# ❖ DMI：計算資源の動的な増減に対応した大規模分散共有メモリインタフェース ❖

近山・田浦研究室　B4　原健太朗

2008.11.14

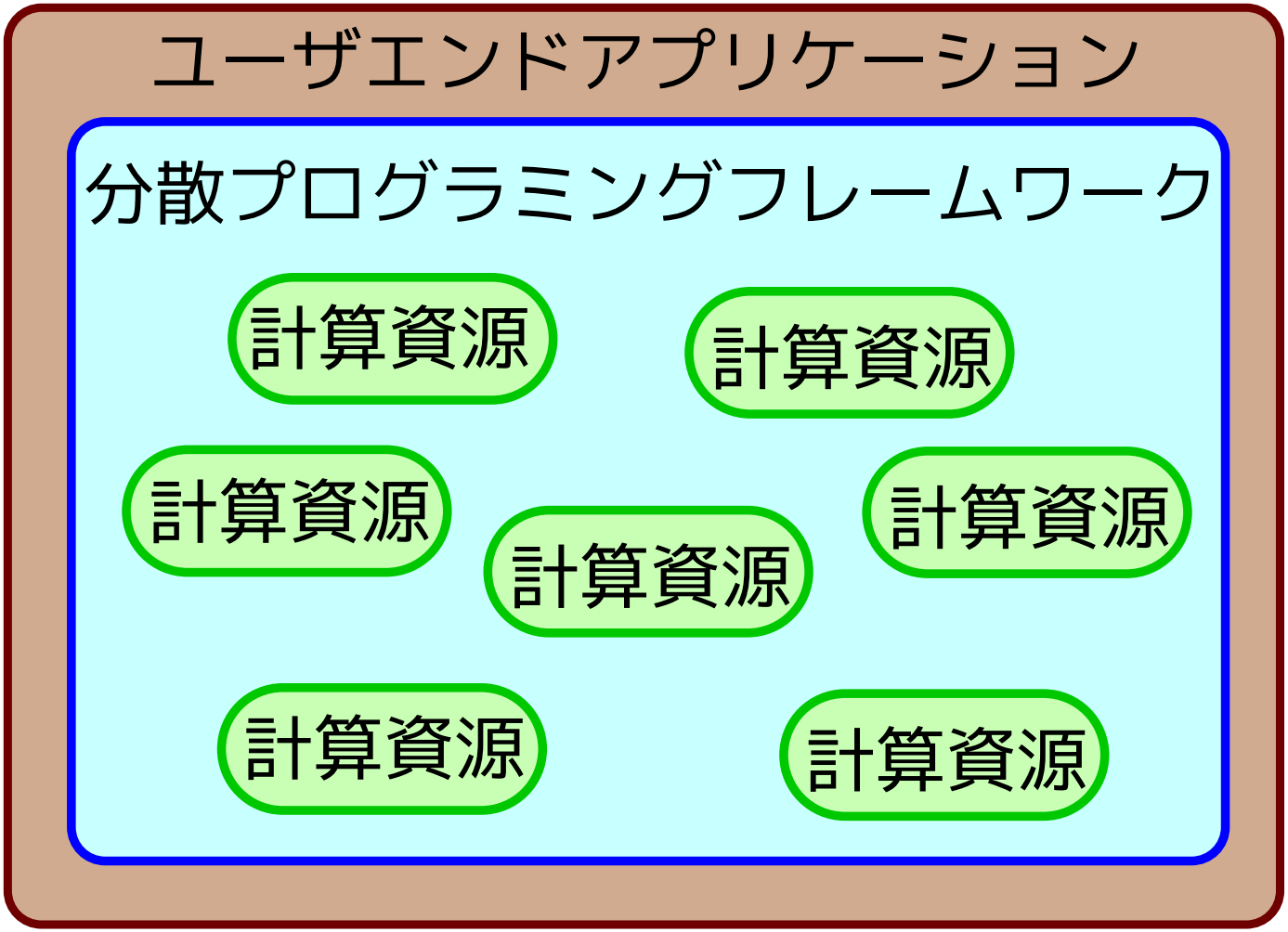
# 発表の流れ

- はじめに
- DSM の特徴
- DMI のコンセプト
- DMI の実装
- 予備的性能評価
- まとめ

# 1. はじめに

# 本研究の背景

- グリッドコンピューティングの発展
  - → 分散プログラミングフレームワークが重要

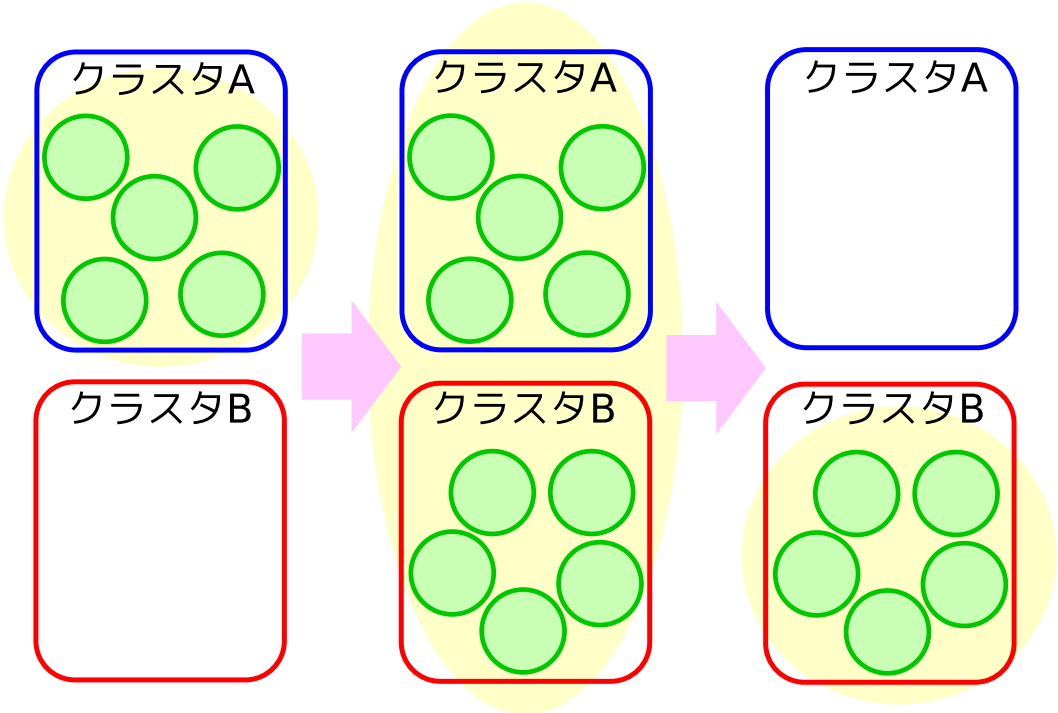


# 分散プログラミングフレームワークへの要請

- 共通の要請：
  - → 記述力
  - → スケーラビリティ
- さまざまな機能面への要請：
  - → 計算資源の動的な増減への対応
  - → マルチコア並列プログラミングから飛躍の小さい記述スタイル
  - → 効率的なオーバーレイ，ルーティング，集合通信のサポート
  - → NAT や Firewall など複雑なネットワーク構成への適応
  - → 耐故障

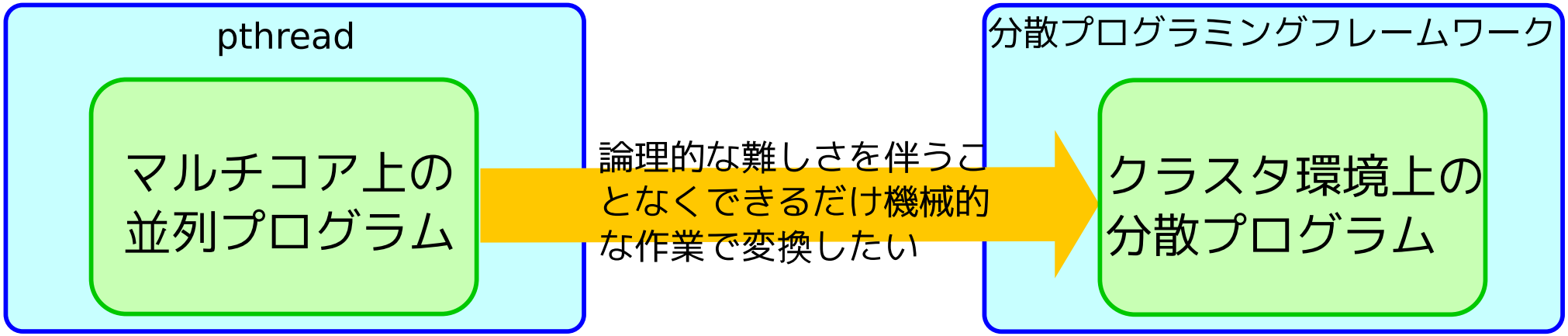
# 計算資源の動的な増減

- 現状：
  - → クラスタ環境には課金制度や運用ポリシーが存在
- 要請：
  - → 必要に応じて計算資源を動的に参加/脱退させたい
  - → 計算環境のマイグレーションを実現したい



# マルチコア並列プログラミングとの類似性

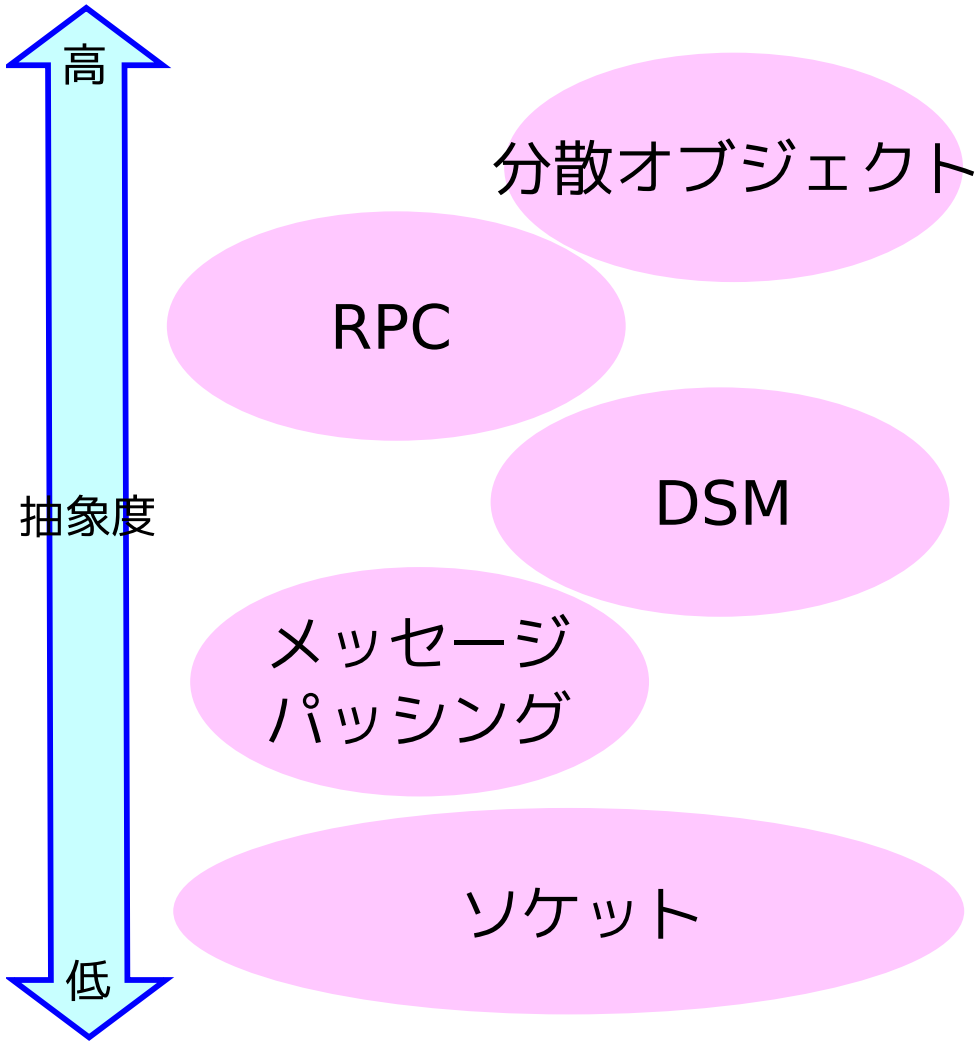
- **現状：**
  - → **マルチコアレベルの並列処理（特に pthread）は実現されているが，分散処理化には至っていないアプリが多数**
- **要請：**
  - → **pthread プログラムからの飛躍が小さい（＝ほぼ機械的な変換作業でコードが得られるような）分散プログラミングフレームワークがあれば便利**



# 本研究の目的

- 分散プログラミングフレームワークの開発：
  - → ノードの動的な参加/脱退をサポート
  - → pthread プログラムからの飛躍が小さい（＝ほぼ機械的な変換作業でコードが得られる）記述スタイル

# 分散プログラミングモデル




- **本研究では** DSM（Distributed Shared Memory）**に着眼**

# 2. DSM の特徴

- DSM とメッセージパッシングの比較
- 既存 DSM のアプローチ

# DSM とメッセージパッシングの比較 Point

**Point 1 :** 記述の容易さ

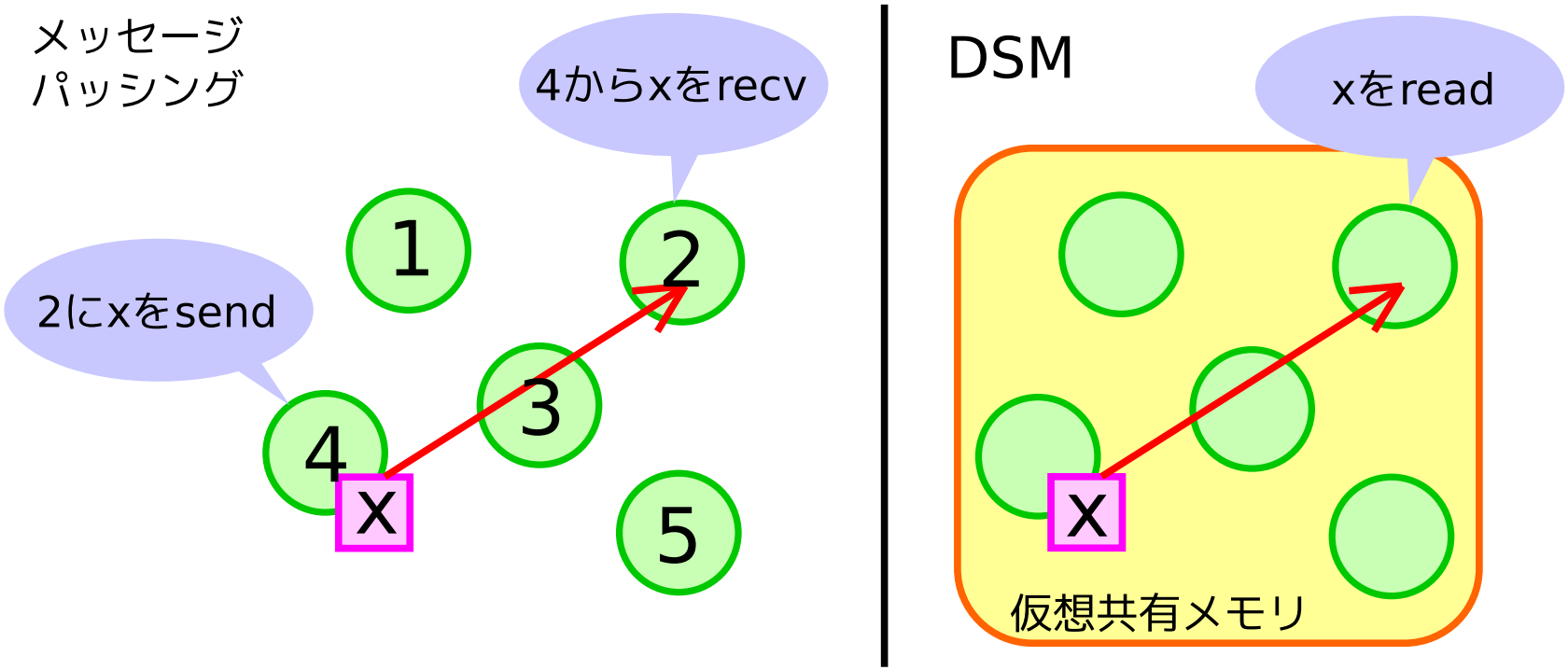
**Point 2 :** ノードの動的な増減への適応力

**Point 3 :** 応用範囲の広さ

**Point 4 :** スケーラビリティ

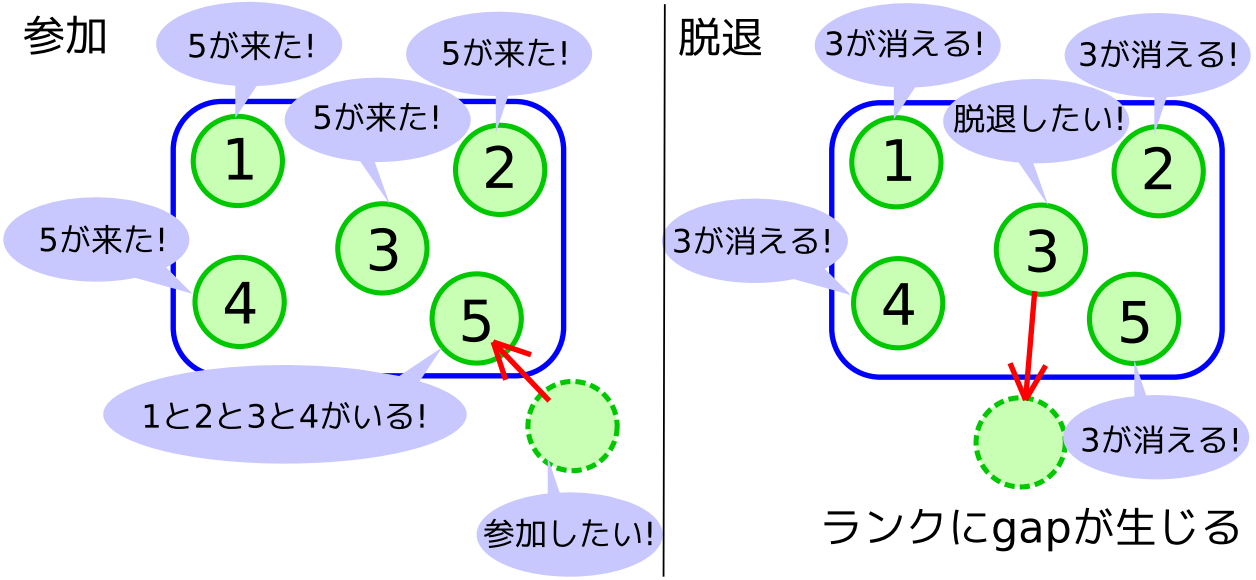
# Point 1：記述の容易さ

- メッセージパッシング：
  - → 送信側と受信側の両者を記述する必要がある
- DSM：
  - → マルチコア上の共有メモリ型プログラミングと同様
  - → （一般的には）記述が容易で論理的にわかりやすい



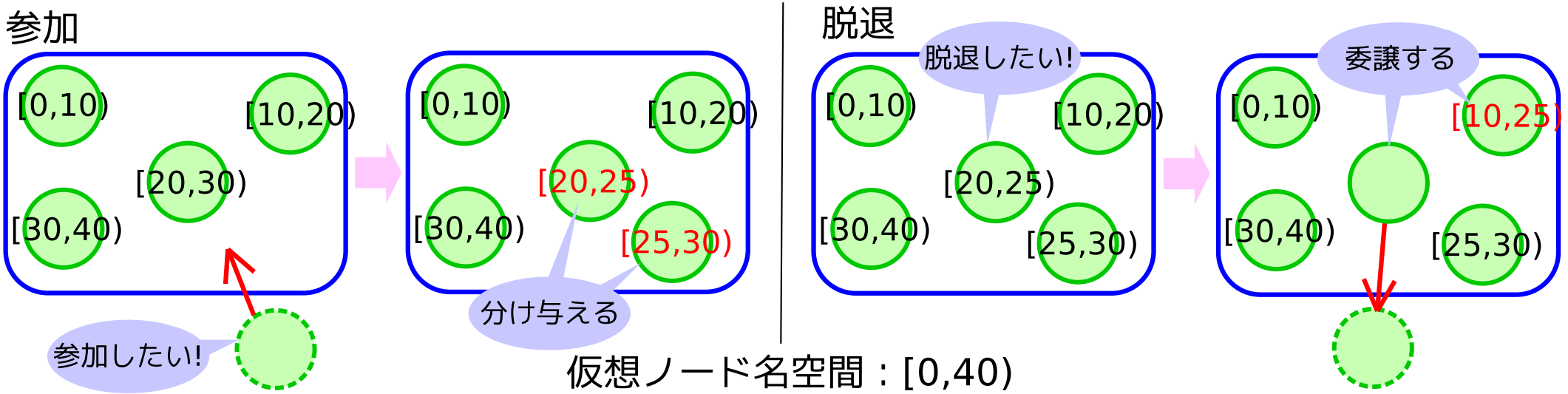
# Point 2：ノードの動的な増減への適応力 (1)

- メッセージパッシング：
  - → 一意的なランクを明示的に使用した通信を記述するため，ユーザプログラム側は全ノードとランクのマッピングを知っている必要がある
  - → 参加/脱退時には，マッピングの変化を取り扱うコーディングがユーザプログラム側に必要
  - → スケーラブルな（＝全員の同期を必要としない）記述は難



# Point 2：ノードの動的な増減への適応力 (2)

- Phoenix[Taura et al,2003]：
  - → **仮想ノード名空間 $D : [0, L)$ を考え，全ノードで $D$ を漏れなく重複なく包むよう管理**
    - ♦ **参加時には実行中のノードから集合の一部をもらう**
    - ♦ **脱退時には実行中のノードに集合を委譲する**
    - ♦ **参加/脱退を局所的な（＝全員の同期が不要な）操作で実現**

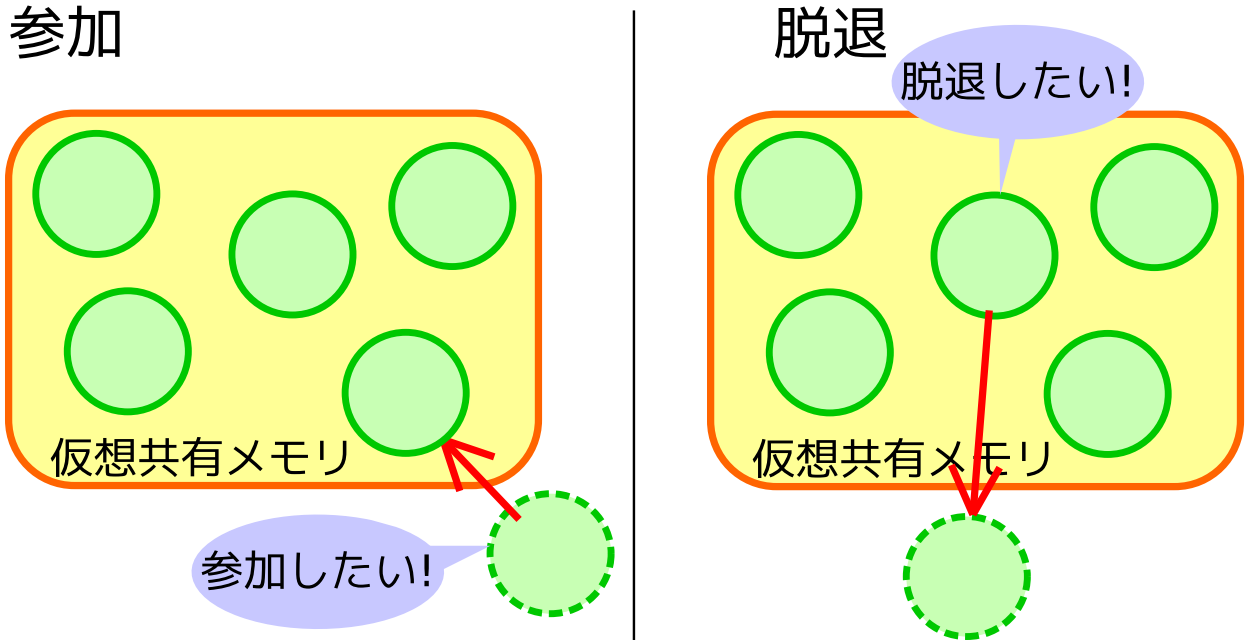


# Point 2：ノードの動的な増減への適応力 (3)

- Phoenix[Taura et al,2003]（続き）：
  - → 仮想ノード名を用いた送受信を行うことで，参加/脱退に関係なくメッセージの到達性を保証
  - → パワフルな記述力とスケーラビリティを兼ね備える
  - → しかし記述は複雑

# Point 2：ノードの動的な増減への適応力 (4)

- DSM：
  - → **処理系内部で管理される仮想的な共有メモリが通信媒体なので，自分以外に誰がいるかをユーザプログラム側で知っている必要が（通信上は）ない**
  - → **参加/脱退時には，「参加したい」「脱退したい」と言うだけでよい**

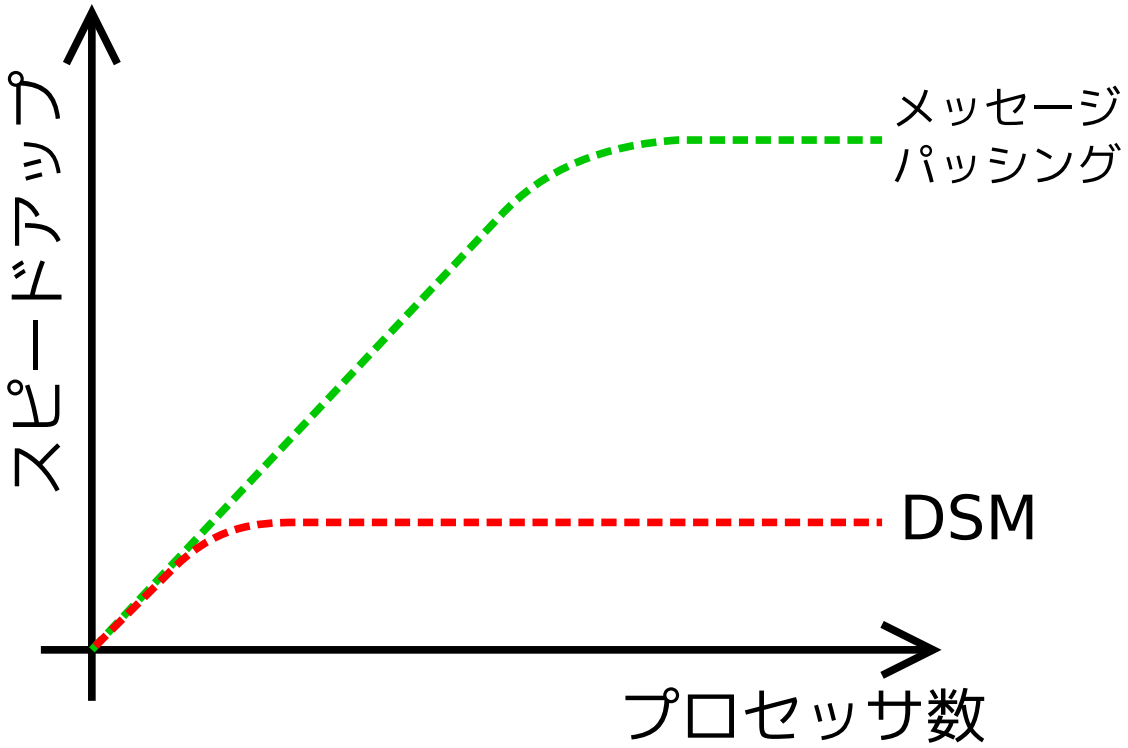


# Point 3：応用範囲の広さ

- DSM は幅広い応用力を持つ
  - → ネットワークページング：
    - ♦ DLM[Midorikawa et al,2007] がローカルな swap アクセスと比較して 5〜10 倍の性能を発揮
  - → プロセスマイグレーション

# Point 4：スケーラビリティ

- メッセージパッシングと比較して DSM はスケールしにくい
  - → DSM の方が抽象度が高いため
  - → DSM では処理系が通信パターンを把握できないため



# 既存 DSM の主なアプローチ

- 過去 20 年以上に渡る多数の実装例：
  - IVY，Munin，TreadMarks，JIAJIA，Midway，UPC，SMS，· · ·

**Approach 1 :** コンシステンシモデルの緩和

**Approach 2 :** データ通信量の削減

**Approach 3 :** OS のメモリ保護違反機構の利用

**Approach 4 :** SPMD 型の記述スタイル

# Approach 1：コンシステンシモデルの緩和

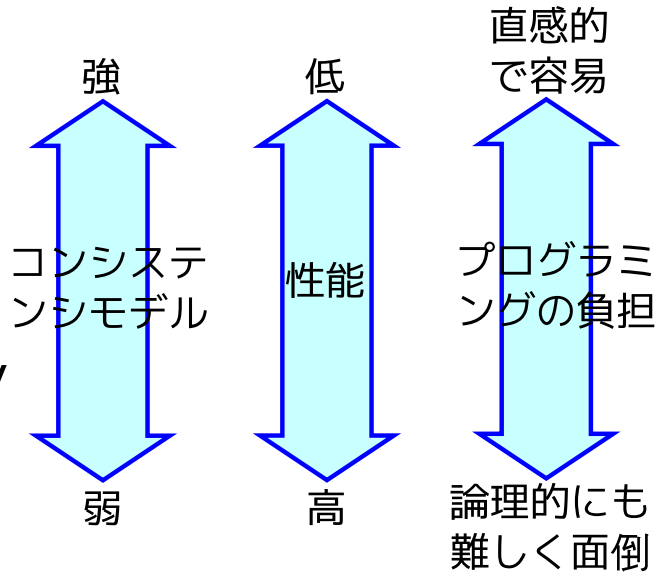
Sequential Consistency

Weak Consistency

Release Consistency

Lazy Release Consistency

Entry Consistency




- **利点：**
  - → **独立性が向上**
  - → **必要な更新データのみの転送を実現**
- **問題点：**
  - → **論理的な意味でプログラミングが難化**
  - → **文法的にも pthread プログラムの飛躍が大きくなる**

# Approach 2：データ通信量の削減

- 手法：
  - → 極力 demand driven に更新データを転送する各種の工夫
  - → タイムスタンプに基づくデータの差分転送
- 利点：
  - → データ転送待ちのストールが減る
- 問題点：
  - → 暗黙的な機構を入れるほど明示的なチューニングが困難に

# Approach 3：OS のメモリ保護違反機構の利用

```
shared int var;
shared int array[100];

var = 123;
array[10] = 123;
```

- 利点：
  - → ローカルメモリへのアクセスと仮想共有メモリへのアクセスを透過的に同じ文法で実現
- 問題点：
  - → コンシステンシ維持の単位が OS のページサイズ（の整数倍）に限定
  - → ネットワークページングは 64bit OS が前提

# Approach 4：SPMD 型の記述スタイル

- 利点：
  - → 時系列的に全員がどう動作するかが明確なアプリ（特に並列計算）に向く
  - → 集団操作が存在し処理系による最適化が可能
- 問題点：
  - → 全員の時系列的な動作がはっきりしないアプリが記述しにくい
  - → 特に，動的な参加/脱退の表現が難しい
    - ♦ MPI-2 が動的プロセス生成をサポートしているが，記述は複雑

# 3. DMI のアプローチ

- DMI のアプローチ
- DMI の API

# DMI のコンセプト

- DMI（Distributed Memory Interface）

**Concept 1 :** **ノードの動的な参加/脱退をサポートする大規模分散共有メモリ**

**Concept 2 :** pthread **プログラムからのほぼ機械的な変換作業で（＝論理的な難しさを伴わない変換作業で）コードが得られるような分散プログラミングフレームワーク**

# DMI のアプローチ

**Approach 1 :** Sequential Consistency の採用

**Approach 2 :** ユーザレベルでのメモリ管理

**Approach 3 :** ノードの動的な参加/脱退

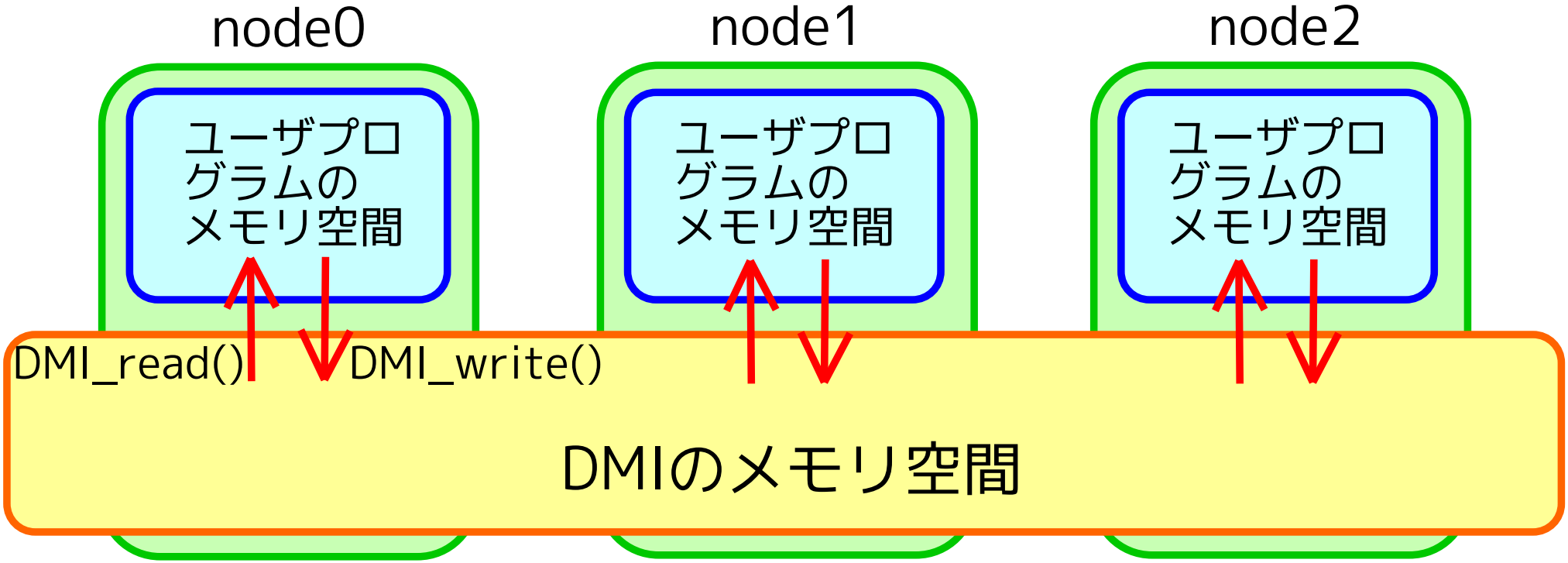
**Approach 4 :** pthread 型の記述スタイル

**Approach 5 :** ページ転送の動的負荷分散

# Approach 1：Sequential Consistency の採用

- 利点：
  - → 動作が直感的で明快
  - → メモリコンシステンシ面では pthread プログラムからの飛躍がない
- 問題点：
  - → 制約が強すぎて性能が出にくい

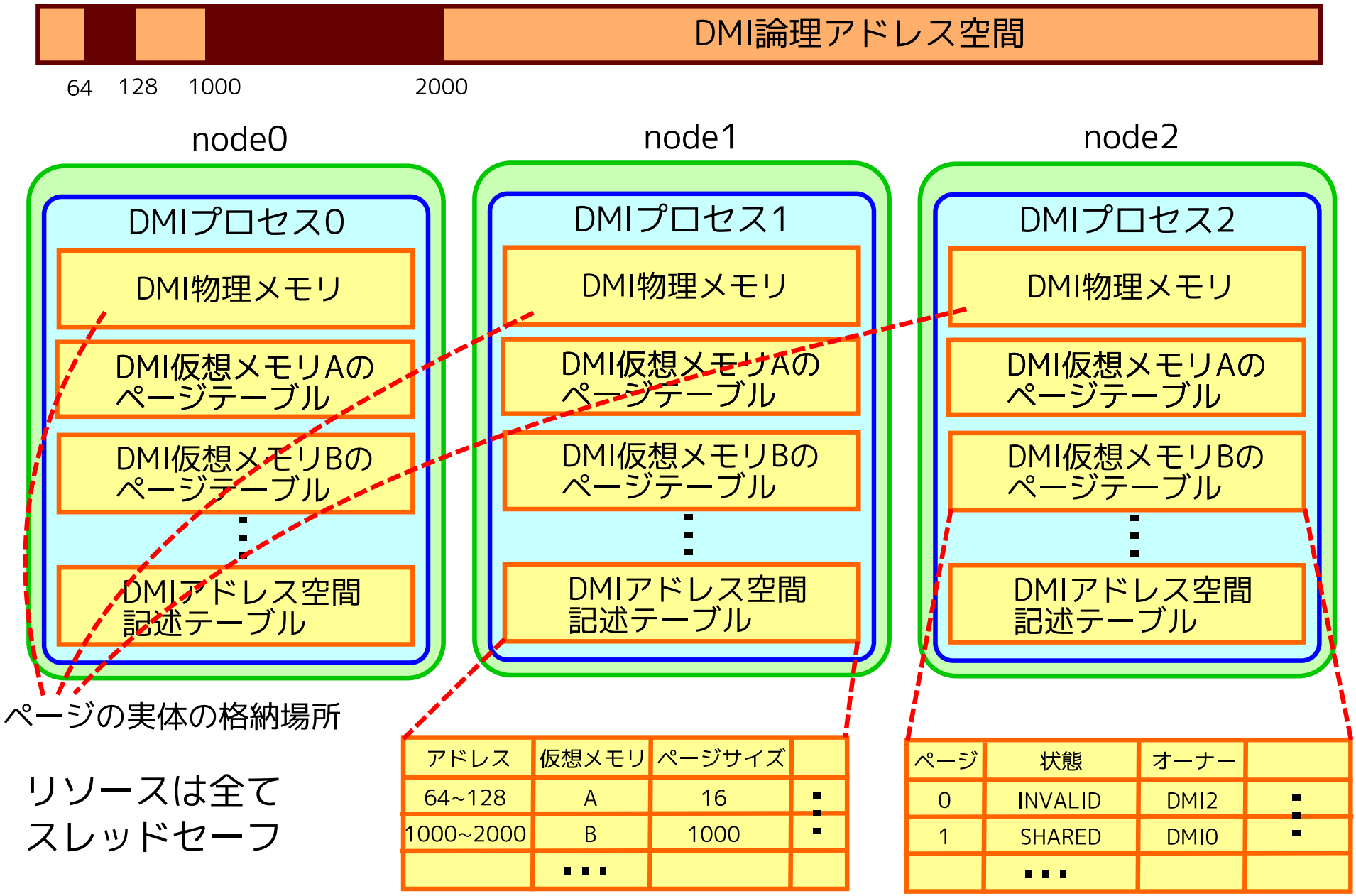
# Approach 2：ユーザレベルでのメモリ管理 (1)




```
DMI_read(address, size, buffer);
DMI_write(address, size, buffer);
```

- **ユーザプログラムのメモリ空間と DMI のメモリ空間は独立**
  - → **API を通じてのみアクセス可能**

# Approach 2：ユーザレベルでのメモリ管理 (2)




リソースは全て
スレッドセーフ

# Approach 2：ユーザレベルでのメモリ管理 (3)

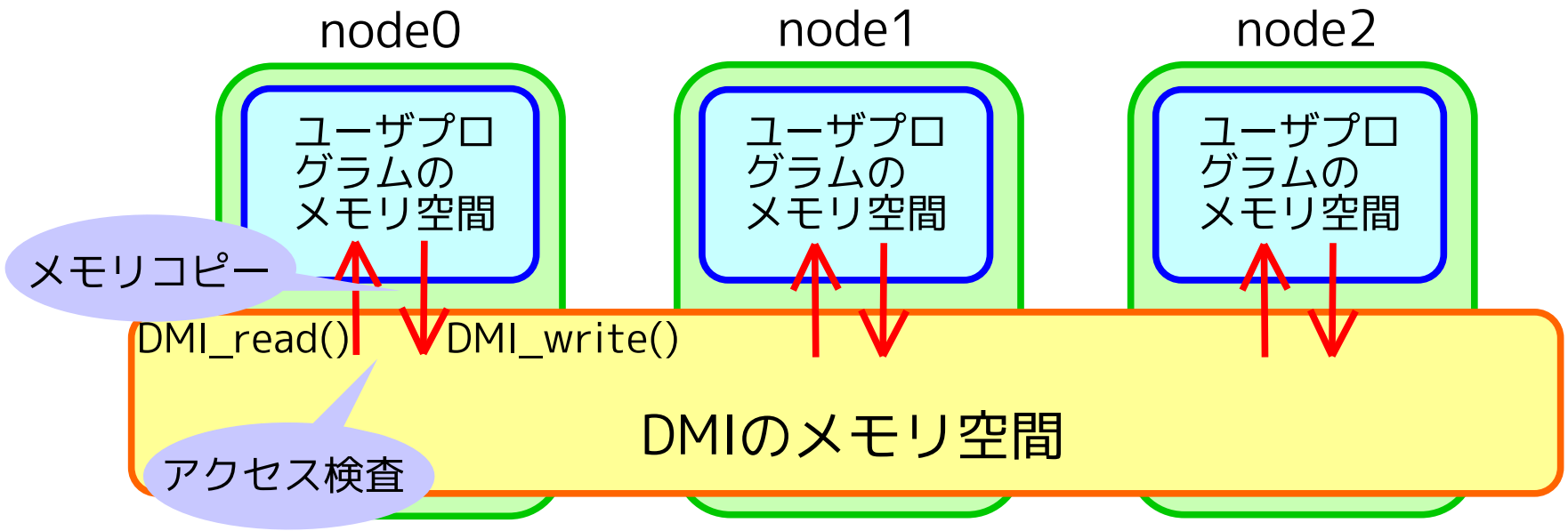
- 任意のページサイズが指定可能
  - → ページフォルトを大幅に抑制可能
    - ♦ 例：行列丸ごと 1 個を 1 ページに設定
  - → Sequential Consistency による性能劣化をある程度補えると期待
- OS のアドレッシング範囲に捉われない
- 非同期 read/write など柔軟性の高い API を提供
  - → 性能チューニングの自由度が高い
  - → インクリメンタルな開発が可能
    - ♦ 例：初期的には通常の read/write で簡易に実装 → 性能を求めるなら非同期 read/write を導入

# Approach 2：ユーザレベルでのメモリ管理 (4)

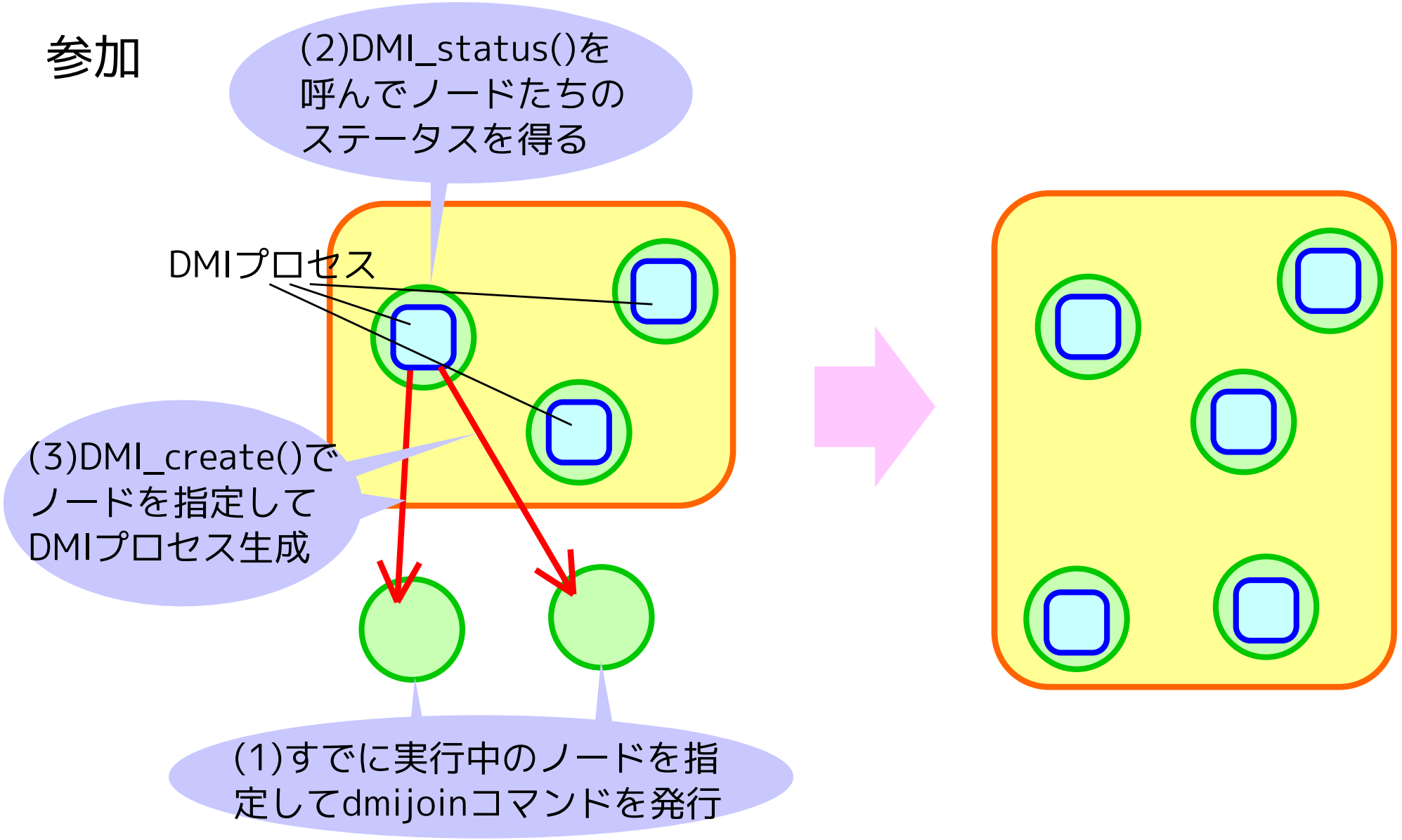
- ユーザプログラムのメモリ空間と DMI のメモリ空間を明確に区別したプログラミングを強制
  - → 効率的なプログラムが開発される可能性が高い
- 機能拡張性に富む

# Approach 2：ユーザレベルでのメモリ管理 (5)

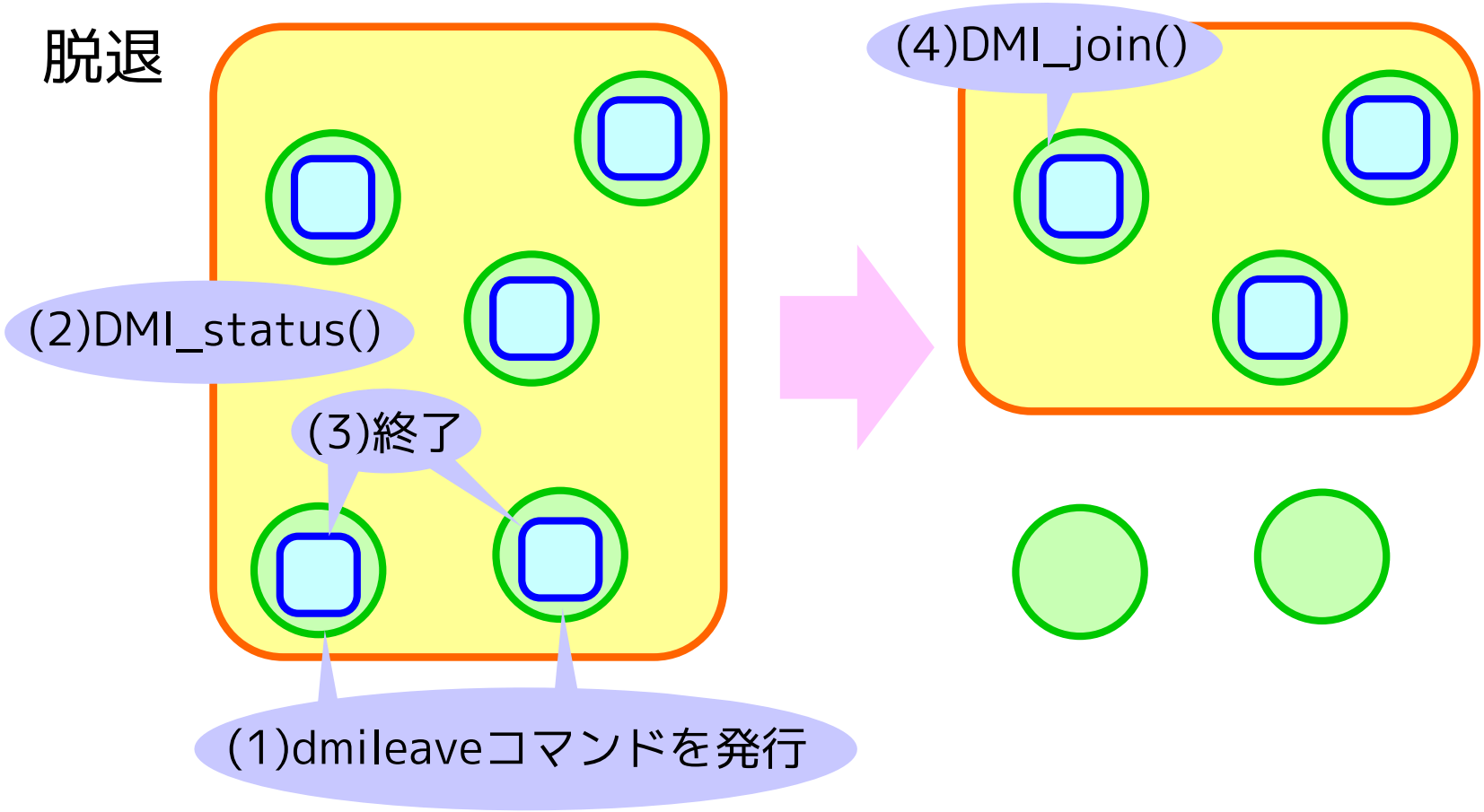
- 欠点：
  - → プログラミングが作業的に面倒
    - ♦ 論理的には容易
  - → DMI のメモリ空間へのアクセスのオーバーヘッド大
    - ♦ 逐一ソフトウェア的なアクセス検査が入る
    - ♦ メモリコピーが必要



# Approach 3：ノードの動的な参加/脱退 (1)



# Approach 3：ノードの動的な参加/脱退 (2)



# Approach 4：pthread 型の記述スタイル (1)

pthreadのスタイル：

```
int global_x;

main() {
  global_x = 123;    ← グローバル変数へwrite
  pthread_create(&hdl, NULL, f, &args); ← スレッド生成
  ...
  pthread_join(hdl, NULL); ← スレッド回収
}

                    引数渡し

void* f(void *args) {
  int local_x = global_x;  ← グローバル変数をread
  ...
}
```

# Approach 4：pthread 型の記述スタイル (2)

DMIのスタイル：

```
#define GLOBAL 1
main(int argc, char **argv) {
  DMI_Init(&argc, &argv, ...);      ← 初期化
  if(argv[1] == "master") {
    int local_x = 123:
    DMI_alloc(..., GLOBAL, &addr);   ← DMI仮想メモリ割当
    DMI_write(addr, ..., &local_x);   ← DMI仮想メモリへwrite
    DMI_status(&node_list, ...);      ← ノードたちのステータスを取得
    node = newcomer in node_list;
    sprintf(args, "worker %d %d ...", ...);  ← 引数を作成
    DMI_create(&hdl, node, args);   ← ノードを指定してDMIプロセスを生成
    ...
    DMI_join(hdl, ...);   ← DMIプロセスを回収
    DMI_free(addr);     ← DMI仮想メモリを解放
  } else if (argv[1] == "worker") {
    int local_x;
    DMI_addr(GLOBAL, &addr); ← DMI仮想メモリのアドレス取得
    DMI_read(addr, ..., &local_x);   ← DMI仮想メモリからread
    ...
  }
  DMI_finalize();  ← 終了
}
```

引数渡し

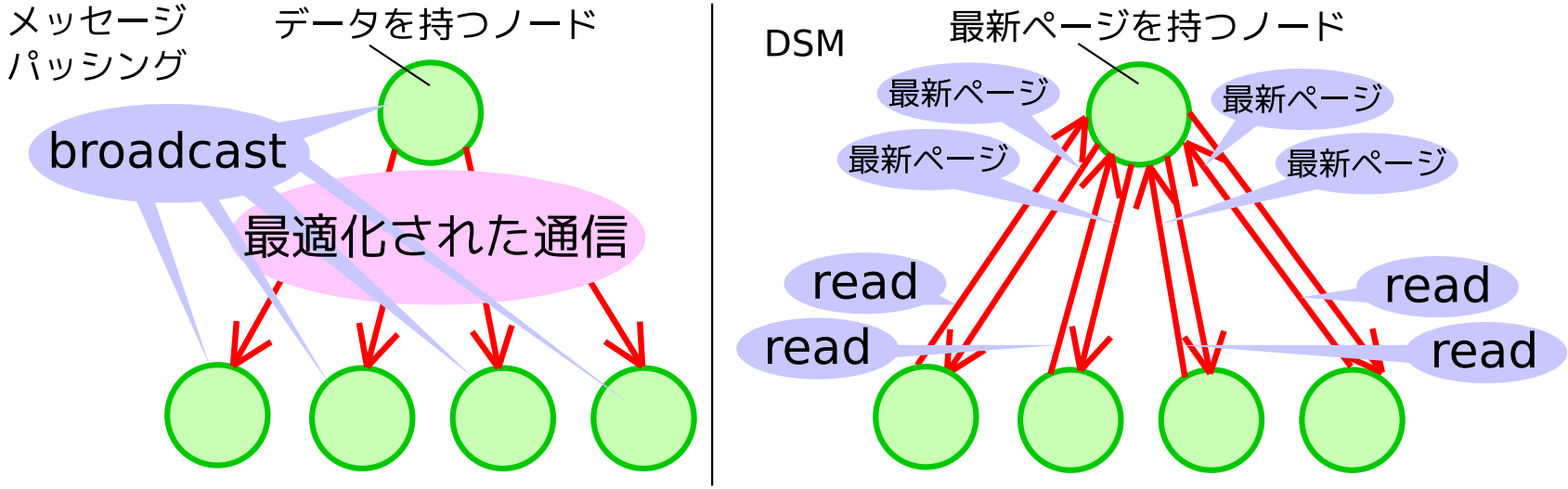
ラベル

# Approach 4：pthread 型の記述スタイル (3)

- pthread プログラムと DMI プログラムの対応付け
  - → スレッド生成 $\Longleftrightarrow$ プロセス生成
  - → スレッド起動時の関数への引数 $\Longleftrightarrow$ プロセス起動時のコマンドライン引数
  - → グローバル変数への read/write $\Longleftrightarrow$ DMI メモリ空間への DMI_read()/DMI_write()
- 並列プログラミングモデル的には pthread プログラムからほぼ機械的に変換可能
  - → 当然 OS 的にはたくさん問題が起きる
    - ♦ I/O，プロセス間通信，各種システムコール

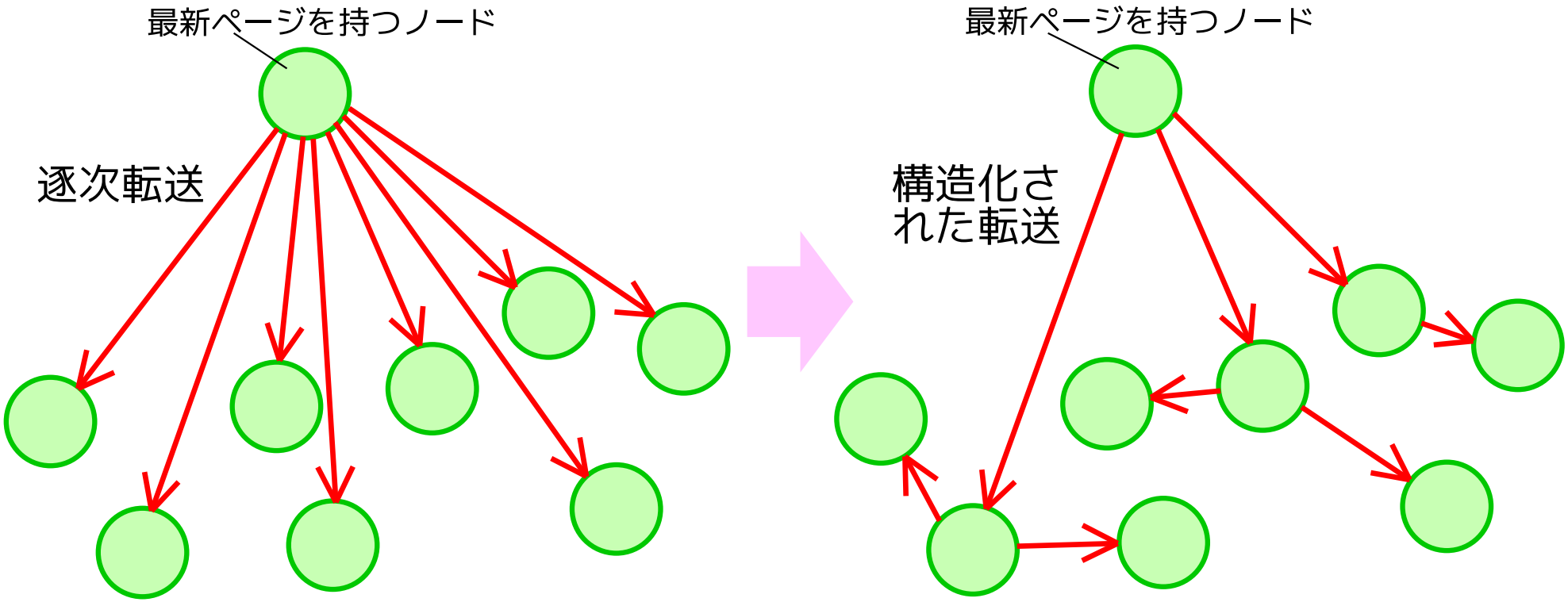
# Approach 5：ページ転送の動的負荷分散 (1)

- 性能上，集合通信（特に broadcast，alltoall）の最適化が重要
- メッセージパッシングでは：
  - → 専用の集合通信 API が存在し処理系による最適化が可能
- DSM では：
  - → 通常の read/write のシンタックスのみでは処理系側で通信パターンを把握できない
  - → pthread 型のスタイルに集団操作は（安直には）存在しえない



# Approach 5：ページ転送の動的負荷分散 (2)

- ページ転送を動的に構造化
  - → 各ノードが，自分に届いたページ要求の一部を，すでに自分がページ転送したノードにフォワーディング



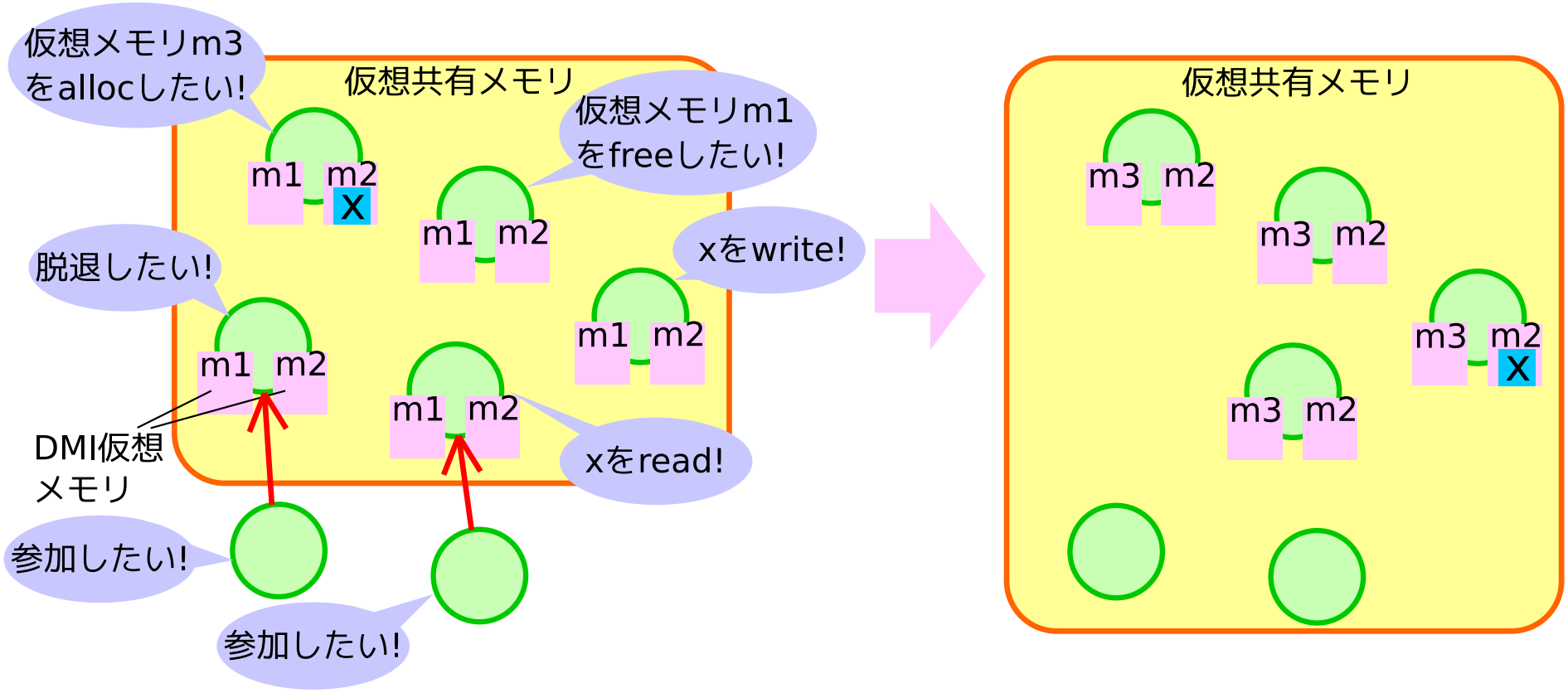
# DMI の API

- 初期化/終了
- メモリ確保/解放
- DMI プロセスの生成/回収
- 通常の read/write
- 非同期 read/write
- ロック機構
- 条件変数
- メモリフェンス
- ... その他ニーズに応じた機能拡張

# 4. DMI の実装

- ページのコンシステンシ管理
- ノードの動的な参加/脱退

# 目標




➤ **何がどういうタイミングで起きようが，（一定のルールの下で）コンシステントに遷移させる**

# ページのコンシステンシ管理 (1)

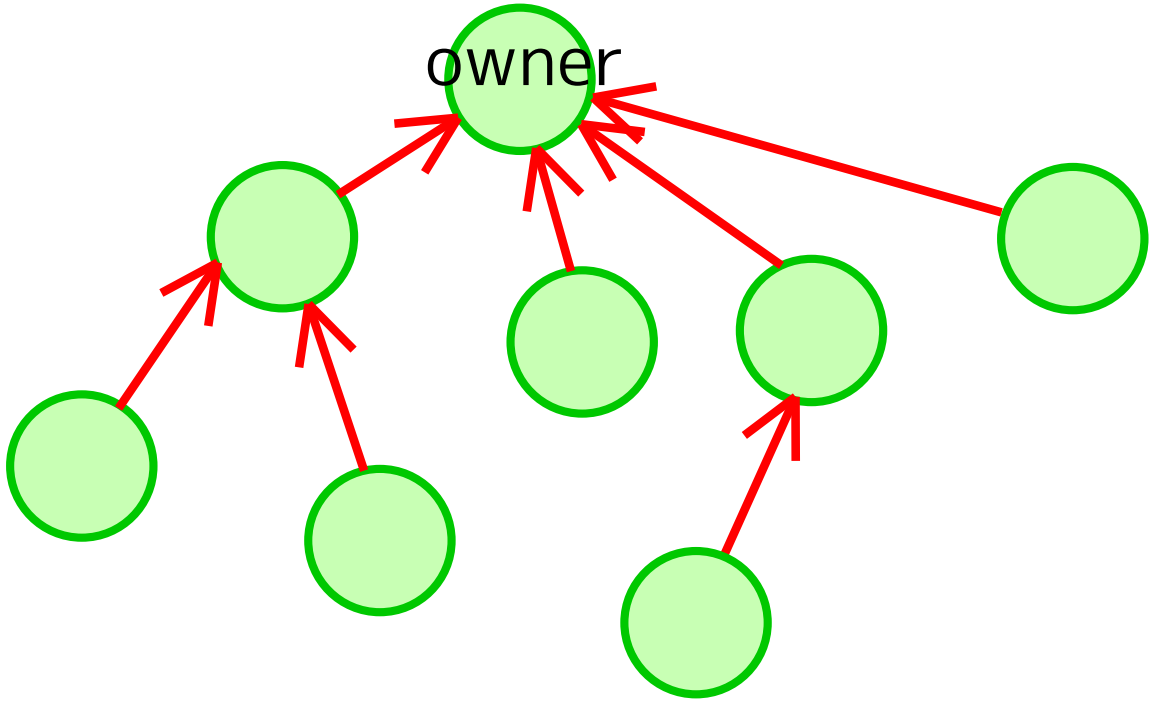
- Sequential Consistency
- Single Writer 型
- Write Invalidation 型
- **コンシステンシ管理はページ単位で独立**
  - → **複数ページへの要求を並列に処理可能**

# ページのコンシステンシ管理 (2)

- ページに関して管理する情報
  - → ページの状態
    - ♦ CLEAN(read：可,write：可)
    - ♦ SHARED(read：可,write：不可)
    - ♦ INVALID(read：不可,write：不可)
  - → オーナーの位置
- 不変条件：任意のページに関して，全ノードを通じて，
  - → オーナーが常に 1 個存在
  - →「CLEAN が 1 個，残りが INVALID」または「CLEAN が 0 個，SHARED が 2 個以上，残りが INVALID」

# ページのコンシステンシ管理 (3)

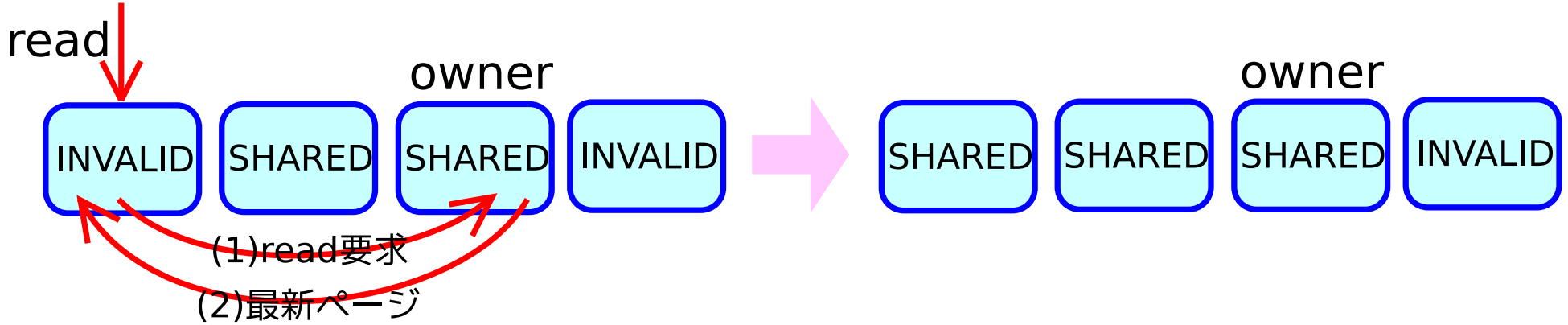
- オーナーの位置を常に把握するホームノードを設けない
- オーナー追跡グラフを形成
  - → オーナーの参照関係を辿ることで真のオーナーに到達可能
  - → ページフォルト時にはリクエストをフォワーディング
- Li らのアルゴリズム [Li et al,1989] を改善した上で実装



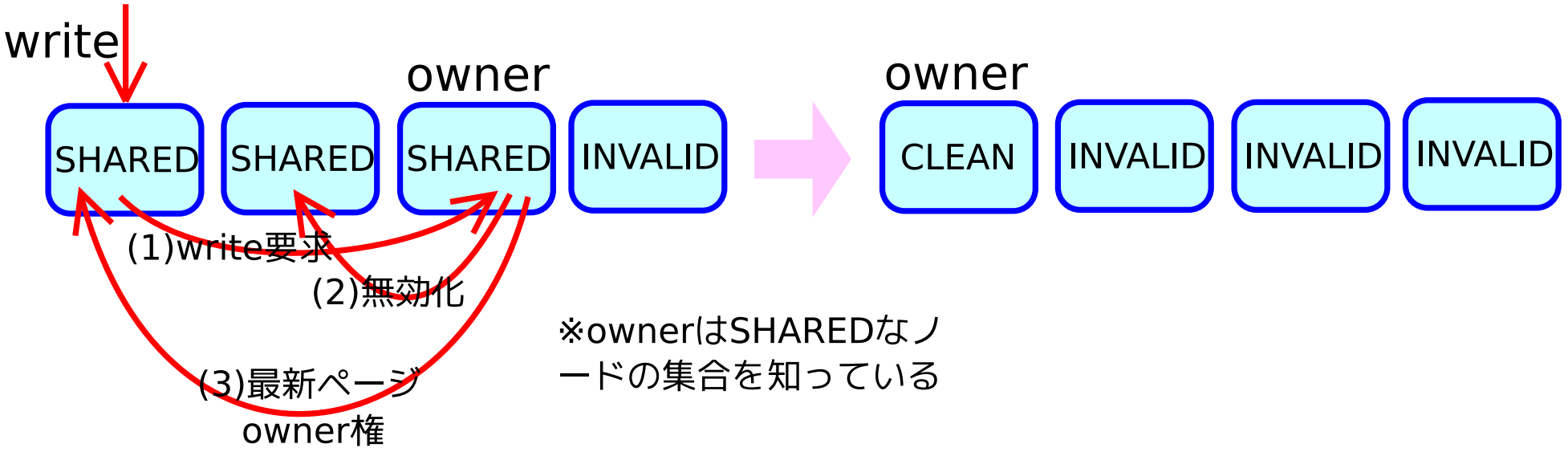
# ページのコンシステンシ管理 (4)

## ページフォルト時の挙動の例：

[リード違反の例]

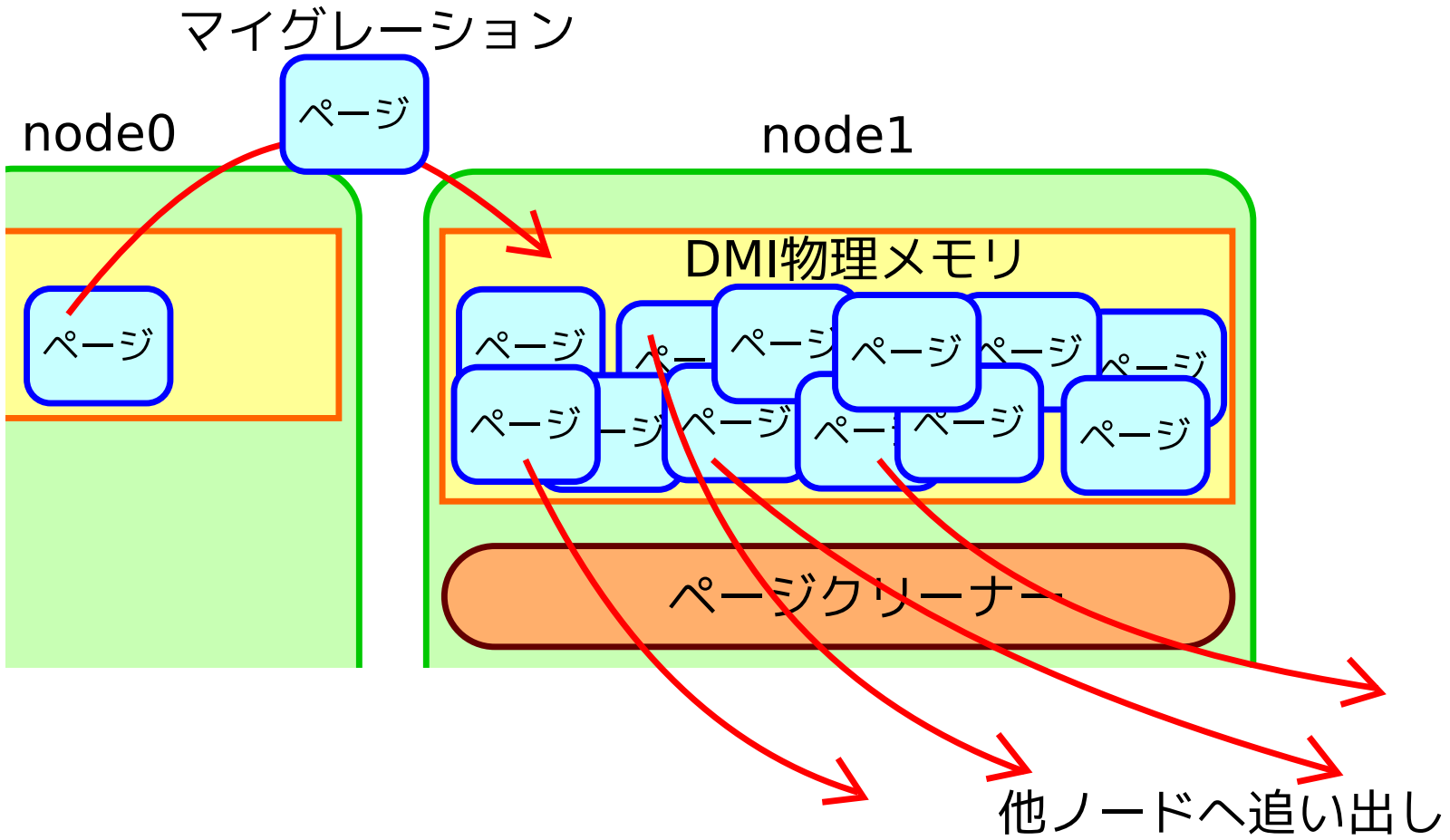



[ライト違反の例]



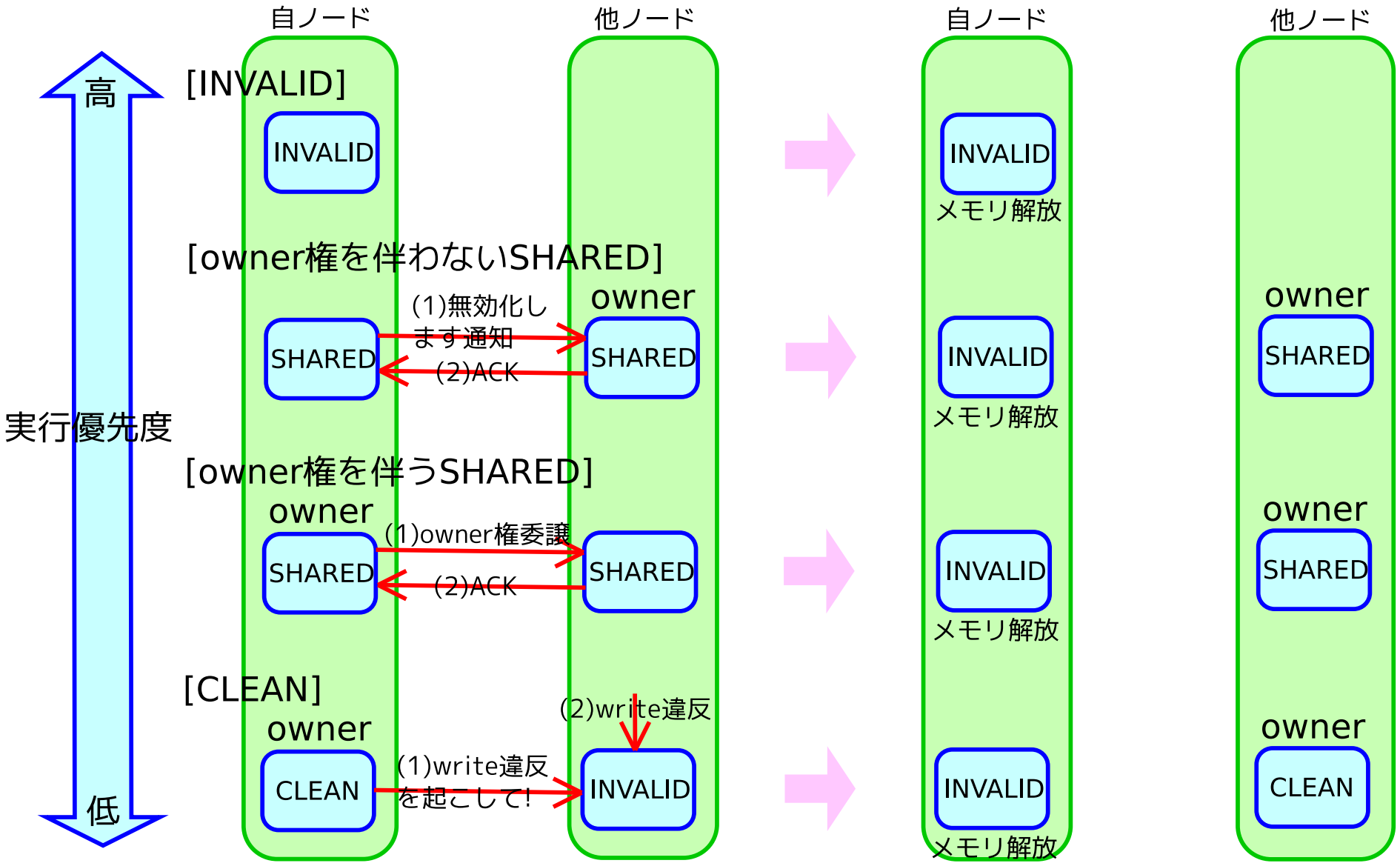
# ページ置換 (1)

- アクセスの度に DMI メモリ空間の使用メモリ量は増加
- ページクリーナー：
  - → 使用メモリ量が既定値に達すると適宜ページを追い出す

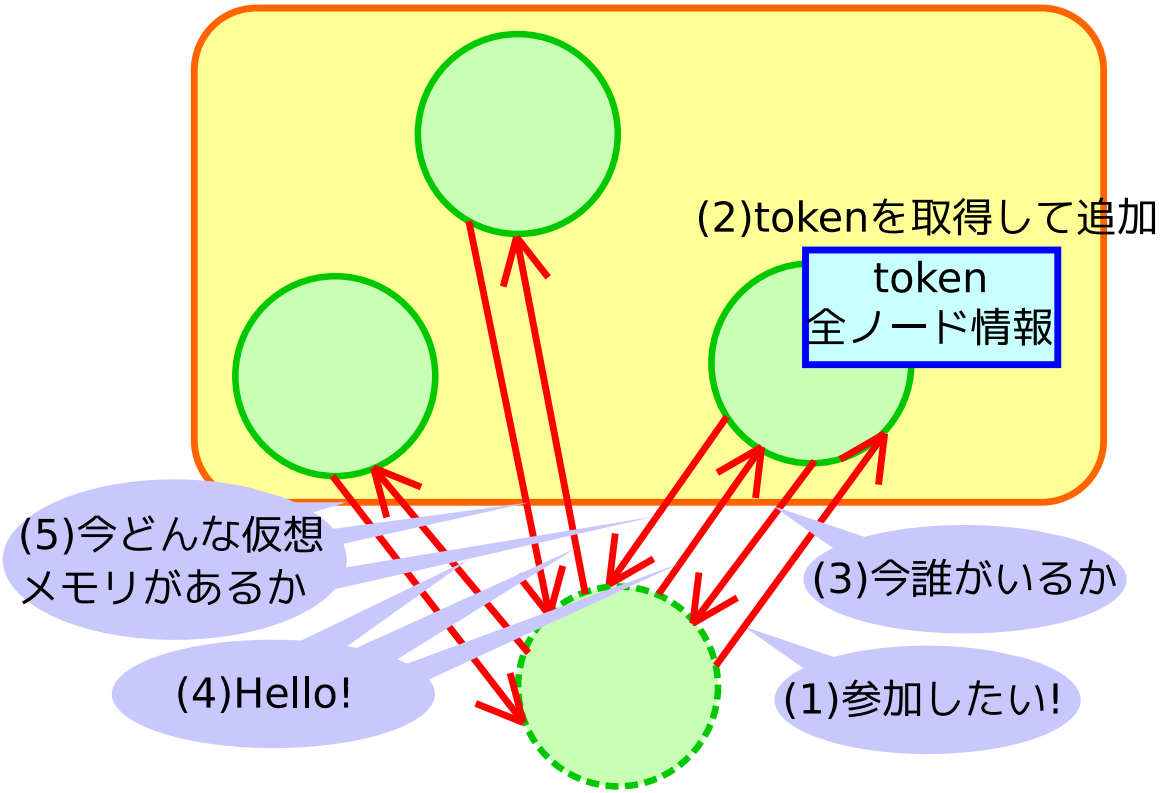


# ページ置換 (2)

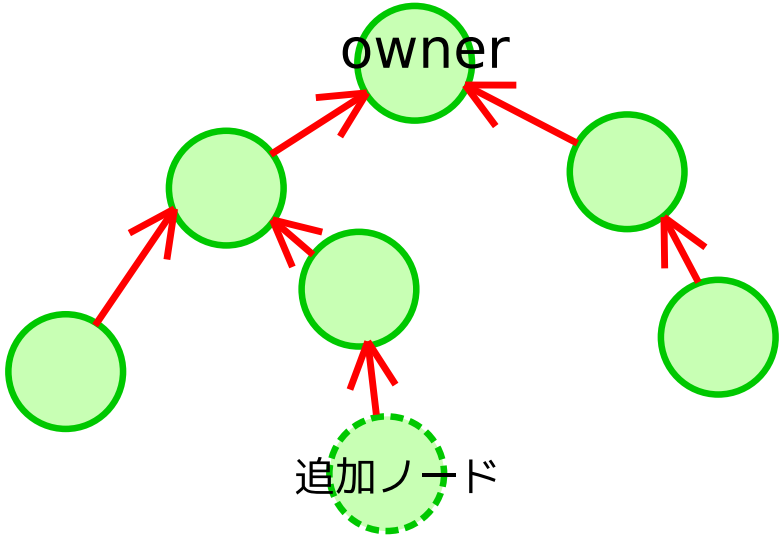
## ➤ ページクリーナーの具体的な動作：



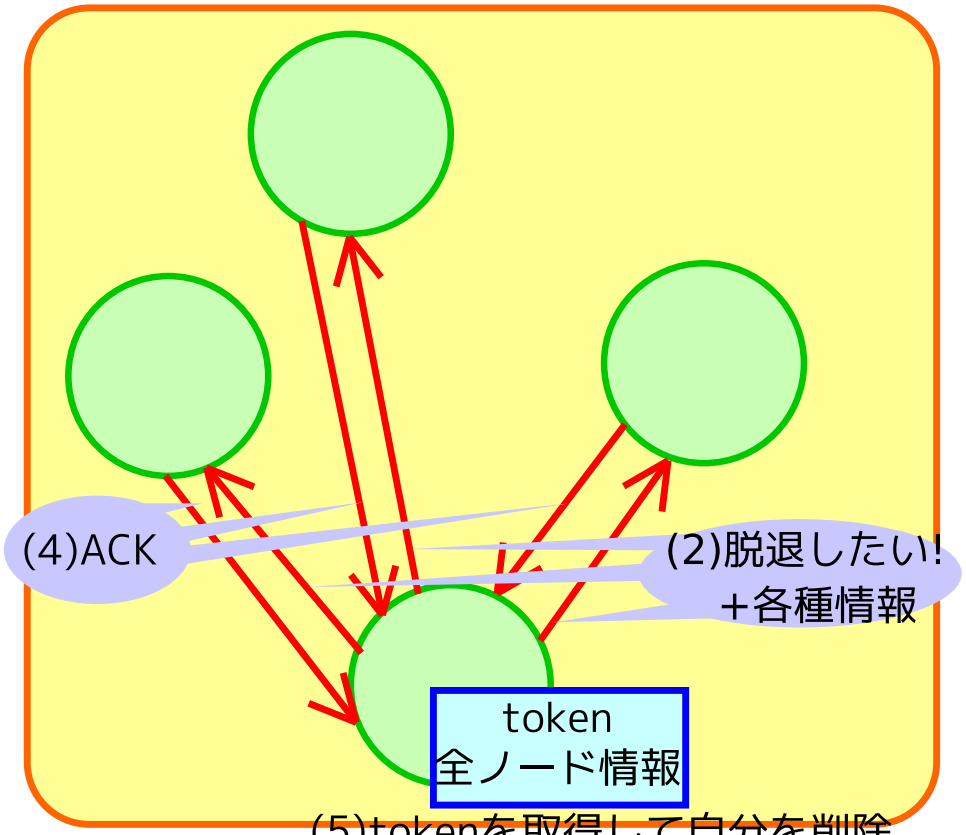
# ノードの参加




(6)仮想メモリを作成
(7)全ページをINVALIDに
(8)各ページのownerを適当に決める

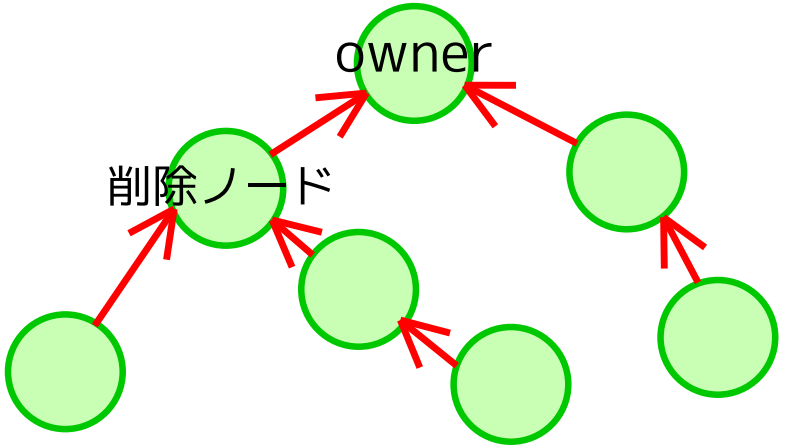


# ノードの脱退

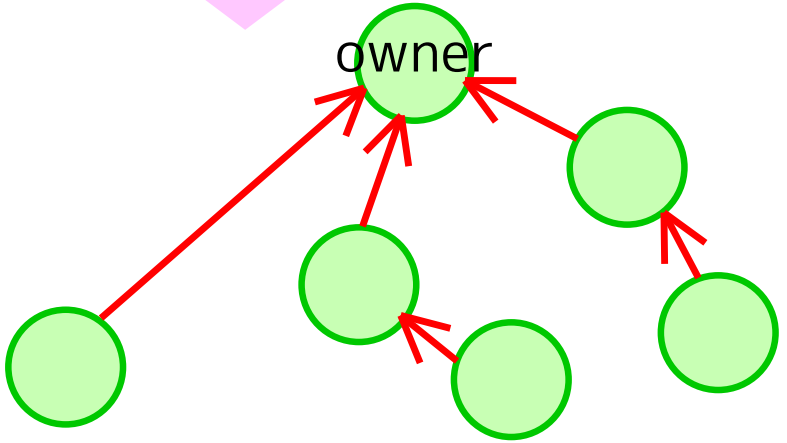



(5)tokenを取得して自分を削除

(1)ページクリーナーが全ページを追い出し




(3)owner追跡グラフの再形成

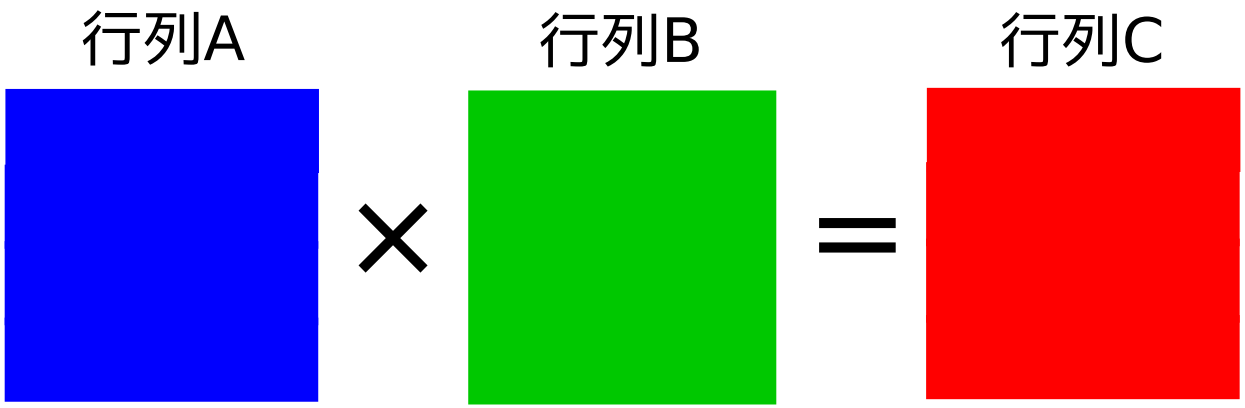


# 5. 予備的性能評価

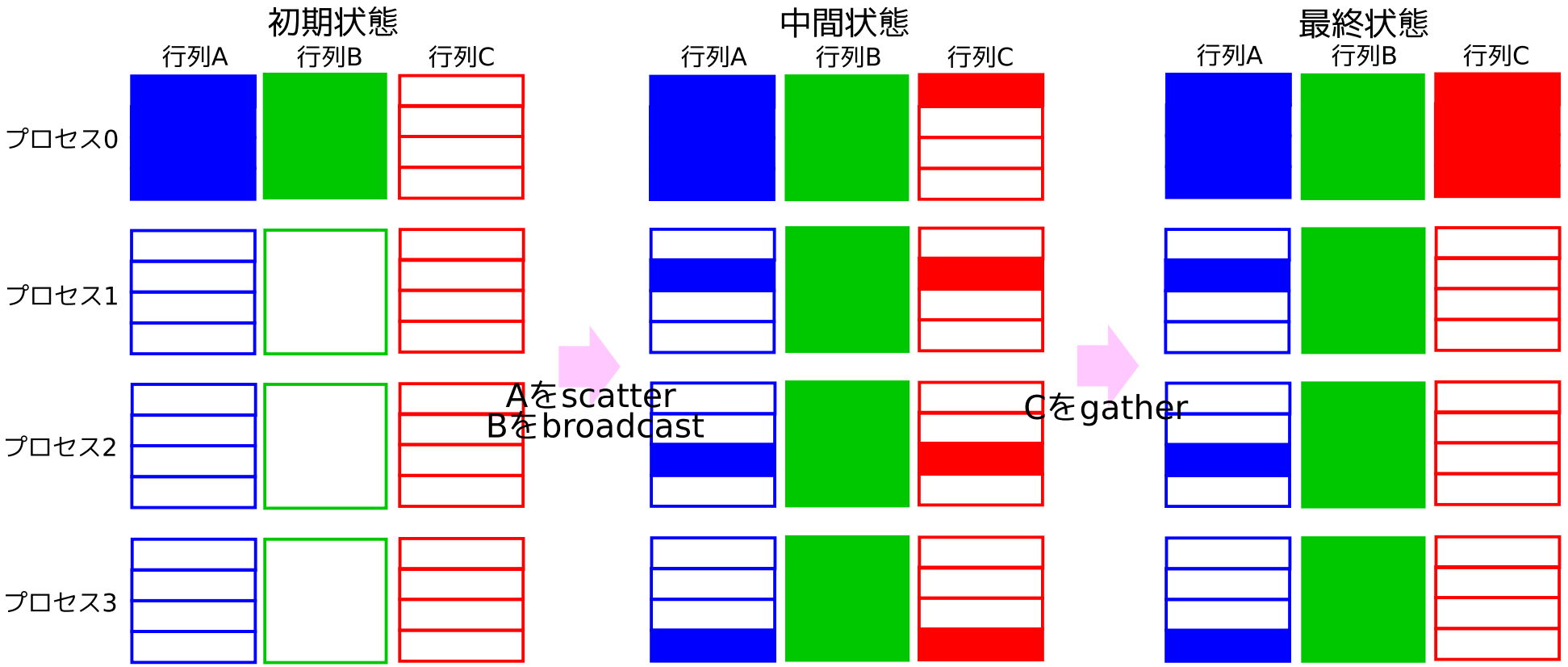
- 行列行列積を題材に MPI と DMI を比較

# 実験：行列行列積

- $2048 \times 2048$ の行列行列積 $AB = C$ を単純なアルゴリズムで実装
- MPI と DMI を性能比較
  - → 注：DMI はノードの増減，ページ置換，一部の API，ページ転送の動的負荷分散は未実装

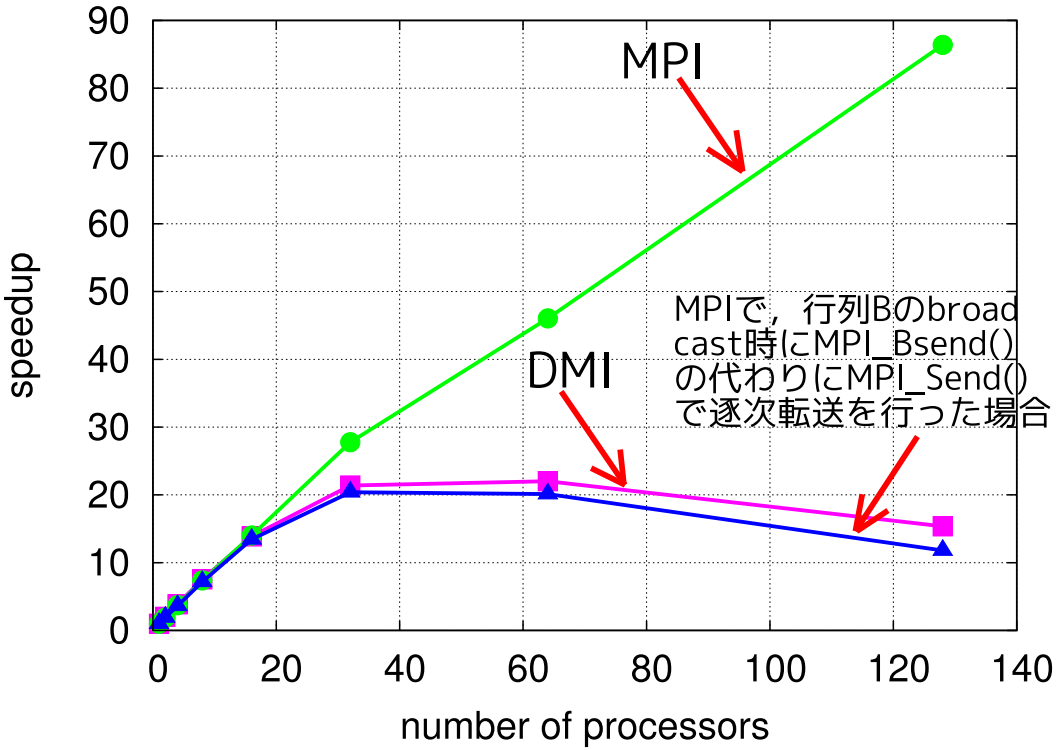


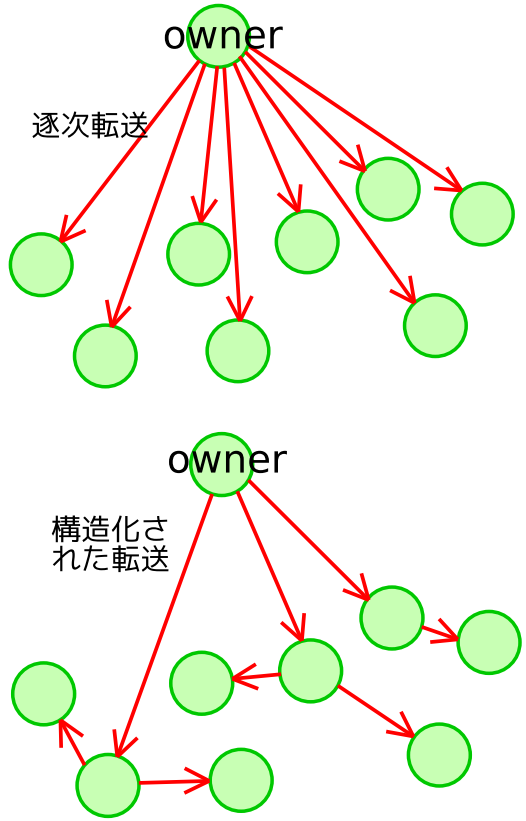
# データのフロー：行列行列積




- DMI では各行列ブロックを 1 ページに設定
  - → 各ページに対して 1 回しかページフォルトが起きない

# 結果




- DMI は 30 台弱までしかスケールせず
- 性能劣化の原因のほぼ全てが，行列 $B$ の broadcast 部分に起因
  - → ページ転送の動的負荷分散が必要

# 6. まとめ

# DMI の特長

- 大規模分散共有メモリ
- ノードの動的な増減をサポート
- pthread プログラムからの飛躍が小さい
- ユーザレベルでメモリ管理を実装
  - → 柔軟性の高い API
  - → 機能拡張の自由度が高い

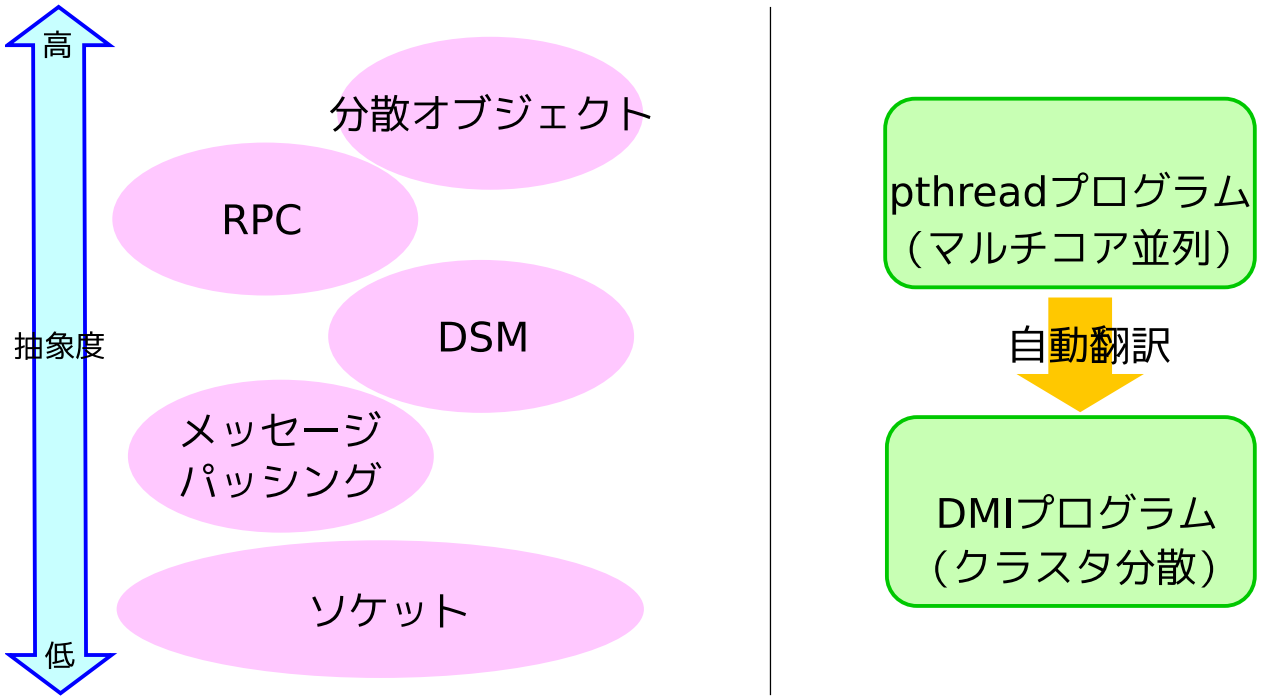
# DMI の応用可能性

- ノードの動的な増減を行いたいアプリ
- 複数ノードにまたがる巨大な共有メモリ空間が欲しいアプリ
- マルチコアレベルでは並列化されているが分散化には至っていないアプリ
- 例：
  - → 並列 model checking
  - → ページランク計算
  - → 巨大なゲーム木探索
  - → ...
- 並列分散ミドルウェアの基盤レイヤーとしての利用

# 現状と今後の予定

- 現状：
  - → 基本機能のプロトコル設計は（机上では）完了
  - → 鋭意コーディング中
- 今後の予定：
  - → ひたすらデバッグ
  - → プロトコルの model checking
  - → 機能拡張
  - → 性能評価
- 卒論締切りまでに「ひとまず動くもの」を，それ以降で「ちゃんと形にしていく」

# 将来的な展望




- DMI の上に，最初から分散処理を前提とした分散オブジェクト指向スクリプト言語処理系を開発
  - → DMI がオブジェクトのマイグレーションを実現
- （ある程度の制約はあるにせよ）pthread プログラムから DMI プログラムへの自動翻訳は可能？