計算科学概論 (田浦先生担当課題)

東京大学 工学部 物理工学科 4 年
大橋 瑞輝 (学籍番号: 03-240540)
Email: ohashi-mizuki0510@g.ecc.u-tokyo.ac.jp

使用した PC の環境は以下の通りである。

- hardware: MacBook Air (Apple M3, arm64, 8-core CPU, 24GB RAM)
- OS: Darwin 24.5.0 (macOS 15.5)

問題設定

2 次元 XY 模型のモンテカルロシミュレーションの 1 つであるパラレルテンパリング法 (レプリカ交換メトロポリス法) を実装し、 命令レベルの並列化、OpenMP によるスレッド並列化、SIMD 命令によるデータ並列化を行い、性能を比較する[1]。

以下、2 次元 XY 模型およびパラレルテンパリング法の概要を、簡単に説明する。

2 次元 XY 模型

2 次元 XY 模型は、スピン i が連続的な角度 θ_i を持つ 2 次元格子上の物理系であり、以下のハミルトニアンで定義される。

$$H = -J \sum_{\langle i,j \rangle} \cos(\theta_i - \theta_j)$$

J は結合定数であり、実装では $J = 1$ とする。また、$\langle i,j \rangle$ は最近接スピンを表す。このハミルトニアンは、スピン間の相互作用を表し、スピンが同じ方向を向くほどエネルギーが低くなる。

低温においては Figure 1 のようにスピンが整列して同じ方向を向く。 一方で、高音になると Figure 2 のようにスピンの向きが乱雑になり、渦 (vortex) や反渦 (anti-vortex) が形成される。これは低音とは異なる相であり、このような相転移を KT (Kosterlitz-Thouless) 相転移と呼ぶ。

この系において重要な秩序変数 (相転移の指標) は、ヘリシティモジュラスと呼ばれる量である。 この量は以下のように定義される。

$$\Gamma = \frac{1}{N} \langle \sum_{\langle i,j \rangle_x} J \cos(\theta_i - \theta_j) \rangle - \frac{\beta}{N} \langle \left(\sum_{\langle i,j \rangle_x} J \sin(\theta_i - \theta_j) \right)^2 \rangle$$

高温領域では Γ は 0 であるが、KT 転移を境として低温側では有限の値を持つことが知られている。

本課題では、このモデルについて KT 転移が生じることを確認するために、モンテカルロシミュレーションを行い、ヘリシティモジュラスの値を計算する。なお、系のサイズは 32×32 とし、周期境界条件を用いる。

[1] 現在、物理工学科の授業 (≈ ミニ卒論) の題材として、2 次元 XY 模型のサンプリング結果を使用する予定である。 それに先駆けて、本課題を通して 2 次元 XY 模型のシミュレーションを行うことにした。

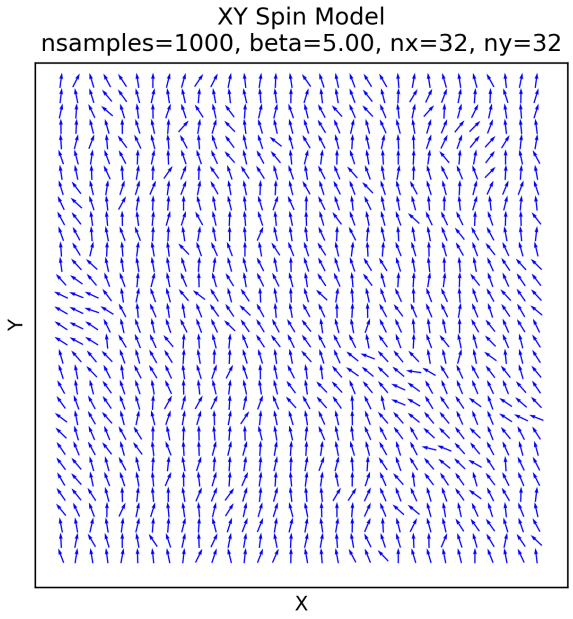

Figure 1: 2 次元 XY 模型の例 (低温領域)

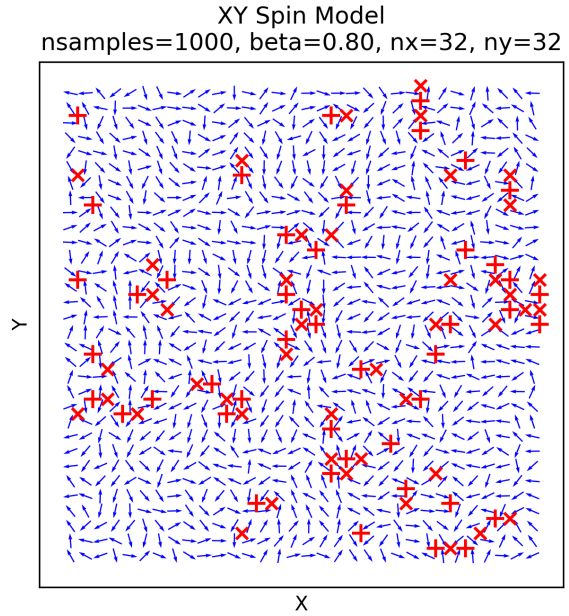

Figure 2: 2 次元 XY 模型の例 (高温領域)。図中の $+ \times$ は渦や反渦を表す。

メトロポリス法とパラレルテンパリング法

パラレルテンパリングは、通常のメトロポリス法を拡張したモンテカルロ手法のひとつであり、複数の温度のサンプルを並行して時間発展させ、一定のステップごとに異なる温度のサンプル間で交換を行うことで、より効率的に相空間を探索することができる。

通常のメトロポリス法ではある 1 つの (逆) 温度 β でシミュレーションを行う。 現在のサンプル X_n があるとき、ランダムな変化 (今回の場合は特定のスピンを選んでそのスピンの角度をランダムに変化させる) を行った後のサンプル X' を作る。 X_n と X' のエネルギーの差を $\Delta E = E(X') - E(X_n)$ とすると、$\min(1, \exp(-\beta \Delta E))$ の確率で X' を受け入れて

$X_{\{n+1\}} \leftarrow X'$ とする。 こうすることで X_n と X_{n+1} の間に詳細釣り合いの条件が成り立つので、十分長い時間このステップを続けると、サンプルは熱平衡状態 (カノニカル分布) に収束する。

パラレルテンパリングでは、複数の温度 $\beta_1, \beta_2, ..., \beta_N$ のサンプルを用意し、それぞれのサンプルに対してメトロポリス法を適用し、発展の途中の一定の間隔で、異なる温度のサンプル間で交換を行う。 この際にはふたつの温度のサンプルに関して温度差を $\Delta\beta$、エネルギー差を ΔE とすると、交換を行う確率は $\min(1, \exp(\Delta\beta\Delta E))$ である。 こうすることで、異なる温度のサンプル間でも詳細釣り合いの条件が成り立つので、全てのサンプルは熱平衡状態に収束する。

実装

ソースコードは `GitHub:Mizuki-OHASHI/xymodel` で公開しているが、本レポートの末尾にもソースコードの一部を掲載する。

ベースとなるコードは、上の XY 模型の説明をプロンプトに付した上で Gemini に生成させた C 言語のコードである (Appendix 1)。上で説明したパラレルテンパリング法が実装されており、各温度でのヘリシティモジュラスのサンプリング平均や実行時間 (wall time) を出力する。

ここでは、並列化の理解のために、処理の流れを簡単な疑似コードで示す。

```algorithm
時間の計測開始
各温度のサンプリングを初期化する
物理量_total = 0.0
i = 0
while (サンプリングした数 < サンプリング数) {
  for (各温度のサンプリング) { ・・・(1)
    全てのサイトのスピンについてメトロポリス法を適用する ・・・(2)
  }
  for (複数回) { ・・・(3)
    隣り合った温度のサンプリング間でレプリカ交換を行う ・・・(4)
  }
  if (初期緩和後のサンプリング間隔に達したら) { ・・・(5)
    各温度のサンプリングから物理量を測定する ・・・(6)
    物理量_total += 測定した物理量
  }
  i ++
}
出力ファイルに物理量の平均値 (物理量_total / サンプリング数) を書き込む
時間の計測終了
```

実行結果を Figure 3 に示す。 実行時間は **135 秒**であった。 プロットを見ると、$\beta = 0.9$ の付近でヘリシティモジュラスが 0 から有限の値に変化していることがわかり、KT 転移が確認できる。

なお、無限大の格子サイズにおける転移 (逆) 温度は 1.1 程度であることが知られているが、

格子サイズが有限である[2] ことから、転移温度が高温側にずれていると考えられる。

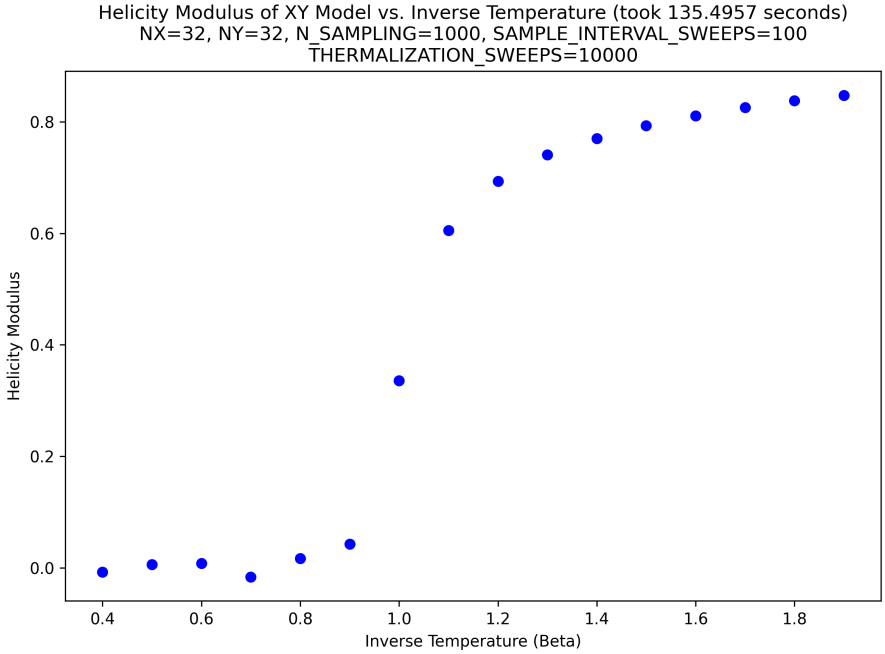

Figure 3: ヘリシティモジュラスのサンプリング平均 (オリジナルのコード)
(コンパイル: `clang original.c -o original -lm`)

以下、並列化などの工夫を行なった場合の結果が正しいことを確認する際には、(乱数を含む計算のため) 完全に同じ結果が得られないため、このプロットと同様のデータが得られることを確認する。

命令レベルの並列化

ソースコード: 変更なし

まずは、コンパイル時に最適化オプションを指定して、命令レベルの並列化を行う。 `clang -O3 original.c -o original_noopt -lm` としてコンパイルしたコードを実行すると、得られる結果は Figure 3 と全く同じで、実行時間は **97.2 秒** まで短縮された (最初と比較して 0.72 倍) 。

OpenMP によるスレッド並列化

ソースコード: `openmp.c` (GitHub)

続いて、OpenMP を用いてスレッド並列化を行う。 具体的には上の疑似コードにおいて以下の 2 箇所のループを並列化する。

- `(2)` のメトロポリス法の処理が重いので、各温度のサンプリングを並列化する (ループ `(1)` の並列化)
- `(6)` の物理量の計算は各々独立に実行可能なので、各温度のサンプリングごとに並列化する (ループ `(5)` の並列化)

なお、並列化を行わなかった箇所があるが、それは以下の理由による。

[2] 2 次元 XY モデルは相関長が無限大に発散することから、サイズが有限であることの影響を大きく受ける。

- (2) のメトロポリス法の処理の内部については、並列化の余地はあるものの、素朴にスピンごとの更新を並列化すると、隣り合ったスピンの更新が競合してしまう恐れがあるので、今は並列化しない。
- (4) のレプリカ交換については、アドレスの入れ替えをしているだけなので各処理はそこまで重くなく、しかも並列化することで隣り合ったレプリカ同士の競合が発生する可能性があるため、並列化しない。

なお、標準的に用意されている乱数生成器は、グルーバルに状態を保つことから並列化すると競合が発生する恐れがあるため、 スレッドごとに独立な乱数生成器を用意して、各スレッドで独立に乱数を生成するようにした。 そのためにオープンソースで公開されている PCG Random Number Generation, C Edition を用いた。

以上を並列化を行なって得られた結果を Figure 4 に示す。 実行時間は **25.3 秒**であった (最初と比較して 0.19 倍)。 プロットから、正しくシミュレーションが行われていると考えられる。

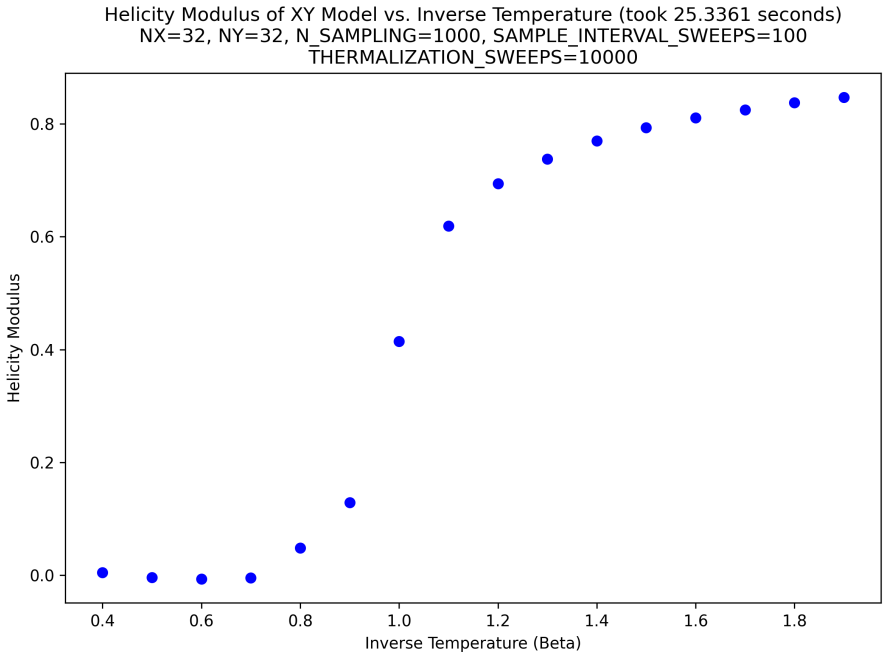

Figure 4: ヘリシティモジュラスのサンプリング平均 (OpenMP による並列化)

(コンパイル: `clang -Xpreprocessor -fopenmp -Rpass=loop-vectorize -march=native -ffast-math -lomp -I"$(shell brew --prefix libomp)/include" -L"$(shell brew --prefix libomp)/lib" -O3 simd.c -o simd -lm`)

SIMD 命令によるデータ並列化

ソースコード: `simd.c` (GitHub または Appendix 2)

次に上の並列化に加えて、SIMD 命令を用いてデータ並列化を行う。

(2) メトロポリス法の処理は、素朴に並列化すると、隣同士のスピンを同時に更新して競合してしまう (隣のスピンに依存して更新が行われるので、同時に更新されると不適当な値が計算される可能性がある)。 そこで、スピンの更新順序を工夫することで競合を回避する。具体的な工夫は Figure 5 の通りである。

オリジナルのコードでは、左上から順番にひとつずつスピンを更新していた。 そのため、そのままの順番で並列化すると、隣り合ったスピンの更新が同期してしまう。 そこで、市松模

様上にスピンを更新するように順序を変更した。スピンの更新が依存するのは、上下左右の隣接スピンのみであるため、市松模様上のスピンは互いに独立に更新できる。

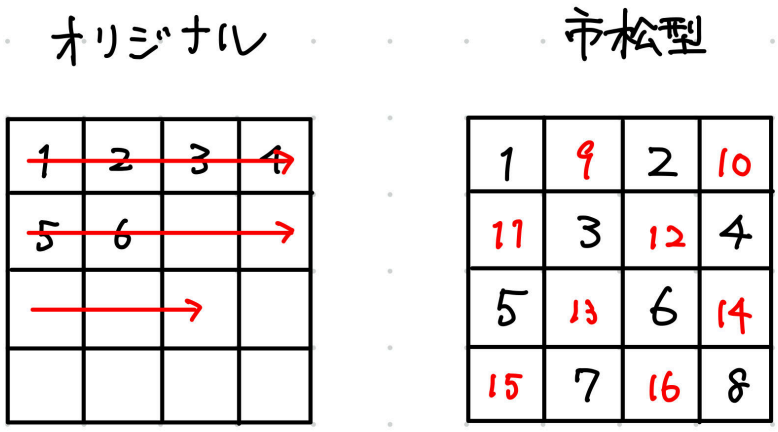

Figure 5: スピンの更新順序の工夫

順序を変更した上で、

1. 1 個飛ばしのスピンを連続するメモリ領域にコピーして (gather load)
2. IMD 命令によって一括で更新する。
3. 最後にメトロポリス法の判定で受け入れられたスピンを更新する (scatter store)

このようにすることで、スピンの更新を SIMD 命令で並列化することができる。 該当箇所のみ、ソースコードを示すと以下のようになる。

```c
// メトロポリス法の SIMD 化
void metropolis_sweep(Replica *rep, pcg32_random_t *rng)
{
  const int N_SITES = rep->nx * rep->ny;
  const float DELTA = 1.0f; // スピン角度の更新幅

  int odd, site_idx, idx, ix, iy, r; // loop variables

  // block variables for SIMD
  float spin[NL], new_spin[NL];                                    // spin block
  float neighborsR[NL], neighborsL[NL], neighborsD[NL], neighborsU[NL]; // neighbors blocks
  float delta_e[NL];                                               // energy change and acceptance probability
  float rand_val_acc[NL], rand_val_delta[NL];                      // random values for acceptance and delta
  int accept[NL];                                                  // acceptance flags

  // update each site in blocks of NL
  for (odd = 0; odd < 2; odd++)
  {
    // odd = 0 : even sites
    // odd = 1 : odd sites
    for (site_idx = odd; site_idx < N_SITES; site_idx += NL * 2)
    {
      // gather load
      for (r = 0; r < NL; ++r)
      {
        idx = site_idx + r * 2;
        ix = idx % rep->nx;
        iy = idx / rep->nx;

```

```
30          // load spin
31          spin[r] = rep->spin[idx];
32
33          // load neighbors
34          neighborsR[r] = rep->spin[iy * rep->nx + (ix + 1) % rep->nx];             // Right
35          neighborsL[r] = rep->spin[iy * rep->nx + (ix - 1 + rep->nx) % rep->nx];   // Left
36          neighborsD[r] = rep->spin[((iy + 1) % rep->ny) * rep->nx + ix];           // Down
37          neighborsU[r] = rep->spin[((iy - 1 + rep->ny) % rep->ny) * rep->nx + ix]; // Up
38
39          rand_val_delta[r] = (float)pcg32_random_r(rng) / (float)UINT32_MAX;
40          rand_val_acc[r] = (float)pcg32_random_r(rng) / (float)UINT32_MAX;
41        }
42
43  #pragma omp simd
44        for (int r = 0; r < NL; ++r)
45        {
46          new_spin[r] = fmodf(spin[r] + (rand_val_delta[r] - 0.5f) * DELTA + 2.0f * M_PI, 2.0f * M_PI);
47
48          // calculate energy change
49          delta_e[r] = 0.0f;
50          // calculate energy change with neighbors
51          delta_e[r] += -J * (cosf(new_spin[r] - neighborsR[r]) - cosf(spin[r] - neighborsR[r]));
52          delta_e[r] += -J * (cosf(new_spin[r] - neighborsL[r]) - cosf(spin[r] - neighborsL[r]));
53          delta_e[r] += -J * (cosf(new_spin[r] - neighborsD[r]) - cosf(spin[r] - neighborsD[r]));
54          delta_e[r] += -J * (cosf(new_spin[r] - neighborsU[r]) - cosf(spin[r] - neighborsU[r]));
55
56          accept[r] = (delta_e[r] < 0.0f || rand_val_acc[r] < expf(-rep->beta * delta_e[r]));
57        }
58
59        // scatter store
60        for (int r = 0; r < NL; ++r)
61        {
62          int idx = site_idx + r * 2;
63          if (accept[r])
64          {
65            // accept the new spin
66            rep->spin[idx] = new_spin[r];
67            rep->energy += delta_e[r];
68          }
69        }
70      }
71    }
72  }
```

この SIMD 化を行った上で、`clang -Xpreprocessor -fopenmp -Rpass=loop-vectorize -march=native -ffast-math -lomp -I"$(shell brew --prefix libomp)/include" -L"$(shell brew --prefix libomp)/lib" -O3 simd.c -o simd -lm` としてコンパイルした。 まず、コンパイル時に出た以下のメッセージから、正常に SIMD 化が行われたことがわかる。

```
1  simd.c:143:1: remark: vectorized loop (vectorization width: 4, interleaved count:
   1) [-Rpass=loop-vectorize]
2    143 | #pragma omp simd
3        | ^
4  simd.c:318:1: remark: vectorized loop (vectorization width: 2, interleaved count:
   4) [-Rpass=loop-vectorize]
5    318 | #pragma omp simd reduction(+ : E_x_sweep, J_x_sweep, E_y_sweep,
     J_y_sweep)
6        | ^
```

得られた結果を Figure 6 に示す。 結果はオリジナルのソースコードと同様に KT 転移が確認でき、問題なくシミュレーションが行われていることがわかる。

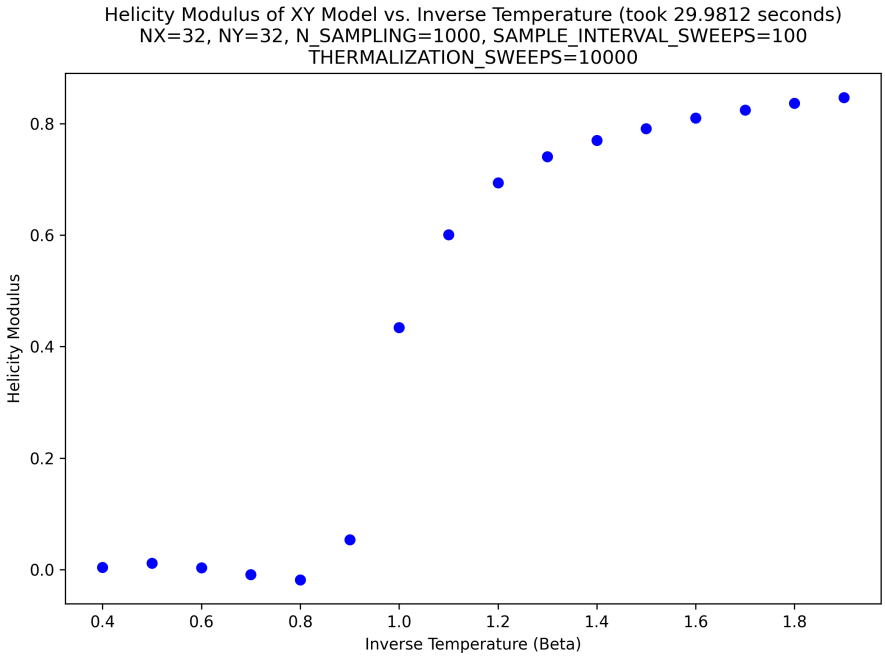

Figure 6: ヘリシティモジュラスのサンプリング平均 (SIMD 命令によるデータ並列化)
(コンパイル: `clang -Xpreprocessor -fopenmp -Rpass=loop-vectorize -march=native -ffast-math -lomp -I"$(shell brew --prefix libomp)/include" -L"$(shell brew --prefix libomp)/lib" -O3 simd.c -o simd -lm`)

実行時間は **30.0 秒** であった (最初と比較して 0.22 倍)。 SIMD 化を行ったことでむしろ実行時間が増加してしまい、並列化の効果よりオーバーヘッドの方が大きくなってしまった。これには、いくつかの要因が考えられるので、以下に挙げる。

- SIMD 命令による一括計算をする工程があまり多くなく (エネルギーの差分を計算するだけ) 、並列化の効果が薄いこと。
- SIMD 命令を用いるために連続したメモリ領域にスピンのデータを複製した (gather load、上のコードの 23 行目以降のブロック) が、このオーバーヘッドが大きいこと。
- メトロポリス法の性質上、最後に受容確率に応じてスピンを更新する・しないの `IF` 判定が必須である。実装では scatter store の際にこの判定を行うことで、メインの計算部分を一括化しているが、この判定が並列化の阻害要因となって、十分に並列化できなかった。

これらについては、より詳細に原因を調査し、改善する余地があると考えられる。 さらに高速化する提案を以下に挙げる。

Gather load のオーバーヘッドとスピンの保持の仕方

ひとつ飛ばし (市松模様) のスピンを連続したメモリ領域にコピーする (gather load) のオーバーヘッドが大きいことが原因の一つであると考えられる。そのため、そもそものスピン情報の格納方法を工夫することで、gather load のオーバーヘッドを減らすことができる。 つまり、ひとつ飛ばしのスピンが元々連続したメモリ領域に格納されれば良い。

さらに、周期境界条件を考慮するにあたって、現在は例えば x 方向の隣のスピンを取得する際に

```
spin[iy * nx + (ix + 1) % nx]
```

のように、% 演算子を用いて周期境界条件を適用しているが、これもオーバーヘッドとなる。 なぜなら、アドレス計算のたびに % 演算を行う負荷があるだけでなく、メモリが不連続になるからである。 そこで、境界より一回りだけ大きいサイズのメモリ領域を確保して、境界の値が反対の端と同じになるように管理する。

そのようにすることで、境界のスピンを取得する際に % 演算を行う必要がなり、境界付近も含めてスピンのデータを連続したメモリ領域に格納することができる (Figure 7)。

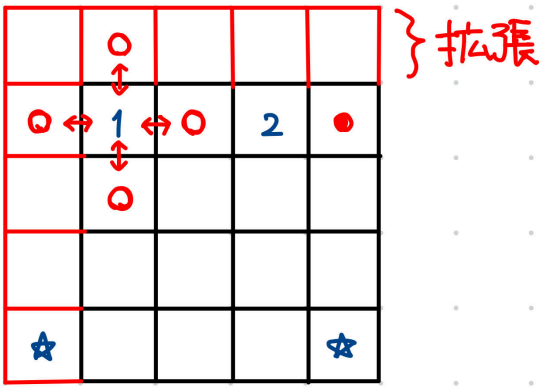

Figure 7: スピンの格納方法の工夫

境界部分のセルを一回り大きく拡張する。 この際周期境界条件に注意する (図においては、⋆ マーク同士のセルは等価なセルとして扱い、同じ値を持つように管理する)。このように拡張することで、境界のスピンの更新を行う際にも、境界から離れたスピンと同様に、`spin[iy * nx + (ix + 1)]` のように周期境界条件を考慮せずに取得できる (図においては セル 1 の更新をするときは白抜きされた赤丸を参照すればよく、 塗りつぶされた赤丸のセルを参照しなくて良くなる。 これによって % 演算が不要になるとともに、 セル 1,2, … を並列に計算するにあたって、対象のセルの隣のセル同士もメモリ領域において必ず連続するようになり、効率的な並列計算が可能になると考えられる)。

条件分岐の回避 (組み込み関数の利用)

メトロポリス法の受容確率の判定において、`IF` 文を用いてスピンを更新する・しないを決定しているが、この条件分岐が並列化の阻害要因となっている。 組み込み関数 mask を用いることで、この条件分岐を回避することができそうである。 調べたところによると、ハードウェア環境の依存なども大きいようであり、今回は断念した。

Appendix: ソースコード

Appendix 1. 並列化等の工夫をしていないオリジナルのコード

ファイル名: `original.c`

```c
// オリジナルの XY モデルのパラレルテンパリング（レプリカ交換）シミュレーション
// Gemini によって生成されたコードをベースとして，部分的に修正を加えた（目安としてコメントが英語の部分は修正した箇所）

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>

#define NX 32                       // 格子のXサイズ
```

```c
#define NY 32                       // 格子のYサイズ
#define N_REPLICAS 16               // レプリカの数
#define N_SAMPLING 1000             // サンプリング数
#define SAMPLE_INTERVAL_SWEEPS 100  // サンプリング間隔
#define THERMALIZATION_SWEEPS 10000 // 初期緩和のスイープ数

// 相互作用の強さ
#define J (1.0f)

/**
 * @brief レプリカの状態を保持する構造体
 */
typedef struct
{
  float *spin;  // スピンの角度θを格納する配列 (サイズ: N_SITES)
  float beta;   // このレプリカの逆温度 β = 1/T
  float energy; // このレプリカの現在のエネルギー
  int nx;       // 格子のX方向のサイズ
  int ny;       // 格子のY方向のサイズ
} Replica;

/**
 * @brief 系全体のエネルギーを計算する
 * @param rep 計算対象のレプリカへのポインタ
 * @return 計算された全エネルギー
 * @note 周期境界条件を適用します
 */
float calculate_total_energy(Replica *rep)
{
  float total_energy = 0.0f;
  const int N_SITES = rep->nx * rep->ny;

  for (int i = 0; i < N_SITES; ++i)
  {
    int ix = i % rep->nx;
    int iy = i / rep->nx;

    // 右の隣接スピンとの相互作用
    int right_neighbor_idx = iy * rep->nx + (ix + 1) % rep->nx;
    total_energy -= J * cosf(rep->spin[i] - rep->spin[right_neighbor_idx]);

    // 下の隣接スピンとの相互作用
    int down_neighbor_idx = ((iy + 1) % rep->ny) * rep->nx + ix;
    total_energy -= J * cosf(rep->spin[i] - rep->spin[down_neighbor_idx]);
  }
  return total_energy;
}

/**
 * @brief パラレルテンパレリング用のレプリカ群を初期化する
 * @param replicas レプリカの配列へのポインタ
 * @param n_replicas レプリカの数
 * @param nx 格子のX方向のサイズ
 * @param ny 格子のY方向のサイズ
 * @param betas 各レプリカに設定する逆温度の配列
 * @note 乱数シードは事前に設定しておく必要があります
 */
void initialize_replicas(Replica *replicas, int n_replicas, int nx, int ny, const float *betas)
{
  const int N_SITES = nx * ny;
  for (int i = 0; i < n_replicas; ++i)
  {
    replicas[i].spin = (float *)malloc(sizeof(float) * N_SITES);
```

```
74      if (replicas[i].spin == NULL)
75      {
76        fprintf(stderr, "Error: Failed to allocate memory for spins.\n");
77        exit(EXIT_FAILURE);
78      }
79
80      replicas[i].beta = betas[i];
81      replicas[i].nx = nx;
82      replicas[i].ny = ny;
83
84      // スピンをランダムな角度 [0, 2π) で初期化
85      for (int j = 0; j < N_SITES; ++j)
86      {
87        replicas[i].spin[j] = ((float)rand() / (float)RAND_MAX) * 2.0f * M_PI;
88      }
89
90      // 初期エネルギーを計算
91      replicas[i].energy = calculate_total_energy(&replicas[i]);
92    }
93  }
94
95  /**
96   * @brief 1つのレプリカに対して1モンテカルロスイープを実行する（メトロポリス法）
97   * @param rep 更新するレプリカへのポインタ
98   * @note 全てのスピンサイトを0からN_SITES-1まで順番に1回ずつ更新します。
99   */
100 void metropolis_sweep(Replica *rep)
101 {
102   const int N_SITES = rep->nx * rep->ny;
103   const float DELTA = 1.0f; // スピン角度の更新幅
104
105   // 全てのサイトを0からN_SITES-1まで順番にループ
106   for (int site_idx = 0; site_idx < N_SITES; ++site_idx)
107   {
108     int ix = site_idx % rep->nx;
109     int iy = site_idx / rep->nx;
110
111     float old_spin = rep->spin[site_idx];
112     // 乱数生成器はスピンの新しい角度を試すために使用
113     float new_spin = fmodf(old_spin + ((float)rand() / (float)RAND_MAX - 0.5f) * DELTA, 2.0f * M_PI);
114     if (new_spin < 0.0f)
115     {
116       new_spin += 2.0f * M_PI;
117     }
118
119     // エネルギー変化量を計算
120     float delta_e = 0.0f;
121     // 4つの隣接サイトをループ
122     int neighbors[4];
123     neighbors[0] = iy * rep->nx + (ix + 1) % rep->nx;             // Right
124     neighbors[1] = iy * rep->nx + (ix - 1 + rep->nx) % rep->nx;   // Left
125     neighbors[2] = ((iy + 1) % rep->ny) * rep->nx + ix;           // Down
126     neighbors[3] = ((iy - 1 + rep->ny) % rep->ny) * rep->nx + ix; // Up
127
128     for (int j = 0; j < 4; ++j)
129     {
130       int neighbor_idx = neighbors[j];
131       // new_spinと隣接スピンとの相互作用エネルギーと、
132       // old_spinと隣接スピンとの相互作用エネルギーの差を計算
133       delta_e += -J * (cosf(new_spin - rep->spin[neighbor_idx]) - cosf(old_spin - rep->spin[neighbor_idx]));
134     }
135
136     // メトロポリス判定
```

```
    if (delta_e < 0.0f || ((float)rand() / (float)RAND_MAX) < expf(-rep->beta * delta_e))
    {
      rep->spin[site_idx] = new_spin;
      rep->energy += delta_e;
    }
  }
}

/**
 * @brief レプリカ交換を実行する
 * @param replicas レプリカの配列
 * @param n_replicas レプリカの数
 */
void replica_exchange(Replica *replicas, int n_replicas)
{
  // 隣接するレプリカのペア (i, i+1) をランダムに選択
  int i = (rand() % (n_replicas - 1));

  float delta_beta = replicas[i].beta - replicas[i + 1].beta;
  float delta_energy = replicas[i].energy - replicas[i + 1].energy;

  // 交換確率を計算: min(1, exp(Δβ * ΔE))
  float acceptance_prob = expf(delta_beta * delta_energy);

  if (rand() < (float)RAND_MAX * acceptance_prob)
  {
    // スピン配列とエネルギーを交換
    float *temp_spin = replicas[i].spin;
    replicas[i].spin = replicas[i + 1].spin;
    replicas[i + 1].spin = temp_spin;

    float temp_energy = replicas[i].energy;
    replicas[i].energy = replicas[i + 1].energy;
    replicas[i + 1].energy = temp_energy;
  }
}

/**
 * @brief 確保したメモリを解放する
 * @param replicas レプリカの配列
 * @param n_replicas レプリカの数
 */
void free_replicas(Replica *replicas, int n_replicas)
{
  for (int i = 0; i < n_replicas; ++i)
  {
    if (replicas[i].spin != NULL)
    {
      free(replicas[i].spin);
      replicas[i].spin = NULL;
    }
  }
}

int main(int argc, char *argv[])
{
  // 1. パラメータ設定
  const int N_SITES = NX * NY;
  Replica replicas[N_REPLICAS];
  float betas[N_REPLICAS];
  int i, sweep;

  float min_beta = 0.4f;
```

```
200   float max_beta = 1.9f;
201
202   struct timeval start_time, end_time;
203   double wall_time;
204
205   FILE *fp; // file pointer for output
206
207   // start time measurement
208   gettimeofday(&start_time, NULL);
209
210   if (argc != 2)
211   {
212     fprintf(stderr, "Usage: %s <output_file>\n", argv[0]);
213     return EXIT_FAILURE;
214   }
215
216   // 出力ファイルを開く
217   fp = fopen(argv[1], "w");
218   if (fp == NULL)
219   {
220     fprintf(stderr, "Error: Could not open output file %s\n", argv[1]);
221     return EXIT_FAILURE;
222   }
223
224   fprintf(fp, "<version\noriginal\nversion>\n");
225
226   fprintf(fp, "<input_parameters\n");
227   fprintf(fp, "N_SITES=%d N_REPLICAS=%d NX=%d NY=%d N_SAMPLING=%d SAMPLE_INTERVAL_SWEEPS=%d
      THERMALIZATION_SWEEPS=%d\n",
228           N_SITES, N_REPLICAS, NX, NY, N_SAMPLING, SAMPLE_INTERVAL_SWEEPS, THERMALIZATION_SWEEPS);
229   fprintf(fp, "min_beta=%.6f max_beta=%.6f\n", min_beta, max_beta);
230   fprintf(fp, "input_parameters>\n");
231
232   for (i = 0; i < N_REPLICAS; ++i)
233   {
234     betas[i] = min_beta + (max_beta - min_beta) * i / (N_REPLICAS - 1);
235   }
236
237   // 2. 初期化
238   srand(48);
239   initialize_replicas(replicas, N_REPLICAS, NX, NY, betas);
240
241   // 全レプリカの物理量を格納する配列
242   double total_Ex[N_REPLICAS] = {0.0};
243   double total_Jx_squared[N_REPLICAS] = {0.0};
244   double total_Ey[N_REPLICAS] = {0.0};
245   double total_Jy_squared[N_REPLICAS] = {0.0};
246   long measurement_count = 0;
247
248   // 3. シミュレーションループ
249   printf("<simulation_progress\n");
250   sweep = 0;
251   while (measurement_count < N_SAMPLING)
252   {
253     // 各レプリカでメトロポリス更新
254     for (i = 0; i < N_REPLICAS; ++i)
255     {
256       metropolis_sweep(&replicas[i]);
257     }
258
259     // レプリカ交換
260     for (i = 0; i < N_REPLICAS; ++i)
261     { // 交換頻度を上げるためにループ
```

```
262       replica_exchange(replicas, N_REPLICAS);
263     }
264
265     // 物理量の測定(初期緩和後)
266     if (sweep > THERMALIZATION_SWEEPS && (sweep - THERMALIZATION_SWEEPS) % SAMPLE_INTERVAL_SWEEPS == 0)
267     {
268       for (i = 0; i < N_REPLICAS; ++i)
269       {
270         float E_x_sweep = 0.0f;
271         float J_x_sweep = 0.0f;
272         float E_y_sweep = 0.0f;
273         float J_y_sweep = 0.0f;
274
275         for (int j = 0; j < N_SITES; ++j)
276         {
277           int ix = j % NX;
278           int iy = j / NX;
279
280           // x方向 (右の隣人)
281           int right_neighbor_idx = iy * NX + (ix + 1) % NX;
282           float delta_theta_x = replicas[i].spin[j] - replicas[i].spin[right_neighbor_idx];
283           E_x_sweep += J * cosf(delta_theta_x);
284           J_x_sweep += J * sinf(delta_theta_x);
285
286           // y方向 (下の隣人)
287           int down_neighbor_idx = ((iy + 1) % NY) * NX + ix;
288           float delta_theta_y = replicas[i].spin[j] - replicas[i].spin[down_neighbor_idx];
289           E_y_sweep += J * cosf(delta_theta_y);
290           J_y_sweep += J * sinf(delta_theta_y);
291         }
292         total_Ex[i] += E_x_sweep;
293         total_Jx_squared[i] += (double)J_x_sweep * J_x_sweep;
294         total_Ey[i] += E_y_sweep;
295         total_Jy_squared[i] += (double)J_y_sweep * J_y_sweep;
296       }
297       measurement_count++;
298       if (measurement_count % (N_SAMPLING / 10) == 0)
299       {
300         printf("sweep=%d measurement_count=%ld total_Ex=%f total_Jx_squared=%f total_Ey=%f
          total_Jy_squared=%f\n",
301                sweep, measurement_count, total_Ex[0], total_Jx_squared[0], total_Ey[0], total_Jy_squared[0]);
302       }
303     }
304
305     sweep++;
306   }
307   printf("simulation_progress>\n");
308
309   // 最終結果の計算と表示
310   fprintf(fp, "<helicity_modulus_results\n");
311   for (i = 0; i < N_REPLICAS; ++i)
312   {
313     if (measurement_count > 0)
314     {
315       double avg_Ex = total_Ex[i] / measurement_count;
316       double avg_Jx_squared = total_Jx_squared[i] / measurement_count;
317       double upsilon_x = (avg_Ex / N_SITES) - (replicas[i].beta / N_SITES) * avg_Jx_squared;
318
319       double avg_Ey = total_Ey[i] / measurement_count;
320       double avg_Jy_squared = total_Jy_squared[i] / measurement_count;
321       double upsilon_y = (avg_Ey / N_SITES) - (replicas[i].beta / N_SITES) * avg_Jy_squared;
322
323       double helicity_modulus = (upsilon_x + upsilon_y) / 2.0;
```

```
      double energy_per_site = replicas[i].energy / (double)N_SITES;
      fprintf(fp, "replica=%d beta=%.6f energy_per_site=%.6f helicity_modulus=%.6f\n",
              i, replicas[i].beta, energy_per_site, helicity_modulus);
    }
  }
  fprintf(fp, "helicity_modulus_results>\n");

  // 4. メモリ解放
  free_replicas(replicas, N_REPLICAS);

  // end time measurement
  gettimeofday(&end_time, NULL);
  wall_time = (end_time.tv_sec - start_time.tv_sec) +
              (end_time.tv_usec - start_time.tv_usec) / 1000000.0;
  fprintf(fp, "<execution_time\n%.4f\nexecution_time>\n", wall_time);

  fclose(fp);

  return 0;
}
```

Appendix 2. SIMD 命令によるデータ並列化を施したコード

ファイル名: `simd.c`

```c
// CPUでの性能向上 --- マルチコア向上 (OpenMP) + SIMD 化

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <omp.h>
#include "pcg_variants.h"

#define NX 32                       // 格子のXサイズ
#define NY 32                       // 格子のYサイズ
#define NL 16                       // SIMD化のブロックサイズ
#define N_REPLICAS 16               // レプリカの数
#define N_SAMPLING 1000             // サンプリング数
#define SAMPLE_INTERVAL_SWEEPS 100  // サンプリング間隔
#define THERMALIZATION_SWEEPS 10000 // 初期緩和のスイープ数

// 相互作用の強さ
#define J (1.0f)

/**
 * @brief レプリカの状態を保持する構造体
 */
typedef struct
{
  float *spin;  // スピンの角度θを格納する配列 (サイズ: N_SITES)
  float beta;   // このレプリカの逆温度 β = 1/T
  float energy; // このレプリカの現在のエネルギー
  int nx;       // 格子のX方向のサイズ
  int ny;       // 格子のY方向のサイズ
} Replica;

/**
 * @brief 系全体のエネルギーを計算する
 * @param rep 計算対象のレプリカへのポインタ
 * @return 計算された全エネルギー
 * @note 周期境界条件を適用します
```

```
 */
float calculate_total_energy(Replica *rep)
{
  float total_energy = 0.0f;
  const int N_SITES = rep->nx * rep->ny;

  for (int i = 0; i < N_SITES; ++i)
  {
    int ix = i % rep->nx;
    int iy = i / rep->nx;

    // 右の隣接スピンとの相互作用
    int right_neighbor_idx = iy * rep->nx + (ix + 1) % rep->nx;
    total_energy -= J * cosf(rep->spin[i] - rep->spin[right_neighbor_idx]);

    // 下の隣接スピンとの相互作用
    int down_neighbor_idx = ((iy + 1) % rep->ny) * rep->nx + ix;
    total_energy -= J * cosf(rep->spin[i] - rep->spin[down_neighbor_idx]);
  }
  return total_energy;
}

/**
 * @brief パラレルテンパレリング用のレプリカ群を初期化する
 * @param replicas レプリカの配列へのポインタ
 * @param n_replicas レプリカの数
 * @param nx 格子のX方向のサイズ
 * @param ny 格子のY方向のサイズ
 * @param betas 各レプリカに設定する逆温度の配列
 * @note 乱数シードは事前に設定しておく必要があります
 */
void initialize_replicas(Replica *replicas, int n_replicas, int nx, int ny, const float *betas)
{
  const int N_SITES = nx * ny;
  for (int i = 0; i < n_replicas; ++i)
  {
    replicas[i].spin = (float *)malloc(sizeof(float) * N_SITES);
    if (replicas[i].spin == NULL)
    {
      fprintf(stderr, "Error: Failed to allocate memory for spins.\n");
      exit(EXIT_FAILURE);
    }

    replicas[i].beta = betas[i];
    replicas[i].nx = nx;
    replicas[i].ny = ny;

    // スピンをランダムな角度 [0, 2π) で初期化
    for (int j = 0; j < N_SITES; ++j)
    {
      replicas[i].spin[j] = ((float)rand() / (float)RAND_MAX) * 2.0f * M_PI;
    }

    // 初期エネルギーを計算
    replicas[i].energy = calculate_total_energy(&replicas[i]);
  }
}

/**
 * @brief 1つのレプリカに対して1モンテカルロスイープを実行する（メトロポリス法）
 * @param rep 更新するレプリカへのポインタ
 * @note 全てのスピンサイトを更新します
 */
```

```
102 void metropolis_sweep(Replica *rep, pcg32_random_t *rng)
103 {
104   const int N_SITES = rep->nx * rep->ny;
105   const float DELTA = 1.0f; // スピン角度の更新幅
106 
107   int odd, site_idx, idx, ix, iy, r; // loop variables
108 
109   // block variables for SIMD
110   float spin[NL], new_spin[NL];                                        // spin block
111   float neighborsR[NL], neighborsL[NL], neighborsD[NL], neighborsU[NL]; // neighbors blocks
112   float delta_e[NL];                                                   // energy change and acceptance
      probability
113   float rand_val_acc[NL], rand_val_delta[NL];                          // random values for acceptance and
      delta
114   int accept[NL];                                                      // acceptance flags
115 
116   // update each site in blocks of NL
117   for (odd = 0; odd < 2; odd++)
118   {
119     // odd = 0 : even sites
120     // odd = 1 : odd sites
121     for (site_idx = odd; site_idx < N_SITES; site_idx += NL * 2)
122     {
123       // gather load
124       for (r = 0; r < NL; ++r)
125       {
126         idx = site_idx + r * 2;
127         ix = idx % rep->nx;
128         iy = idx / rep->nx;
129 
130         // load spin
131         spin[r] = rep->spin[idx];
132 
133         // load neighbors
134         neighborsR[r] = rep->spin[iy * rep->nx + (ix + 1) % rep->nx];             // Right
135         neighborsL[r] = rep->spin[iy * rep->nx + (ix - 1 + rep->nx) % rep->nx];   // Left
136         neighborsD[r] = rep->spin[((iy + 1) % rep->ny) * rep->nx + ix];           // Down
137         neighborsU[r] = rep->spin[((iy - 1 + rep->ny) % rep->ny) * rep->nx + ix]; // Up
138 
139         rand_val_delta[r] = (float)pcg32_random_r(rng) / (float)UINT32_MAX;
140         rand_val_acc[r] = (float)pcg32_random_r(rng) / (float)UINT32_MAX;
141       }
142 
143       // scatter store
144 #pragma omp simd
145       for (int r = 0; r < NL; ++r)
146       {
147         new_spin[r] = fmodf(spin[r] + (rand_val_delta[r] - 0.5f) * DELTA + 2.0f * M_PI, 2.0f * M_PI);
148 
149         // calculate energy change
150         delta_e[r] = 0.0f;
151         // calculate energy change with neighbors
152         delta_e[r] += -J * (cosf(new_spin[r] - neighborsR[r]) - cosf(spin[r] - neighborsR[r]));
153         delta_e[r] += -J * (cosf(new_spin[r] - neighborsL[r]) - cosf(spin[r] - neighborsL[r]));
154         delta_e[r] += -J * (cosf(new_spin[r] - neighborsD[r]) - cosf(spin[r] - neighborsD[r]));
155         delta_e[r] += -J * (cosf(new_spin[r] - neighborsU[r]) - cosf(spin[r] - neighborsU[r]));
156 
157         accept[r] = (delta_e[r] < 0.0f || rand_val_acc[r] < expf(-rep->beta * delta_e[r]));
158       }
159 
160       // store results back to replica
161       for (int r = 0; r < NL; ++r)
162       {
```

```
163         int idx = site_idx + r * 2;
164         if (accept[r])
165         {
166           // accept the new spin
167           rep->spin[idx] = new_spin[r];
168           rep->energy += delta_e[r];
169         }
170       }
171     }
172   }
173 }
174
175 /**
176  * @brief レプリカ交換を実行する
177  * @param replicas レプリカの配列
178  * @param n_replicas レプリカの数
179  */
180 void replica_exchange(Replica *replicas, int n_replicas)
181 {
182   // 隣接するレプリカのペア (i, i+1) をランダムに選択
183   int i = (rand() % (n_replicas - 1));
184
185   float delta_beta = replicas[i].beta - replicas[i + 1].beta;
186   float delta_energy = replicas[i].energy - replicas[i + 1].energy;
187
188   // 交換確率を計算: min(1, exp(Δβ * ΔE))
189   float acceptance_prob = expf(delta_beta * delta_energy);
190
191   if (rand() < (float)RAND_MAX * acceptance_prob)
192   {
193     // スピン配列とエネルギーを交換
194     float *temp_spin = replicas[i].spin;
195     replicas[i].spin = replicas[i + 1].spin;
196     replicas[i + 1].spin = temp_spin;
197
198     float temp_energy = replicas[i].energy;
199     replicas[i].energy = replicas[i + 1].energy;
200     replicas[i + 1].energy = temp_energy;
201   }
202 }
203
204 /**
205  * @brief 確保したメモリを解放する
206  * @param replicas レプリカの配列
207  * @param n_replicas レプリカの数
208  */
209 void free_replicas(Replica *replicas, int n_replicas)
210 {
211   for (int i = 0; i < n_replicas; ++i)
212   {
213     if (replicas[i].spin != NULL)
214     {
215       free(replicas[i].spin);
216       replicas[i].spin = NULL;
217     }
218   }
219 }
220
221 int main(int argc, char *argv[])
222 {
223   // 1. パラメータ設定
224   const int N_SITES = NX * NY;
225   Replica replicas[N_REPLICAS];
```

```
226   float betas[N_REPLICAS];
227   int i, sweep;
228
229   float min_beta = 0.4f;
230   float max_beta = 1.9f;
231
232   struct timeval start_time, end_time;
233   double wall_time;
234
235   FILE *fp;
236
237   // start time measurement
238   gettimeofday(&start_time, NULL);
239
240   if (N_SITES % (NL * 2) != 0)
241   {
242     fprintf(stderr, "Error: N_SITES must be a multiple of %d.\n", NL * 2);
243     return EXIT_FAILURE;
244   }
245
246   if (argc != 2)
247   {
248     fprintf(stderr, "Usage: %s <output_file>\n", argv[0]);
249     return EXIT_FAILURE;
250   }
251
252   // 出力ファイルを開く
253   fp = fopen(argv[1], "w");
254   if (fp == NULL)
255   {
256     fprintf(stderr, "Error: Could not open output file %s\n", argv[1]);
257     return EXIT_FAILURE;
258   }
259
260   fprintf(fp, "<version\nsimd\nversion>\n");
261
262   fprintf(fp, "<input_parameters\n");
263   fprintf(fp, "N_SITES=%d N_REPLICAS=%d NX=%d NY=%d N_SAMPLING=%d SAMPLE_INTERVAL_SWEEPS=%d
      THERMALIZATION_SWEEPS=%d\n",
264           N_SITES, N_REPLICAS, NX, NY, N_SAMPLING, SAMPLE_INTERVAL_SWEEPS, THERMALIZATION_SWEEPS);
265   fprintf(fp, "min_beta=%.6f max_beta=%.6f\n", min_beta, max_beta);
266   fprintf(fp, "input_parameters>\n");
267
268   for (i = 0; i < N_REPLICAS; ++i)
269   {
270     betas[i] = min_beta + (max_beta - min_beta) * i / (N_REPLICAS - 1);
271   }
272
273   // 2. 初期化
274   srand(48);
275   int max_threads = omp_get_max_threads();
276   pcg32_random_t rngs[max_threads];
277   for (i = 0; i < max_threads; ++i)
278   {
279     pcg32_srandom_r(&rngs[i], time(NULL), (intptr_t)&rngs[i]);
280   }
281   initialize_replicas(replicas, N_REPLICAS, NX, NY, betas);
282
283   // 全レプリカの物理量を格納する配列
284   double total_Ex[N_REPLICAS] = {0.0};
285   double total_Jx_squared[N_REPLICAS] = {0.0};
286   double total_Ey[N_REPLICAS] = {0.0};
287   double total_Jy_squared[N_REPLICAS] = {0.0};
```

```
288   long measurement_count = 0;
289
290   // 3. シミュレーションループ
291   printf("<simulation_progress\n");
292   sweep = 0;
293   while (measurement_count < N_SAMPLING)
294   {
295     // 各レプリカでメトロポリス更新
296 #pragma omp parallel for
297     for (i = 0; i < N_REPLICAS; ++i)
298     {
299       metropolis_sweep(&replicas[i], &rngs[omp_get_thread_num()]);
300     }
301
302     // レプリカ交換
303     for (i = 0; i < N_REPLICAS; ++i)
304     { // 交換頻度を上げるためにループ
305       replica_exchange(replicas, N_REPLICAS);
306     }
307
308     // 物理量の測定(初期緩和後)
309     if (sweep > THERMALIZATION_SWEEPS && (sweep - THERMALIZATION_SWEEPS) % SAMPLE_INTERVAL_SWEEPS == 0)
310     {
311 #pragma omp parallel for
312       for (i = 0; i < N_REPLICAS; ++i)
313       {
314         float E_x_sweep = 0.0f;
315         float J_x_sweep = 0.0f;
316         float E_y_sweep = 0.0f;
317         float J_y_sweep = 0.0f;
318
319 #pragma omp simd reduction(+ : E_x_sweep, J_x_sweep, E_y_sweep, J_y_sweep)
320         for (int j = 0; j < N_SITES; ++j)
321         {
322           int ix = j % NX;
323           int iy = j / NX;
324
325           // x方向 (右の隣人)
326           int right_neighbor_idx = iy * NX + (ix + 1) % NX;
327           float delta_theta_x = replicas[i].spin[j] - replicas[i].spin[right_neighbor_idx];
328           E_x_sweep += J * cosf(delta_theta_x);
329           J_x_sweep += J * sinf(delta_theta_x);
330
331           // y方向 (下の隣人)
332           int down_neighbor_idx = ((iy + 1) % NY) * NX + ix;
333           float delta_theta_y = replicas[i].spin[j] - replicas[i].spin[down_neighbor_idx];
334           E_y_sweep += J * cosf(delta_theta_y);
335           J_y_sweep += J * sinf(delta_theta_y);
336         }
337         total_Ex[i] += E_x_sweep;
338         total_Jx_squared[i] += (double)J_x_sweep * J_x_sweep;
339         total_Ey[i] += E_y_sweep;
340         total_Jy_squared[i] += (double)J_y_sweep * J_y_sweep;
341       }
342       measurement_count++;
343       if (measurement_count % (N_SAMPLING / 10) == 0)
344       {
345         printf("sweep=%d measurement_count=%ld total_Ex=%f total_Jx_squared=%f total_Ey=%f
          total_Jy_squared=%f\n",
346                sweep, measurement_count, total_Ex[0], total_Jx_squared[0], total_Ey[0], total_Jy_squared[0]);
347       }
348     }
349
```

```
350       sweep++;
351     }
352     printf("simulation_progress>\n");
353 
354     // 最終結果の計算と表示
355     fprintf(fp, "<helicity_modulus_results\n");
356     for (i = 0; i < N_REPLICAS; ++i)
357     {
358       if (measurement_count > 0)
359       {
360         double avg_Ex = total_Ex[i] / measurement_count;
361         double avg_Jx_squared = total_Jx_squared[i] / measurement_count;
362         double upsilon_x = (avg_Ex / N_SITES) - (replicas[i].beta / N_SITES) * avg_Jx_squared;
363 
364         double avg_Ey = total_Ey[i] / measurement_count;
365         double avg_Jy_squared = total_Jy_squared[i] / measurement_count;
366         double upsilon_y = (avg_Ey / N_SITES) - (replicas[i].beta / N_SITES) * avg_Jy_squared;
367 
368         double helicity_modulus = (upsilon_x + upsilon_y) / 2.0;
369         double energy_per_site = replicas[i].energy / (double)N_SITES;
370         fprintf(fp, "replica=%d beta=%.6f energy_per_site=%.6f helicity_modulus=%.6f\n",
371                 i, replicas[i].beta, energy_per_site, helicity_modulus);
372       }
373     }
374     fprintf(fp, "helicity_modulus_results>\n");
375 
376     // 4. メモリ解放
377     free_replicas(replicas, N_REPLICAS);
378 
379     // end time measurement
380     gettimeofday(&end_time, NULL);
381     wall_time = (end_time.tv_sec - start_time.tv_sec) +
382                 (end_time.tv_usec - start_time.tv_usec) / 1000000.0;
383     fprintf(fp, "<execution_time\n%.4f\nexecution_time>\n", wall_time);
384 
385     fclose(fp);
386 
387     return 0;
388 }
```