2012-10-01版 (r2465)

powered by ALPS
http://alps.comp-phys.org/

Inside ALPS/looper

中核アプリケーション

大規模並列量子モンテカルロ法

http://wistaria.comp-phys.org/alps-looper/

藤堂眞治　松尾春彦

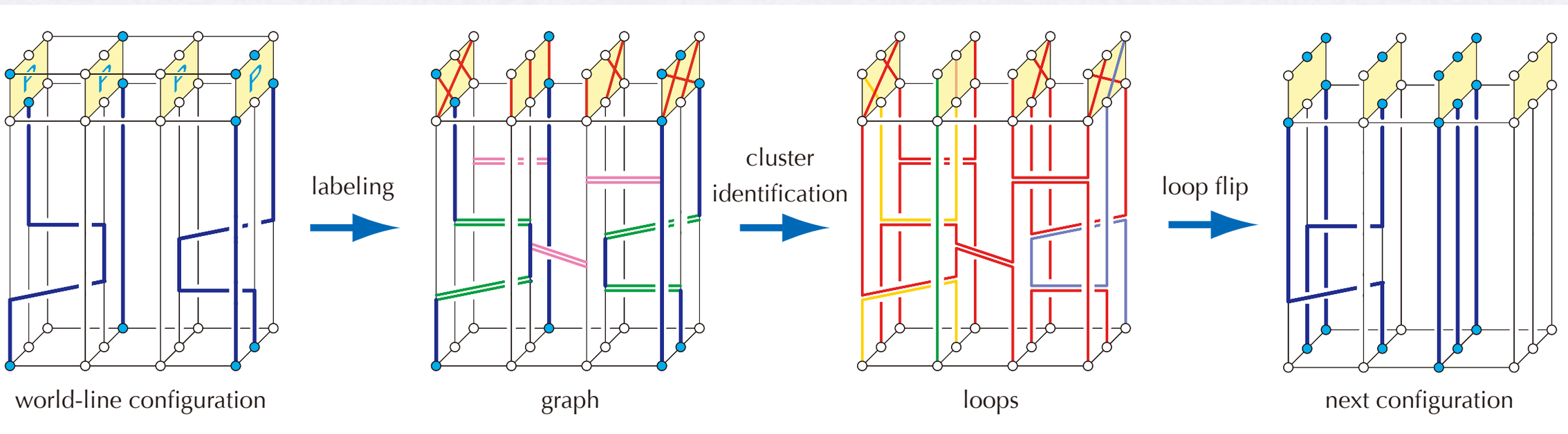
東京大学大学院工学系研究科

東京大学 THE UNIVERSITY OF TOKYO

連続虚時間ループアルゴリズム量子モンテカルロ法

(Continuous imaginary-time loop algorithm quantum Monte Carlo method)

- 量子多体系 (量子スピン系、ボーズ粒子系、etc)
  - 量子格子模型 (quantum lattice models)
    - 格子上(d=1,2,3,...)にスピン自由度 and/or ボゾン粒子数が定義
  - 虚時間経路積分により d+1 次元古典系に焼き直す (世界線表示)
  - スピン系やボゾン系の場合、補助場は必要なし (cf. 電子系、格子QCD)
- クラスター(ループ)アルゴリズムによる非局所更新 - 緩和は非常に速い
- あらかじめ連続虚時間極限を取ることが可能 - 離散化誤差なし



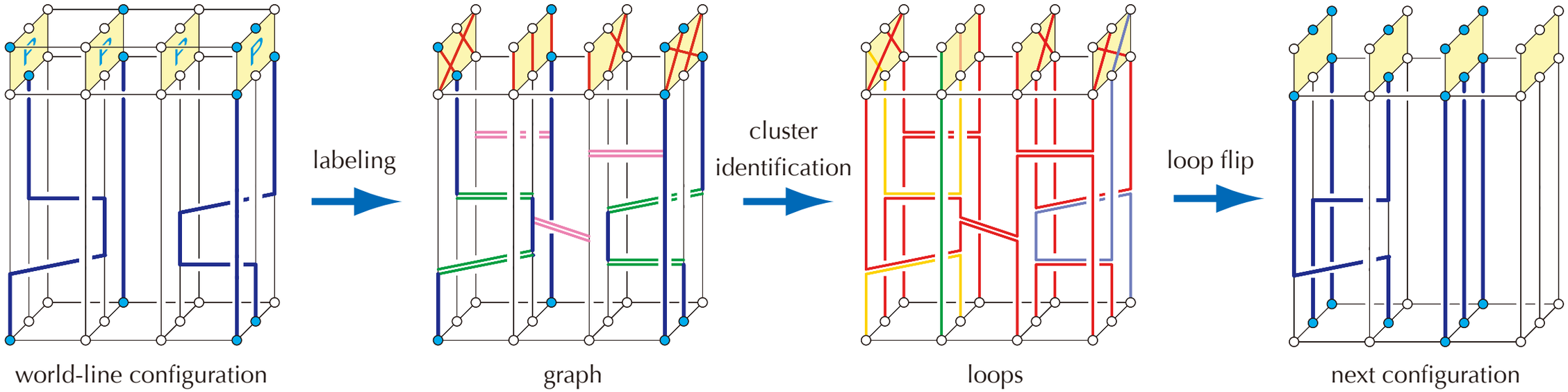
計算規模

- N: ボンド数 (格子の辺の数)
- β: 逆温度 (系の典型的な相互作用定数を単位とする)
- 必要メモリ量 ～ 世界線のキンク(ジャンプ)数 ～ N x β
- CPU (1モンテカルロステップあたり) ～ N x β
  - クラスター(ループ)アルゴリズムでは緩和時間が短かいので、系のサイズ (N, β)が大きくなっても、必要ステップ数はそれほど増えない
- メモリ量が制限となりうるような大規模系の計算が可能
  - 小さなギャップ(質量)の測定
  - 非常に弱い一次相転移
  - 大きな補正項を共なう臨界現象

目標計算規模 (マイルストーン)

- 現時点での最大計算規模 (筑波T2K全ノード 16コア x 640 ノード)
  - S=4 反強磁性鎖のギャップ測定
  - $N \sim 2^{22}$ $\beta \sim 2^{13}$ $\Rightarrow$ $N\beta \sim 2^{35} \sim 3 \cdot 10^{10}$
- 次世代スパコンでの目標 (8コア x 8万ノード)
  - S=5 反強磁性鎖のギャップ測定
  - $N \sim 2^{26}$ $\beta \sim 2^{17}$ $\Rightarrow$ $N\beta \sim 2^{43} \sim 8 \cdot 10^{12}$

One Monte Carlo step in loop algorithm



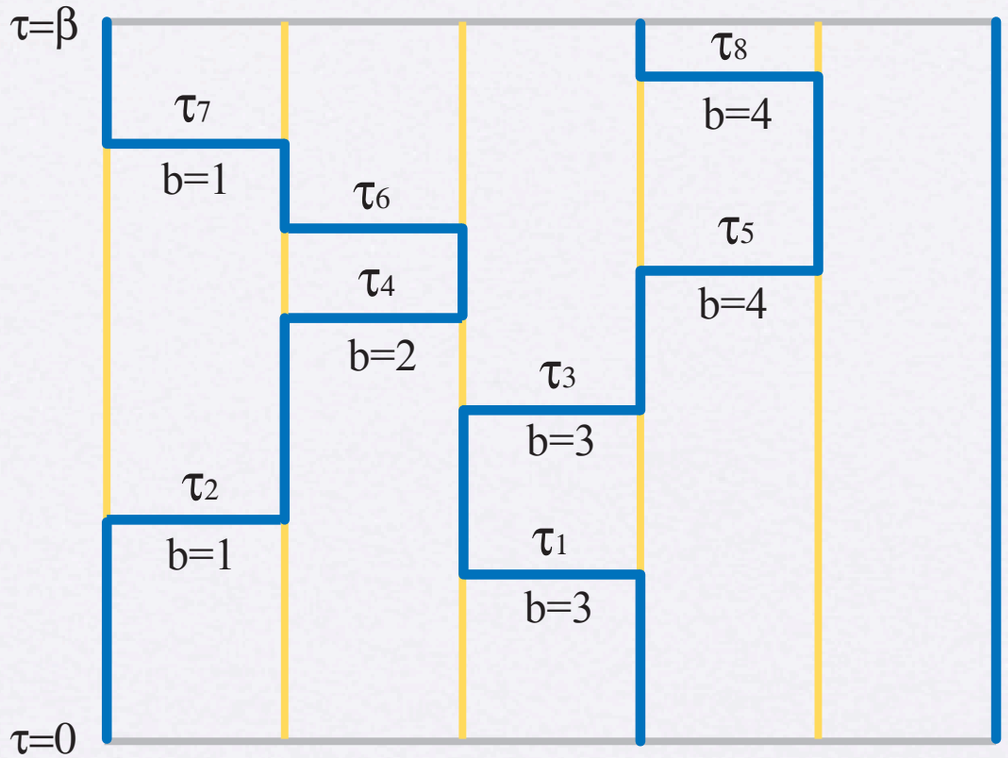
- ラベリング:
  - ある密度で横木を挿入 (両端の状態が反平行の時のみ採択)
- クラスター(ループ)認識
  - 全体をクラスター(ループ)の組に分割
  - もともとのキンク箇所(緑)でループは折り返す
  - 新たに挿入した横木箇所(紫)でもループは折り返す
- クラスター(ループ)反転
  - クラスター(ループ)毎に確率1/2で状態を反転 (図では赤と黄を反転)

状態 (世界線)の格納

- d+1 次元のうち
  - 実空間(d次元)方向には離散的 (実格子)
  - 虚時間(1次元)方向には連続
- メモリ上に格納するもの
  - 虚時間 τ=0 (一番下)の状態(0 or 1)

    std::vector<int> spins;
- キンク(世界線のジャンプ)の位置
  - (虚)時刻と場所(実格子の辺)
  - 全体で一つの一次元配列に保存

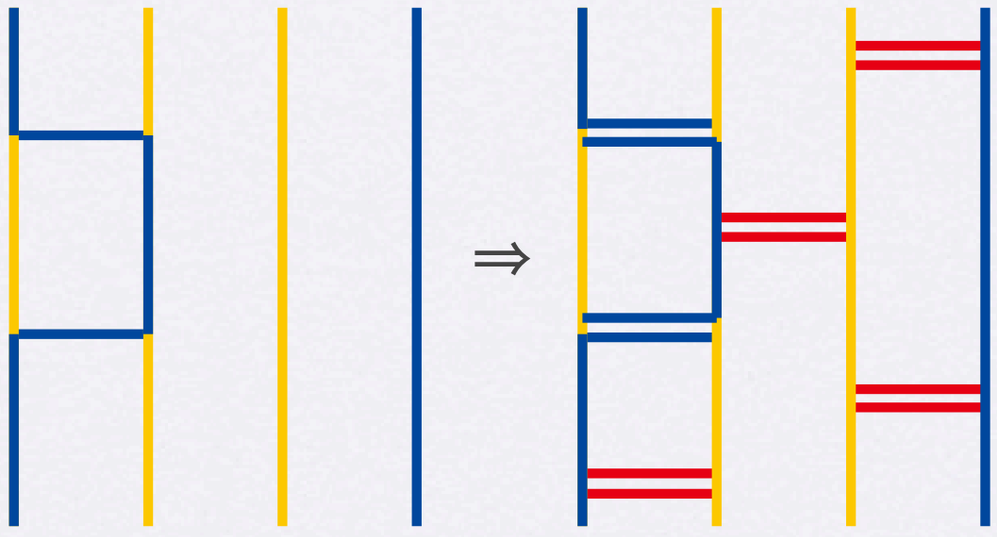
    std::vector<local_operator_t> operators;
  - キンクは虚時間順に並んでいる



連続虚時間経路積分表示

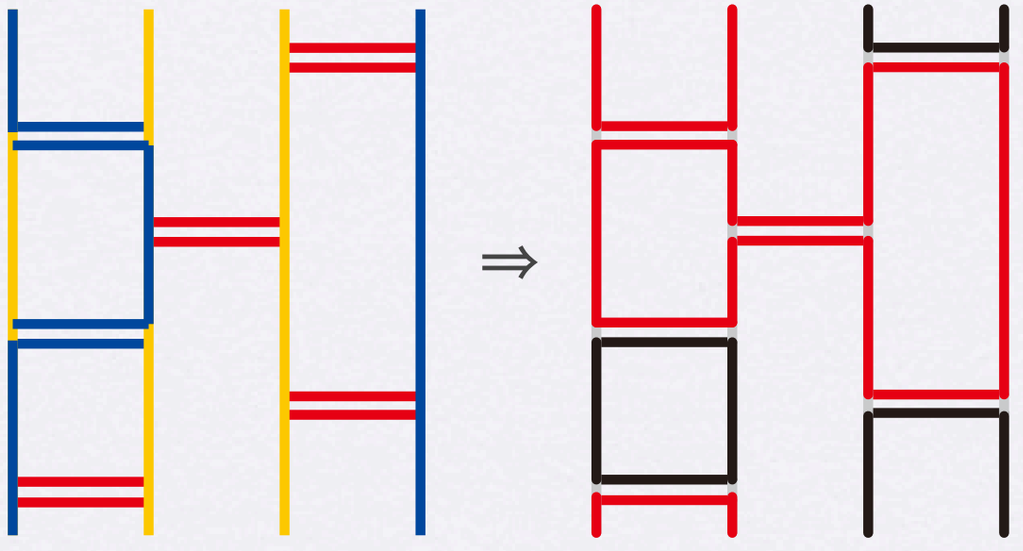
状態更新 (labeling, diagonal update)

- 新たに挿入を試みる横木の虚時刻を τ=0 から指数分布で順に生成、場所(辺)は挿入時にランダムに選ぶ
- キンク(operators_p に退避)と横木(times)の配列を τ=0 から走査
  - キンクは無条件に operators に追加
  - 横木は両端の状態が反平行の場合に限り operators に追加
  - 演算子(キンク + 横木)はキンク・横木の区別なしに虚時間順に並べる
- 横木挿入箇所のスピン状態 (spins)
  - 横木の採択/棄却に必要
  - τ=0 の状態を走査時に同時にアップデートする(キンク箇所で反転)



状態更新 (cluster identification)

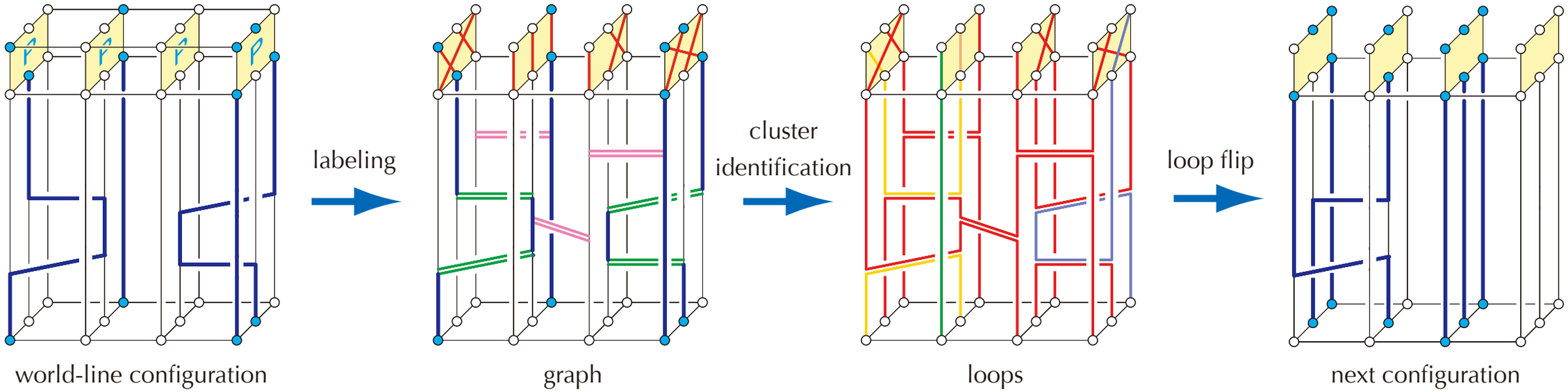
- 個々のノード(演算子を結ぶ世界線) のクラスター分け問題
- クラスター中のノードの並び順に関する情報は不要
- union-find アルゴリズムを使用
- ラベリング処理中の虚時間走査時にクラスター分けも同時に行なう
- 初期状態: サイト(実格子の頂点)数だけの独立したノードを用意(fragments)
- 現在の虚時間における各頂点におけるノード番号: current
- 演算子(キンク・横木)挿入時:
  - 演算子によりつながるノード同士を union
  - 新たなノードを挿入
- 上端: 各頂点のノードを下端のノードと union (周期境界条件)



状態更新(クラスターフリップ)

- ノードの配列 (fragments) を走査
  - ルートノードにクラスター番号を割り当て
  - ルート毎にフリップするかどうかを確率1/2で決める
- スピン状態 (spins) の更新
  - 各頂点のノードからルートノードを find
  - ルートノードのフリップ or not に従って spins を更新
- 演算子の更新
  - 演算子の上下を通っているノードのからルートノードを find
  - キンク ⇒ 横木、横木 ⇒ キンクなどの遷移が発生
  - 横木を取り除く (実際には次回のラベリング時に削除する)
- 物理量の測定

One Monte Carlo step in loop algorithm



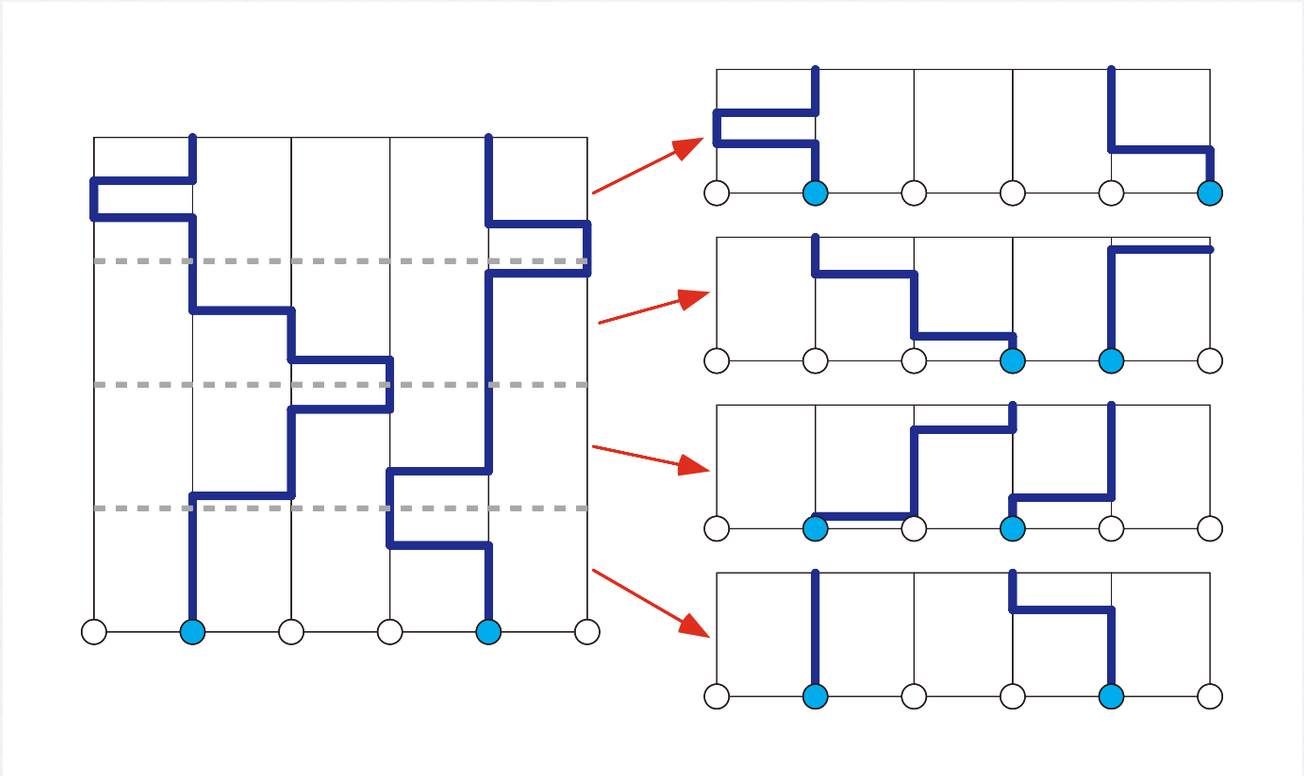
parallelization of loop algorithm is nontrivial!

- insertion of operators/cuts with constant density ⇒ Poisson process
- cluster identification ⇒ non-local operations on trees/lists

- cf) conventional local flip is easy to parallelize, but is suffered from critical slowing down ~(system size)^2

Parallelization of loop algorithm

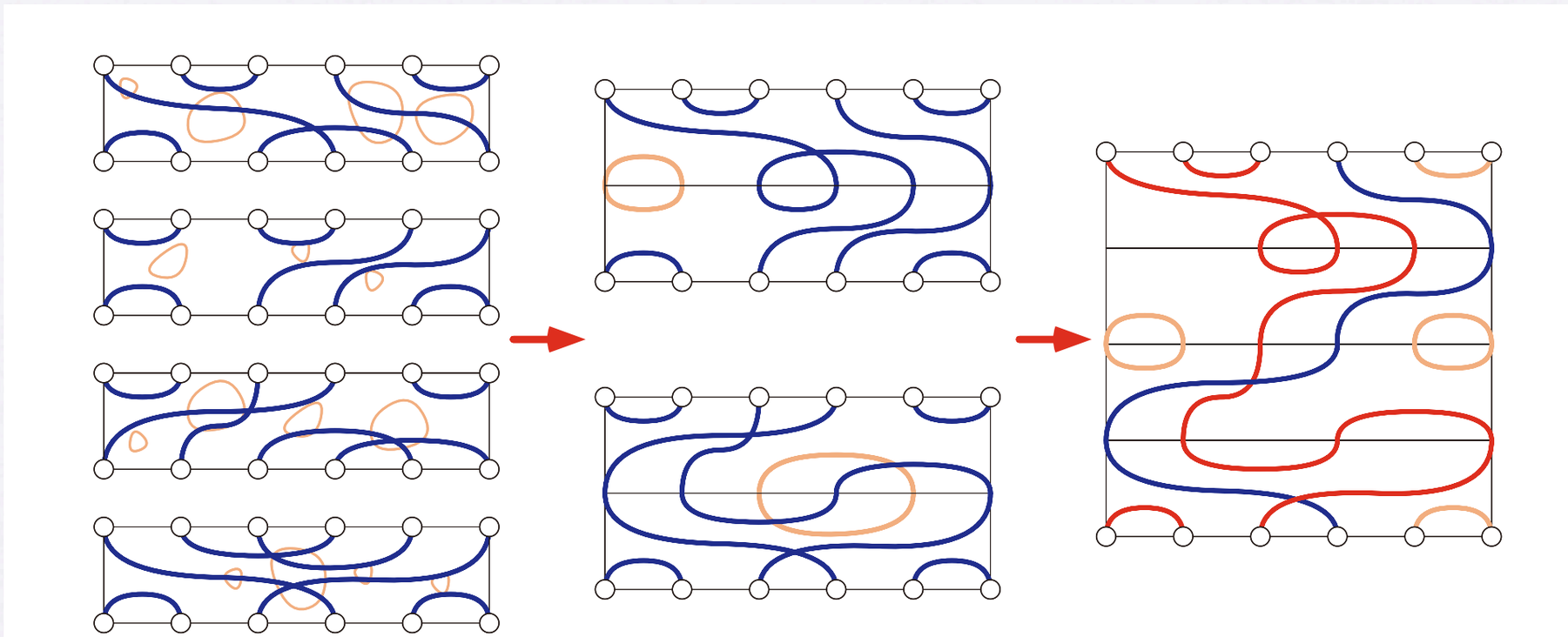
- distributing world-line configuration to several processors



- (d+1)-dimensional lattice is split into Np slices with equal imaginary time thickness
- labeling process can be performed independently on each process

Parallelization of loop algorithm

- binary-tree algorithm for cluster identification

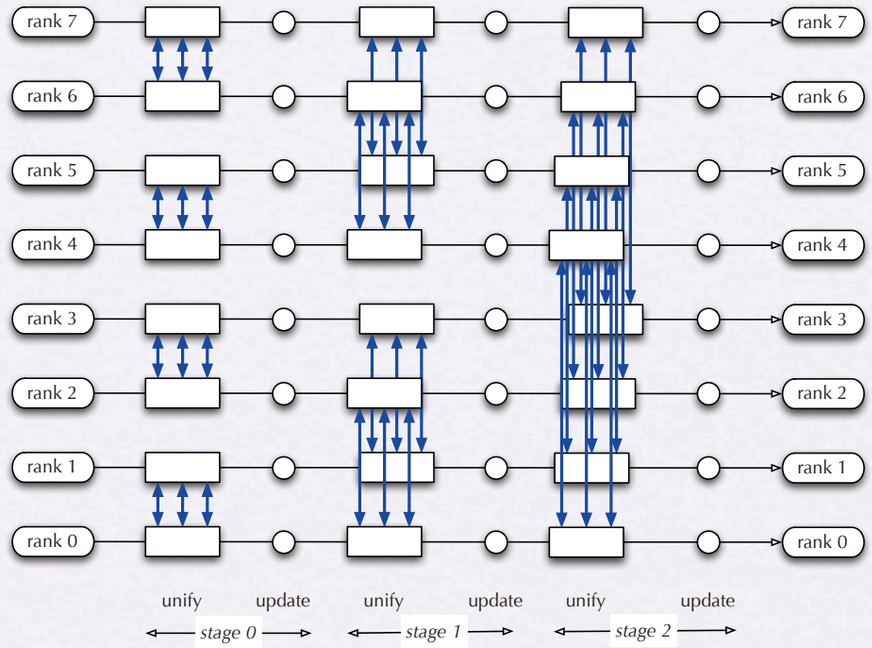


- overhead of parallelization is $(N \log N_{\mathrm{p}})/(\frac{N\beta}{N_{\mathrm{p}}}) = N_{\mathrm{p}} \log N_{\mathrm{p}}/\beta$

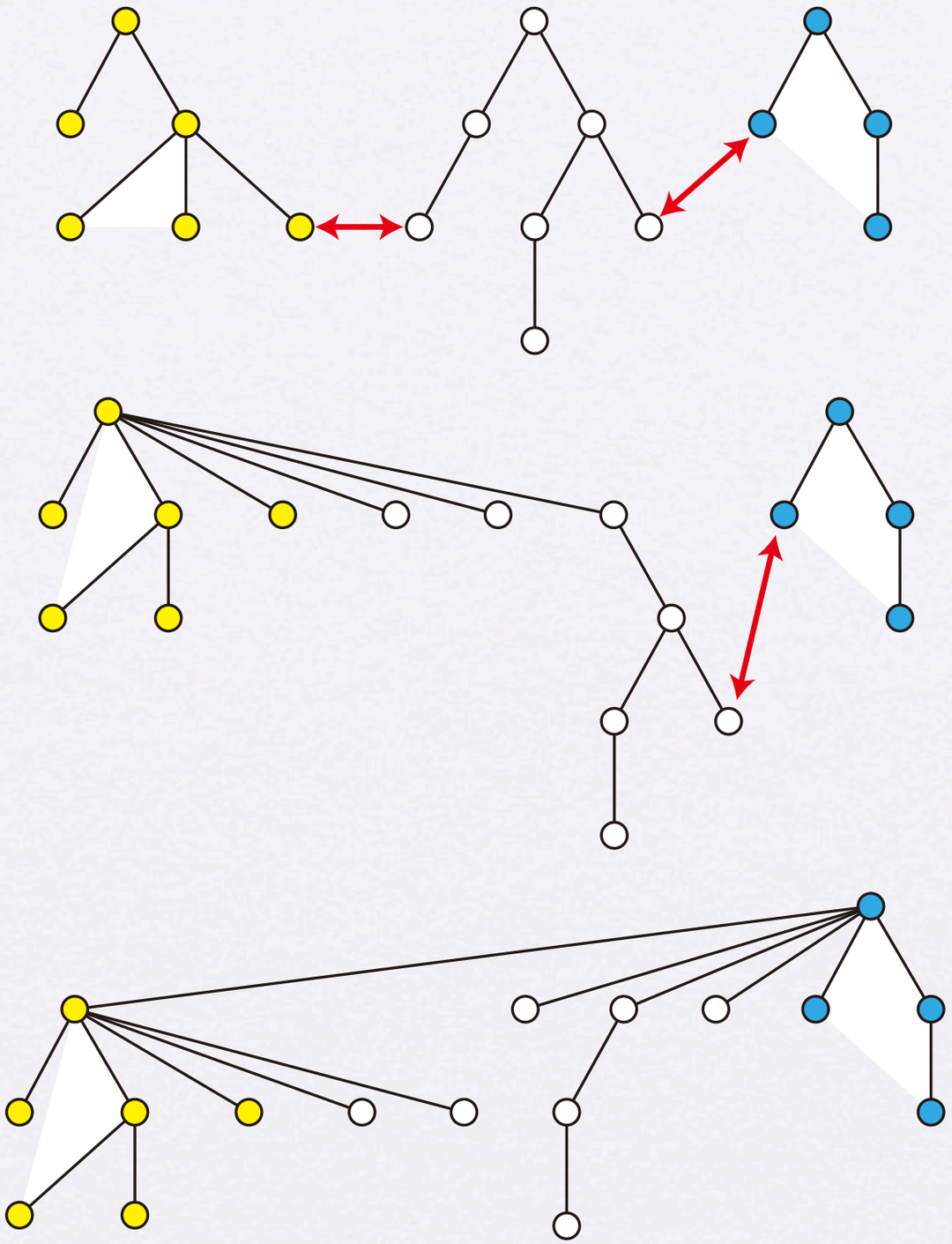
  (negligible at very low temperatures)

Further Optimization

- Introduction of butterfly-type communication
  - eliminates overhead in distribute process
  - speculation trick to determine loop directions with minimum communication
  - efficient for high-dimensional torus
- Reduction of number of stages
  - ‘bucket brigade’ adjacent communication
  - effective for low-dimensional torus
- Hybrid parallelization with MPI and OpenMP
  - multi-thread parallelization with Open MP
  - inter-node parallelization with MPI
  - reduces memory and communication overhead in each node
  - fine-grained parallelization of cluster identification by introducing asynchronous lock-free union-find algorithm



Asynchronous lock-free union-find algorithm



1. find root of each cluster/tree
2. unify two clusters
3. compress path to the new root

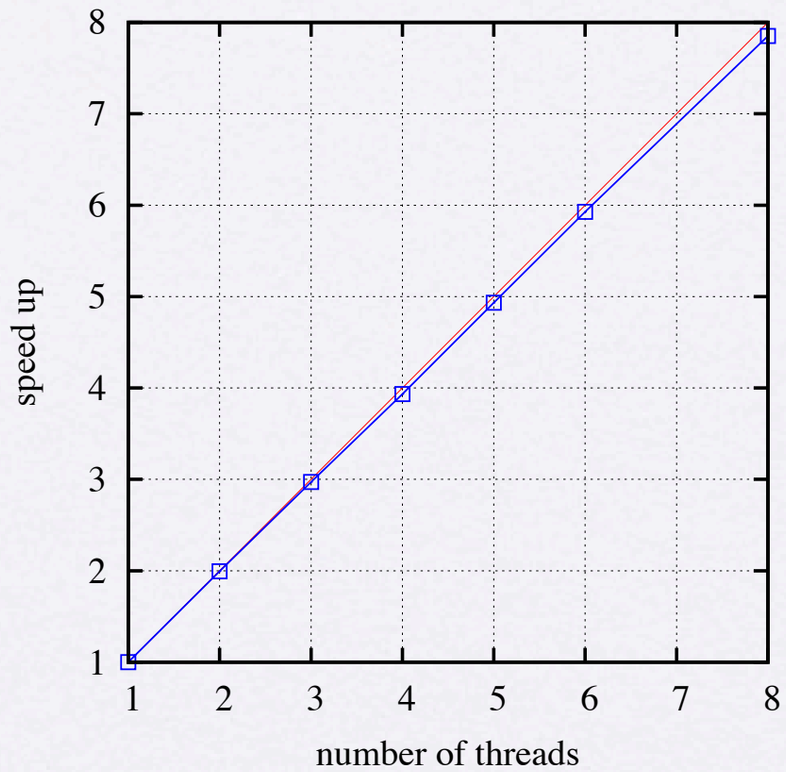
- locking whole clusters (e.g. by using “omp critical”) is too expensive

- finding root and path compression are “thread-safe”
- lock-free unification by using CAS (compare-and-swap) atomic operation

Thread-safe union-find algorithm

- Swendsen-Wang algorithm for 2d Ising model

  N = 8192 x 8192 at T = Tc

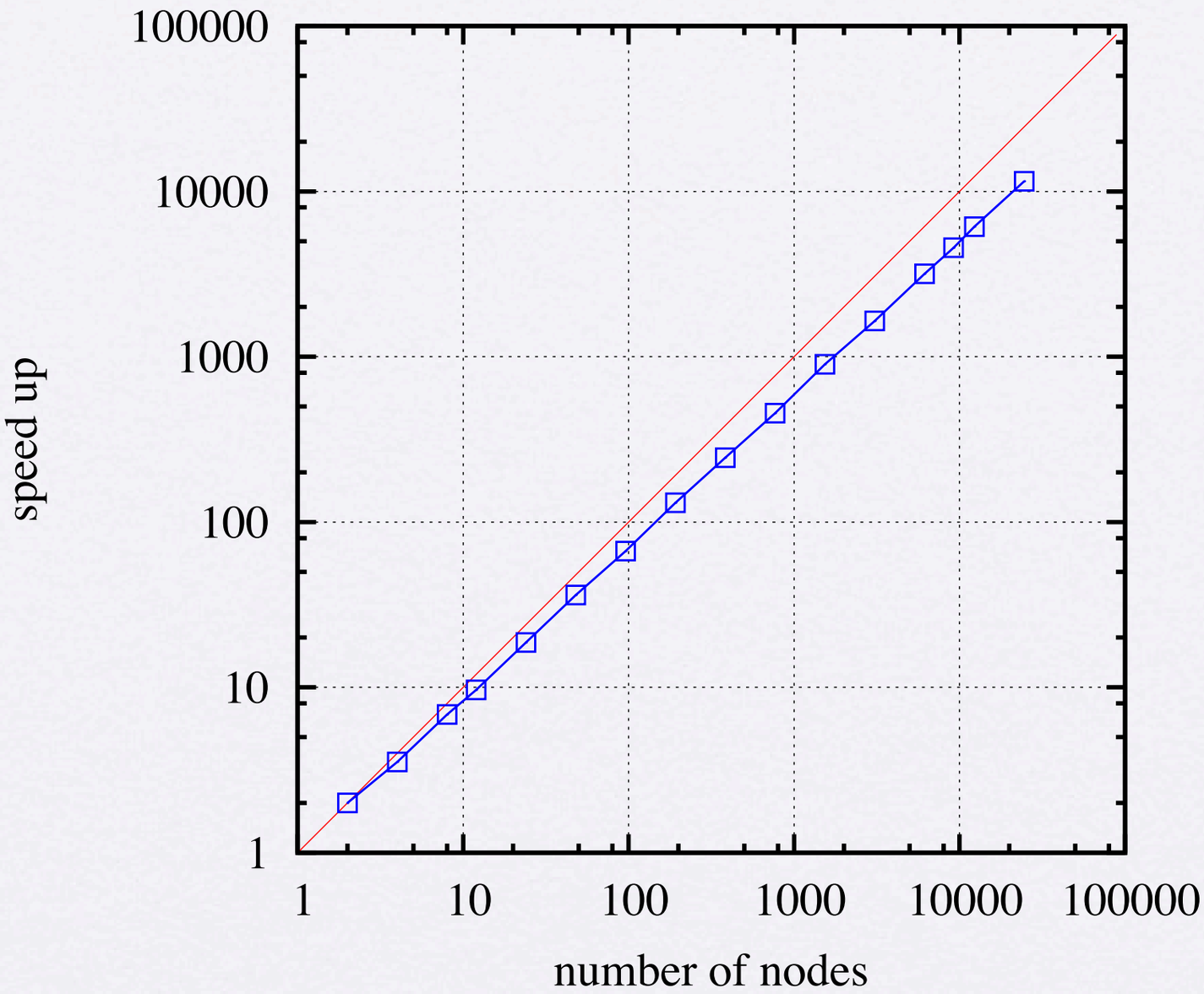


- OpenMP directives (“omp parallel for”) and CAS operations
- No memory overhead
- No need for optimized lattice decomposition
- Benchmark on T2K 20nodes: >190% speed up, >80% memory reduction

Thread parallelization of loop algorithm

- loop.C, loop_mpi.C: 実空間方向にスレッド並列
  - サイトで分割 (サイトを複数のスレッドで共有)
- 配列 operators, times をスレッド数だけ用意
- fragments, spins, current はスレッド間で共有
- クラス lattice_sharing
  - あるサイトが自分以外のスレッドとの間で共有されているかどうか?
  - 共有されている場合、そのスレッドIDは?
- 共有されているサイトに係る演算子を挿入する場合には他のスレッドが先に進んでいるかどうかを確認する必要あり: current_times
  - false sharing を避けるために padding
  - キャッシュが正しく更新されるよう “#pragma omp flush” を挿入

京におけるウィークスケーリングテスト



- 京 196608コア(24576プロセス x 8スレッド)で46.9%の並列化効率を実現