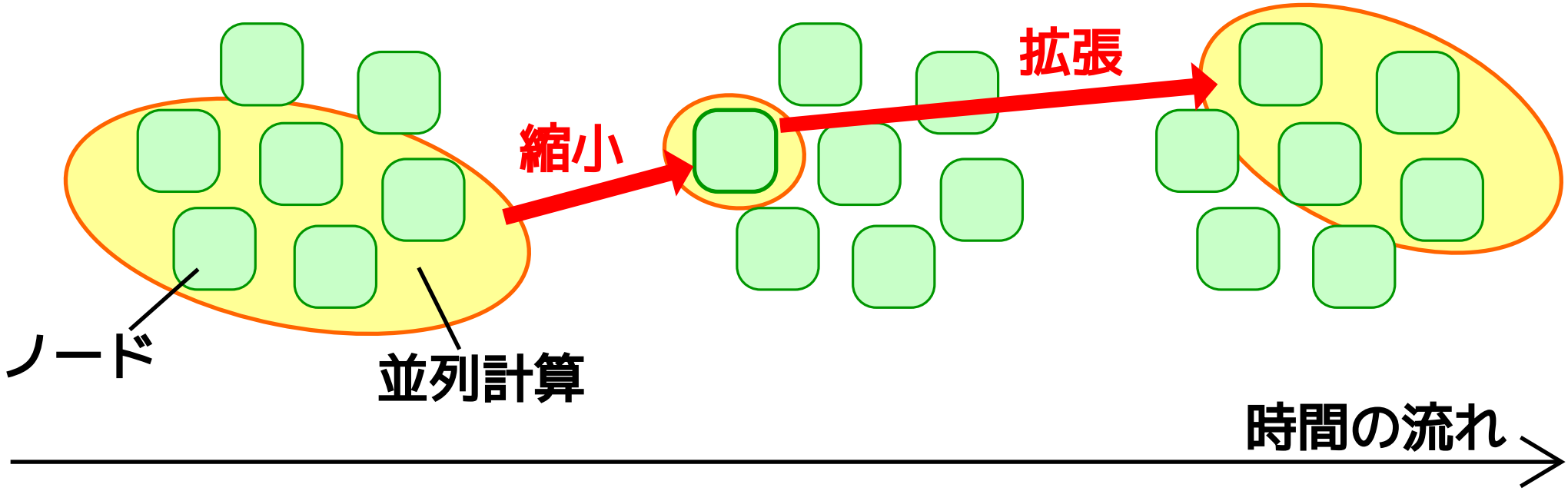
本研究の目的 I

- その意味ではメッセージパッシングモデルが (ひとまず) ベスト
  - → 実際，HPC 分野のデファクトスタンダードは MPI
  - → しかし，プログラマビリティも妥協していてはいけない
- 目的 I：(とくに非定型な) 並列計算に対して，MPI における「性能最適化」の強力さと「実行」時性能を妥協しない範囲で，できるかぎりプログラマビリティを高める
  - → グローバルアドレス空間モデルの採用



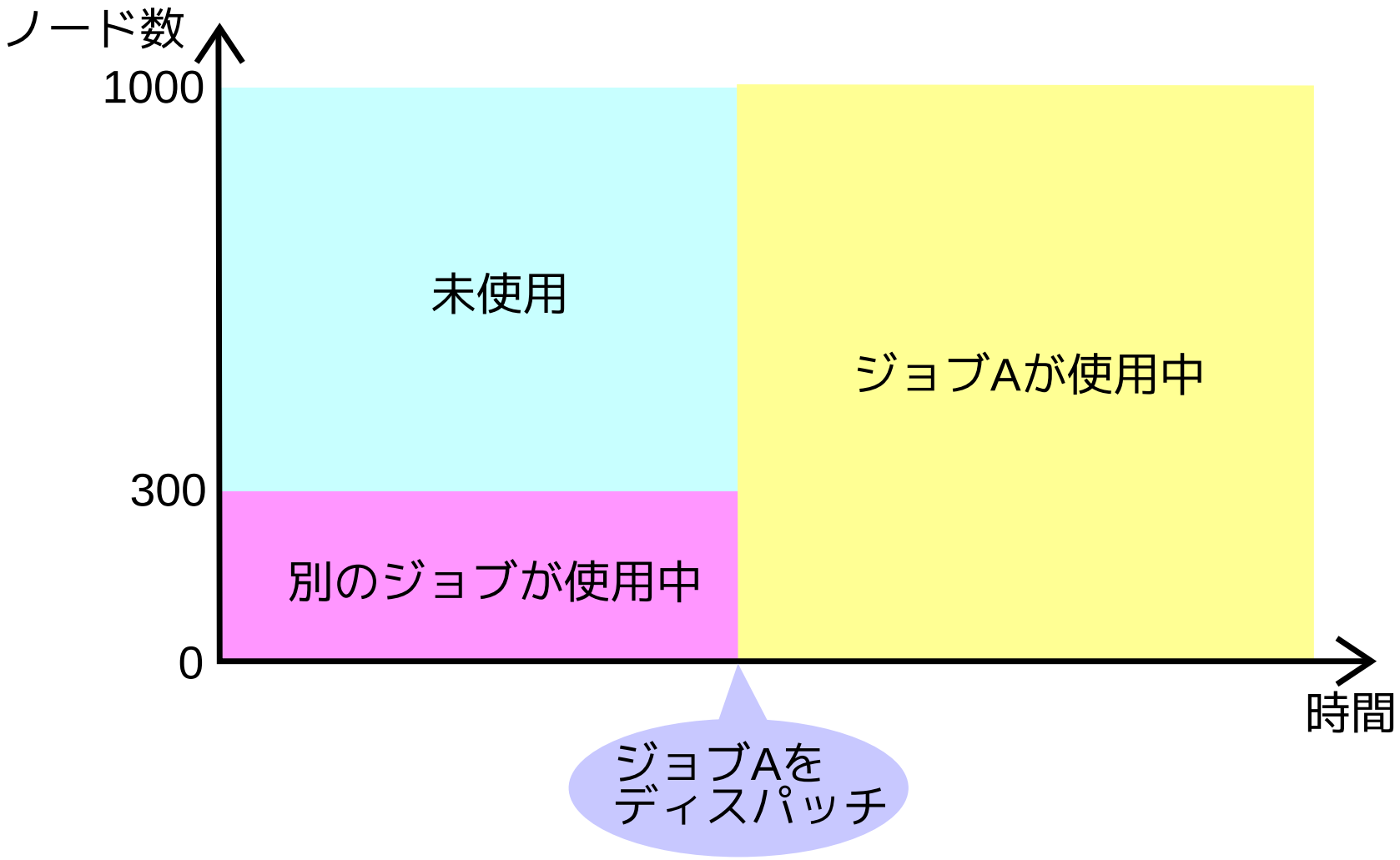
要請 II：再構成可能な並列計算のサポート

- 再構成 = 実行途中に計算規模を拡張/縮小させること



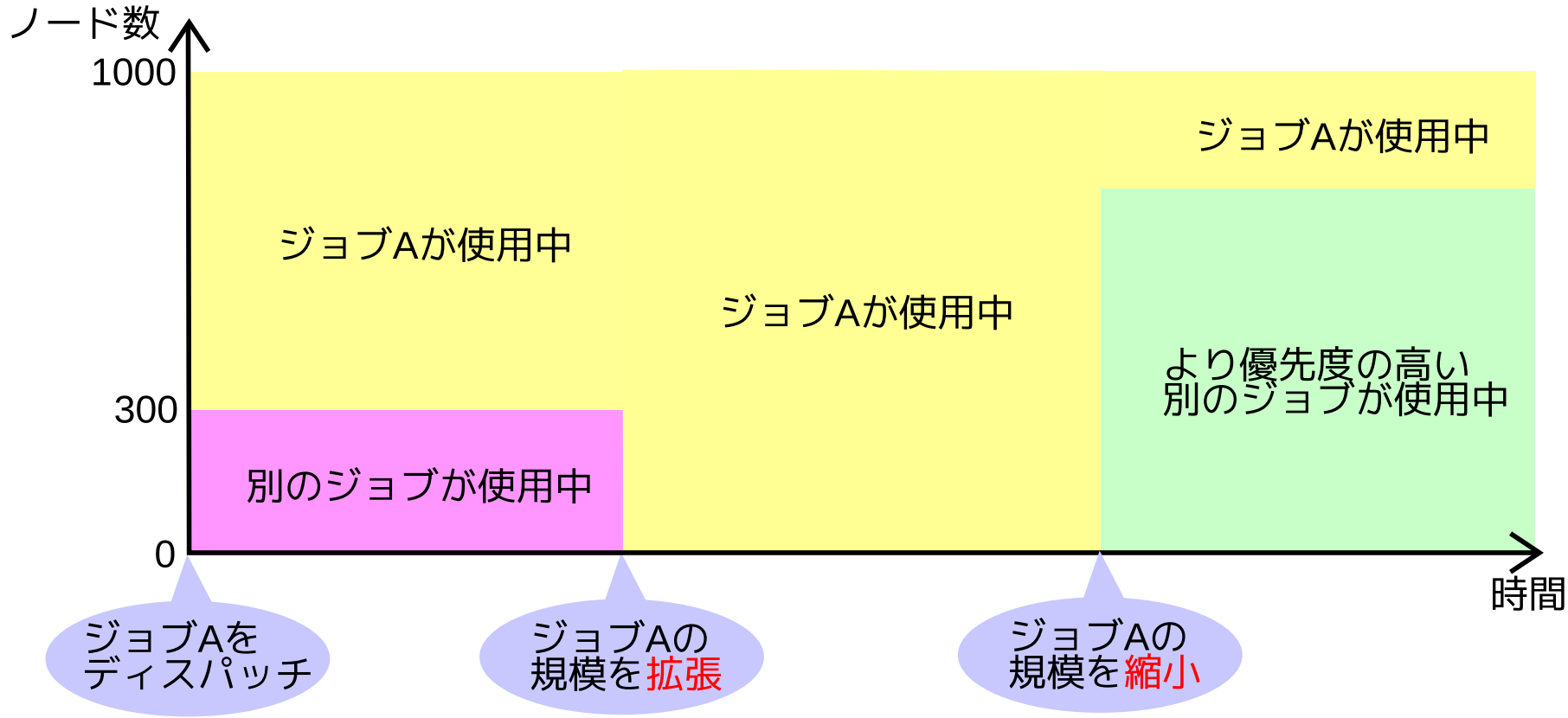
適用例：柔軟なジョブスケジューリング (1)

- **現状のスパコンのジョブスケジューラ** (ex：TORQUE)
  - → **1000 ノードを要求するジョブ A は 1000 ノードが利用可能になるまで実行されない**



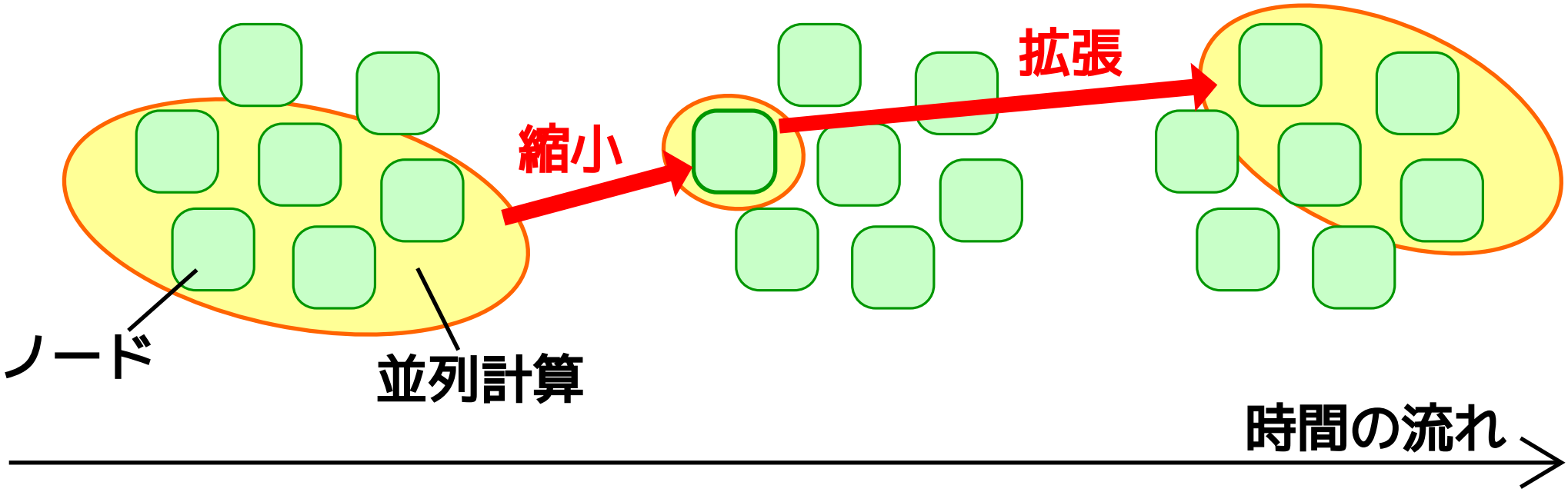
適用例：柔軟なジョブスケジューリング (2)

- ジョブスケジューラのそもそもの目的：
  - → 計算資源の使用率の最大化
  - → ジョブの実行時間の最短化
- その意味で理想的なジョブスケジューラ：
  - → ジョブ (並列計算) を再構成することで，より柔軟に計算資源をスケジューリング



本研究の目的 Ⅱ

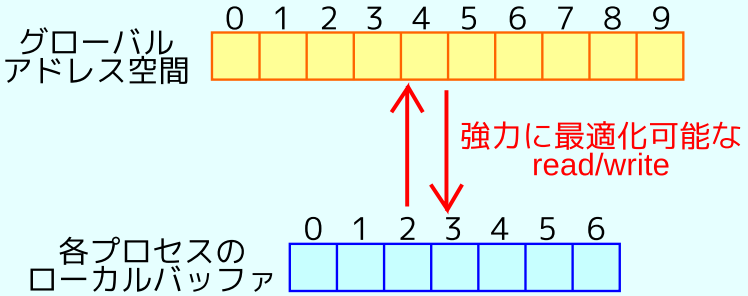
- しかし，再構成可能な並列計算など簡単に書けるものではない
  - →「うまい」プログラミングモデルと処理系が必須
- 目的 Ⅱ：再構成可能な並列計算を簡単に書けるプログラミングモデルを作る



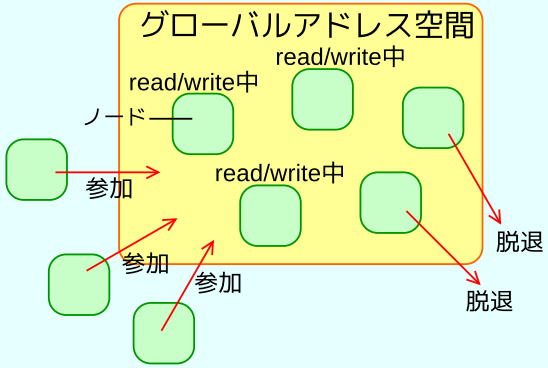
本研究の全体像

➤ **DMI(Distributed Memory Interface)：再構成可能な（とくに非定型な）高性能並列計算のためのグローバルアドレス空間処理系**

(1) 強力に最適化可能なread/writeを備えた高性能なグローバルアドレス空間の設計



(2) ノードが自由なタイミングで参加/脱退できるグローバルアドレス空間のコヒーレンシプロトコルを設計



(3) 再構成可能な並列計算を簡単に書けるようにするためのプログラミングモデルを3種類提案して比較検討

- 1.スレッド増減
- 2.スレッド移動
- 3.スレッド移動 with half-process

(4) 性能とプログラマビリティの評価

発表の流れ

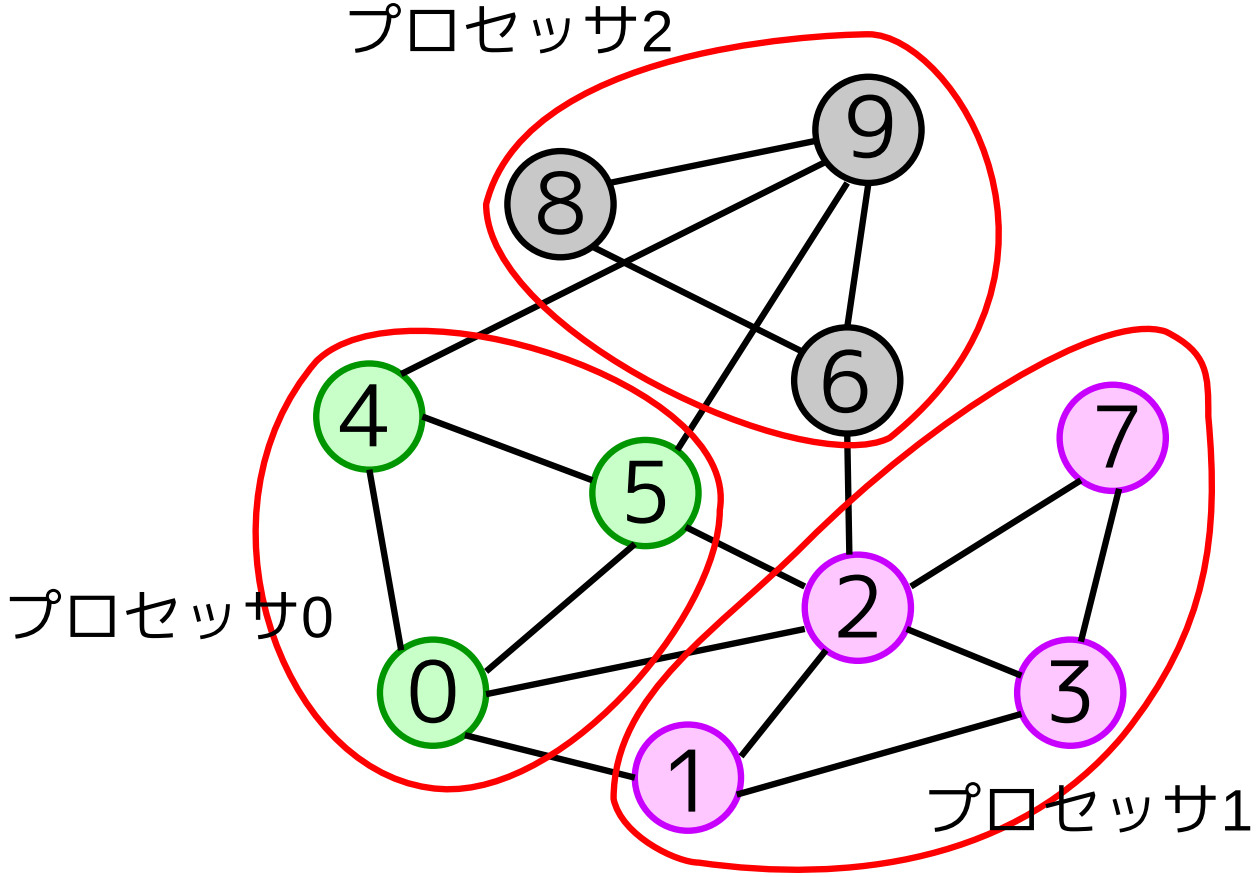
(1) 非定型な並列計算のサポート
- → 関連研究
- → 設計
- → 評価

(2) 再構成可能な並列計算のサポート
- → 関連研究
- → 設計
- → 評価

(3) 結論

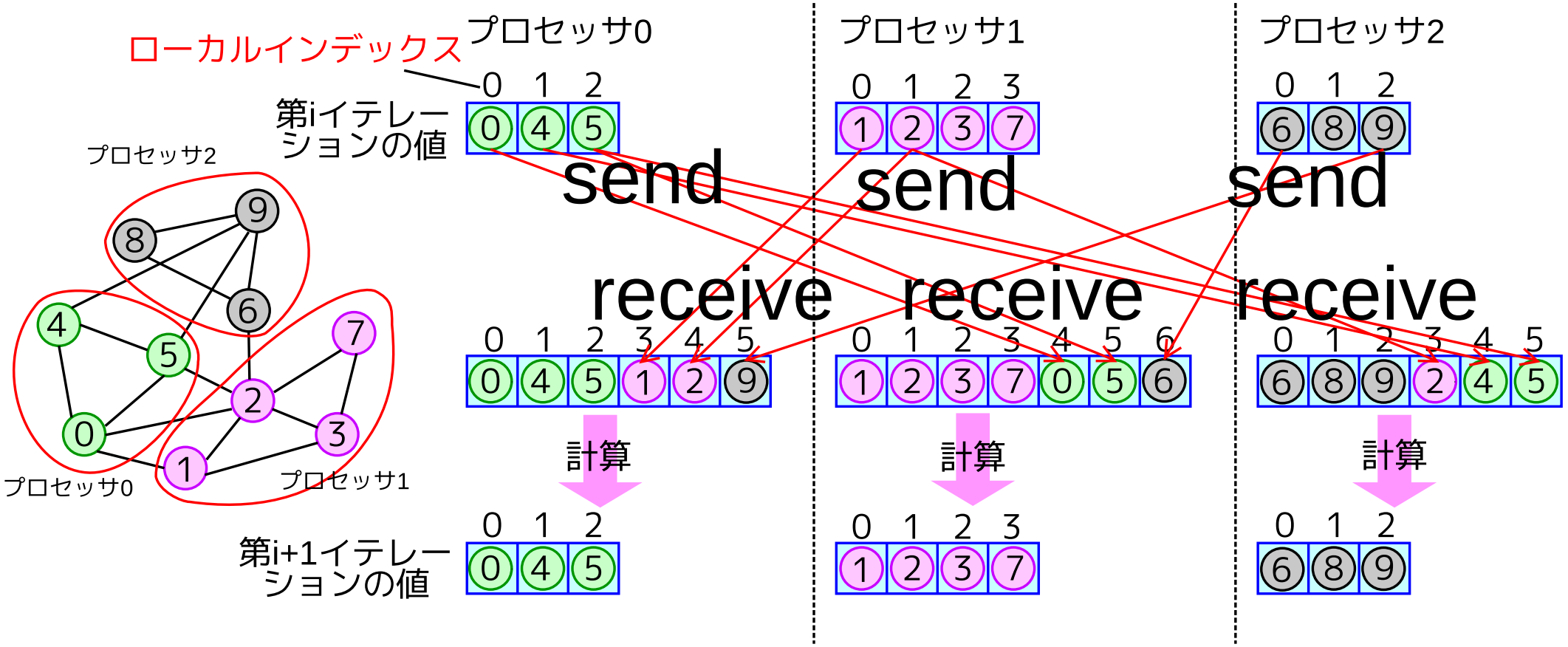
❖ 非定型な並列計算のサポート [関連研究]

非定型な並列計算の例



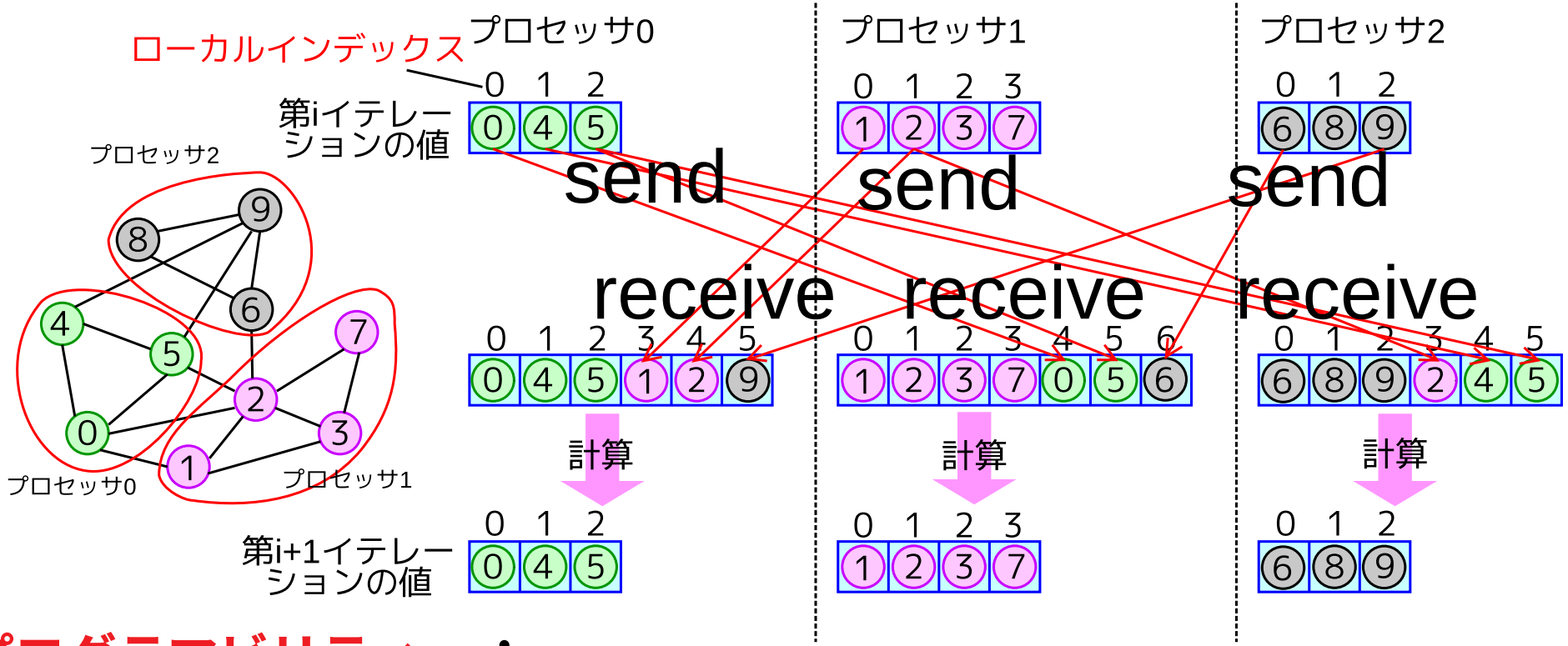
- グラフを 3 個のサブグラフに分割
- ex：節点 5 の値の更新には，節点 0,2,4,9 の値が必要
- メッセージパッシングモデル vs グローバルアドレス空間モデル

メッセージパッシングモデルではどう書けるか?(1)



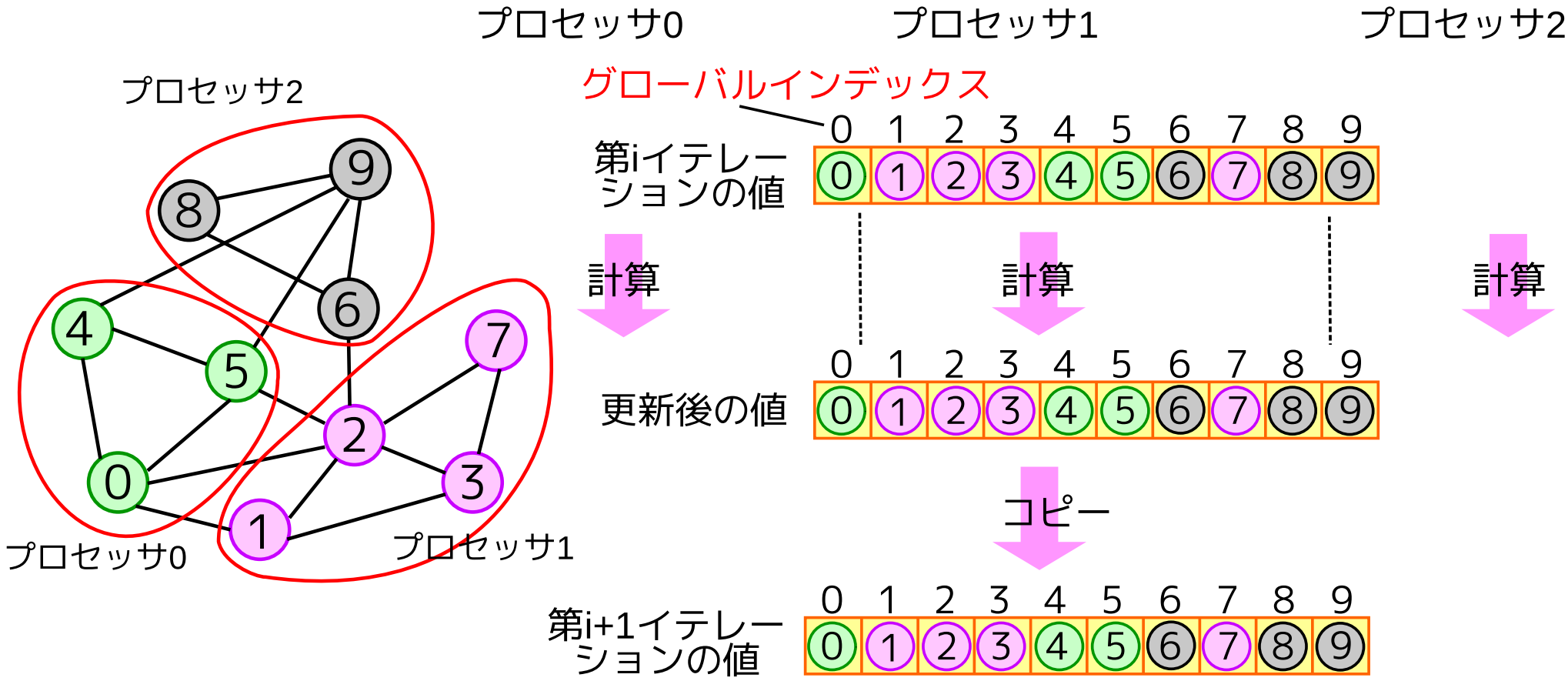
- **性能◎**
  - → 本質的に必要な通信だけが起きる
- **性能最適化◎**
  - → データの配置と通信をすべて明示できる

メッセージパッシングモデルではどう書けるか?(2)



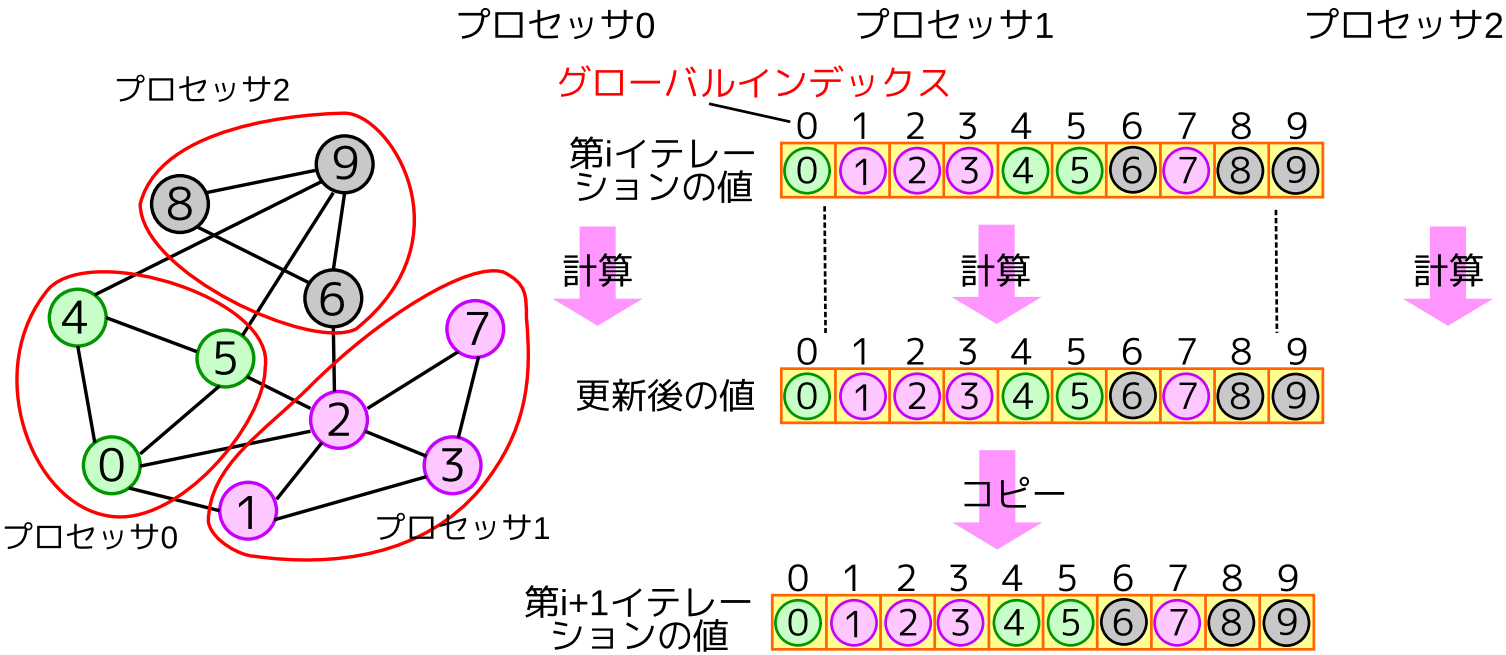
- **プログラマビリティ ×：**
  - → **節点番号とローカルインデックスとを対応付ける煩雑な計算が必要**
  - → **プロセッサ 0 が節点 9 の値を「得る」には，プロセッサ 2 のローカルバッファの 2 番目の要素を send してもらい，プロセッサ 0 のローカルバッファの 5 番目の要素として receive する**
  - → **プロセッサ 0 が節点 9 の値を計算で「使う」には，自分のローカルバッファの 5 番目を read する**

グローバルアドレス空間モデルではどう書けるか?(1)



- **プログラマビリティ◎：**
  - → **いわゆる「共有メモリ」**
  - → **節点 9 の値を「得る」にも「使う」にも，単にグローバルアドレス空間の 9 番目を read すればよい**

グローバルアドレス空間モデルではどう書けるか?(2)



- **性能 ×：**
  - → 処理系による最適化手段（コンシステンシモデルの緩和 [Midorikawa et al,2001]，コヒーレンシ粒度の微細化 [Daniel et al,1997]，inspector/executor[Jimmy et al,2005]，...）は存在するが，結局，メッセージパッシングモデルと同等の通信だけが起きるレベルまで内部的な通信を集約させるのは無理
- **性能最適化 ×**
  - → グローバルアドレス空間に**あまりに透過的に**アクセスできてしまうため，**内部的にいつどんな通信が起きるかがわからない**

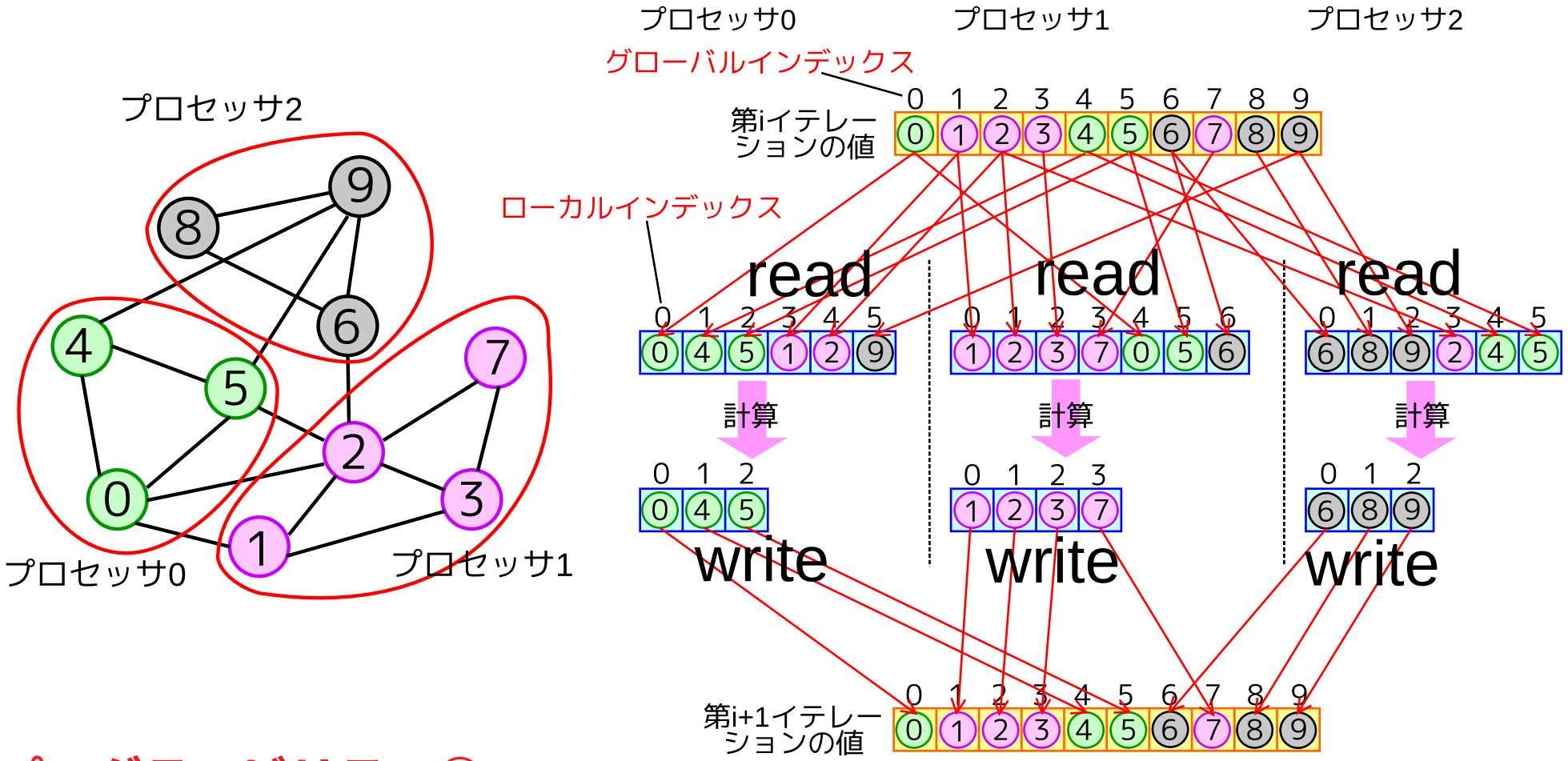
既存の処理系の問題例

- 大学院講義「並列分散プログラミング」で，並列粒子法を UPC，Chapel，X10 で書いてみた受講生の感想
  - →「初期的にプログラムを書くのは簡単」
  - →「異様に遅いが原因がわからない」
  - →「何を書いたときにどんな通信が起きるかがわからない」
- ex：Chapel

```
var a: [1..N] real;
on Locales(1) {
    var b: [1..N] real;
    b = a;
}
```

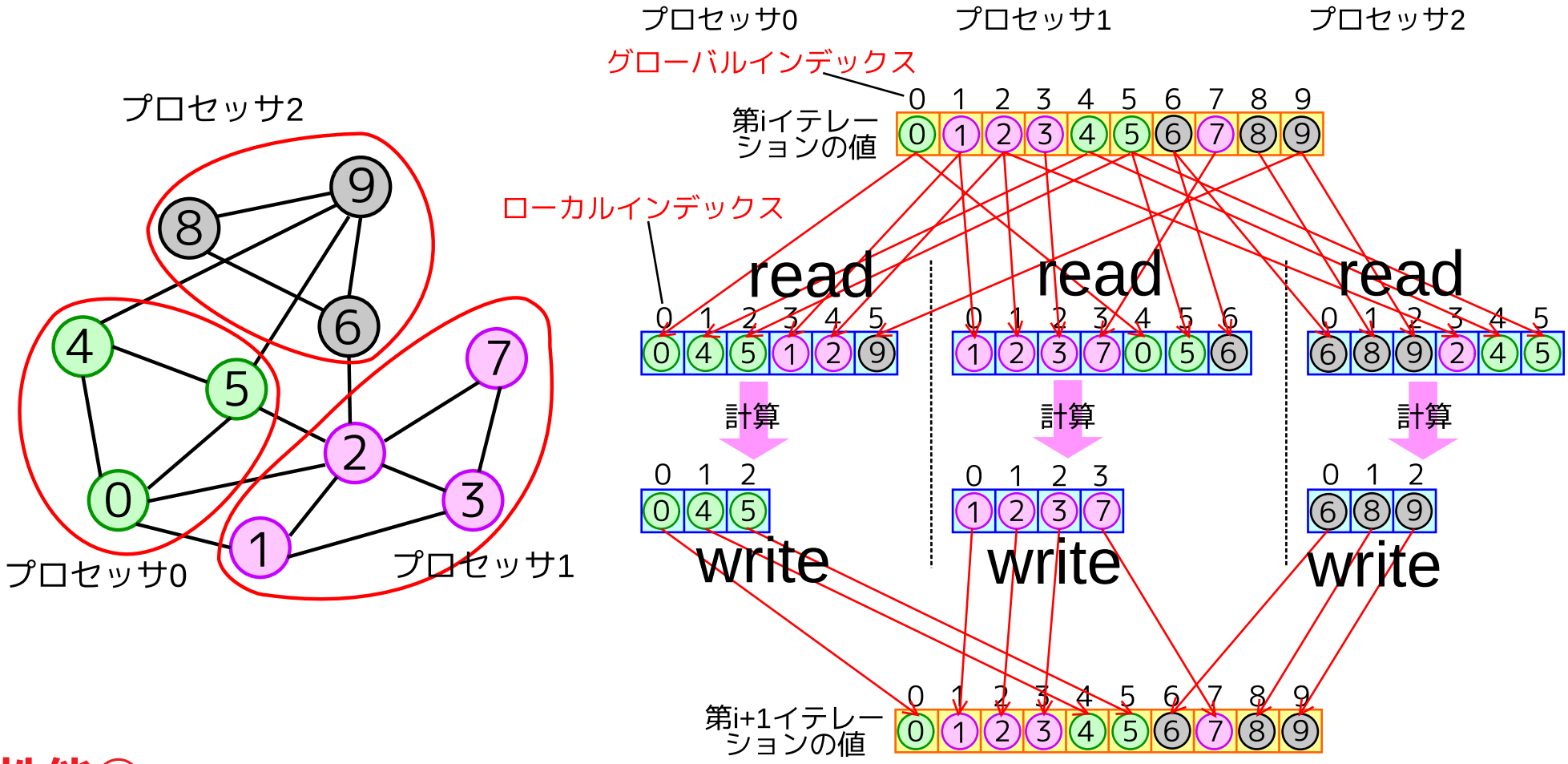
配列の各要素ごとに
N回の通信が発生!!!

やや妥協したグローバルアドレス空間モデル (1)



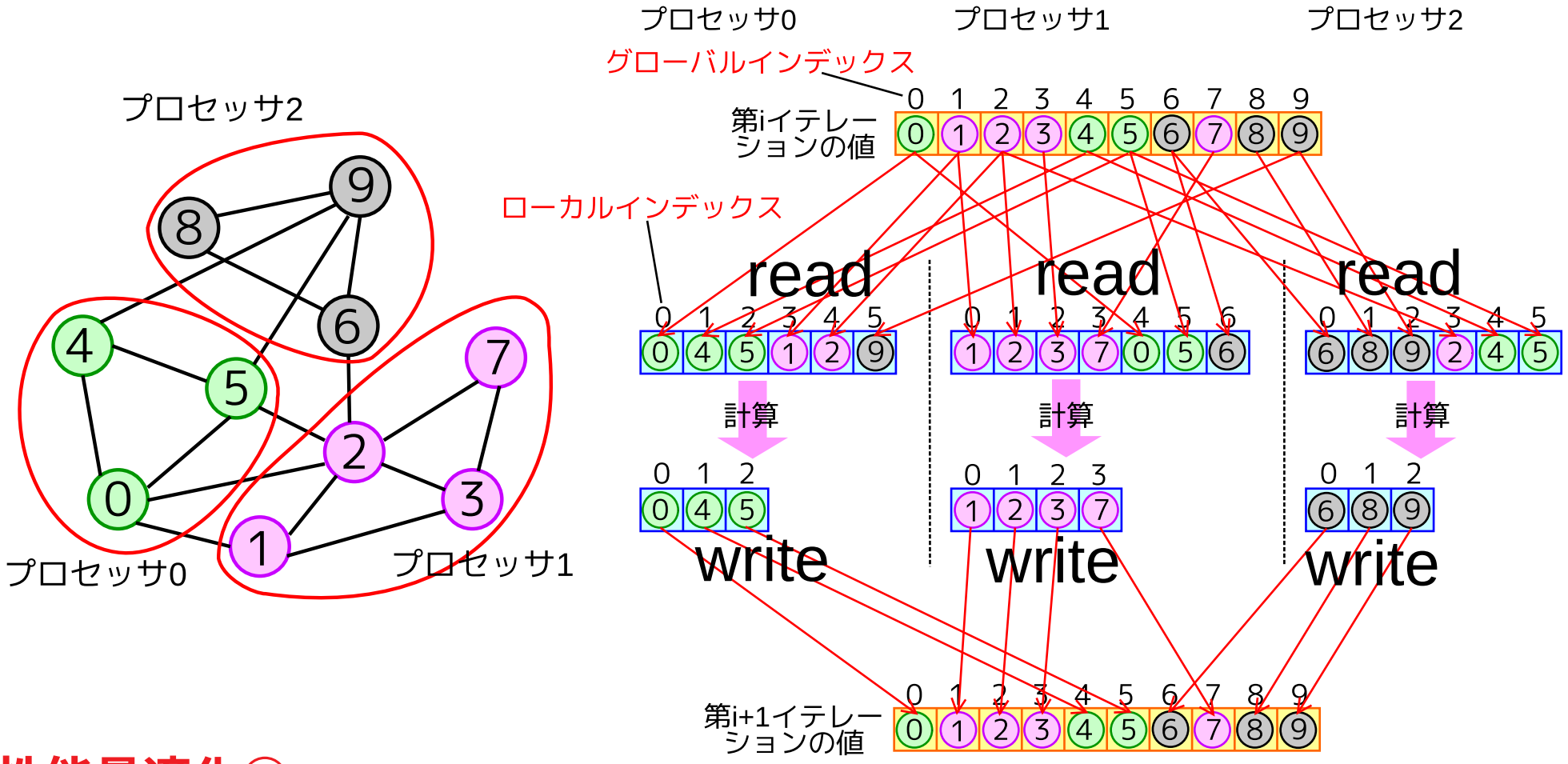
- **プログラマビリティ○：**
  - → **節点 9 の値を「得る」には，単にグローバルアドレス空間の 9 番目を read すればよい**
  - → **節点 9 の値を「使う」には，自分のローカルバッファの 5 番目を read する必要がある**

やや妥協したグローバルアドレス空間モデル (2)



- **性能◎：**
  - → **グローバルアドレス空間への read/write をできるかぎり集約し，できるかぎりローカルバッファで計算を行うように書けば，メッセージパッシングモデルと同等の通信だけを起こすことは可能**

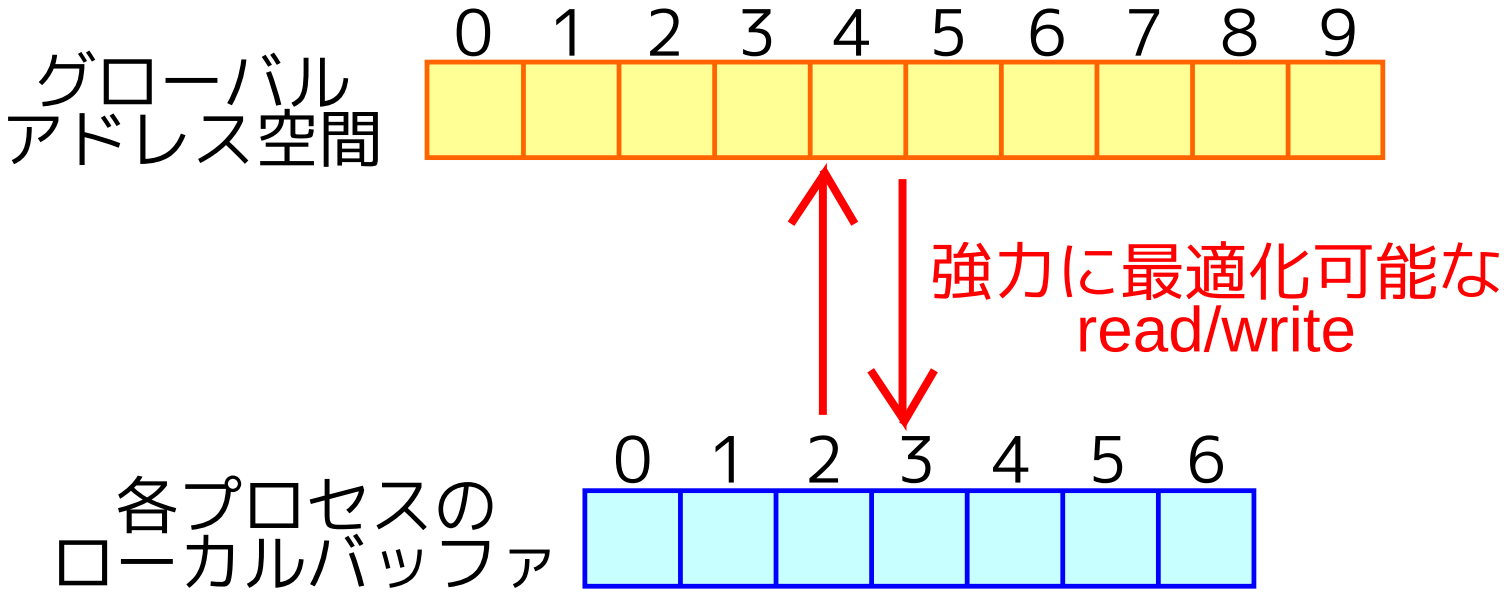
やや妥協したグローバルアドレス空間モデル (3)



- **性能最適化◎**
  - → **グローバルアドレス空間への多様な read/write が引き起こす通信を，強力に最適化できる API が提供することは可能**
  - → **しかし，既存の処理系の API は不十分**
    - ♦ UPC，Titanium，Chapel，X10，Global Arrays，XcalableMP，...

❖ 非定型な並列計算のサポート [設計]

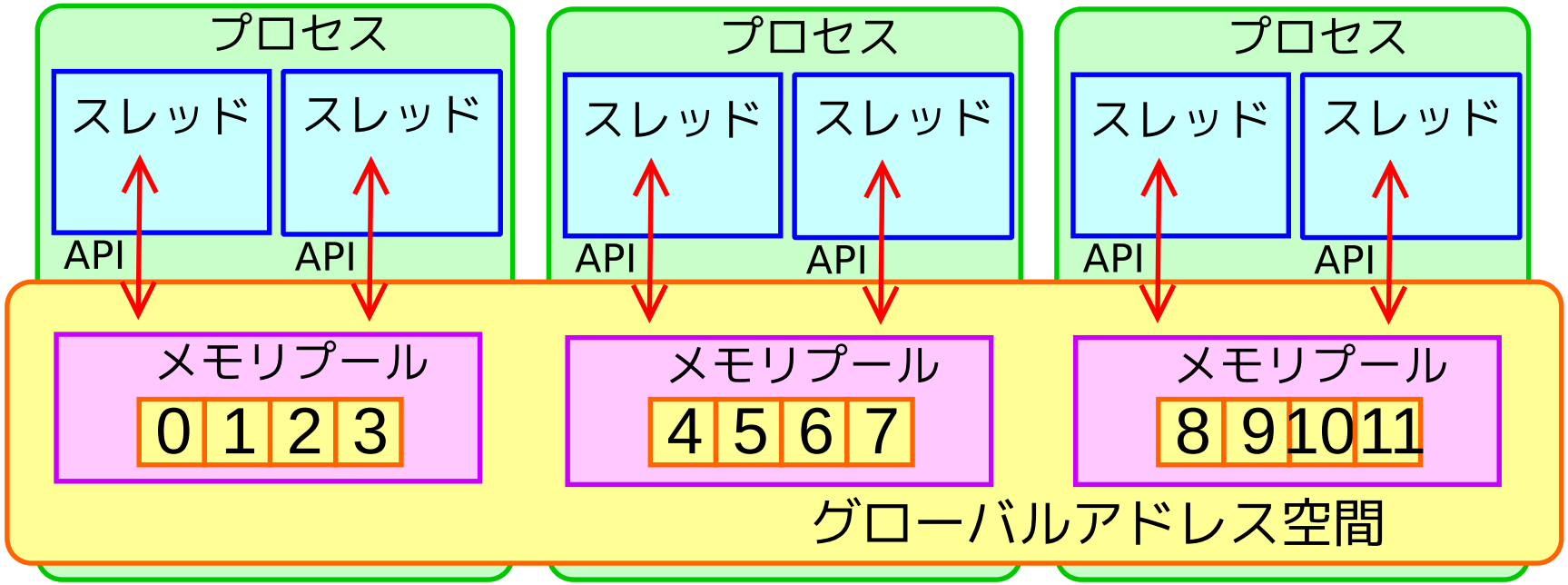
設計の全体像



- **グローバルアドレス空間への非定型で多様なread/write について，**
  - **(1) どんな通信が起きるかがわかりやすい**
  - **(2) 通信を明示的に簡単に制御できる**
  - **(3) 通信をできるかぎり集約させられる**

  ような API を設計する

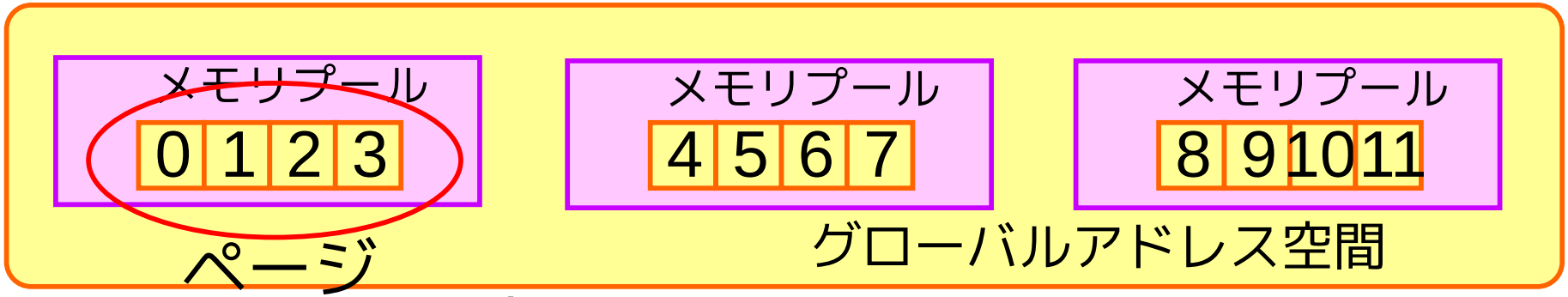
DMI の全体像



- **マルチスレッド型のグローバルアドレス空間処理系**
  - → 多数のノードのメモリプールをかき集めてグローバルアドレス空間を実現
- 同一ノード内のスレッドたちでメモリプールを共有
  - → 透過的に**ハイブリッドプログラミング**を実現
- ノード数を増やせば**遠隔スワップシステム**(大容量メモリ) としても利用可能
- C 言語で実装 (27000 行)

任意のコヒーレンシ粒度

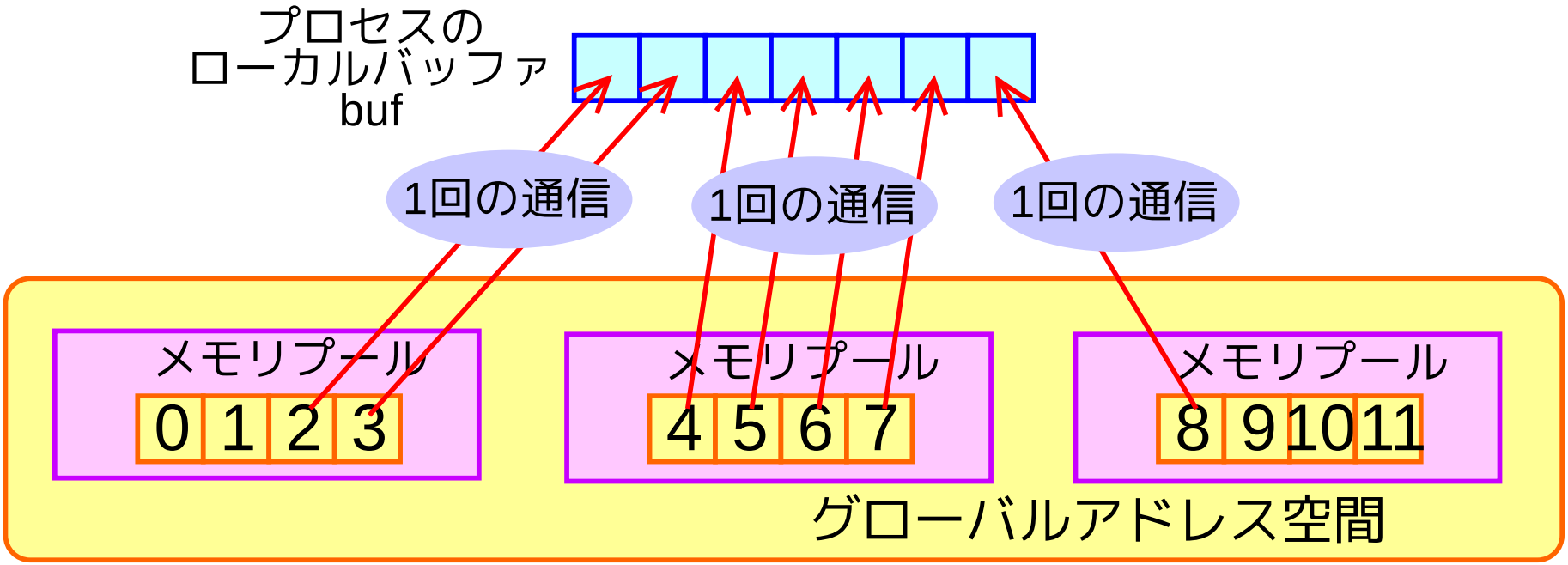
DMI_mmap(4, 3)の結果：



- DMI_mmap($page\_size$, $page\_num$)：
  - → **ページサイズが $page\_size$ のページを $page\_num$ 個確保**
- **内部的な通信上の単位をアプリにとって必要十分なだけ巨大化させられる**
  - → ex：**行列まるごと 1 個を 1 ページにする**

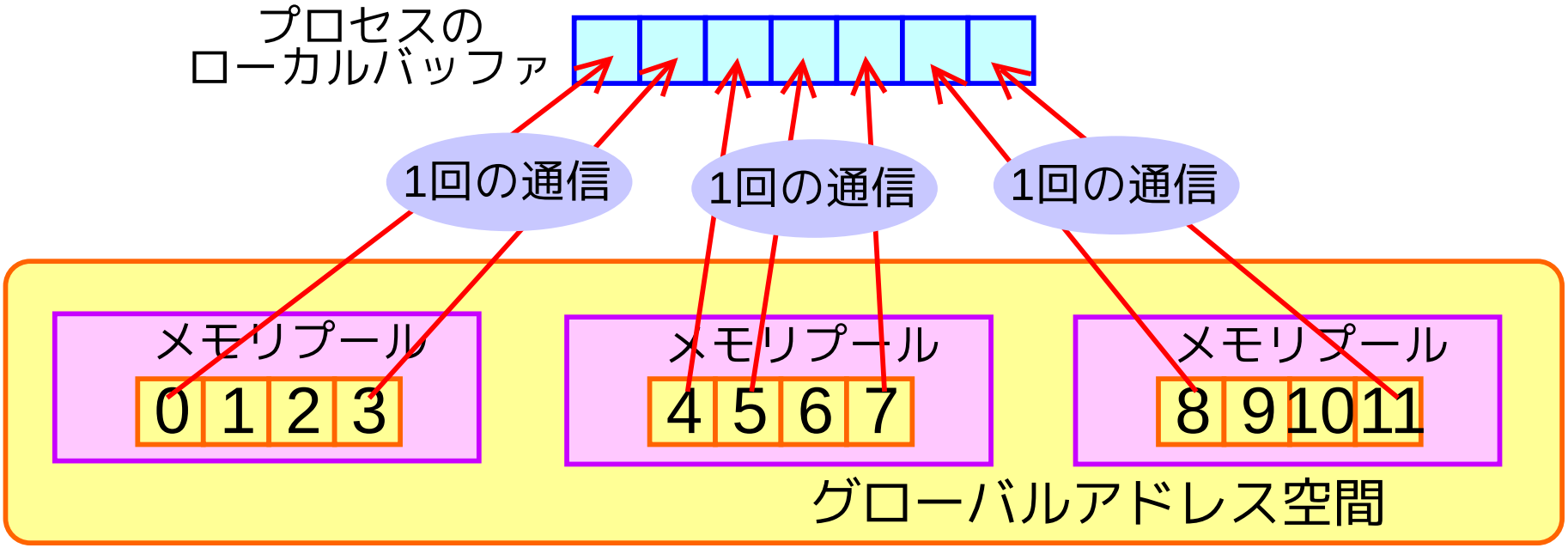
連続した領域を一気に read/write

DMI_read(アドレス2, 7バイト, buf)の結果：



- DMI_read(*addr*, *size*, *buf*, ...)：
  - → **グローバルアドレス** *addr* **から** *size* **バイトを，ローカルアドレス** *buf* **に** read **する**
- DMI_write(*addr*, *size*, *buf*, ...)
- **(複数のページにまたがる) 連続した領域を一気に集約して** read/write

離散的な領域を一気に read/write



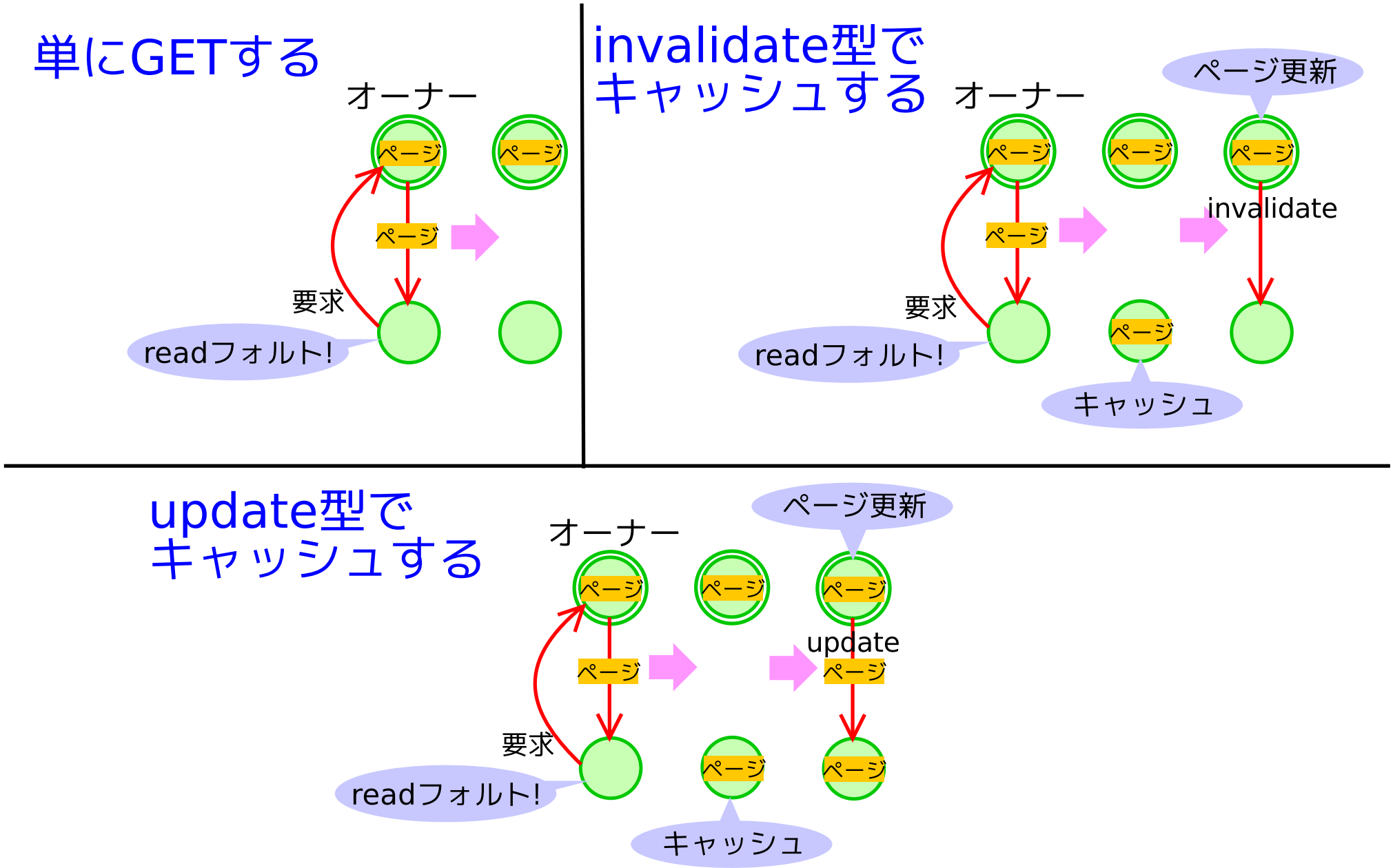
- **(複数のページにまたがる) 離散的な領域を一気に集約して read/write**
  - → 10000 **ヶ所を** read/write **しようが，アクセスしたページの個数の通信しか起きない**

アクセスローカリティの最適化

- 問：そもそも通信を起こさないためにはどうしたらよいか?
- 答：read/write フォルトを最小化すればよい
  - → read フォルトを防ぐにはページをキャッシュすればよい
  - → write フォルトを防ぐには write ローカリティを持ったノードがページのオーナーになっていればよい
- DMI ではこのようなアクセスローカリティを強力に明示的に最適化できる

read ローカリティの最適化

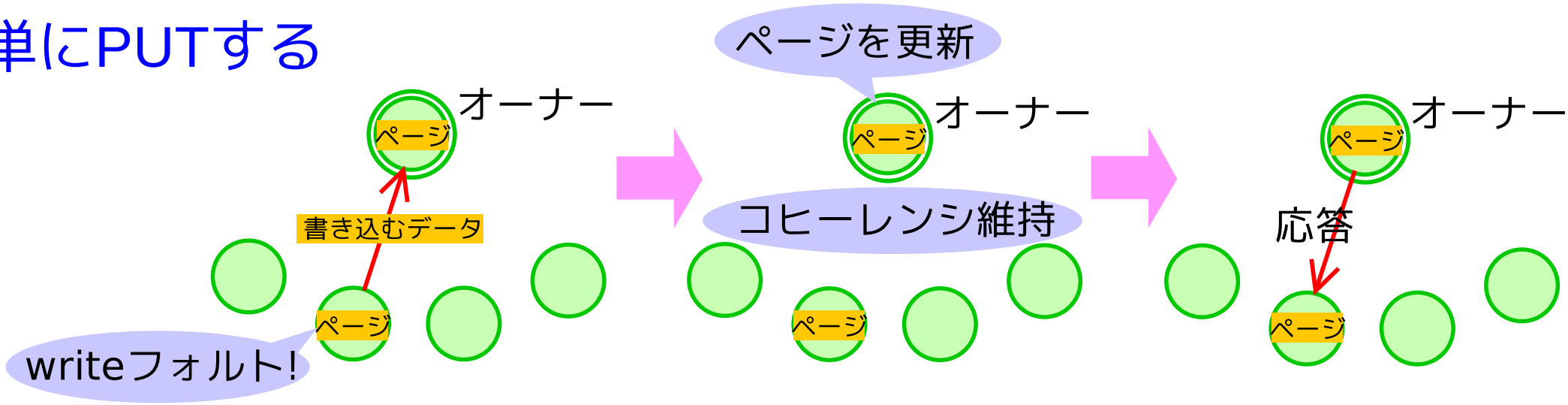
- DMI_read(*addr*, *size*, *buf*, ***mode***)



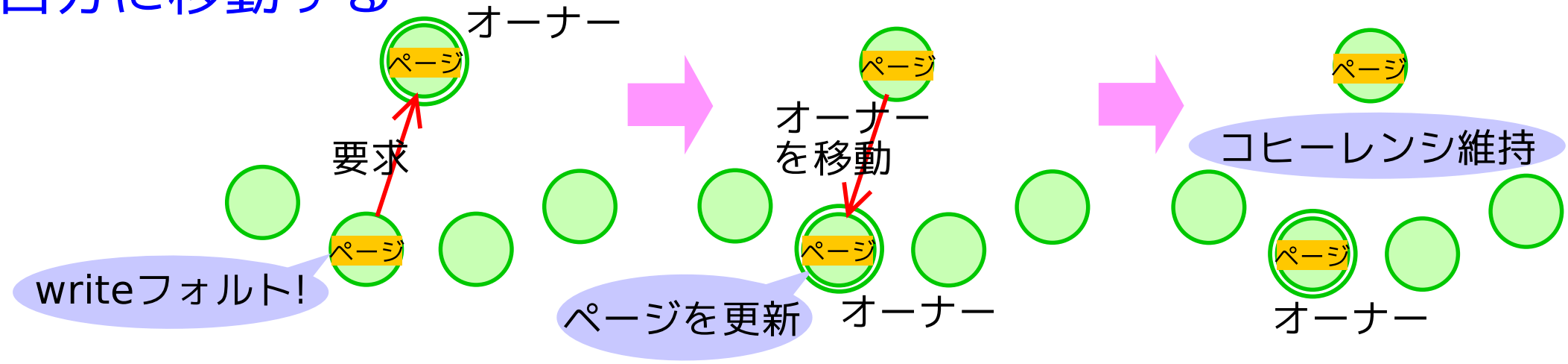
write ローカリティの最適化

➤ DMI_write(*addr*, *size*, *buf*, ***mode***)

単にPUTする

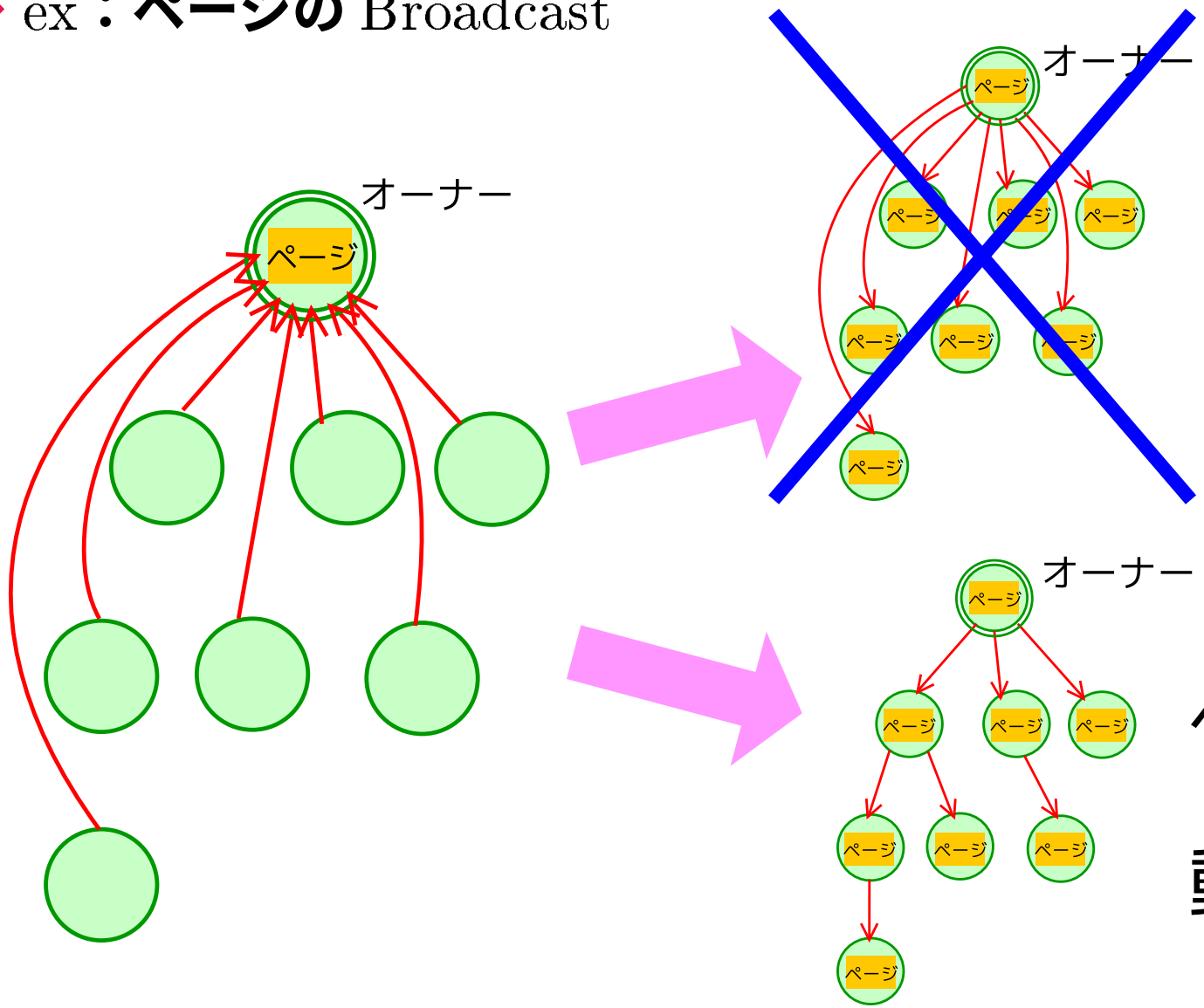


オーナーを自分に移動する



データ転送の動的負荷分散

- **オーナーにページの read 要求が集中した場合，すでにページのキャッシュを持っているノードを使ってページ転送負荷を分散**
  - → ex：**ページの** Broadcast



その他の API

- read/write を非同期化する API(プリフェッチ + ポストストア)
- read-modify-write を自由に定義できる API(高速な同期)
- あるアドレスの値の変更を待機する API(高速な同期)
- 非定型な領域分割に伴う read/write をグローバルビューで表現する API
- ...
- (83 個の C 言語 API)

❖ 非定型な並列計算のサポート [評価]

実験条件

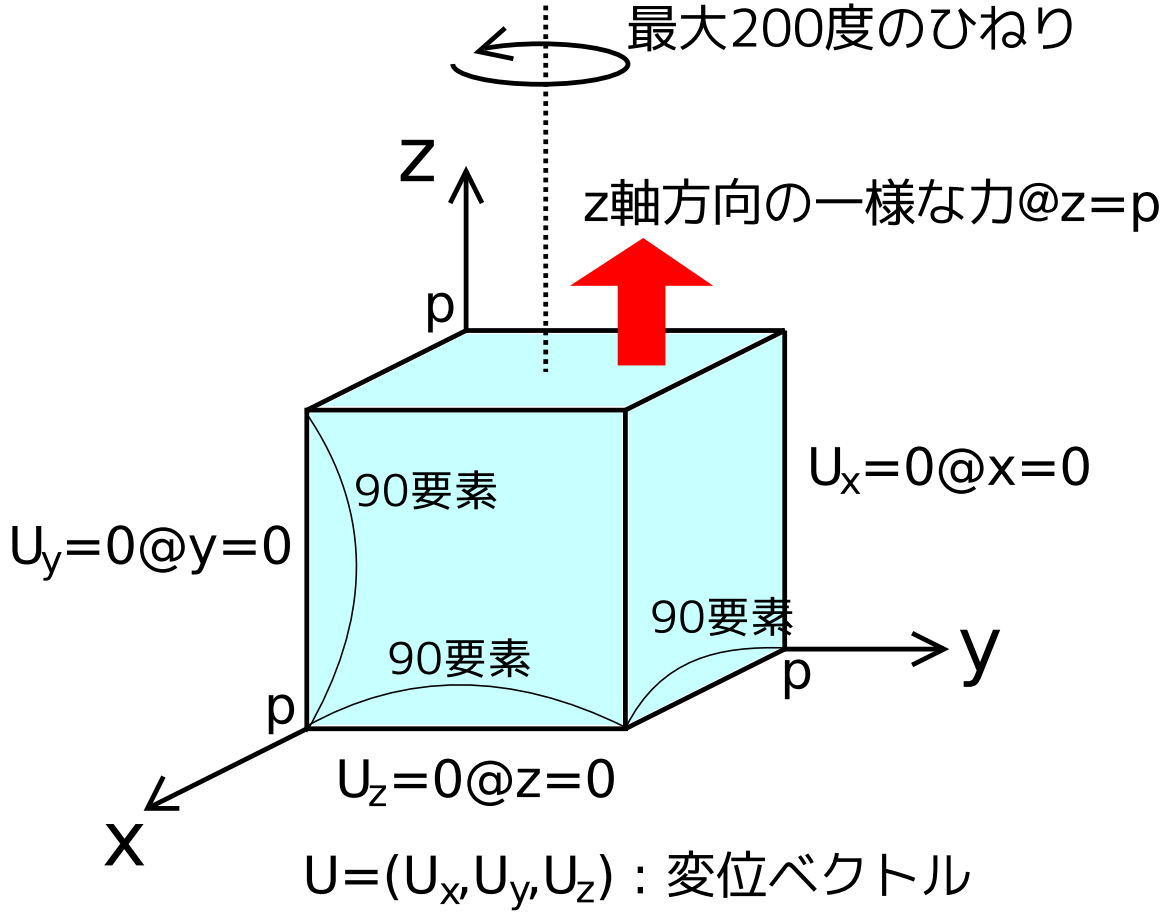
- 環境：8 コア ×16 ノード，10Gbit イーサネット
- 各ノードの構成：
  - CPU：Intel Xeon E5330(4 コア,L1:256KB,L2:1MB,L3:8MB)×2
  - メモリ：2GB×12，1066 MHz
  - OS：Linux カーネル 2.6.26-2-amd64
  - NIC：NetXtreme II 10Gigabit PCIe
- DMI vs (HPC 分野でデファクトスタンダードの)MPI
  - MPI の実装：mpich2 1.2.1p1，OpenMPI 1.4.2
  - コンパイラ：gcc -O3

アプリケーション

- DMI で合計 17000 行記述
- マイクロベンチマーク：
  - → read/write 性能，mutex，Allreduce，Broadcast，STREAM ベンチマークによる遠隔スワップの性能
- 基本的なアプリ：
  - → NAS Parallel Benchmark(EP)，マンデルブロ集合の描画，横ブロック分割による行列行列積，Fox アルゴリズムによる行列行列積，ランダムサンプリングソート，N 体問題，ヤコビ法による PDE ソルバ
- 非定型で応用的なアプリ：
  - → 有限要素法による応力解析，大規模 Web グラフのページランク計算，大規模 Web グラフの最短路計算

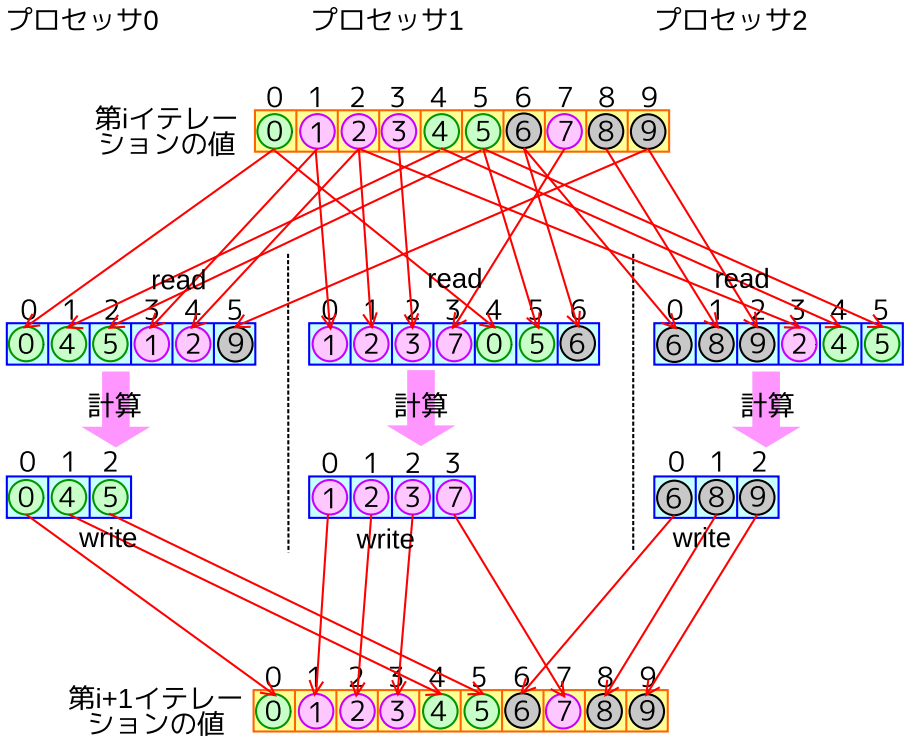
有限要素法：実験

- 有限要素法による応力解析：
  - → 第 2 回並列プログラミングコンテストの題材
  - → 疎行列係数の連立一次方程式 $Ax = b$ の解を反復法で求める
  - → 実世界の工学に基づく非常に収束させづらい問題

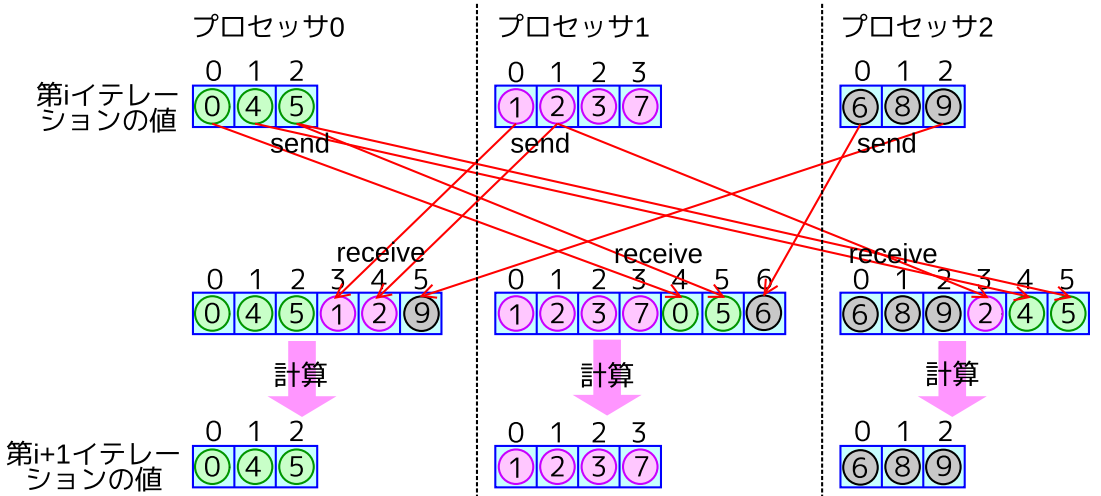


有限要素法：プログラマビリティ

DMI：2368行

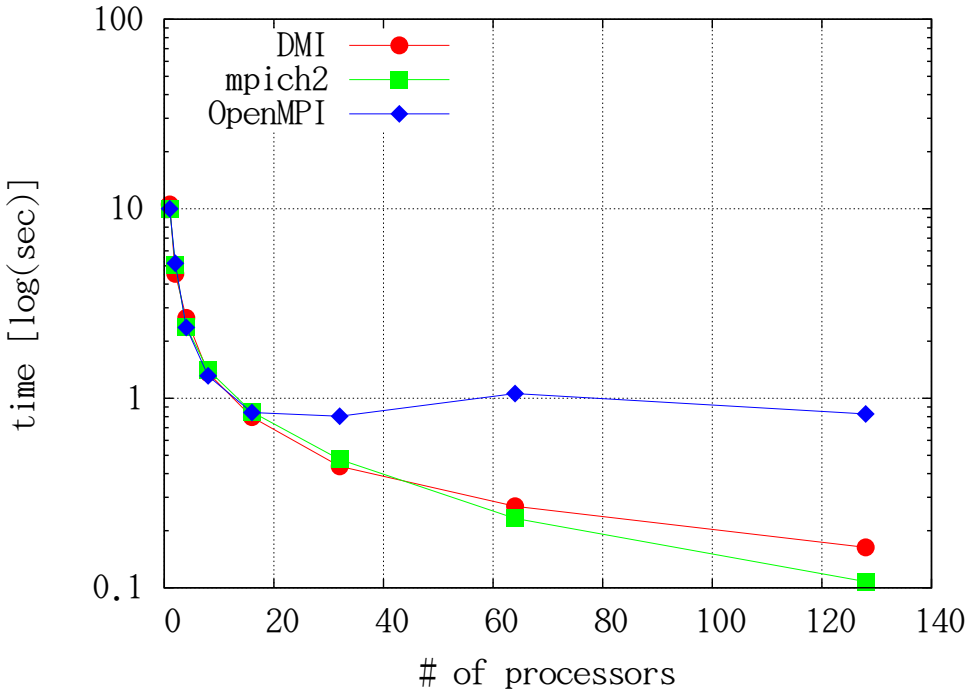


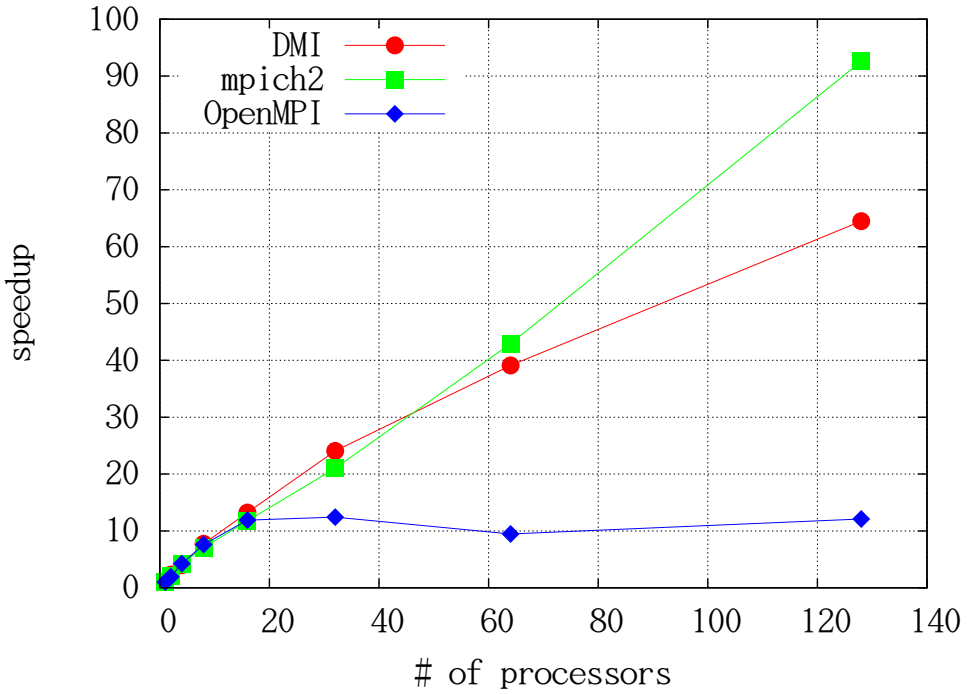
MPI：2572行



- MPI では節点番号とローカルインデックスを対応付ける煩雑な計算が必要
- DMI では不要

有限要素法：性能

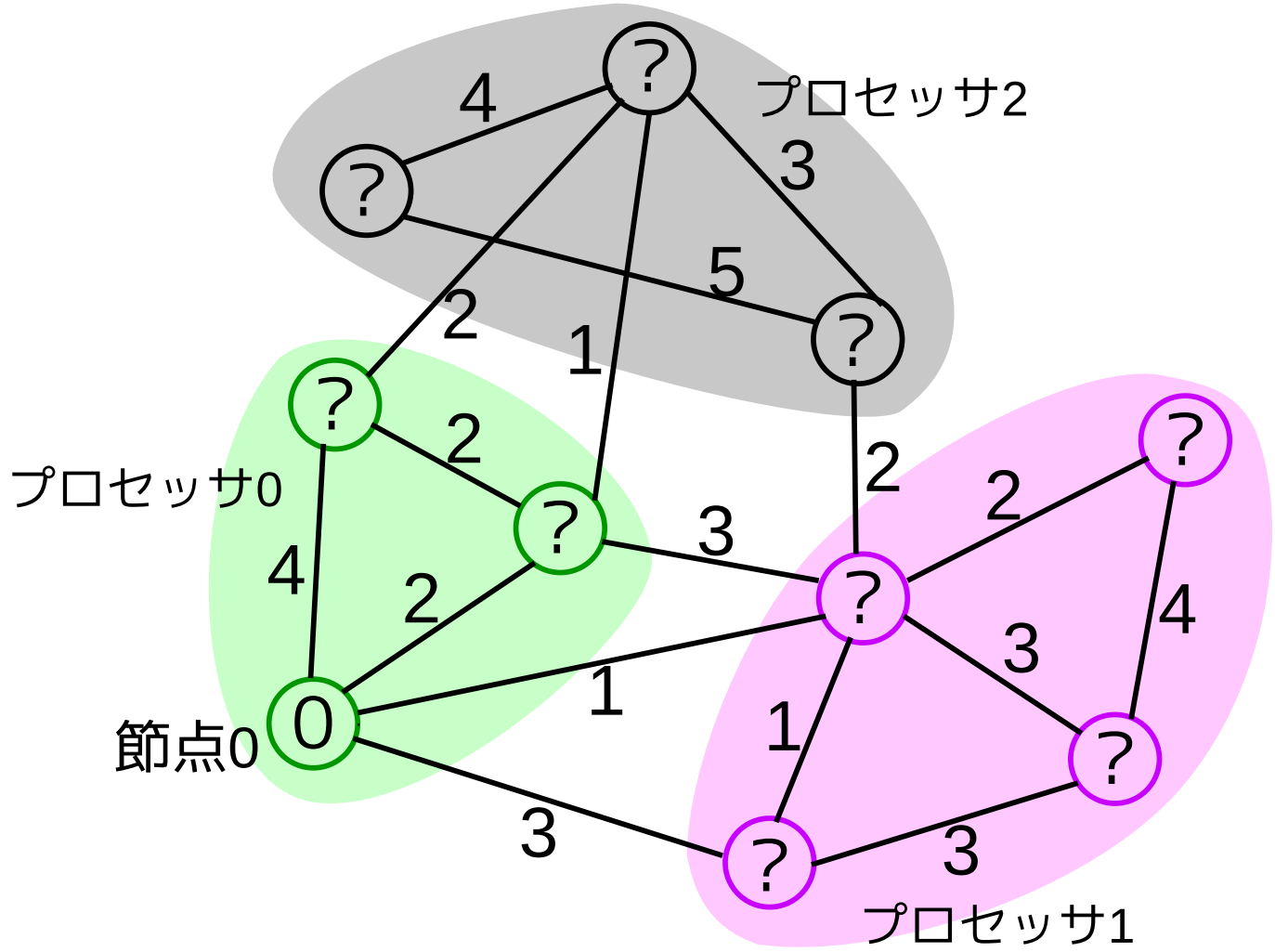




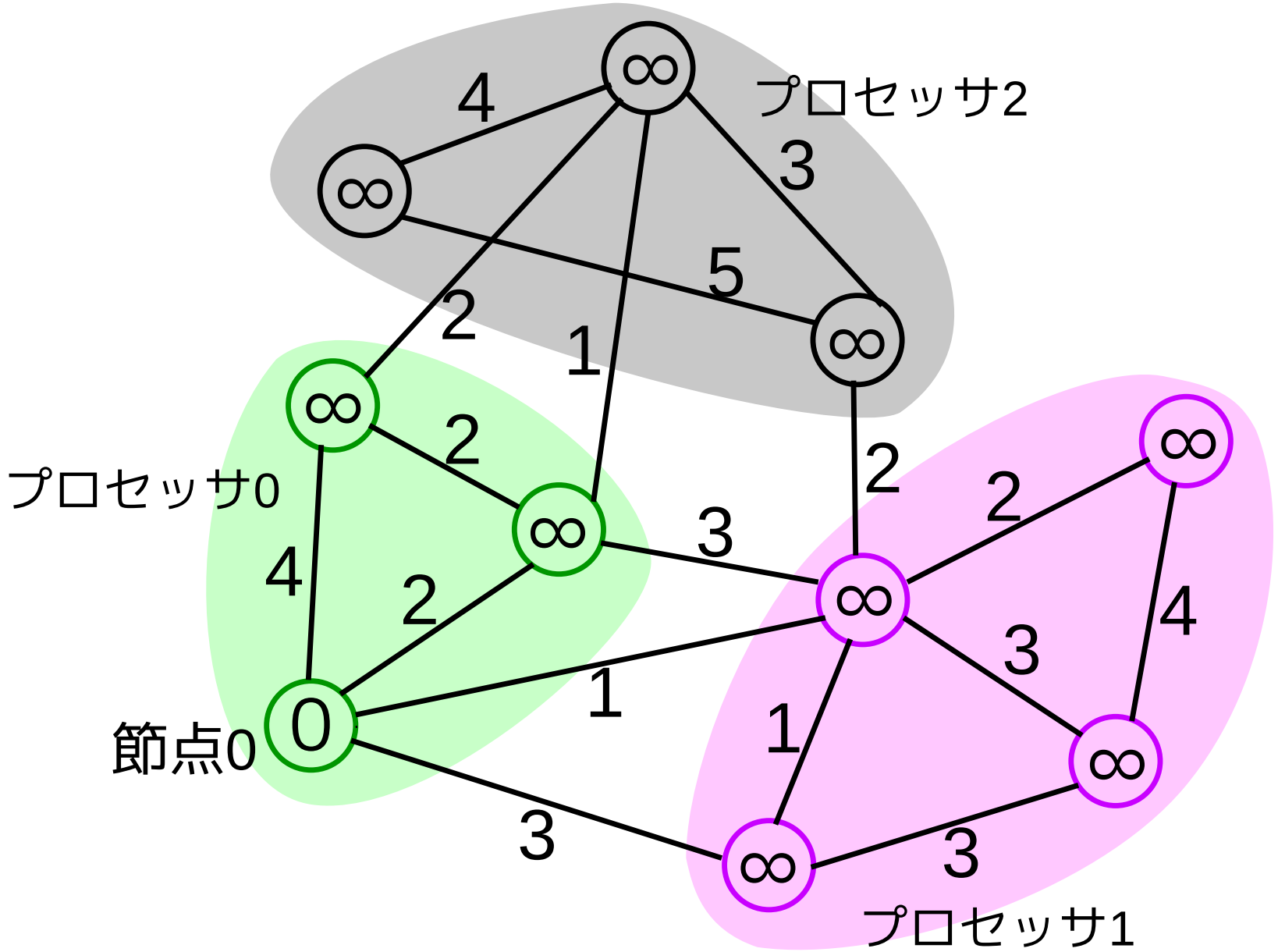
- **mpich2>DMI>OpenMPI**
- DMI と mpich2 の性能差は，バリア操作の性能差が原因
- OpenMPI の遅さは，OpenMPI の 1 対 1 通信が遅いため

Web グラフの最短路計算：実験

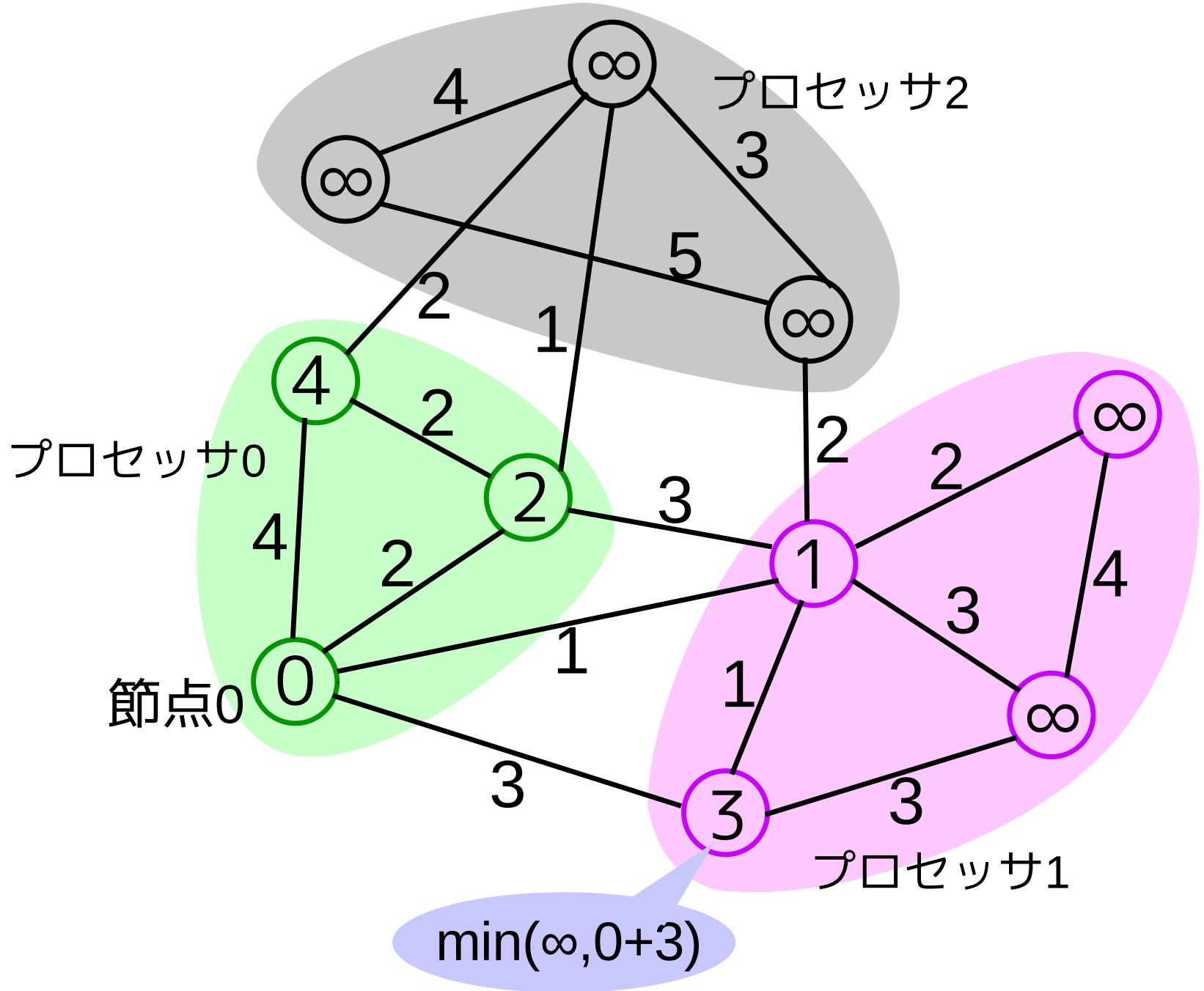
- 数理モデルに基づいて Web グラフを人工的に生成
  - → 節点数 1.28 億，エッジ数 4.48 億，エッジカットは全エッジの 10%
- 節点 0 から他の全節点までの最短路を求める



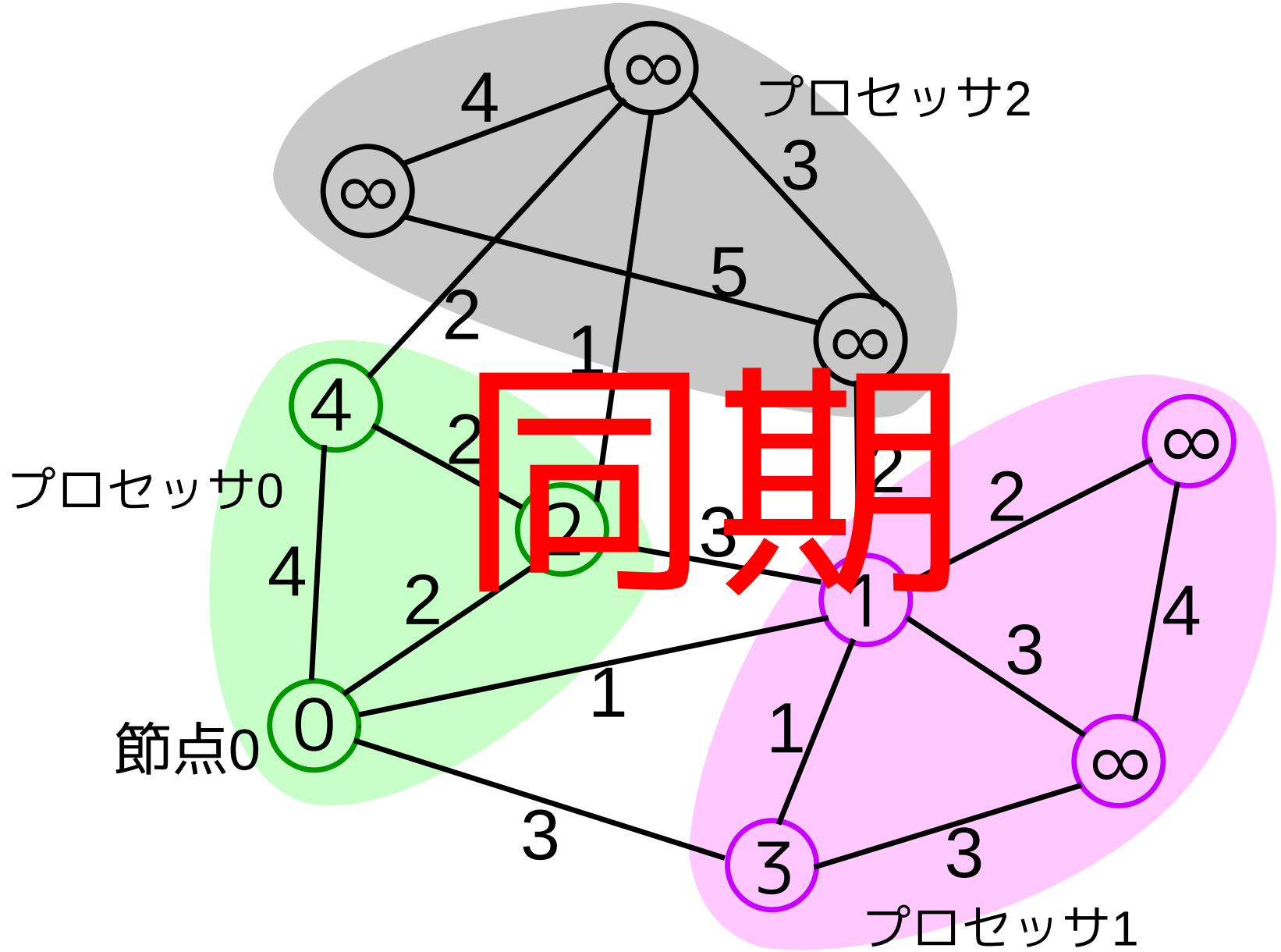
Web グラフの最短路計算：アルゴリズム (1)



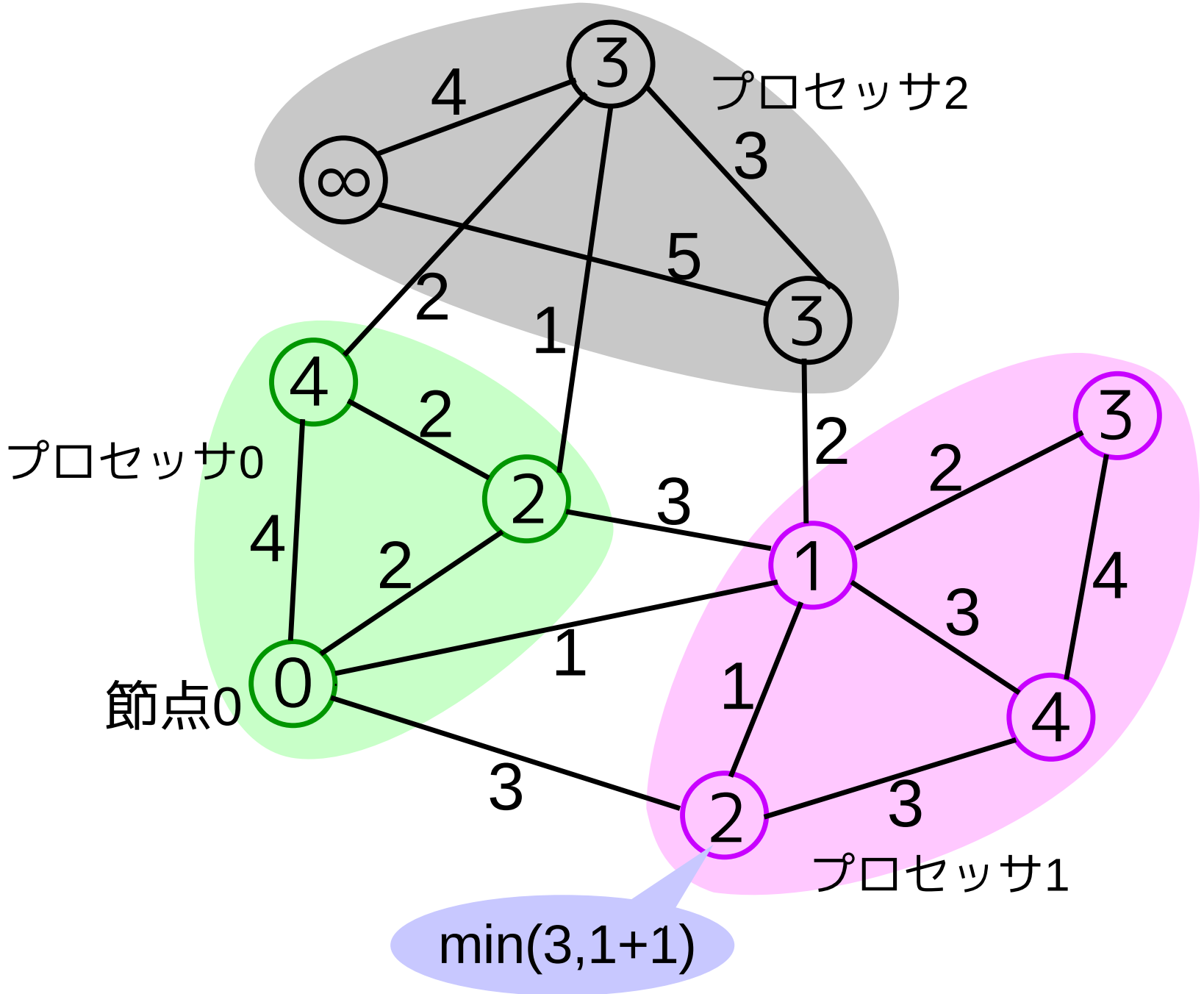
Web グラフの最短路計算：アルゴリズム (2)



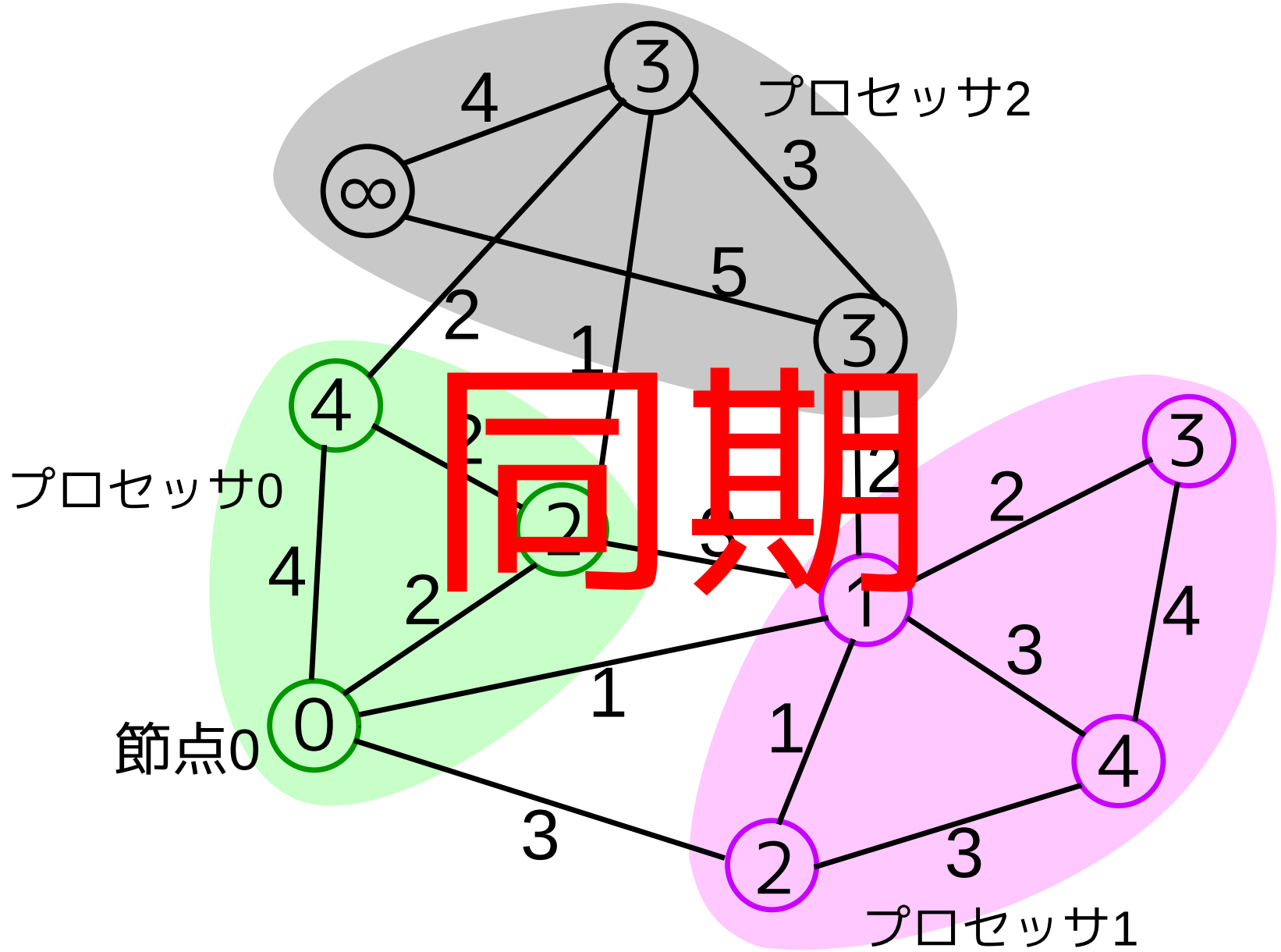
Web グラフの最短路計算：アルゴリズム (3)



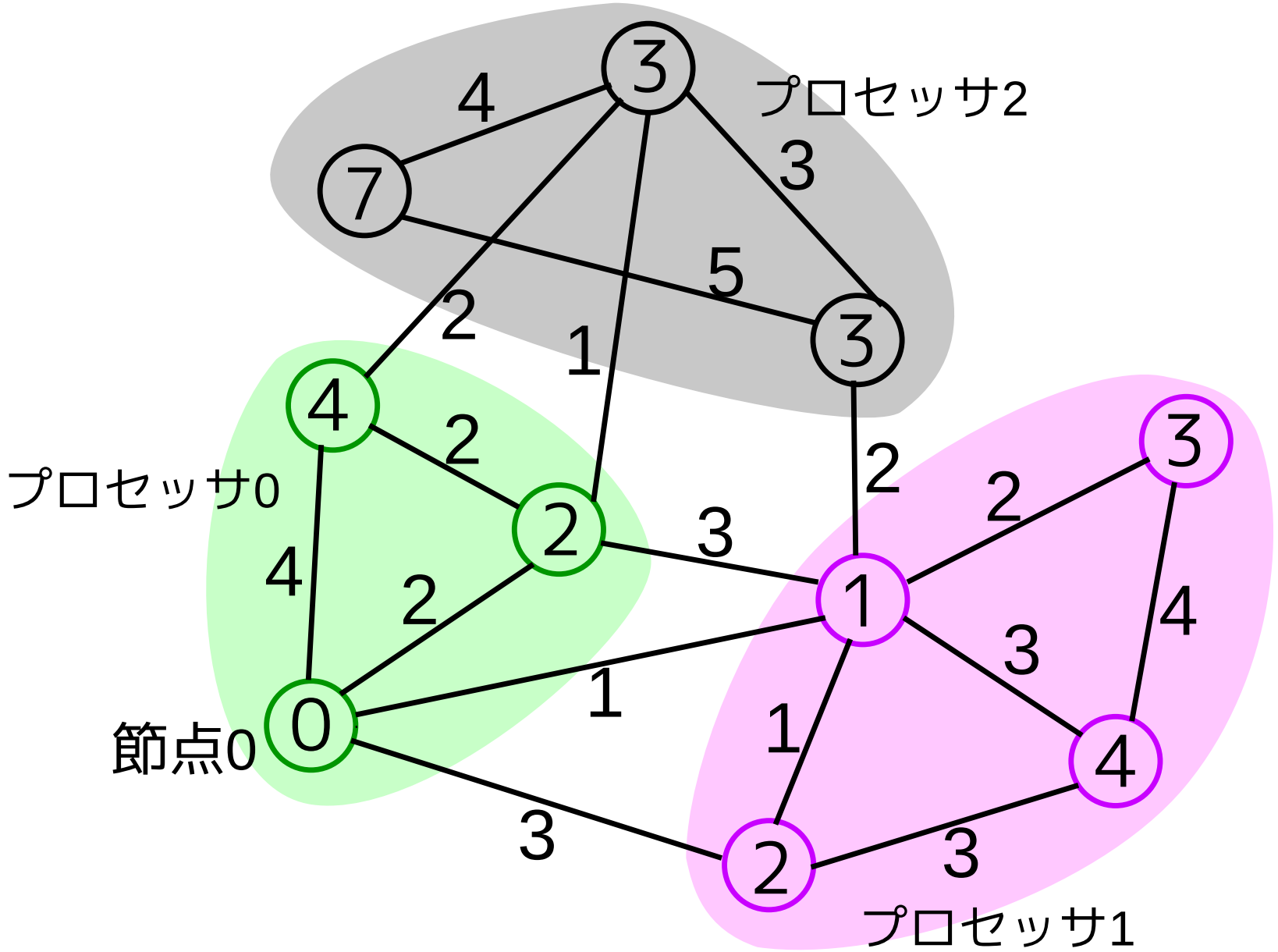
Web グラフの最短路計算：アルゴリズム (4)



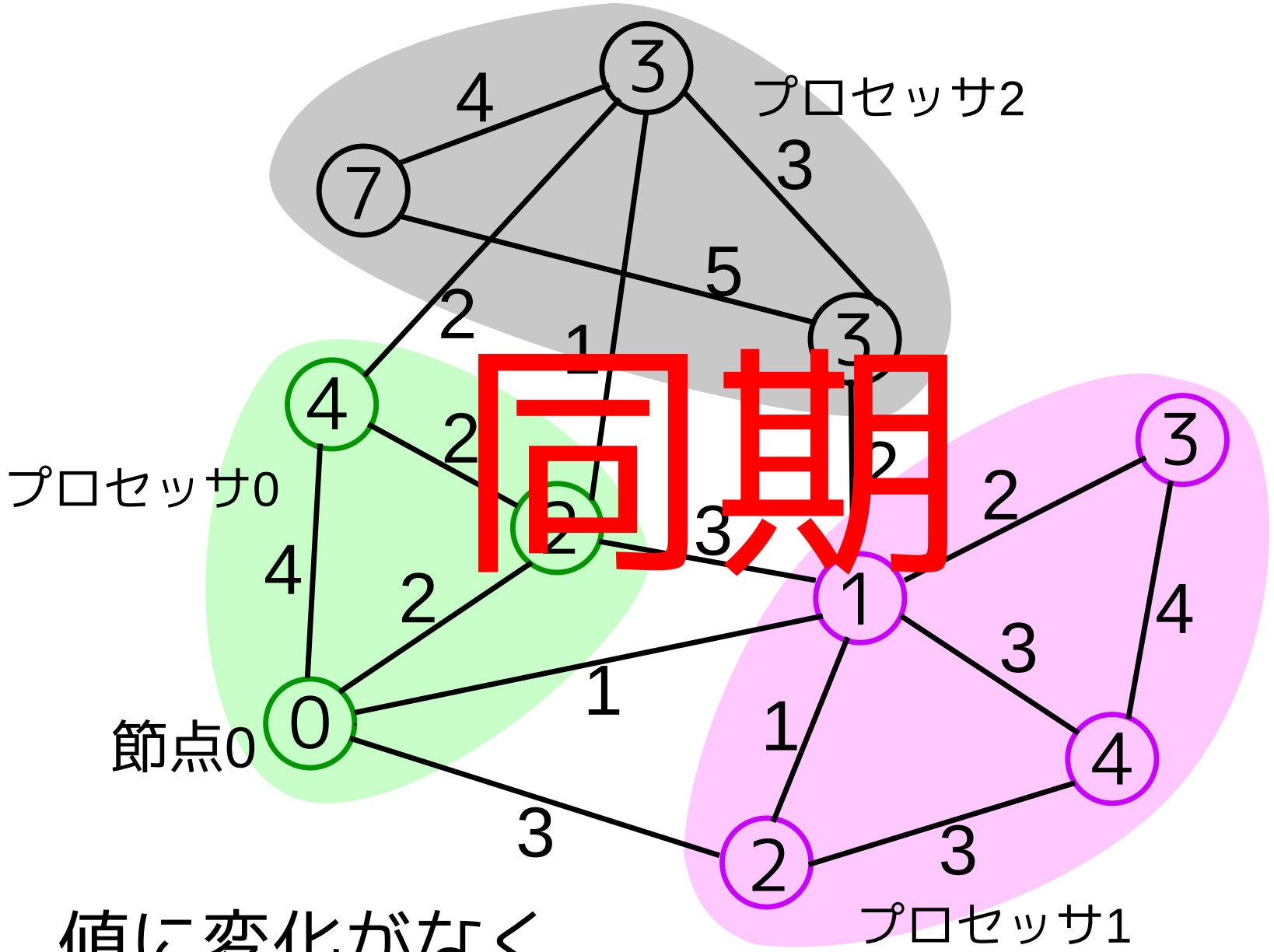
Web グラフの最短路計算：アルゴリズム (5)



Web グラフの最短路計算：アルゴリズム (6)



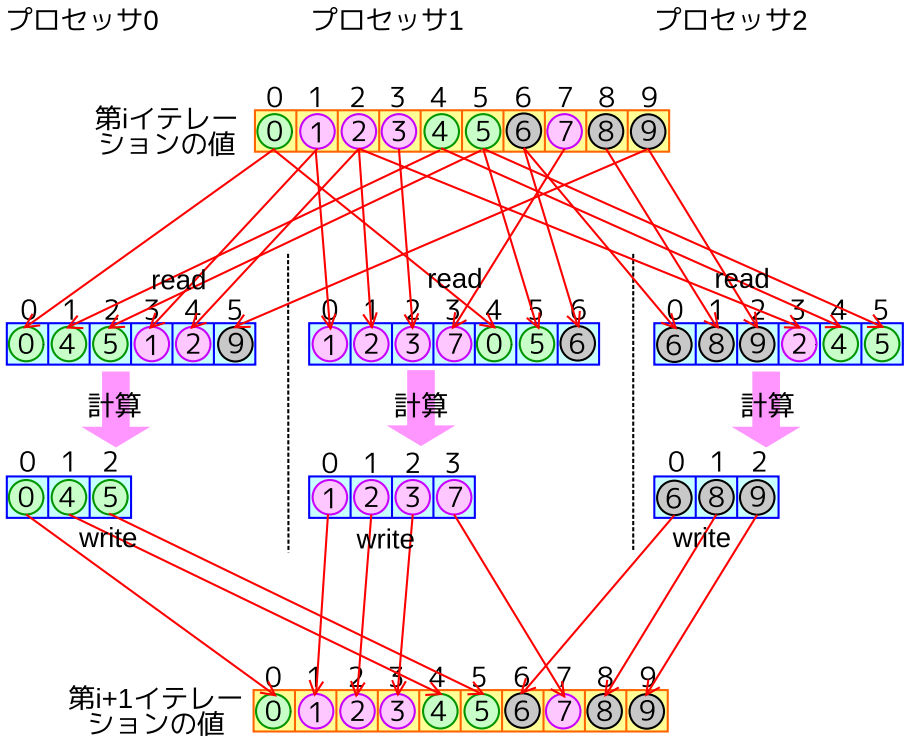
Web グラフの最短路計算：アルゴリズム (7)



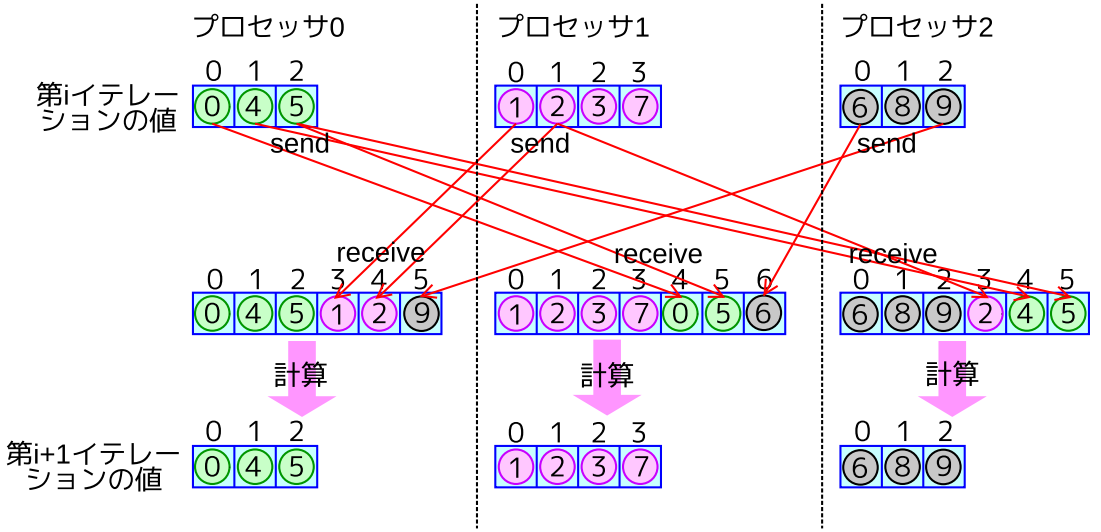
値に変化がなく
なったので終了

Web グラフの最短路計算：プログラマビリティ

DMI：645行

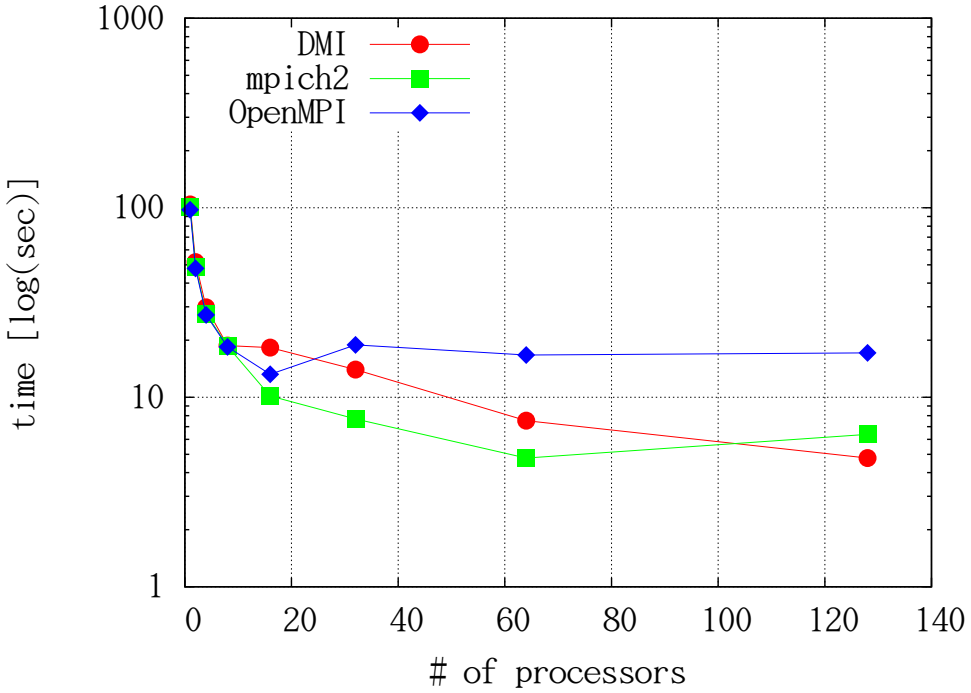


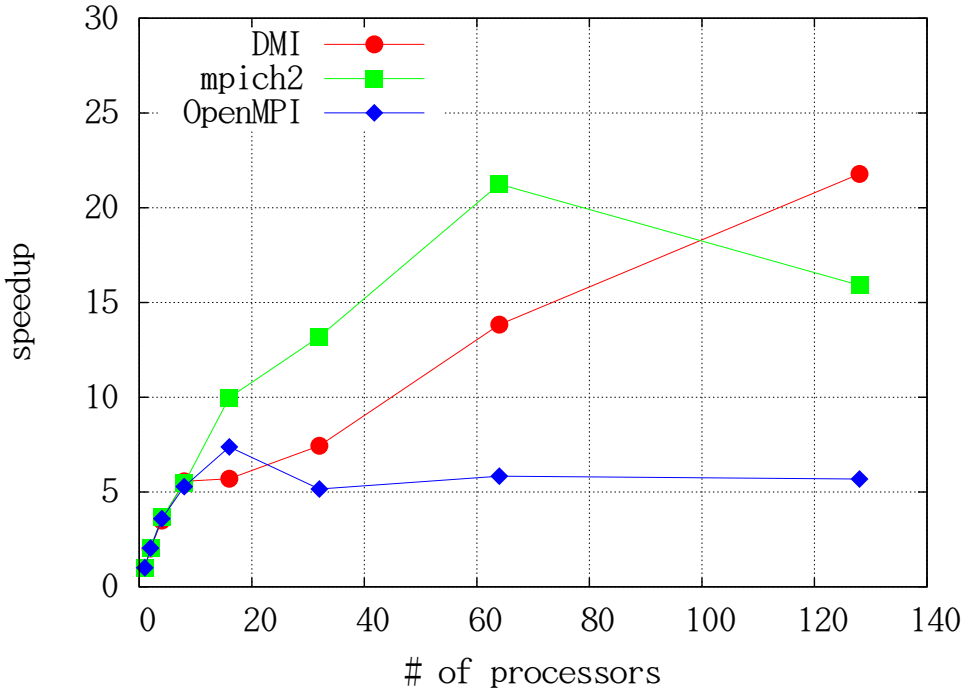
MPI：747行



- MPI では節点番号とローカルインデックスを対応付ける煩雑な計算が必要
- DMI では不要

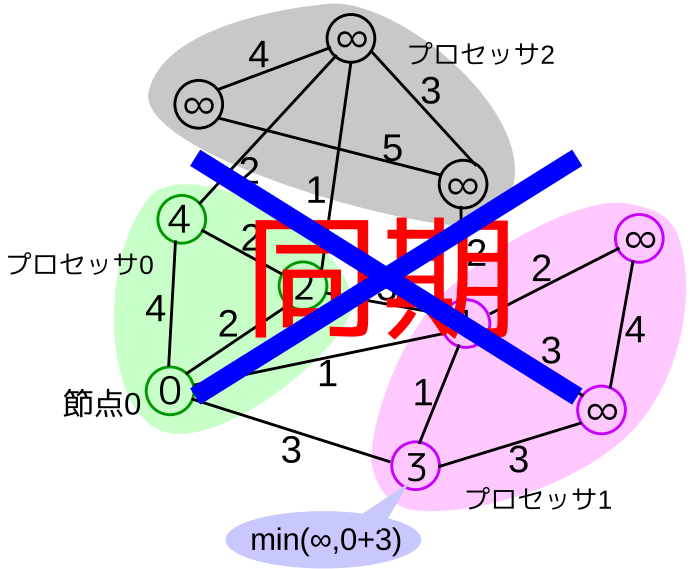
Web グラフの最短路計算：性能





- **DMI>mpich2>OpenMPI**
- mpich2 と OpenMPI の性能が低い理由は，密な全対全通信に弱いため
  - → 128 個の各プロセッサが自分以外の 127 個の各プロセッサに約 21.5KB を送信する

Web グラフの最短路計算：非同期的なアルゴリズム



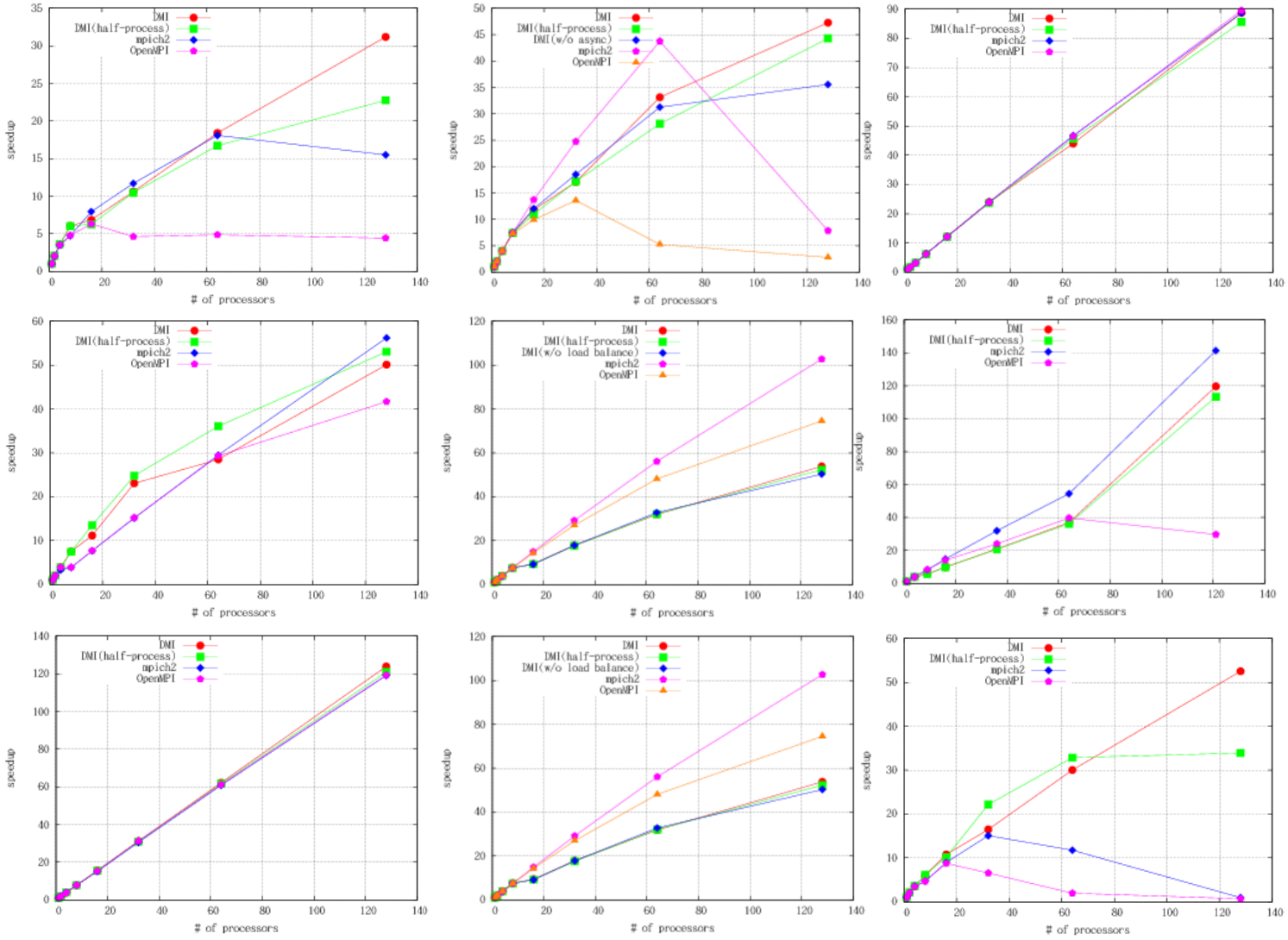
- 観察：演算子 min には可換性と結合性があるので，min の適用順序は結果に影響しない
  - → 1 イテレーションごとに同期する必要はない
  - → 各プロセッサは (周囲のプロセッサと同期することなく) 勝手なタイミングで節点の値を read しては最短路を更新
  - → 終了検知だけは「うまく」やる
- DMI では単方向通信 (read/write) が可能だからこそ記述できるアルゴリズム
  - → MPI の双方向通信 (send/receive) で記述するのは困難

Web グラフの最短路計算：128 プロセッサ実行時の性能

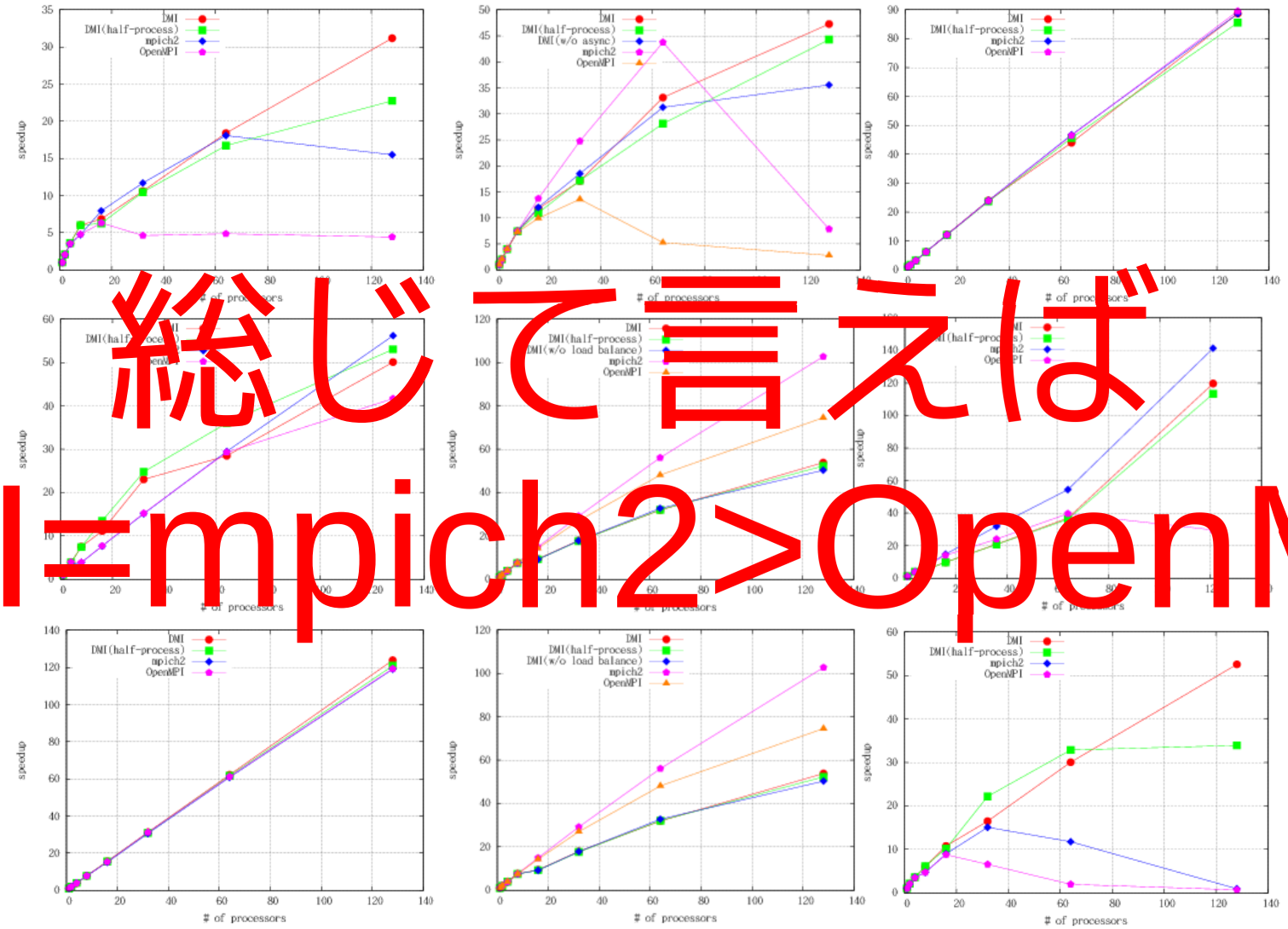
処理系	mpich2	OpenMPI	DMI(同期的)	DMI(非同期的)
時間 [sec]	31.2	105.4	21.8	**17.3**

- DMI の単方向通信を活かした非同期的なアルゴリズムによってさらに高速化できた

さまざまなアプリでの性能 (1)



非定型な並列計算のサポート [評価]
さまざまなアプリでの性能 (2)



総じて言えば

DMI≒mpich2>OpenMPI

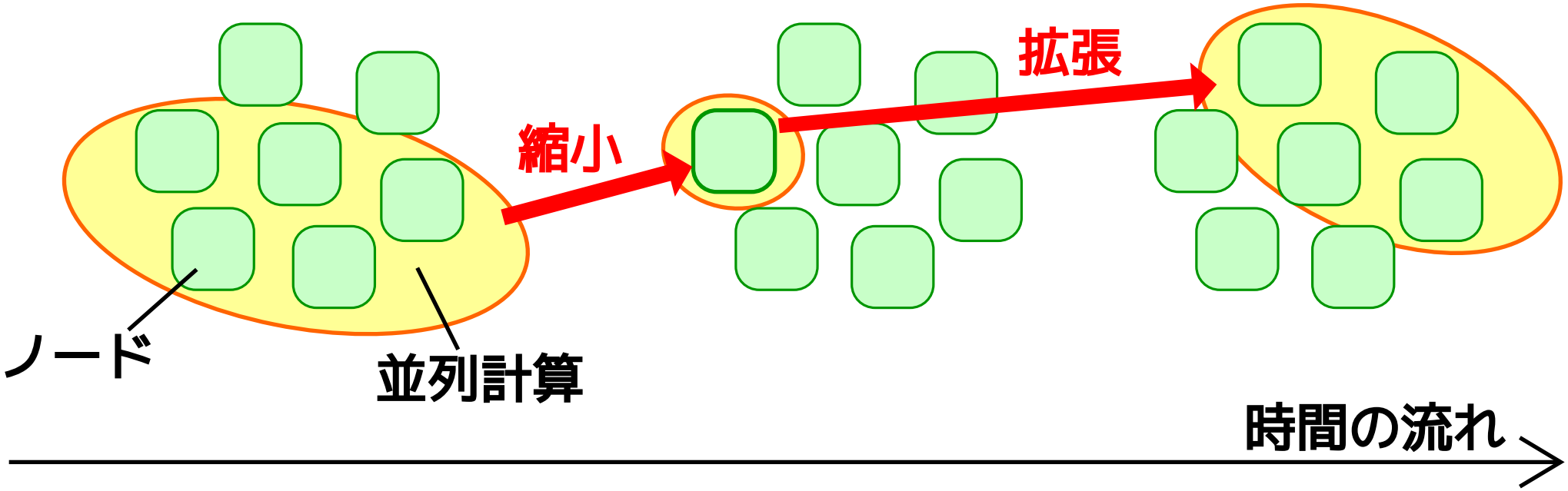
ここまでのまとめ

- 目的 I：MPI における「性能最適化」の強力さと「実行」時性能を妥協しない範囲で，できるかぎりプログラマビリティを高める
  → グローバルアドレス空間モデルを採用
- グローバルアドレス空間への read/write をわかりやすく強力に最適化できるAPI を設計
- 性能：DMI=mpich2>OpenMPI
- 非定型な並列計算に対するプログラマビリティ：DMI>MPI

❖ 再構成可能な並列計算のサポート [関連研究]

目的の確認

➤ 目的Ⅱ：**再構成可能な並列計算を簡単に書ける**ようにする



何を「単位」として再構成すべきか?

- 問：何を「単位」として再構成すべきか?
  - → 候補：仮想マシン，プロセス，スレッド
- 観察：
  - → 再構成 (= 計算の移動) のコストは，移動する「単位」の資源消費量に比例
  - → 資源消費量は，仮想マシン > プロセス > スレッド
  - → たいていの並列科学技術計算は，プロセス/スレッドのレベルで処理が完結しているので，OS レベルでの移動までは要求されない [Chaudhart et al,2006]
    - ♦ 仮想マシンの移動は「必要以上に重すぎる」
    - ♦ ex：Amazon EC2，Windows Azure
- 結論：プロセス/スレッドを「単位」とした再構成が適切

プロセス/スレッドを「単位」とした再構成

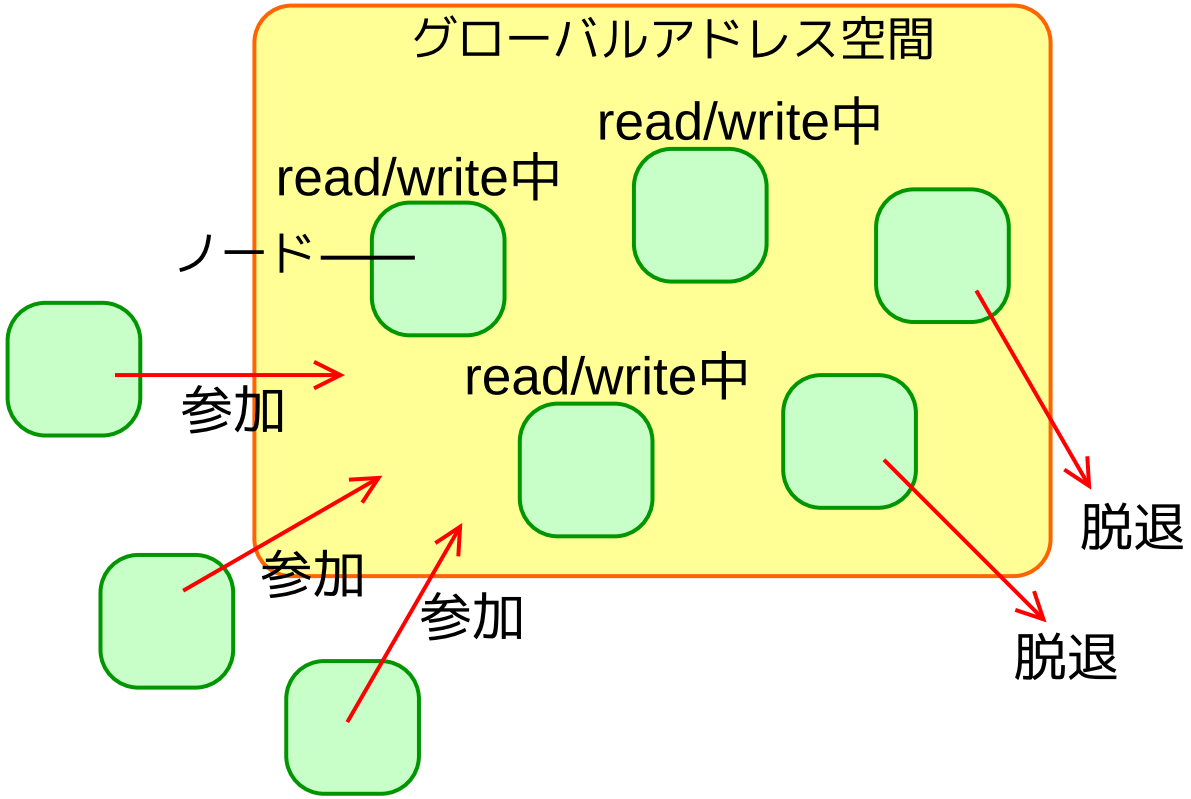
- 既存研究：
  - MapReduce[Jeffrey et al,2004]：プロセス間の密な通信を必要とする並列計算には不適
  - Satin[Rob et al,2000]：分割統治型の並列計算に特化
  - 研究 [Sriram et al,2005]，MPI-Mitten[Cong et al,2006]：MPI のチェックポイント/リスタート
  - Adaptive MPI[Chao et al,2003]，SRS[Sathish et al,2003]，DyRecT [Etienne,2000]，DRMS[Vijay,1997]，PCM[Kaoutar et al,2005]：MPI の再構成
  - ...
- (MPI よりもプログラマビリティの高い) グローバルアドレス空間モデルに基づいて再構成を実現した例は存在しない

❖ 再構成可能な並列計算のサポート [設計]

設計の全体像

(1) ノードが自由なタイミングで参加/脱退できるようなグローバルアドレス空間のコヒーレンシプロトコルを設計

→ 新規的 (説明略)

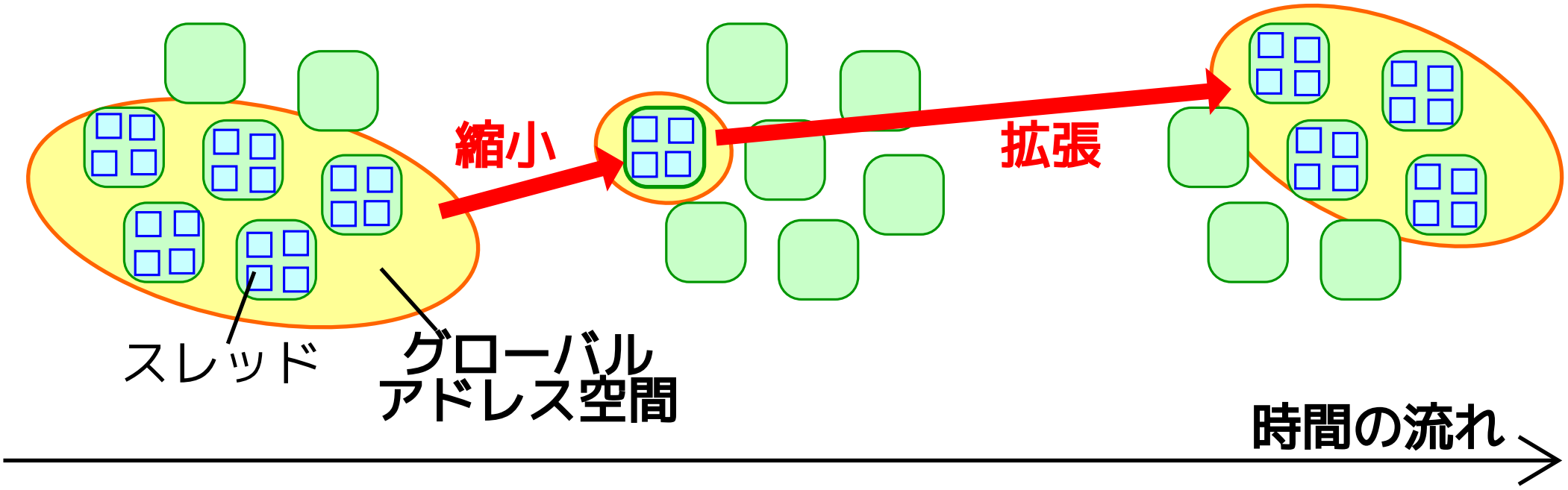


(2) 再構成可能な並列計算のための簡単なプログラミングモデルを設計

→ やり方 1：スレッド増減に基づくモデル

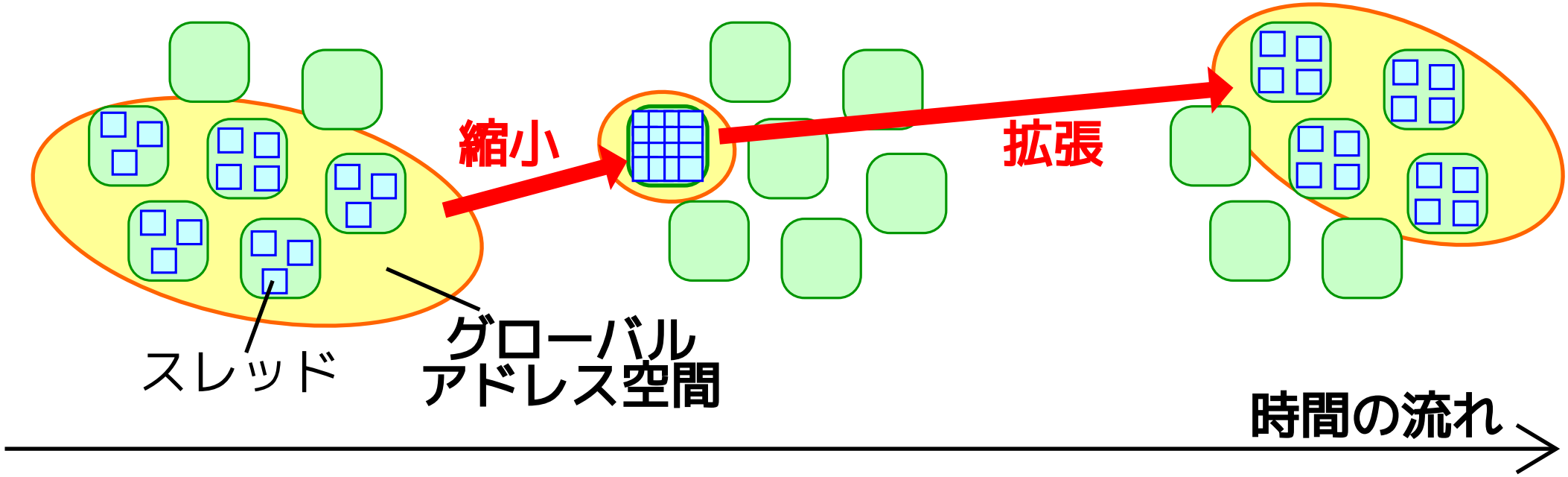
→ やり方 2：スレッド移動に基づくモデル

やり方 1：スレッド増減に基づくモデル



- モデル：再構成のたびに 1 プロセッサあたり 1 スレッドとなるようにスレッドを増減させる
- 特徴：
  - → 性能◎：1 プロセッサあたり 1 スレッド
  - → プログラマビリティ ×：(途中で実行スレッド数が変わるので) データのチェックポイント/リスタートのためのコードをプログラマに書かせざるをえない

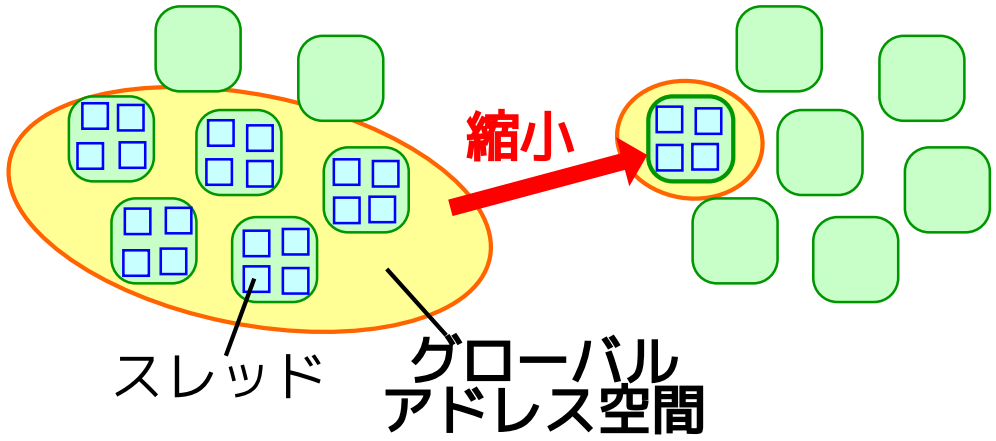
やり方 2：スレッド移動に基づくモデル



- モデル：プログラマは単に大量のスレッドを生成しておけばよく，あとは処理系が，スレッド移動によって，それら大量のスレッドを利用可能なノードにマッピングしてくれる
- 特徴：
  - → プログラマビリティ◎：プログラマが再構成を意識する必要がない
  - → 性能△：1 プロセッサあたり多スレッドを割り当てることのオーバヘッド

提案する 3 種類のプログラミングモデル

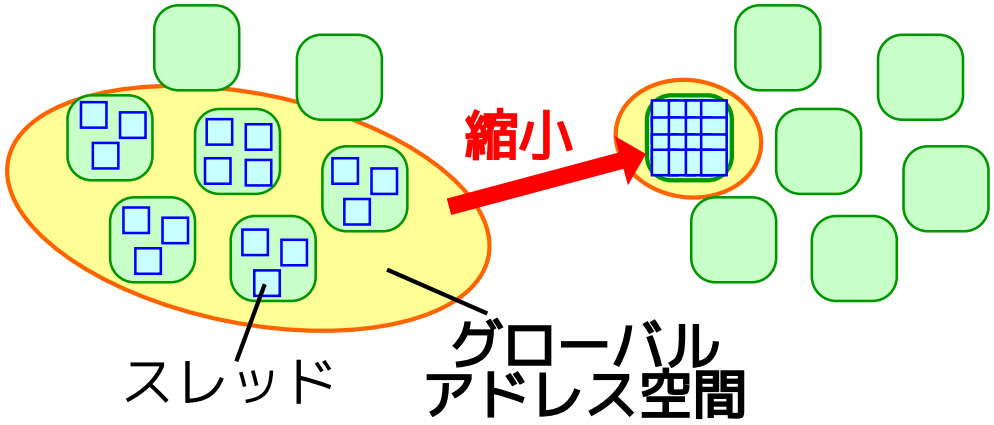
スレッド増減に基づくモデル



1.スレッド増減

性能◎
プログラマビリティ×

スレッド移動に基づくモデル



2.スレッド移動

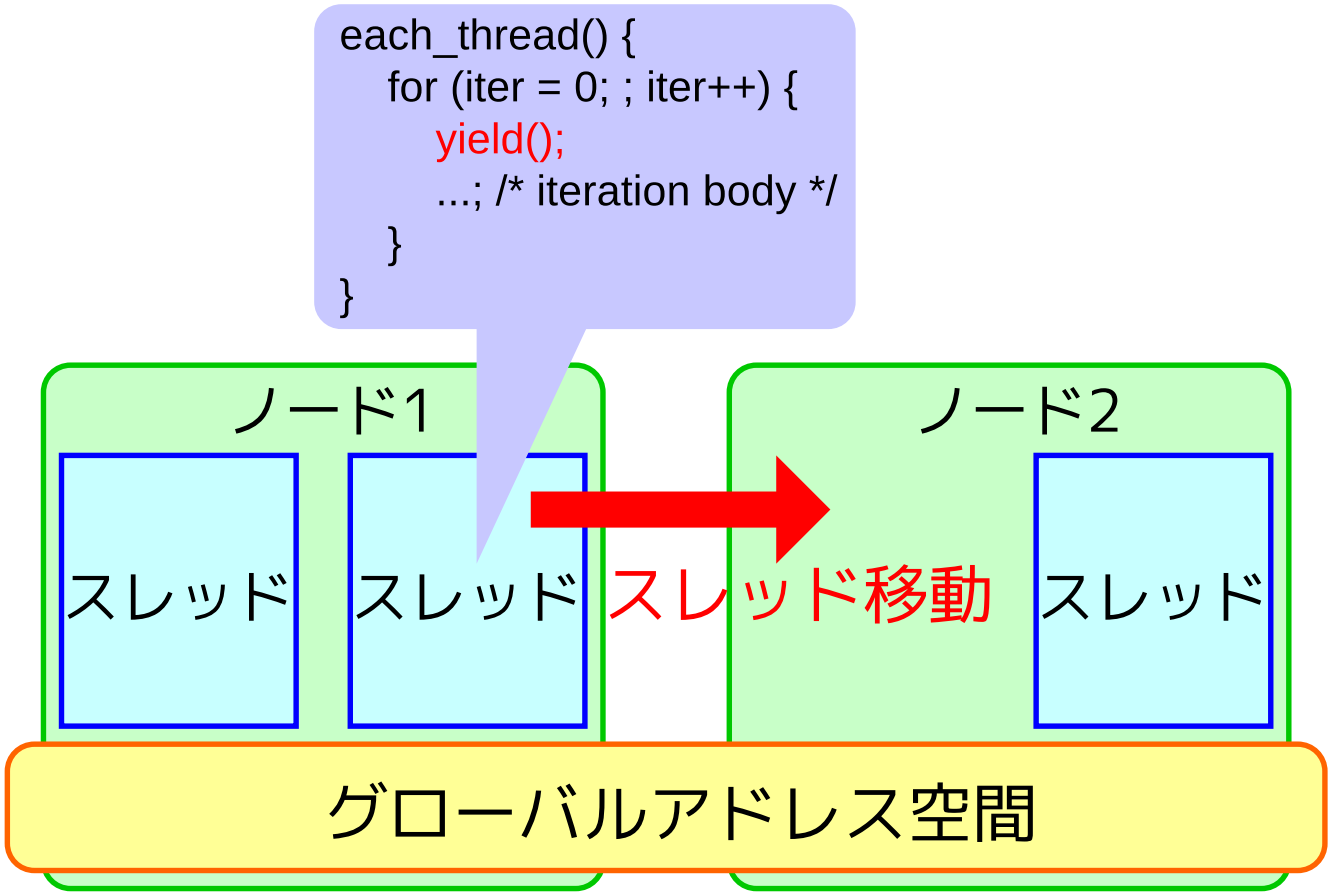
性能△
プログラマビリティ◎
ただし「制約」あり

3.スレッド移動
with half-process

性能△
プログラマビリティ◎

➤ **以降では，「スレッド移動」における「制約」が何なのかを指摘したうえで，「スレッド移動** with half-process**」について説明**

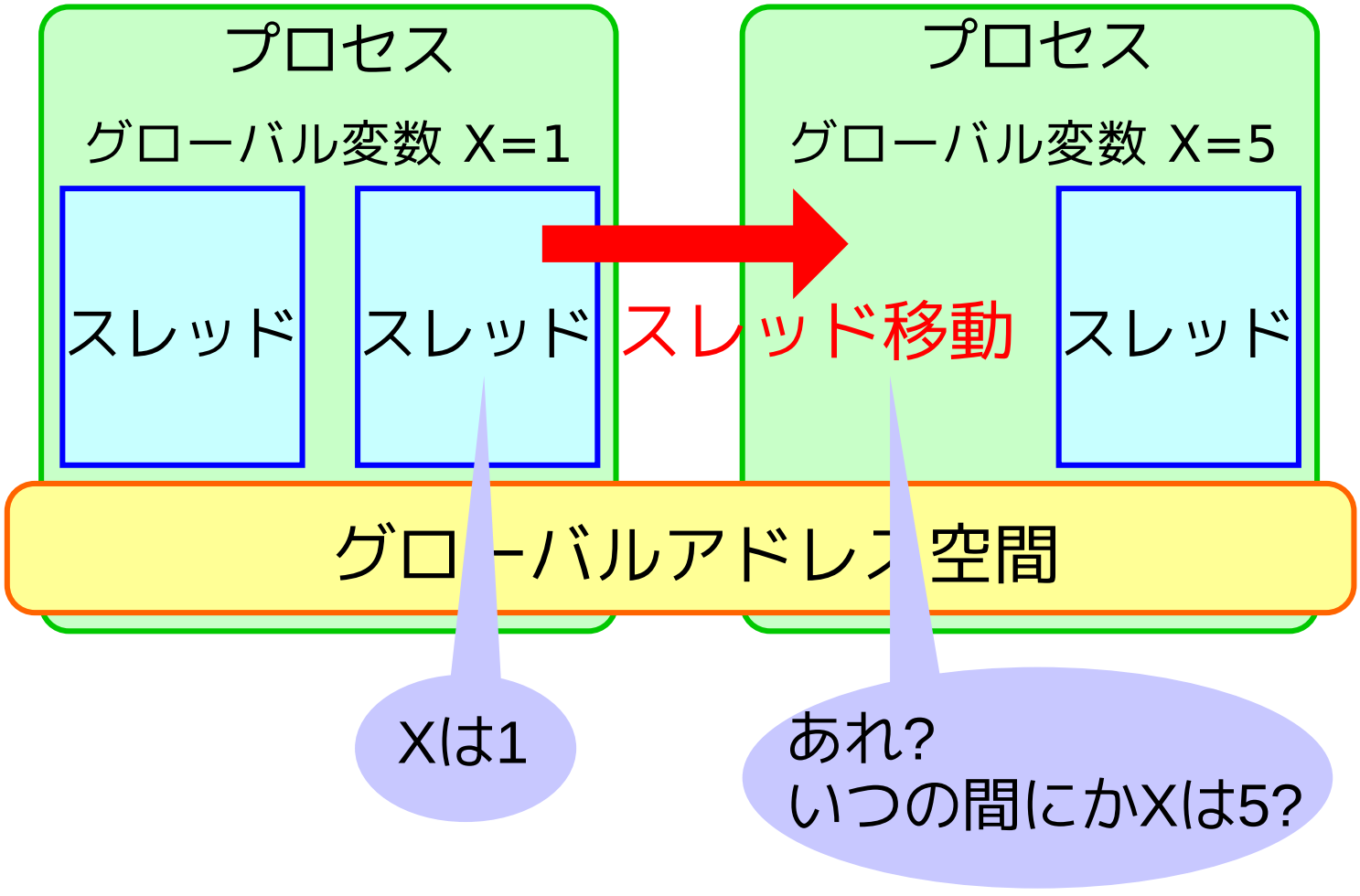
スレッド移動のプログラミングモデル



- プログラマ視点で見えるものは 2 つ：
  - → 各スレッド固有のアドレス空間
  - → グローバルアドレス空間
- yield() 関数を書いておくだけで，(スレッド移動の必要があれば) その「中」で透過的にスレッド移動が起きる

スレッドで実装すると何が起きるか?

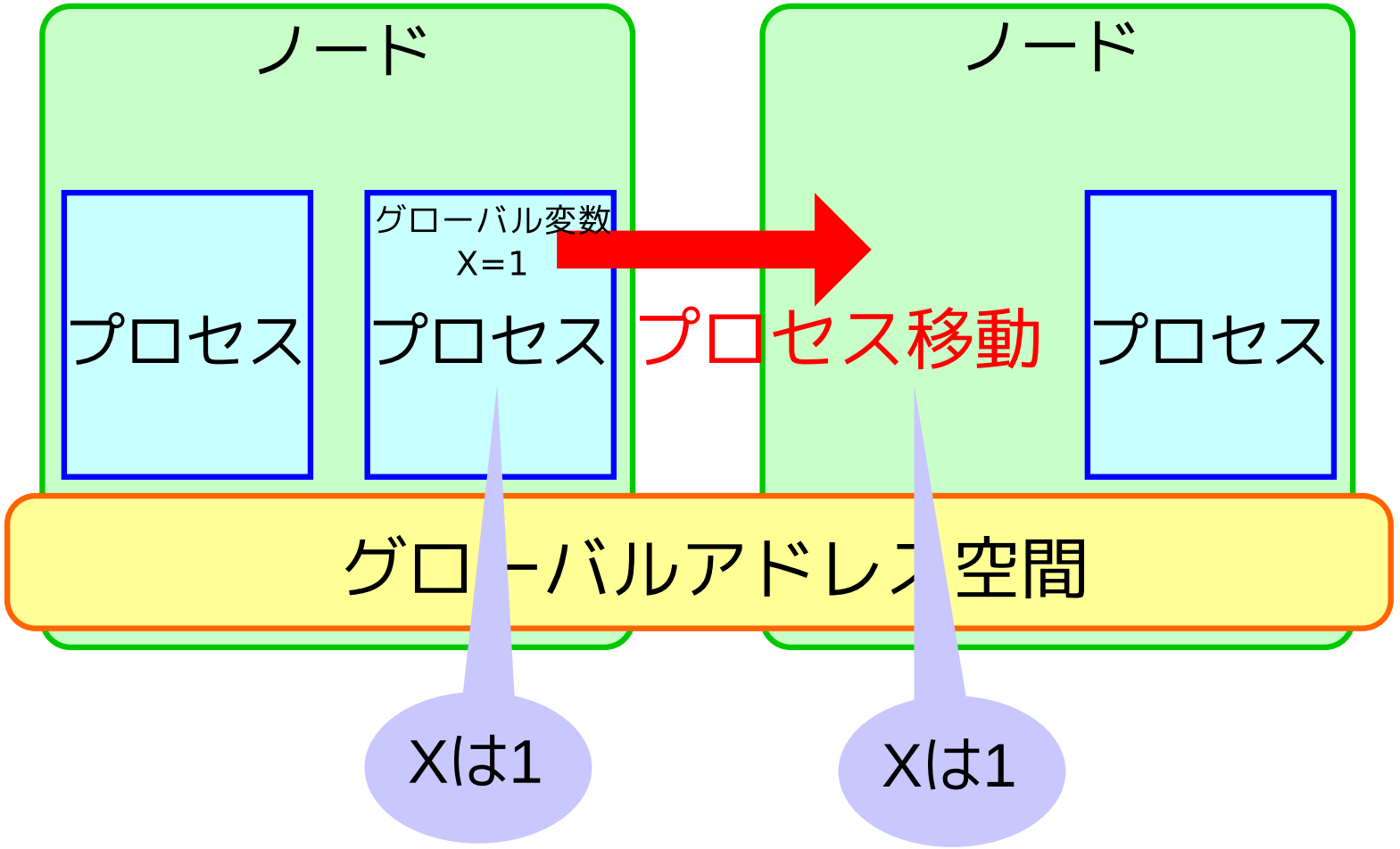
- プログラマ視点では「各スレッド固有のアドレス空間」が見えている必要がある
- しかし，実際にはスレッドどうしはアドレス空間を共有している
- プログラマがグローバル変数を使うと「変なこと」が起きる



既存の処理系における解決策

- スレッド移動の安全性を保証するために，各スレッドが使ってもいいメモリ領域に「制約」を加える[Gabriel et al,1999],[Chao et al,2003],[Bozhidar,1998],[Hai,2002]
  - → ex：そもそもグローバル変数は使用禁止!
    - よって，グローバル変数を使用しうるライブラリも使用禁止
    - printf()，malloc()，sin()，cos() も (理論上は) ダメ
- 問題点：「制約」は不便
- 「制約」を撤廃させましょう!

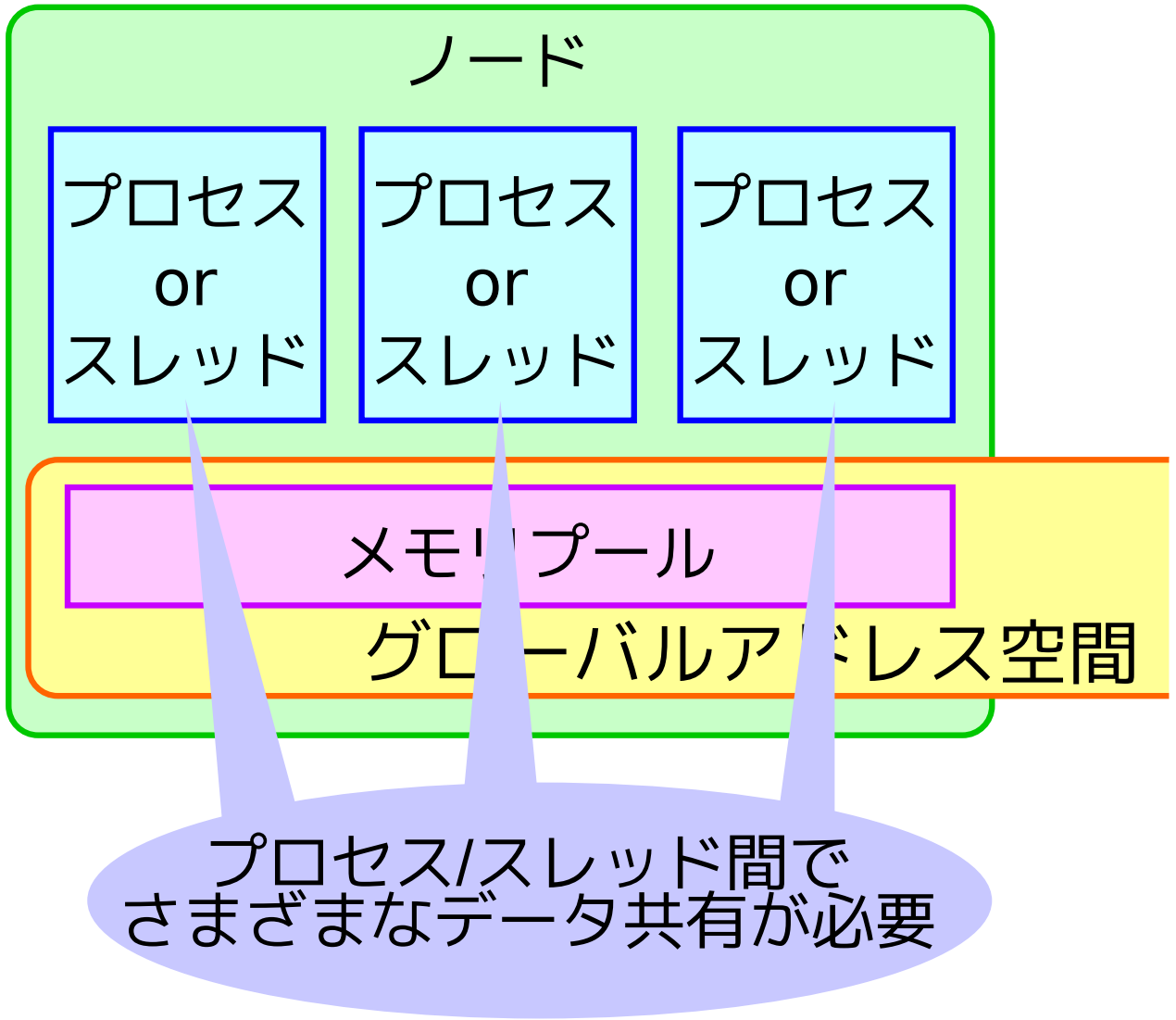
案：プロセスを使うのはどうか?



- スレッドのかわりにプロセスを使う
  - → プロセスどうしはアドレス空間が独立しているので，各プロセスが使ってもいいメモリ領域に関する「制約」は不要に
  - → 解決???

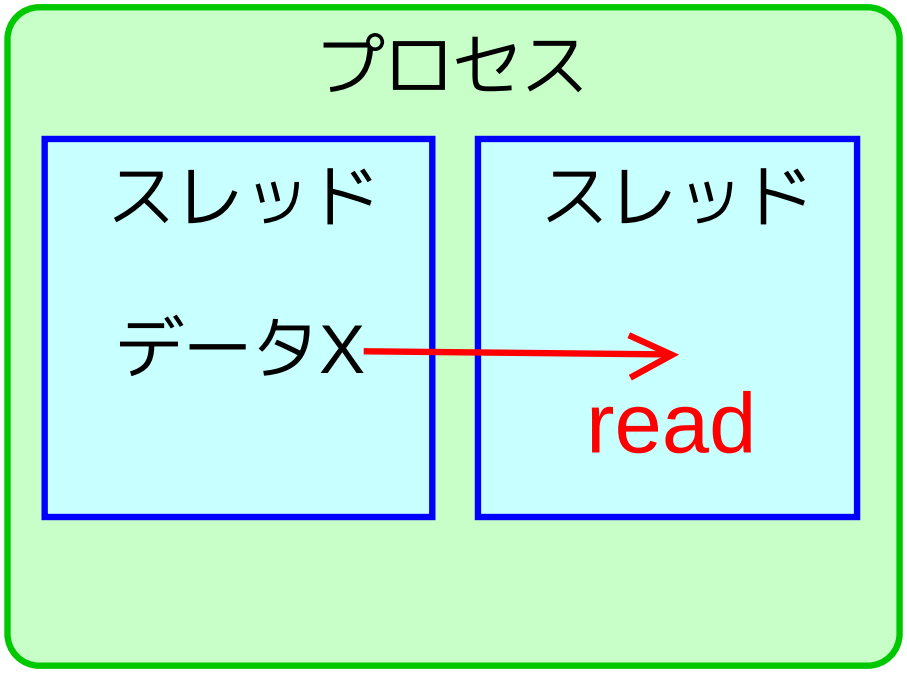
プロセスを使うことの問題点 (1)

- ノード内のプロセス/スレッドたちはメモリプールやページテーブルなどたくさんのデータを共有する必要がある

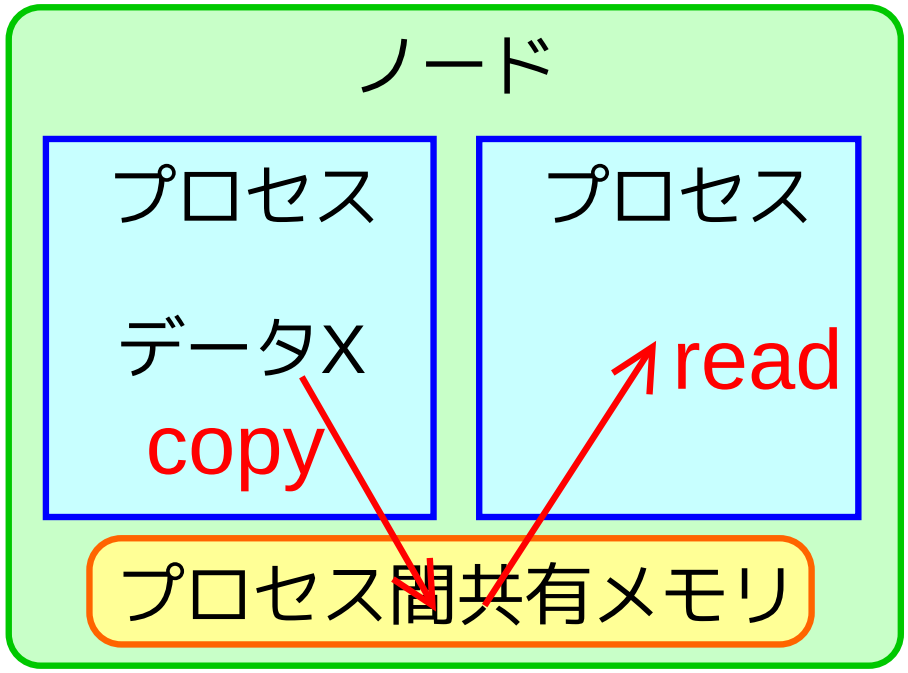


プロセスを使うことの問題点 (2)

- **問題点１：プロセスどうしのデータ共有は，スレッドどうしのデータ共有よりも<u>オーバヘッドが大きい</u>**



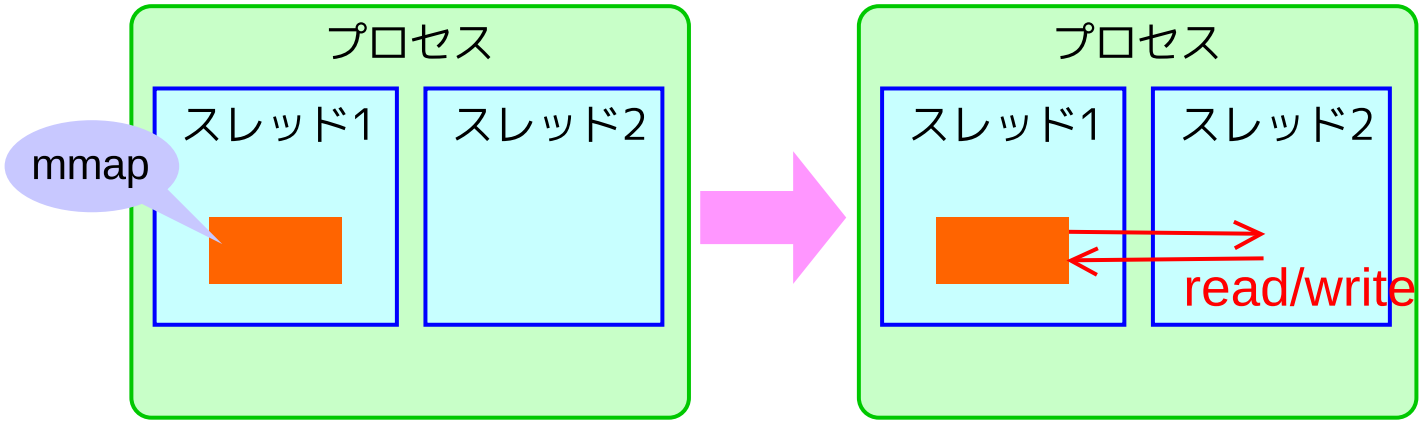
他のスレッドのデータを
直接read/writeできる



いったんプロセス間共有メモリに
コピーする必要あり

プロセスを使うことの問題点 (3)

- **問題点 2：プロセス間共有メモリ上のデータ共有は，スレッドどうしのデータ共有よりも<u>プログラミングしにくい</u>**

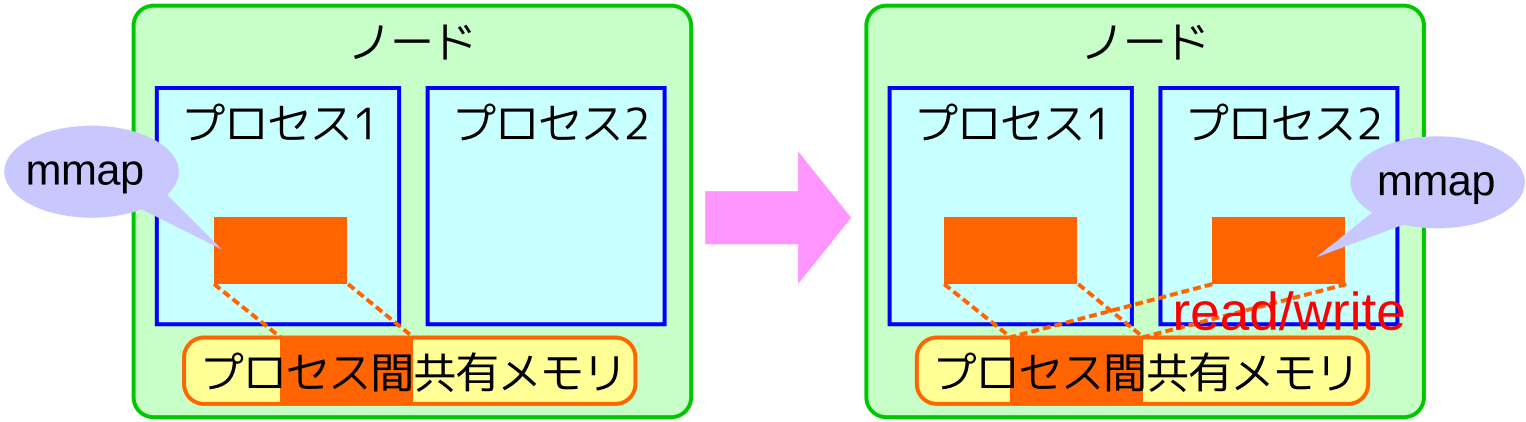


スレッド1がメモリ確保したら...

「自然と」スレッド2から
read/writeできる

スレッド間では，
メモリのメタ操作
(mmap,mprotect,...)
が共有されている

スレッドプログラミングが
「簡単な」理由の1つ

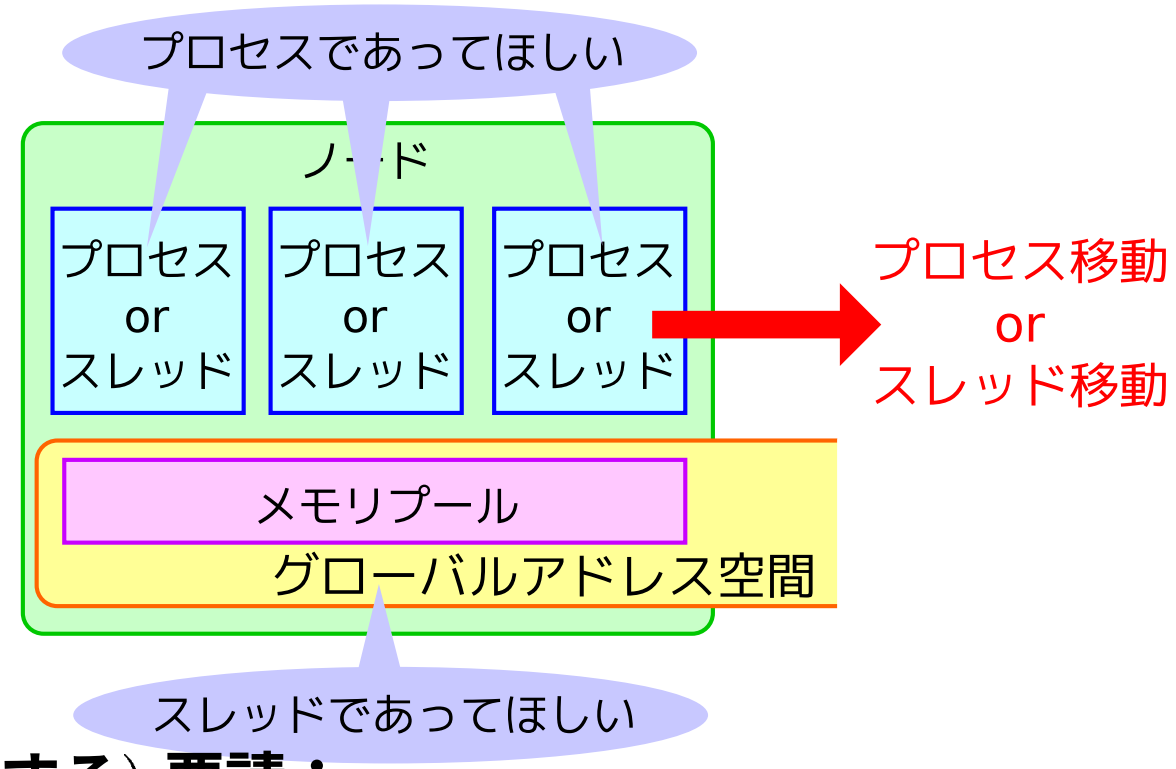


プロセス1がメモリ確保しても...

「プロセス2もメモリ確保しないかぎり」
プロセス2はread/writeできない

プロセス間では，
メモリのメタ操作
(mmap,mprotect,...)
が共有されない

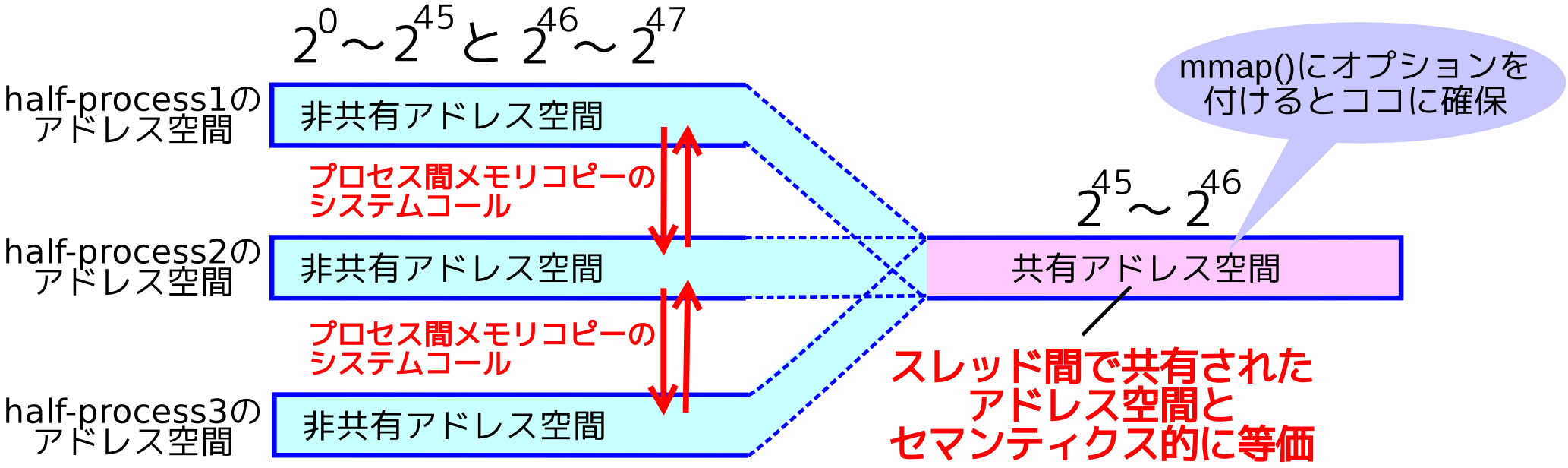
要件を整理すると



- (本質的に矛盾する) 要請：
  - → プログラマ視点では「各スレッド固有のアドレス空間」が見えている必要がある → プロセスにしたい
  - → データ共有を性能よく簡単に実現したい → スレッドにしたい
- 言い換えると：
  - → プログラマが触る部分 → プロセスにしたい
  - → 処理系が触る部分 → スレッドにしたい

新しいカーネルプリミティブの提案

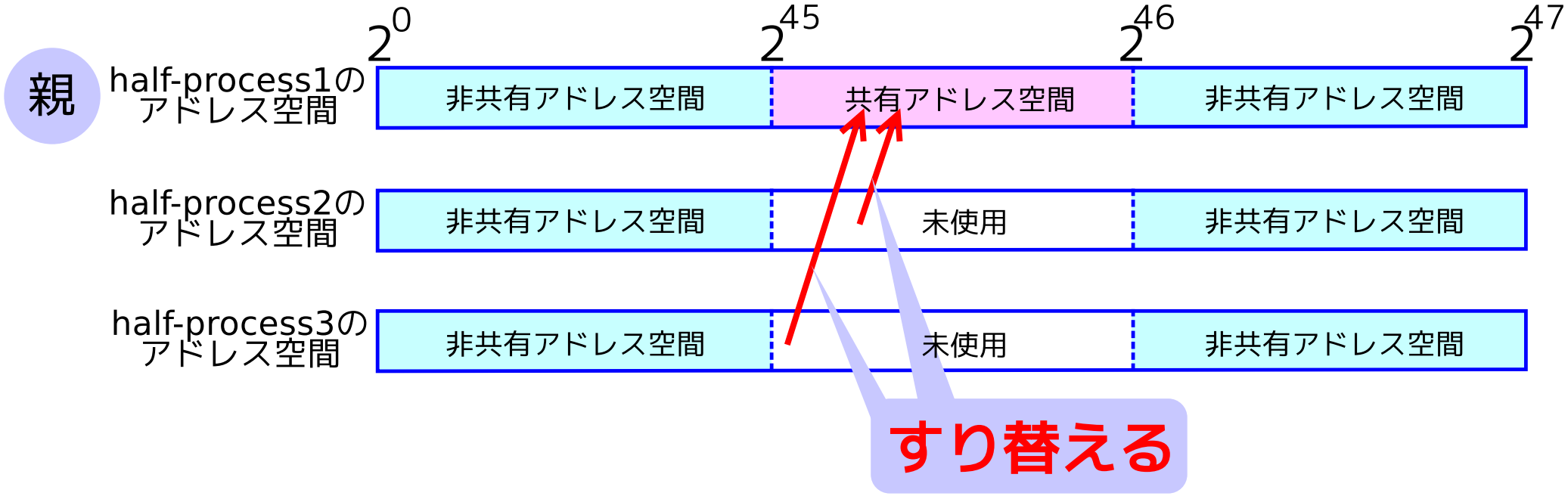
- **half-process**：スレッドとプロセスの「中間」



- **デフォルトでは普通のプロセス**で非共有アドレス空間だけを使う
- mmap() にオプションを付けると共有アドレス空間にメモリを確保できる

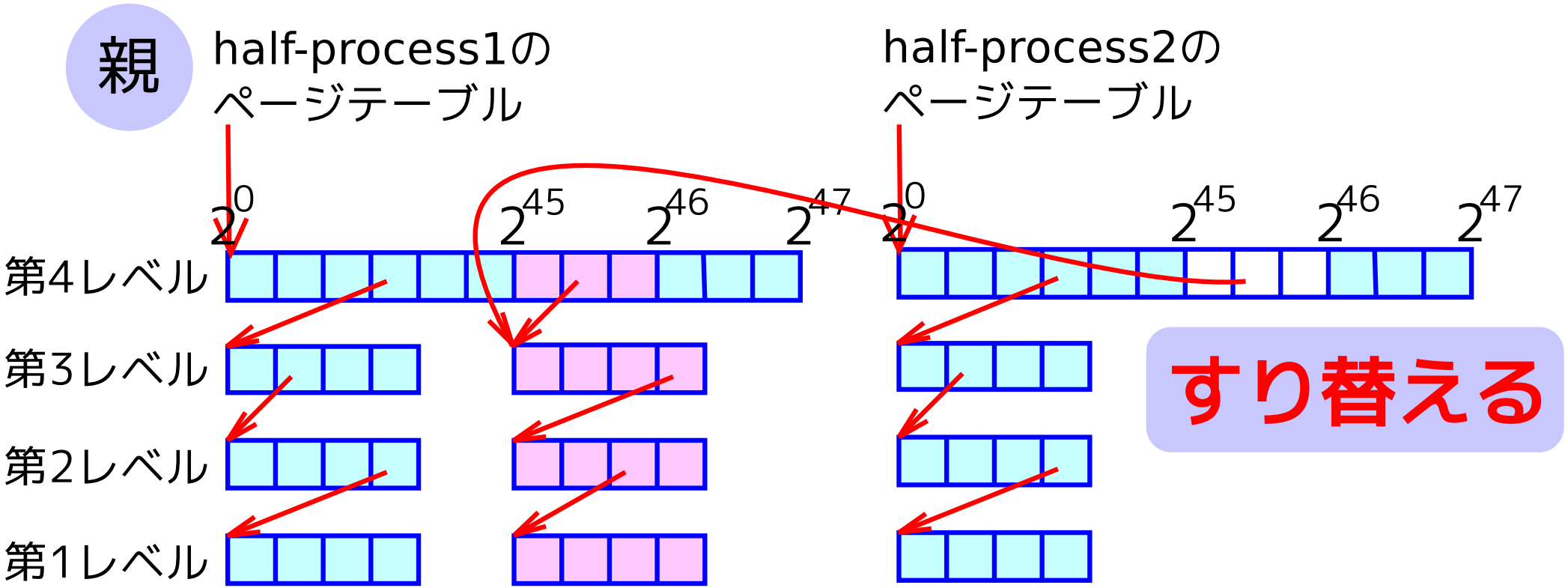
half-process の実装 (1)

- 各 half-process はそれぞれのアドレス空間を持っている
- 親 half-process を 1 個決めておく
- 共有アドレス空間に対するメモリメタ操作 (mmap,mprotect,...) が発行されるたびに，その操作の対象となるアドレス空間を親 half-process のアドレス空間にすり替える

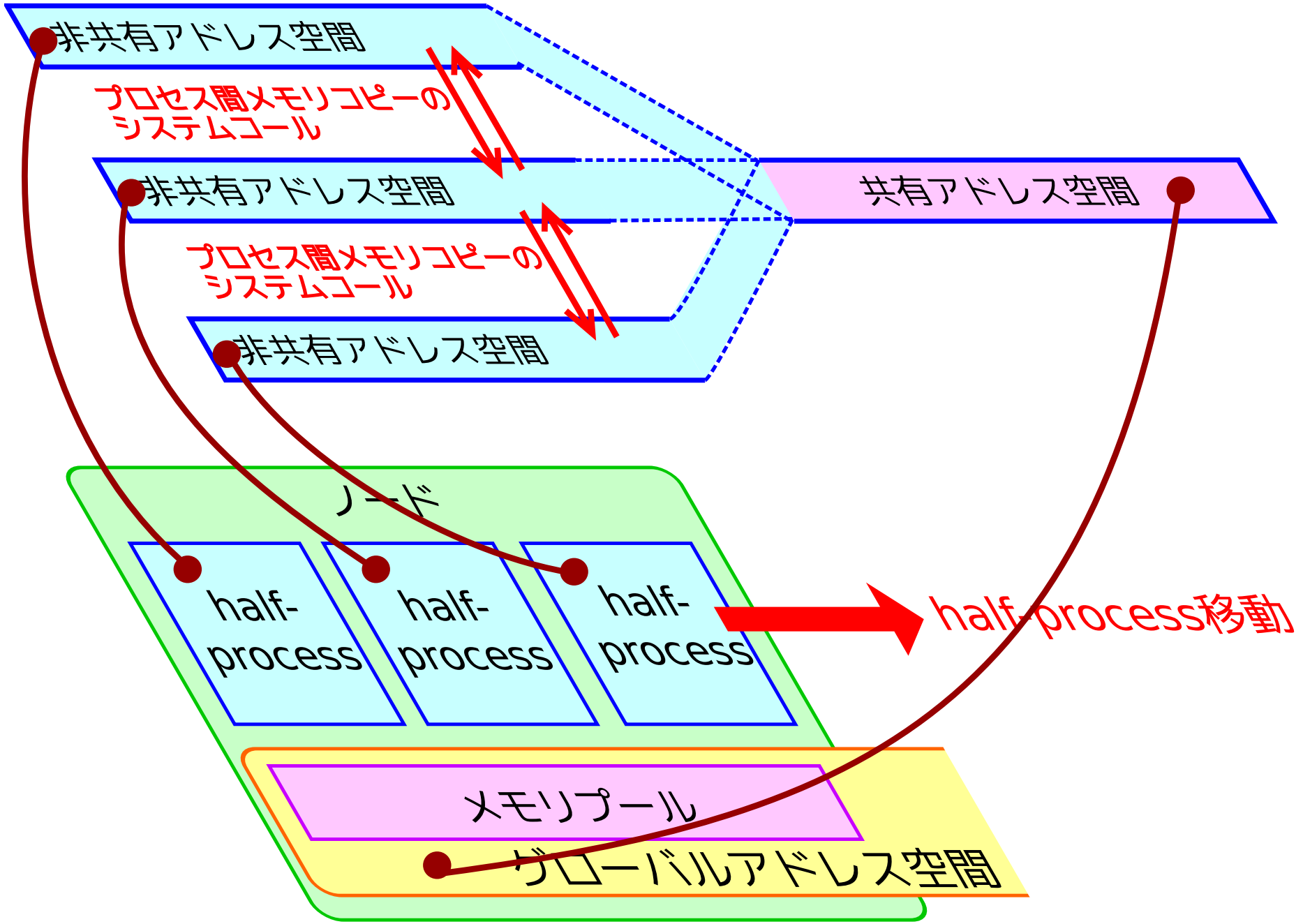


half-process の実装 (2)

- 共有アドレス空間に対するページフォルトが起きるたびに，各 half-process のページテーブルエントリを親 half-process のページテーブルエントリにすり替える



スレッド移動 with half-process



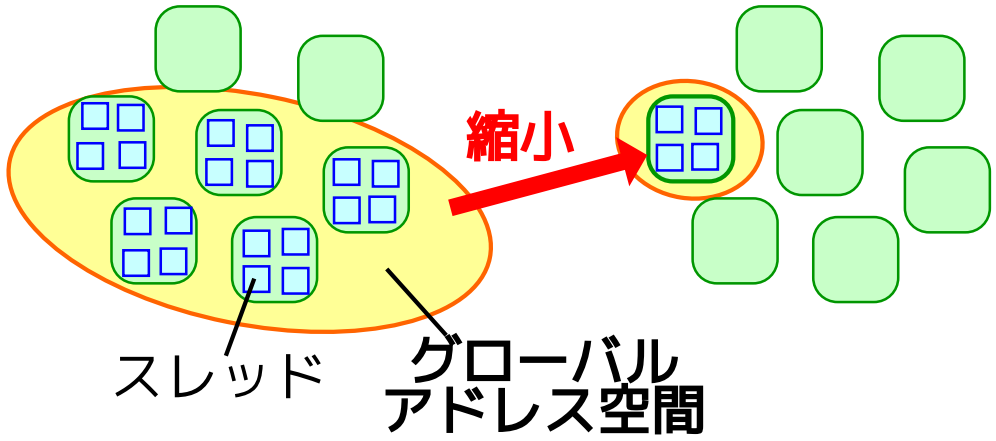
half-process の潜在的な応用可能性

- そもそも half-process は，プロセス間のデータ共有を (既存のプロセス間共有メモリよりも) 性能よく簡単に実現するための汎用的なカーネルプリミティブ
- 応用可能性：
  - → スレッド移動
  - → スレッドアンセーフなライブラリをスレッドプログラミングで使う
  - → 柔軟なハイブリッドプログラミング
  - → 並列言語処理系の開発者の負担減
  - → ...

❖ 再構成可能な並列計算のサポート [評価]

3 種類のプログラミングモデルを比較

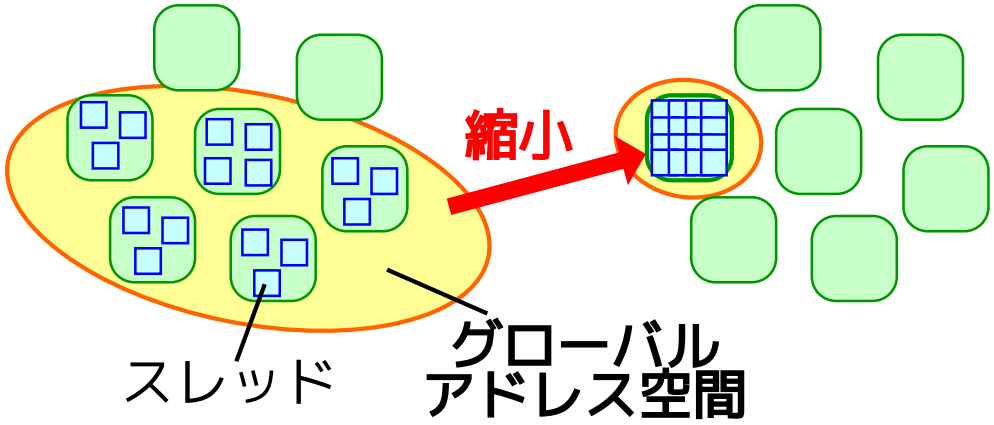
スレッド増減に基づくモデル



1.スレッド増減

性能◎
プログラマビリティ×

スレッド移動に基づくモデル



2.スレッド移動

性能△
プログラマビリティ◎
ただし「制約」あり

3.スレッド移動
with half-process

性能△
プログラマビリティ◎

プログラマビリティの比較

➤ **再構成を行わないプログラムを再構成可能にするために必要な変更行数**

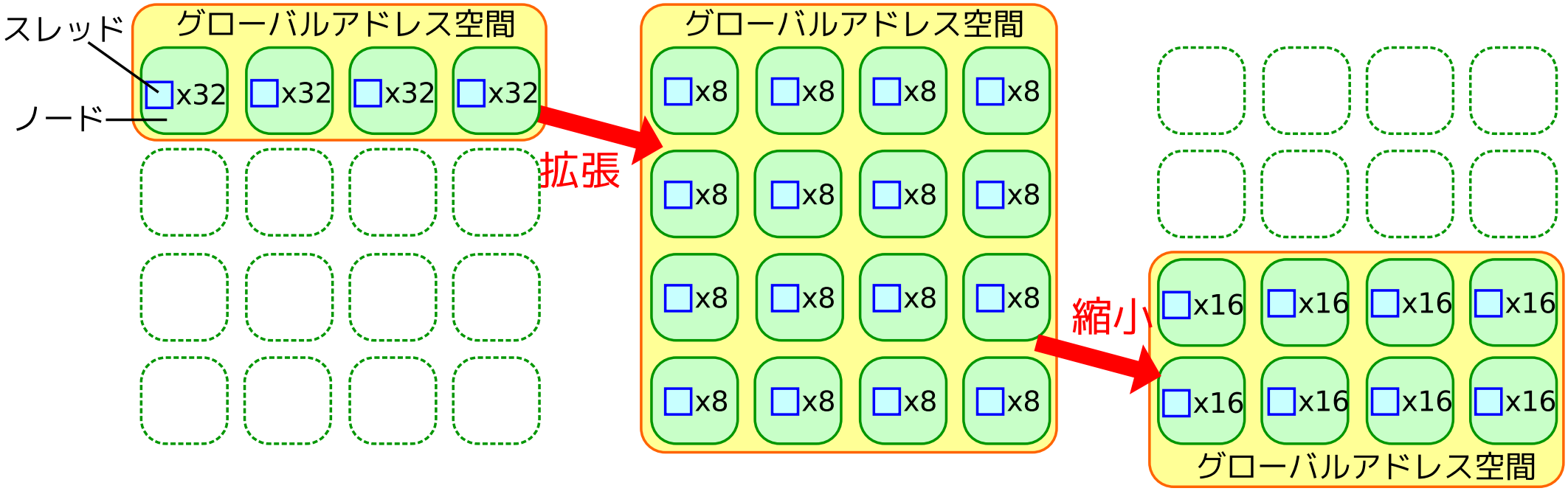
	スレッド増減	スレッド移動	スレッド移動 with half-process
N体問題	44行	5行	1行
ヤコビ法	11行	5行	1行
有限要素法	187行	62行	1行
ページランク	29行	18行	1行
最短路計算	29行	16行	1行

データのチェック
ポイント/リストア
のためのコードが
必要

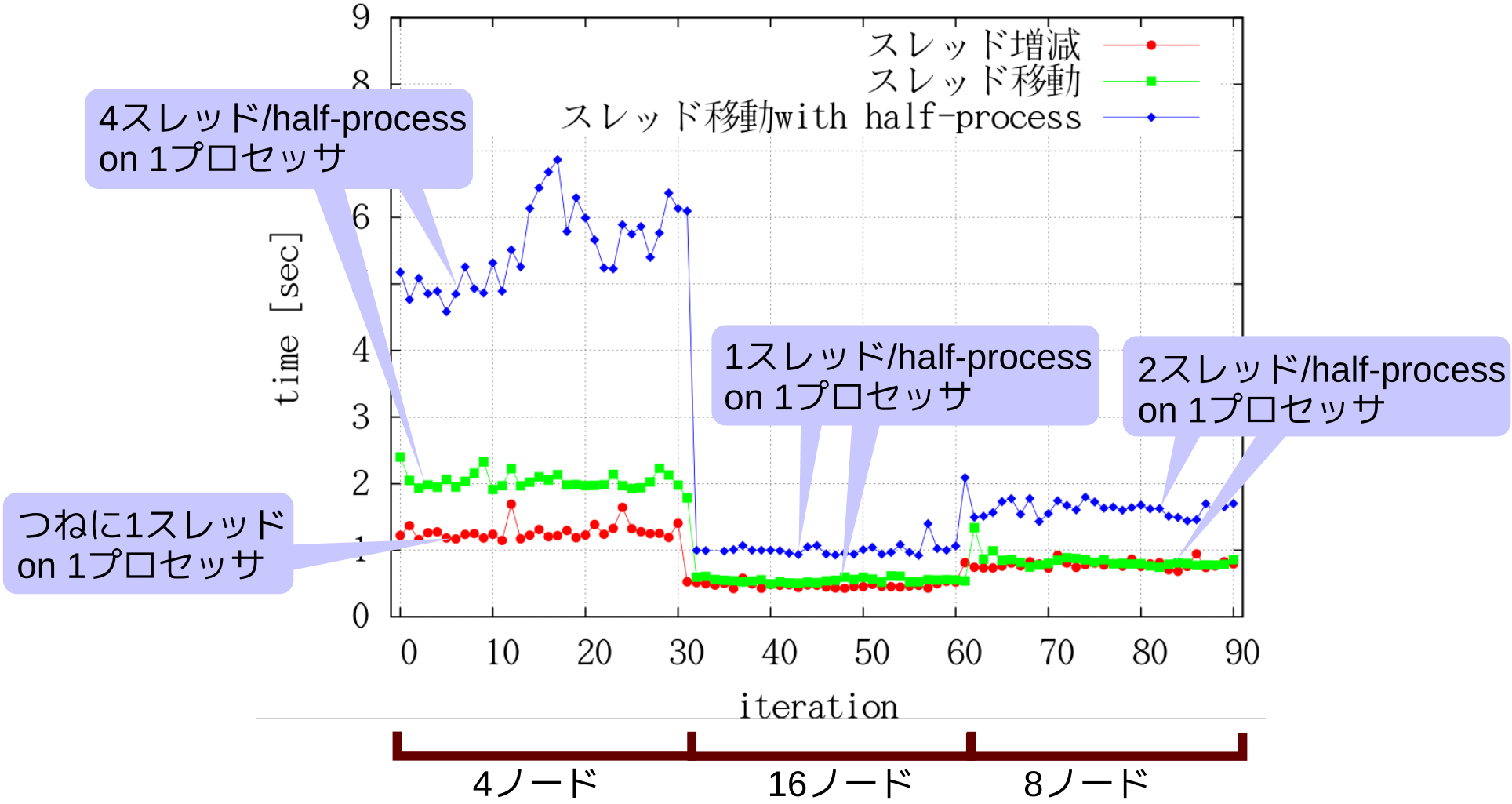
yield()関数を
1行追加するだけ

実験のシナリオ

- ノード数を 4 ノード (32 プロセッサ)→ 16 ノード (128 プロセッサ)→8 ノード (64 プロセッサ) と増減させる
- スレッド移動 (with half-process) の場合には，128 個のスレッド/half-process を生成



Web グラフの最短路計算



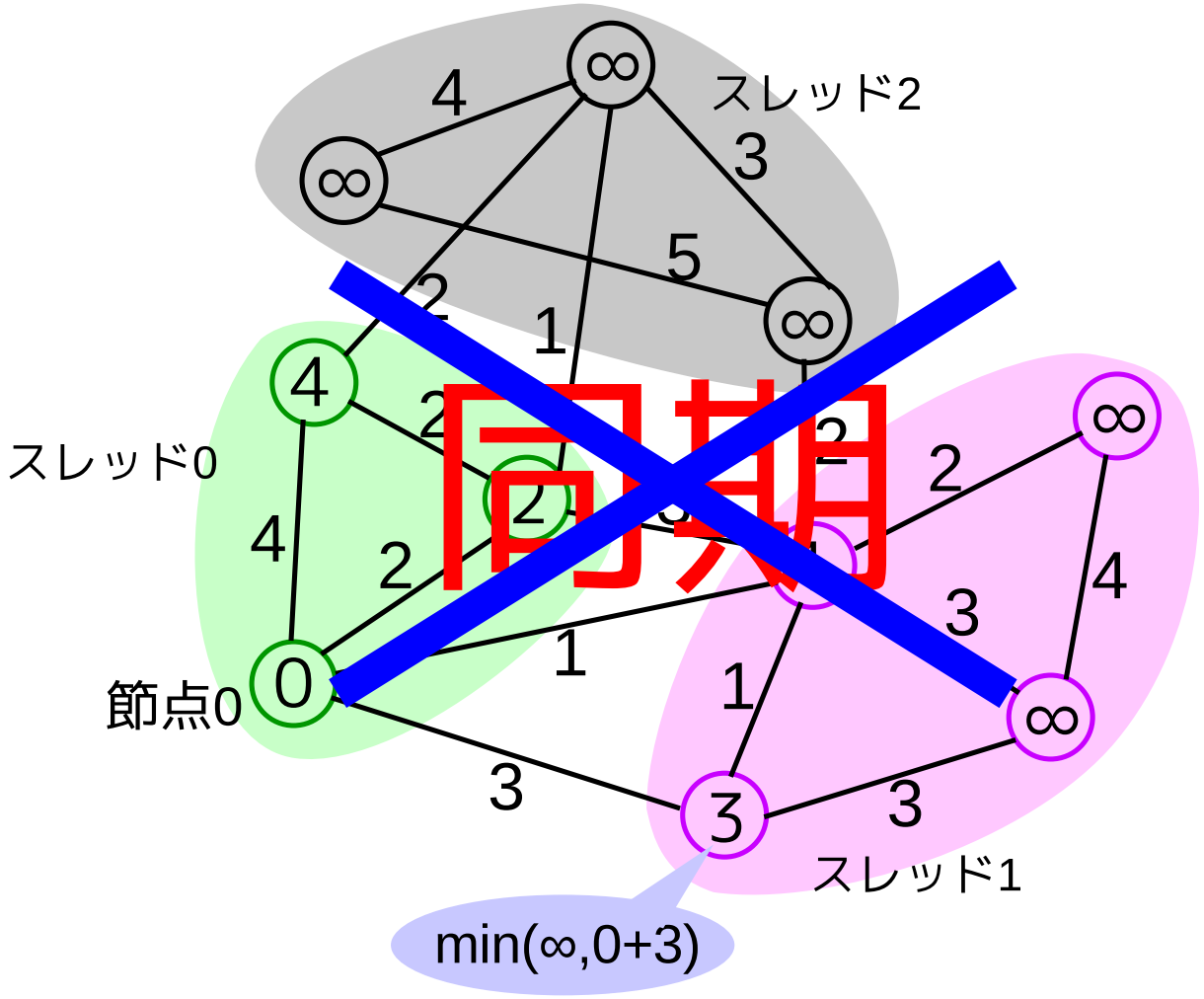
- 利用可能なノード数の増減に応じて効果的に並列度を増減できる
- 1 プロセッサあたり複数のスレッド/half-process を割り当てたときのオーバヘッドが大きい

half-process のオーバヘッドの原因

- いくつかある
  - → 非共有アドレス空間どうしのプロセス間メモリコピーにおけるコンテキストスイッチ
  - → 共有アドレス空間の malloc アルゴリズム
  - → ...
- オーバヘッド削減がきわめて重要

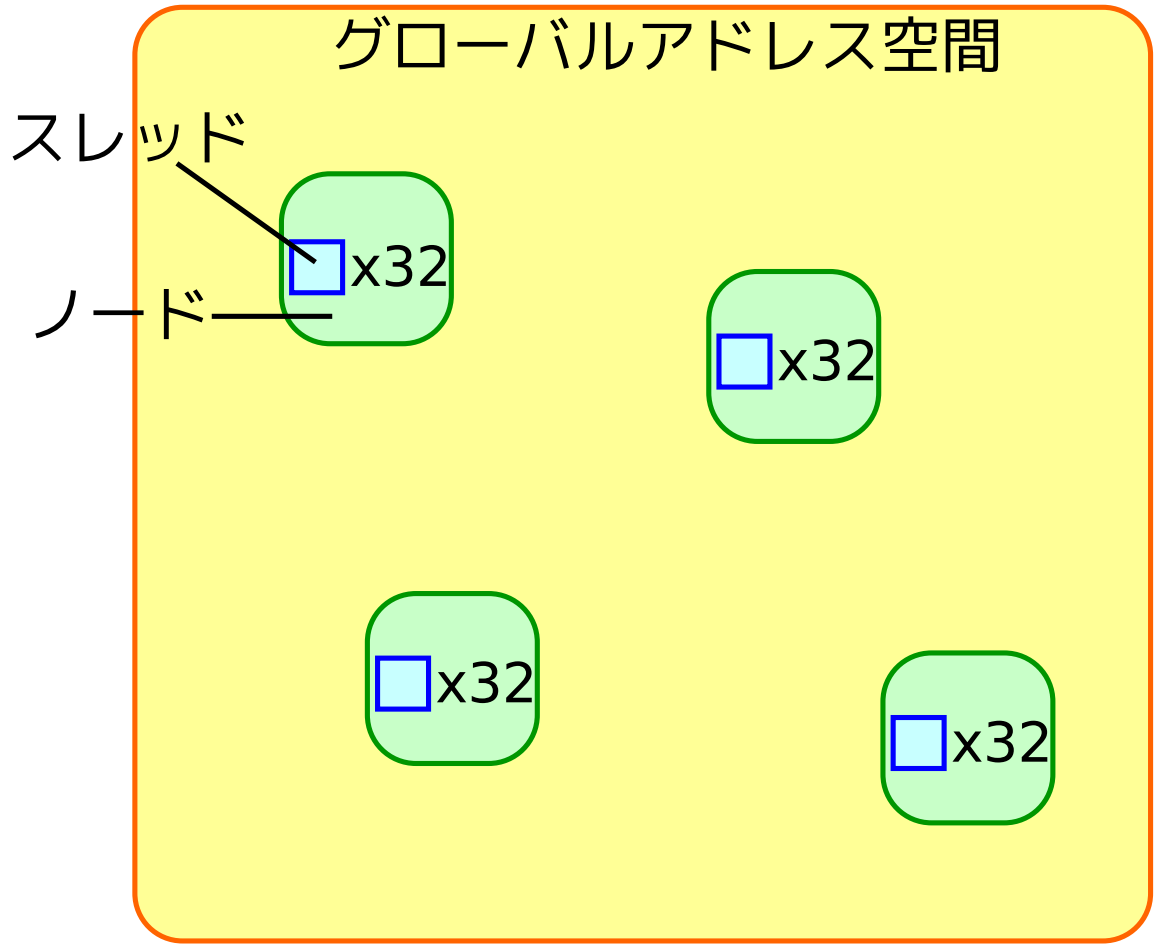
非同期的な最短路計算の再構成

- 以上の評価は「同期的な」反復並列計算の再構成
- 128 個のスレッドが「非同期的に」最短路計算をしている最中にノード数を増減させる

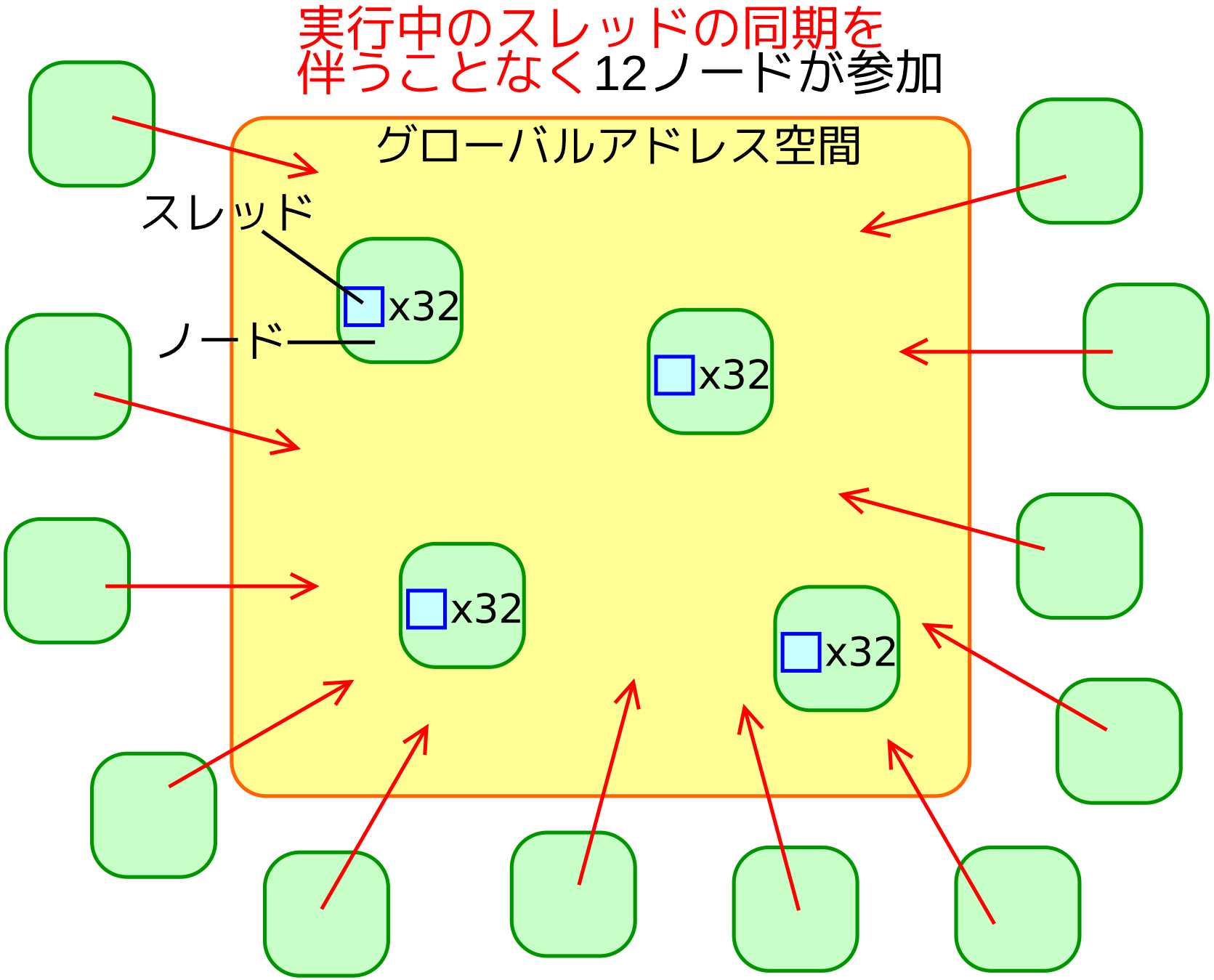


Web グラフの非同期的な最短路計算：結果 (1)

128スレッド on 4ノードで実行

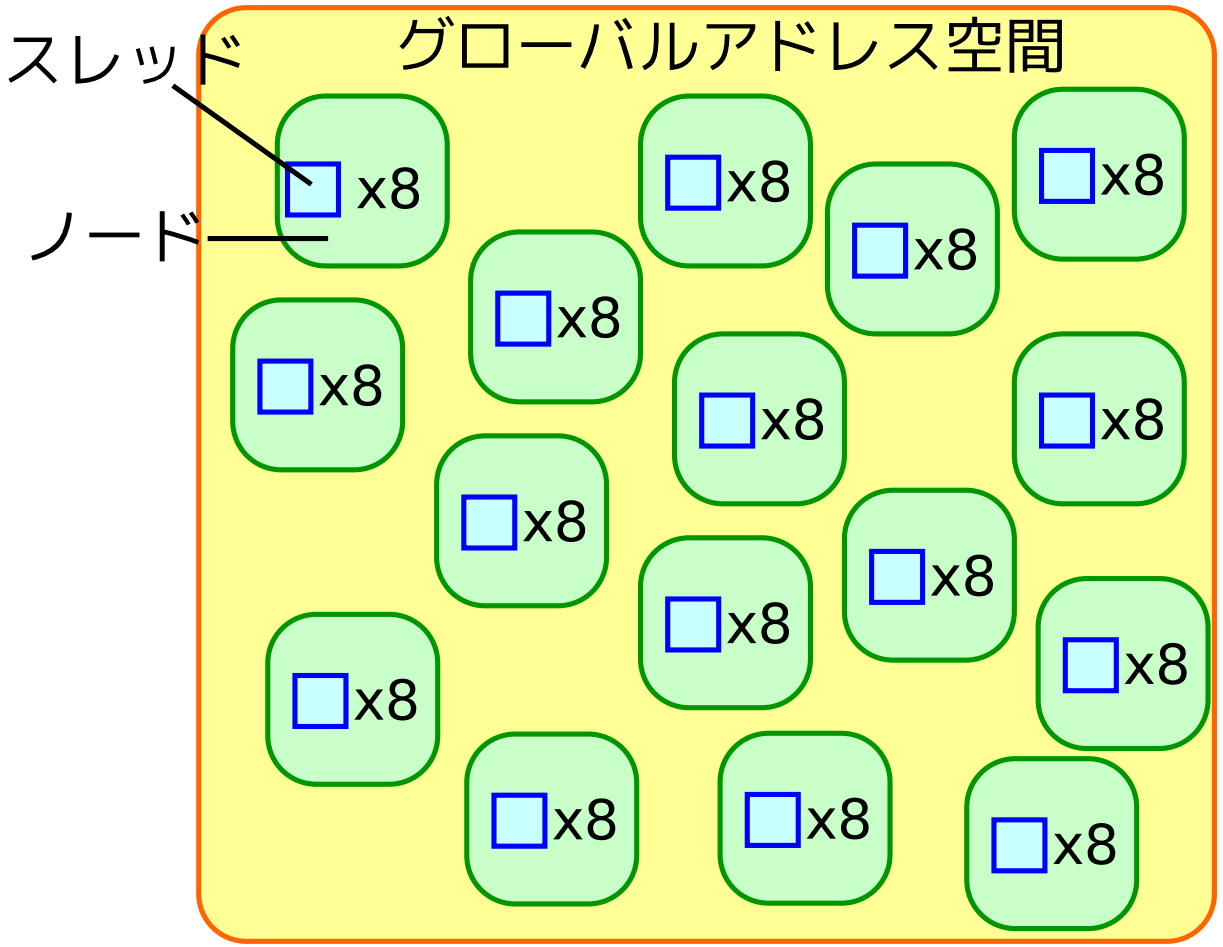


Web グラフの非同期的な最短路計算：結果 (2)



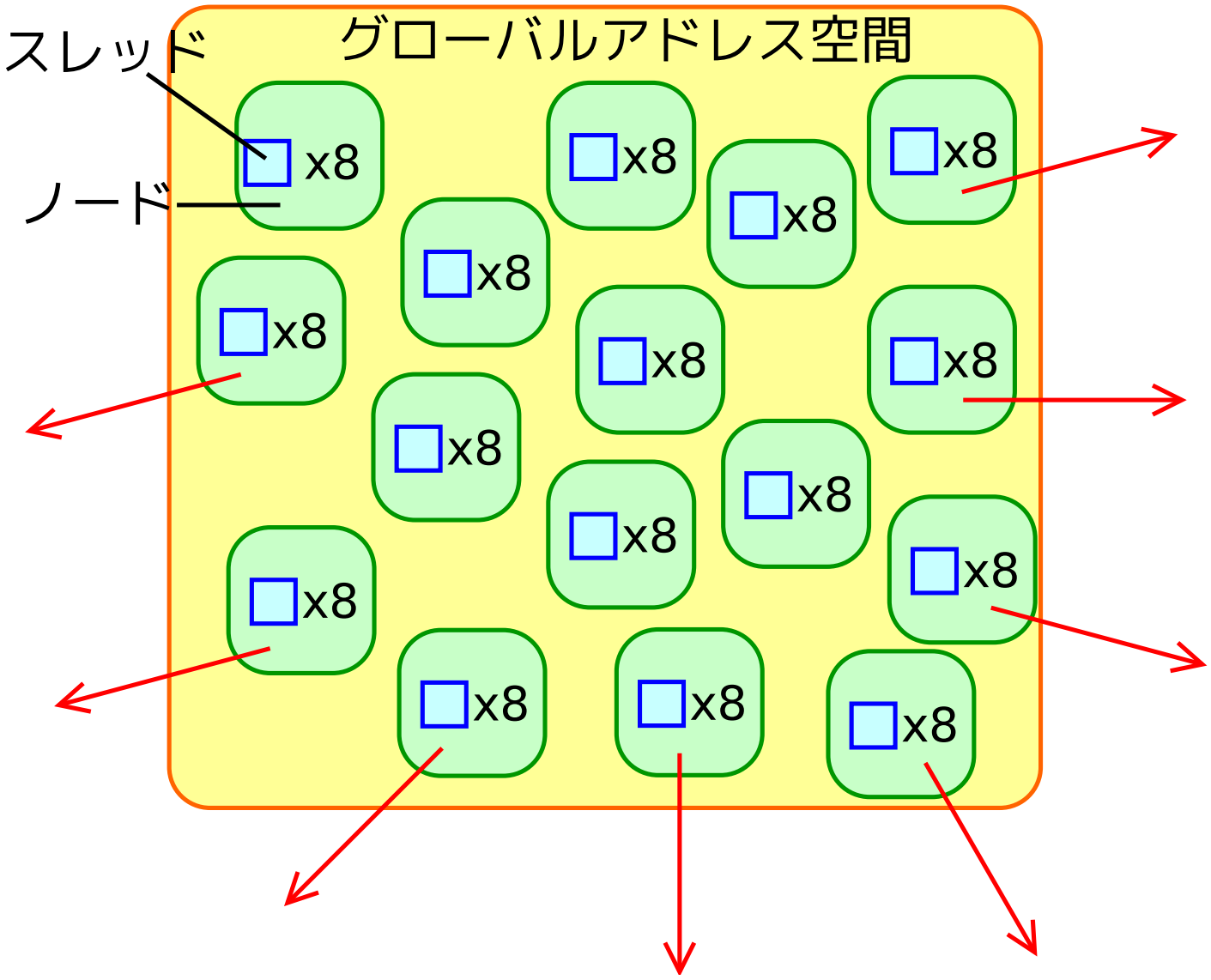
Web グラフの非同期的な最短路計算：結果 (3)

128スレッド on 16ノードで実行



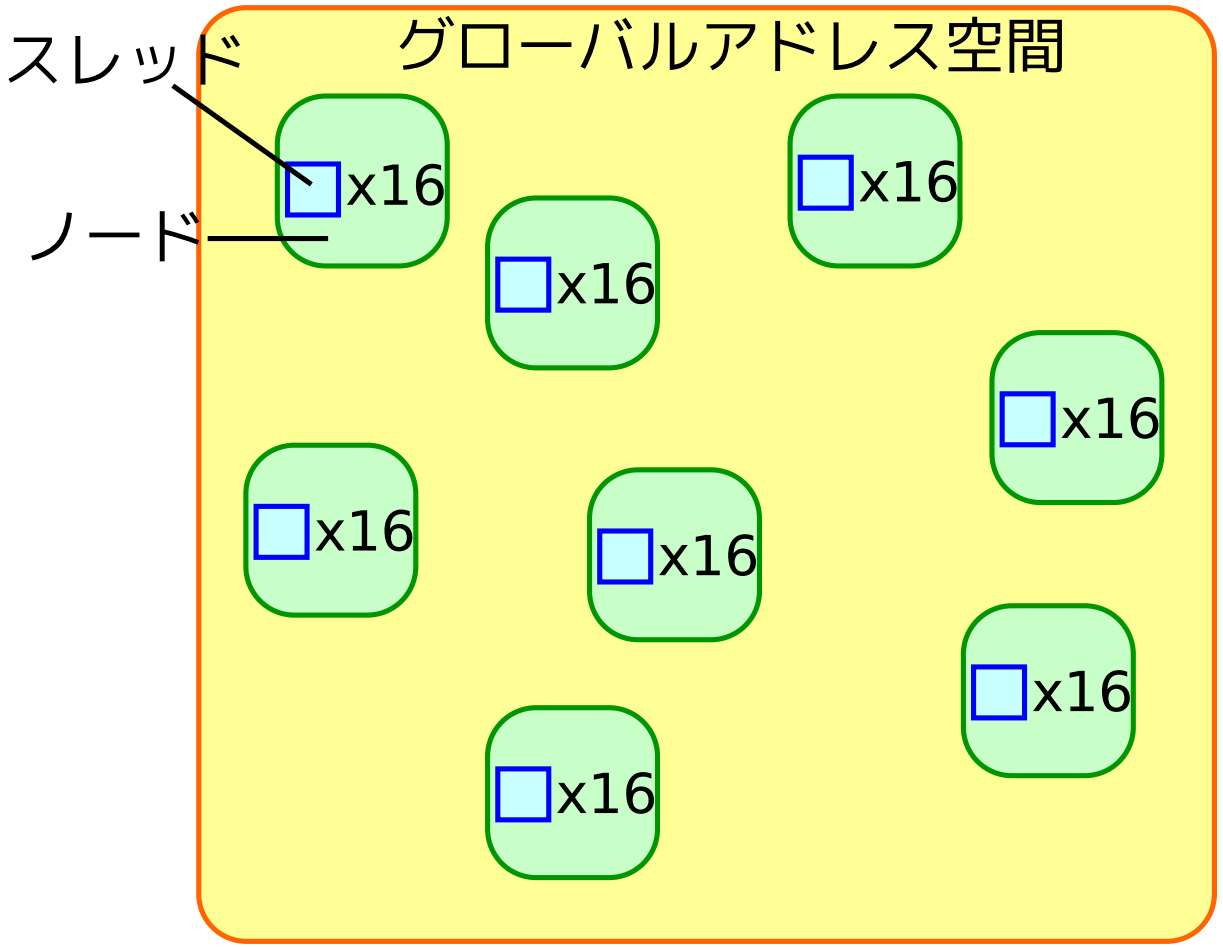
Web グラフの非同期的な最短路計算：結果 (4)

実行中のスレッドの同期を
伴うことなく8ノードが脱退



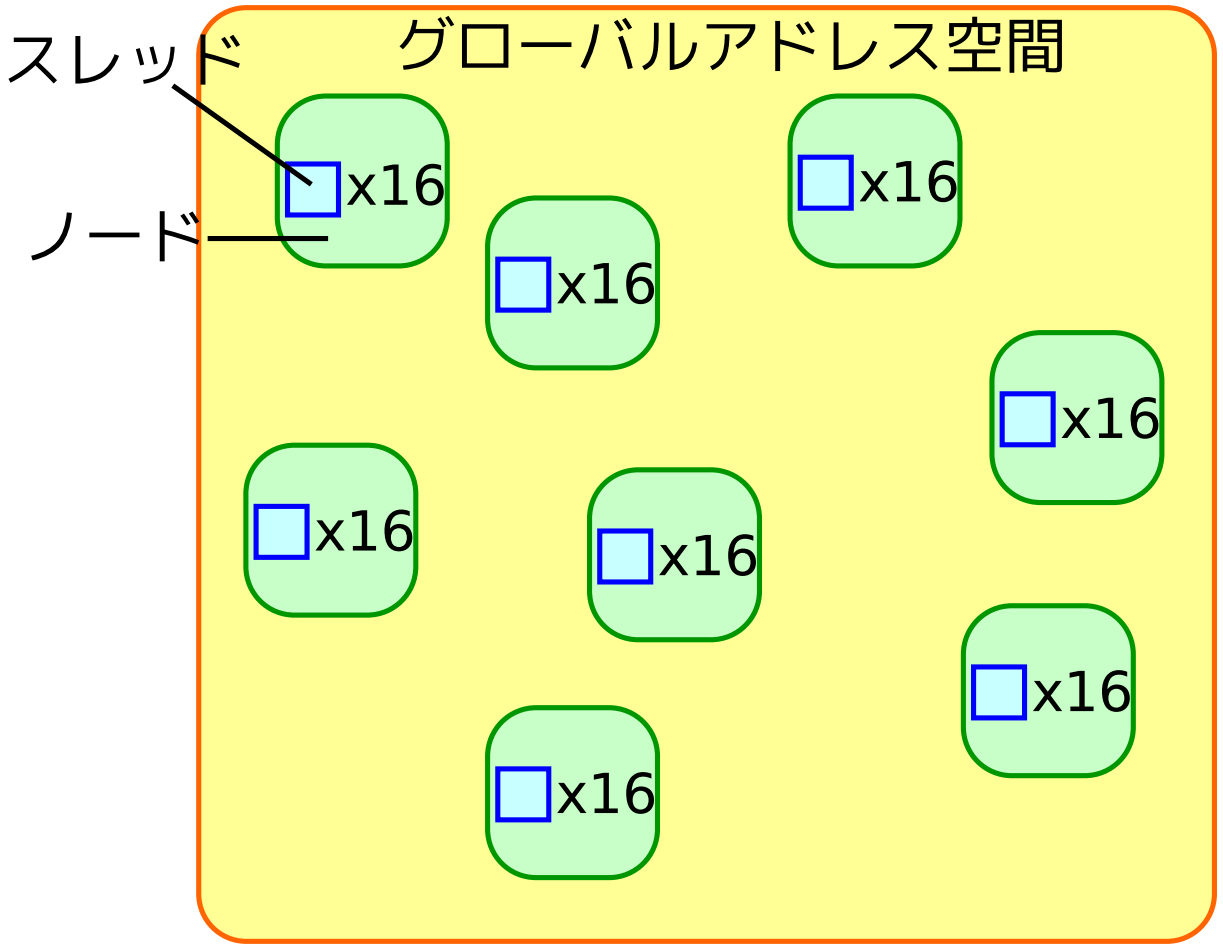
Web グラフの非同期的な最短路計算：結果 (5)

128スレッド on 8ノードで実行



Web グラフの非同期的な最短路計算：結果 (5)

128スレッド on 8ノードで実行



➤ **グローバルアドレス空間モデルに基づき，実行中のスレッドの同期を伴うことなくノードを自由に参加/脱退させられる処理系ははじめて**

ここまでのまとめ

- 目的 II：**再構成可能な並列計算を簡単に書ける**ようにする
- ノードが**自由なタイミングで参加/脱退**できるようなグローバルアドレス空間のコヒーレンシプロトコルを設計
- 再構成可能な並列計算のためのプログラミングモデルを 3 種類提案して比較
  - → スレッド移動 with **half-process** では，**yield() 関数 1 行を追加するだけ**で再構成を実現可能
  - → half-process のオーバヘッド削減が課題

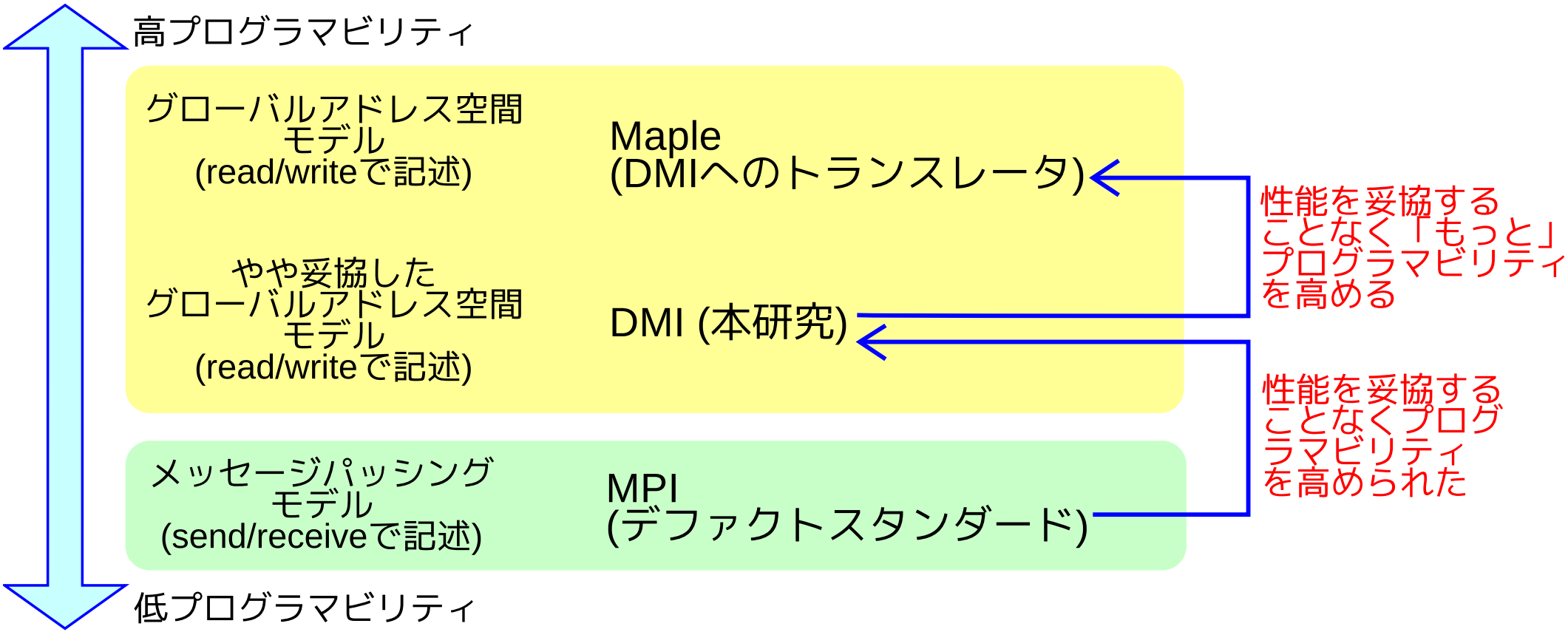
結論

本研究の貢献

- 汎用的な並列言語処理系 DMI
  - → たくさんの新規的な要素技術
- 非定型な並列計算：
  - → グローバルアドレス空間への read/write をわかりやすく強力に最適化する API を設計
- 再構成可能な並列計算：
  - → ノードが自由なタイミングで参加/脱退できるグローバルアドレス空間のコヒーレンシプロトコルを設計
  - → 再構成可能な並列計算のためのプログラミングモデルを 3 種類提案して比較

今後の課題

➤ **DMI の上に「もっと」プログラマビリティが高い並列言語処理系を開発中**



発表文献

- 論文誌：
  - 原健太朗，中島潤，田浦健次朗．アドレス空間の大きさに制限されないスレッド移動を実現する PGAS 処理系．情報処理学会論文誌（プログラミング）．Vol.4, No.1．pp.1-40．2011/3
  - 原健太朗，田浦健次朗，近山隆．DMI：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース．情報処理学会論文誌（プログラミング）．Vol.3，No.1，pp.1-40．2010/3
- 国際学会（査読あり）：
  - Kentaro Hara, Kenjiro Taura. A Global Address Space Framework for Irregular Applications. 2010 International ACM Symposium on High Performance Distributed Computing（HPDC2010）. pp.296-299. 2010/6
- 国際学会（投稿中）：
  - Kentaro Hara, Kenjiro Taura. Parallel Computational Reconfiguration Based on a PGAS Model. 2011 International ACM Symposium on High Performance Distributed Computing (HPDC2011). 12 pages（投稿中）.
- 国内学会（査読あり）：
  - 原健太朗，塩谷亮太，田浦健次朗．メモリアクセス最適化を適用した汎用プロセッサと Cell の性能比較．先進的計算基盤シンポジウム（SACSIS 2008）．pp.157-166．2008/6
- 国内学会（査読なし）：
  - 原健太朗，中島潤，田浦健次朗．アドレス空間の大きさに制限されないスレッド移動を実現する PGAS 処理系．SWoPP2010．2010/8
  - 原健太朗，田浦健次朗，近山隆．DMI：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース．SWoPP2009．2009/8
  - 原健太朗．有限要素法における連立方程式ソルバの並列化（第 2 回クラスタシステム上の並列プログラミングコンテスト成果報告）．第 9 回 PC クラスタシンポジウム．2009/12
  - 原健太朗．ホモロジー検索の並列最適化（クラスタシステム上の並列プログラミングコンテスト非数値計算部門成果報告）．PC クラスタワークショップ in 広島．2009/5
- 受賞：
  - 2009 年度 CS 領域奨励賞（プログラミング研究会）．2011/1
  - 2009 年 学士卒業論文 東京大学工学部長賞．2009/3
- プログラミングコンテスト：
  - 第 2 回 クラスタシステム上の並列プログラミングコンテスト 第 2 位．2009/10
  - SACSIS2009 併設企画 クラスタシステム上の並列プログラミングコンテスト非数値計算部門 第 1 位．2009/5