I-032

# 大規模並列計算アーキテクチャに基づく画像処理プログラムの高速化に関する考察

# Consideration on Accelerating Image Processing Programs Based on Large-scale Parallel Computational Architectures

河畑 則文 †
Norifumi Kawabata

## 1 まえがき

大学を始めとする高等教育機関においては，事務・教育・研究を含め，業務や作業の効率化が必要とされており，特に，ディジタルトランスフォーメーション (DX) 技術の導入は急務である．オフィスワークにおいては，事前に計画的にスケジューリングしてできるのであれば問題はないだろうが，何かしら提出〆切がある．タスク処理においては，勿論，正確性も重要ではあるが，速い方が望ましい．特に，画像を始めとするマルチメディア処理を扱う際には，通常の文書処理より大きなデータが必要であるため，処理速度や処理内容は重要である．画像処理を始めとするグラフィックスに関するプログラムを実行する際には，計算時間と処理能力がパラメータとして必要となる．特に，計算アーキテクチャを並列計算処理して実行することで高速化するとされるが，画像処理毎にどの程度の並列処理や実行環境であればよいかは明らかではない．

本研究では，まず，複数の画像処理プログラムを準備し，その後，プログラムの高速化を目的として，大規模並列計算のアルゴリズムとアーキテクチャを考慮して OpenMP [1] による画像処理プログラムの検証実験を行い，その結果を考察した．

## 2 関連研究

### 2.1 分散システム分野における並列計算

分散システム分野においては，OpenMP アプリケーションにおける高速化ノードへの移植 [2], コンパイラ支援型適応型実行時システムによる組み込みマルチコアにおける OpenMP のサポート [3]，OpenMP プログラムからタスクグラフ [4]，OpenMP における自動スケジューリングアルゴリズムの選択とチャンクパラメータの計算 [5]，メモリ効率の良いフロー蓄積手法と OpenMP 並列化 [6]，OpenMP における変数分類のための機械学習手法 [7]，OpenMP マルチスレッドアプリケーションにおける実行時隠蔽行動の人間工学的な診断 [8]，ロックレスワークスティールリングを用いた極限の細粒度タスク分割 [9]，パフォーマンス最適化のための自動化された OpenMP 変異テストフレームワーク [10]，AMALTHEA と OpenMP を活用した自動車システム向け細粒度適応型並列化 [11]，MPI/OpenMP ベースの並列ソルバーを用いたインプリント成形シミュレーション [12]，ディープラーニングを活用したハイブリッドメッセージパッシングインターフェース（MPI）とオープンマルチプロセッシング（OpenMP）並列プログラムにおけるソフトウェア欠陥予測のための自動モデル [13], OpenMP, OpenACC, SYCL を用いた疎行列ベクトル乗算のパフォーマンスポータビリティ [14]，Fortran コードの高速化：Coarray Fortran と CUDA Fortran, OpenMP を統合する手法 [15]，バックトラッキングを回避し、入力データを最適化：OpenMP を用いた CONUS 規模の流域境界抽出 [16]，に関する研究が行われている．

### 2.2 グラフィクス・画像分野における並列計算

グラフィクス・画像分野においては，ギガピクセル解像度組織画像における並列データアクセスと分散データアクセスの比較 [17]，OpenMP と MPI を用いたラスタ化計算ベースの並列ベクトルポリゴンオーバーレイ解析アルゴリズム [18]，OpenMP を用いた画像ブロック表現の並列実装 [19]，埋め込み GPU に基づくフォーメーションフライング SAR の分散型リアルタイム画像処理 [20]，x86/x64 プロセッサにおける 2 次元畳み込みの設計と実装 [21]，ハイパースペクトラル画像からの局所共分散行列を用いた空間・スペクトル特徴抽出のハイブリッド並列化 [22]，OpenMP と MPI を組み合わせた Sentinel-1 SAR 時系列画像のマルチセグメント並列コレジストレーション [23]，効率的な OpenMP ベースの Z 曲線符号化・復号化アルゴリズム [24]，断層画像診断のための分散型エッジ AI [25]，MPI, OpenMP/OpenACC, および非同期マルチ GPU プログラミングを用いた粒子インセルモンテカルロシミュレーションの高速化 [26]，GPU を活用した極限スケール乱流シミュレーション：OpenMP オフロードを用いたエクサスケールでのフーリエ擬スペクトルアルゴリズム [27]，に関する研究が行われている．

### 2.3 まとめ

全体として，並列計算や分散処理に関する高性能計算に関する研究は行われてきているが，画像処理プログラムの内容と並列計算の具体的な関係は明らかではないので，本研究では，この部分を考察していく．

† 金沢大学 学術メディア創成センター
Emerging Media Initiative, Kanazawa University

# 3　大規模並列計算アーキテクチャ

## 3.1　概要

本研究では，並列化 (Parallelization) に関する技術を用いて画像処理プログラムの高速化を試みる．並列化とは，同時に処理できるようにすることである．並列化はコンピュータに限らず一般社会における概念であり，あらゆることに適用できる．例えば，料理やオフィスワークにおける手順を考える際には並列化が可能である．しかしながら，「並列化できる処理」と「並列化できない処理」があり，これを明らかにする必要がある．本研究で扱う並列プログラミングにおいては，「並列化できる場所」と「並列化できない場所」の検討をすることが最初の手順となる．

並列化の利点としては，プログラムが高速に実行できるようになることである．プログラムは計算機上で実行され，通常，1 つの計算機しか使わないと思われているが，実際には，内部に 8 つ以上の計算をできる仕組みを持っている，すなわち計算機の内部に 8 台の PC があるのと同じである．それゆえに，8 台の計算機を同時に使える「並列化」をした方が高速化できる．例えば，並列化されていないプログラムによる実行が 8 時間かかる場合，並列化をしたプログラムを用いて実行すると，8 つの計算要素を持つ PC では，同じ処理をするのに 1 時間で終了できる．

次の利点は，大規模な問題を取り扱えるようになることである．並列化により複数の計算機を同時に使えるようになるが，使える計算機の数に比例して計算機内のメモリの量も大きくなっていく．例えば，1 つの計算機で 32GB のメモリを持っているとすると，並列化しないと 32GB 以下の問題しか実行できない．一方で，並列化して 10 台の計算機が使えるとすると，合計で 320GB の問題を実行できるようになる．使う計算機数に比例して，大規模な問題を扱えるようになる．

このように，複数の計算機を利用するような並列化であるノード間並列化，複数の計算機を使うのではなく，1 つの計算機内にある同時に計算できる要素を用いた並列化であるノード内並列化，これら 2 点が並列化することの利点である．

## 3.2　並列コンピュータと並列処理

並列コンピュータ (Parallel Computer) とは，複数のコンピュータあるいはその主要部が，データを交換しながら協調して動作する構成方式である．現在，サーバ，デスクトップ，ノート PC などのコンピュータは勿論，スマートフォンや家庭用ゲーム機に内蔵されているコンピュータまで，全て並列コンピュータである．

並列処理の様式は，「命令及びデータの流れが単一か複数か」（Flynn の分類）によって次のように分類できる．

- SISD (Single Instruction stream, Single Data stream)：1 つのデータに対して一度に 1 つの命令を実行（並列処理でない通常のコンピュータ）
- SIMD (Single Instruction stream, Multiple Data stream)：複数のデータに対して同時に同じ命令を実行できる
- MISD (Multiple Instruction stream, Single Data stream)：1 つのデータに対して同時に複数の異なる命令を実行できる
- MIMD (Multiple Instruction stream, Multiple Data stream)：複数のデータに対して同時に複数の異なる命令を実行できる

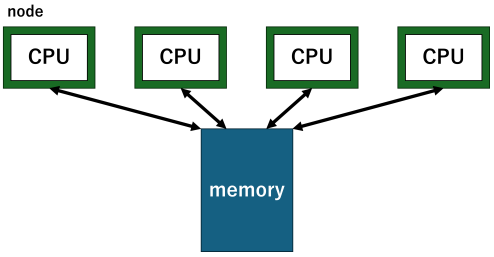


図 1: 共有メモリ型

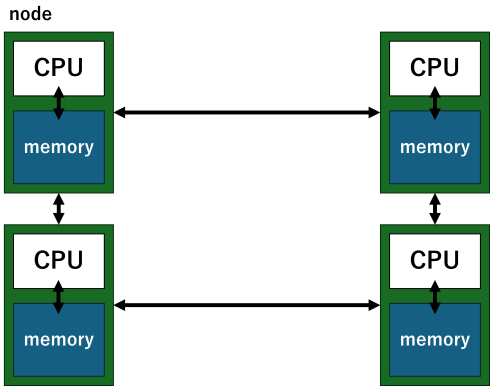


図 2: 分散メモリ型

並列処理における処理単位（CPU など）のことをノード (node) または PE (Processor Element) という．並列処理の方式は，ノードとメモリとの接続方式によって共有メモリ型 (shared memory) と分散メモリ型 (distributed memory) に二分することもある．共有メモリ型はすべてのノードが単一のメモリ領域を共有する方式である（図 1 参照）．共有メモリ型ではノード間でのデータの受け渡しが容易な反面，メモリアクセスが競合して性能が低下しやすい．一方，分散メモリ型は，個々のノードが独立したメモリ領域を持つ方式である（図 2 参照）．分散メモリ型では，メモリアクセス速度を上げられる反面，ノード間でのデータの受け渡しがネックとなる．

## 3.3　並列プログラミング

並列プログラミングを行うには C 言語や Fortran 言語の取得に加えて，Message Passing Interface (MPI) や OpenMP を利用して，並列プログラムを開発していく [28].

並列プログラミングを行うときに最も基本的な概念が並列化 (Parallelization) である．ここで，並列化とは，他の処理を気にすることなく，各処理が独立して処理するようになることである．並列化した処理をプログラミングするのが並列プログラミング (Parallel Computing) である．

並列プログラミングは普段行っているプログラミング，いわゆる逐次プログラミング (Sequential Programming) とは異なる考え方をする必要がある．並列プログラミングでは，並列処理の要素毎に，他の要素と独立して実行できなくてはいけない．そのため，並列処理を阻害する要因，いわゆる依存 (dependency) を知る必要がある．プログラムでは一般に次のような 2 つの構造がある．

- データ構造
- 制御構造 (if 文や for 文といったプログラム構造)

双方とも並列処理を阻害する要因になるが，多くの事例で現れる，データ構造の依存，いわゆるデータ依存について説明していく．

### 3.4 データ構造（流れ依存）

データ依存の中で最も基本的な依存は流れ依存 (flow dependency) である．流れ依存の例を以下に示す．

```
b = a;
c = b;
d = c;
```

`c = b;` の実行には，`b = a;` の実行を終了し，変数 `a` の値を読み取り，変数 `b` に値を収納しないと，次の 2 行目の代入は変数 `b` が確定していないので実行できない．同様なことは，3 行目の変数 `c` にもいえる．変数 `b` と変数 `c` は流れ依存がある，といえる．

### 3.5 データ構造（逆依存）

一方で，流れ依存の逆の依存として，逆依存 (anti-flow dependency) がある．逆依存の例を以下に示す．

```
a = b;
b = c;
c = d;
```

2 行目の `b = c;` の実行には，1 行目の `a = b;` の実行を終了し，変数 `b` の値を読み取り，変数 `a` に値を収納しないと，2 行目の代入は，変数 `b` の値が壊れてしまうため実行できない．変数 `b` と変数 `c` は逆依存がある，といえる．

### 3.6 データ構造（出力依存）

最後のデータ依存として，出力依存 (output dependency) がある．出力依存の例を次に示す．

```
a = b;
c = d;
printf(‘‘c = %d \n’’, c);
c = e;
```

ここで，4 行目の `c = e;` の実行には，2 行目の `c = d;` の実行を終了し，変数 `c` の値を，変数 `d` に収納しないと，4 行目の代入は，変数 `c` の値の変化について，逐次と異なる順番となり表示の結果がおかしくなる．一方，この出力依存は，回避方法があるため偽の依存とも呼ばれる．すなわち，4 行目の代入について，別の変数 `f` を導入し，

```
f = e;
```

とすれば，出力依存がなくなる．次節の制御構造の依存についても，これらのデータ依存の概念を拡張することで定義が可能である．

### 3.7 制御構造（ループ並列性）

プログラムを構成するループにおいて，ここでは，ループ中に現れる式において，データ依存が全くない場合を考える．以下のループを考える．

```
for (i=0; i<N; i++) {
    r = f(i);
    if (r == 1) printf(‘‘find i=%d \n’’, i);
}
```

以上のループは，ある値を与えて，それが解かどうかを探索する処理の骨格を表している．具体的には，関数 `f(i)` に整数 `i`（その範囲は `0` から `N-1`）を与え，何かの解を探索する処理である．関数 `f(i)` の戻り値が 1 かどうかで，解かどうかを判定する．以上の例では，整数 `i` を与えた後の関数 `f(i)` の呼び出し処理について，データ依存はない．よって，上記の処理は，並列化可能である．

### 3.8 制御構造（ジョブレベルの並列性）

ここでは，ジョブレベルでの並列化を説明する．ジョブレベルというのは，プログラムの実行レベルという意味である．同時に並列計算機でプログラムを動かすことで解きたい問題を高速に求解することを目的とする．自明に並列できる処理，いわゆる自明並列 (Embarrassingly Parallel: EP) は原理的に並列化の効率が極めて高いとされ，ここでは，自明並列のループのプログラムは原理的にジョブレベルの並列化ができるという点も重要である．前節のループの例文は，以下のようなプログラムに変更できる．

```
int main(int argc, char* argv[]) {
    int i, r;
    if (argc != 2) return(1);
    i = atoi(argv[1]);  r = f(i);
    if (r == 1) printf(‘‘find i=%d \n’’, i);
    else printf(‘‘It cannot find i=%d \n’’,
        i);
}
```

以上の例は，C 言語の実行時オプションを入手する main 関数の引数である `argc, argv[]` を利用しており，関数 `f(i)` に与えるパラメータである整数値を，ジョブの実行時オプションとして与える仕様に変更されている．この実

行可能ファイル名を job とするとき，整数パラメータ値 10 を指定して実行するには，以下のコマンドを入力して実行する．

```
$ ./job 10
```

またこの処理は，ジョブ間にデータ依存関係がないので，同時に実行できる．

```
$ ./job 10
$ ./job 20
$ ./job 30
```

以上のような実行を，自動で行うように Linux コマンドを実行させると，プログラム中にループで記載した処理と同等の処理ができる．この処理の自動化に関して，シェルスクリプトで処理を記載でき，Linux の bash スクリプトでは以下のように記載できる．

```
#!/bin/bash
for i in ‘seq 0 99’
do
    echo ‘‘i=’’$i
    ./job $i
done
```

ここで，最初の行の指定#!/bin/bash で，bash を指定している．以上のスクリプト記述により，job の実行時のコマンドラインオプションに，0，1，...，99 までを指定することで実行がされる．この実行に関して，並列計算機で行うことで，各ジョブが並列実行され，高速化が期待できる．

### 3.9 OpenMP による並列計算

OpenMP (open MultiProcessing) とは，並列計算機環境において共有メモリ・マルチスレッド型の並列アプリケーションソフトウェア開発をサポートするために標準化された API である．並列コンピューティングに利用される MPI (Message Passing Interface) では，メッセージの交換をプログラム中に明示的に記述しなければならないが，OpenMP ではディレクティブ（指令）を挿入することで並列化を行う（図 3 参照）．OpenMP ではループ部分に指示文を付け加えることにより並列化を実現する．

```
#pragma omp for [指示句[[,]指示句...]
for ループ
```

OpenMP が実行されるとマスタースレッド (master thread) という単一のスレッドのみで実行が開始され，プログラムの実行が並列指示文に到達すると，スレーブスレッド (slave thread) と呼ばれる複数のスレッドを生成し，並列指示文で指定されている範囲の処理が全スレッドにおいて実行される．

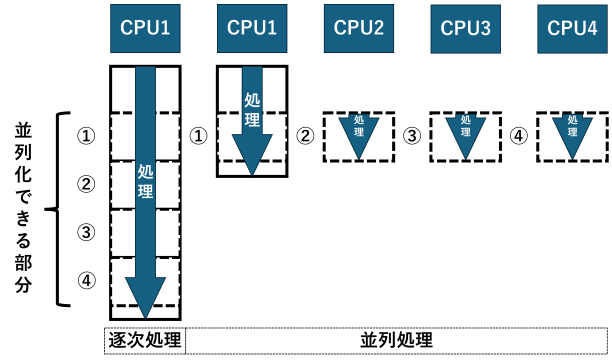


図 3: 逐次処理と並列処理

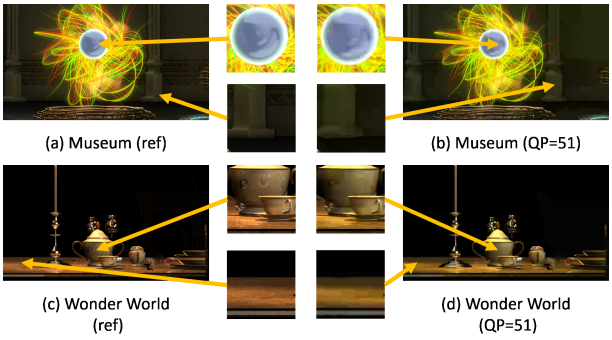


図 4: 本研究で使用した 3DCG 画像コンテンツ

## 4 実験

### 4.1 本研究で使用した画像コンテンツ

本研究で使用した画像コンテンツは，NICT [29] が無償で提供している図 4 のような 3DCG コンテンツ (Museum (M), Wonder World (W)) である．画像の生成に関しては，まず，8 視点分の CG カメラを構築し，カメラワーク，レンダリング [30] を行い，ハイビジョン画質の 8 視点分の静止画像を生成した．本コンテンツは，本来，多視点 3D 画像コンテンツであるが，本研究では，8 視点分の静止画像のうち，1 視点分の単視点画像を用いた．

### 4.2 実験手順

1. 3DCG コンテンツを作成する．
2. 実験のために画像編集（画像サイズの縮小,）を行う．
3. 画像処理プログラム (Visual C++ 2022, OpenCV 4.11) を作成する．
4. 作成した画像処理プログラムをビルドし，実行時間を計測する．
5. 計測したデータに基づきパターン間の実行時間について考察を行う．

### 4.3 実験方法と評価方法

実験方法として，OpenMP を用いた画像処理プログラムを作成し，実行時間を測定する．本研究では，ループ部分の並列化無し／並列化した場合における画像サイズ，画像処

理内容による実行時間に着目した．具体的には，OpenMP あり／無し（2 通り），ループ回数（1 回，10 回，100 回，1000 回の 4 通り），画像サイズ（1280 × 720, 1920 × 1080 の 2 通り），画像処理（回転，メディアンフィルタの 2 通り），3DCG 画像コンテンツ（“Museum,” “WonderWorld” の 2 通り）の計 64 通りとした．本研究で使用した計算機環境は，プロセッサが 13th Gen Intel(R) Core(TM) i7-1360P (2.20 GHz), 実装 RAM（メモリ）が 32GB, グラフィックスカード Intel(R) Iris(R) Xe Graphics 128MB のノート PC を用いて実験を行った．

評価方法として，並列処理した画像処理プログラムに対して実行時間の計測を行い，時間が改善したかどうか，パターン間での差分はどうかを評価する．

# 5　実験結果と考察

## 5.1　実験 1（回転）の結果と考察

実験 1 の結果（回転）を表 1 に示す．表中の数値は実行時間計測 (ms) を表す．実行時間計測には Visual C++のタイマ関数を使用した．画像サイズは画像解像度のことを示している．ループ回数は `for` 文のループ回数のことを示す．並列化は OpenMP ライブラリを使用した場合を「あり」，使用しない場合を「無し」とした．

実験 1 の結果から，3DCG コンテンツ “Museum”, “WonderWorld” の実行時間計測は並列化あり無し，画像サイズの大小，ループ回数のパターンに着目すると，ほとんどの場合において $M > W$ となった．これは，画像の複雑さに関係しているのではないかと考える．“Museum” は画像内容が “WonderWorld” よりも細かい，色彩が鮮やかであることを考慮すると，処理時間がかかったと推測される．OpenMP の並列化あり／無しの影響としては，実験 1 においてはコンテンツによって結果が分かれた．“Museum” においては，OpenMP による並列化が「あり」の方が「無い」場合よりほとんどのパターンで時間がかかった．このことから，“Museum” においては，画像の内容云々が並列化による処理時間の短縮に直接的に関係しなかったといえる．一方で，“WonderWorld” は OpenMP による並列化の効果が見られた．このことから，“WonderWorld” においては，画像の内容云々が並列化による処理時間の短縮に少なくとも直接的に関係していたといえる．本研究では，CPU を使って実験した．使用した画像はコンピュータグラフィックスで作成された画像であったため，CPU による並列化よりも GPU による並列化の方が効果がある．少なくとも本実験結果から，CPU による並列化処理では，3DCG 物体の面積が大きいほど，処理時間の短縮には直接影響しないということが把握できた．

## 5.2　実験 2（メディアンフィルタ）の結果と考察

実験 2 の結果を表 2 に示す．表の見方については，実験 1 と同様である．

表 1: 実験 1 の結果（回転）

| Exp.1 (Rotation) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 実行時間計測 (ms) | | | | | | | | | |
| 並列化 | 画像サイズ (pixel) | 1920x1080 | | | | 1280x720 | | | |
| | ループ回数（回） | 1回 | 10回 | 100回 | 1000回 | 1回 | 10回 | 100回 | 1000回 |
| 無し | Museum | 118 | 243 | 1701 | 16087 | 60 | 119 | 647 | 6077 |
| | WonderWorld | 86 | 217 | 1627 | 15726 | 48 | 103 | 649 | 6099 |
| あり | Museum | 136 | 246 | 1698 | 16871 | 64 | 116 | 697 | 6145 |
| | WonderWorld | 77 | 220 | 1614 | 16491 | 42 | 101 | 642 | 5995 |

表 2: 実験 2 の結果（メディアンフィルタ）

| Exp.2 (Median Filter) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 実行時間計測 (ms) | | | | | | | | | |
| 並列化 | 画像サイズ (pixel) | 1920x1080 | | | | 1280x720 | | | |
| | ループ回数（回） | 1回 | 10回 | 100回 | 1000回 | 1回 | 10回 | 100回 | 1000回 |
| 無し | Museum | 521 | 4555 | 46356 | 476790 | 241 | 1888 | 18698 | 201131 |
| | WonderWorld | 484 | 3399 | 34080 | 349827 | 190 | 1527 | 15254 | 158709 |
| あり | Museum | 616 | 4949 | 51769 | 464131 | 232 | 1954 | 19637 | 201354 |
| | WonderWorld | 369 | 3385 | 33680 | 353238 | 189 | 1561 | 15100 | 158161 |

実験 2 の結果から，メディアンフィルタは回転（実験 1）よりも多くの時間がかかっていることがわかる．これは，メディアンフィルタは `for` 文がプログラム中に 6 つ存在しており，一方，回転はプログラム中に 1 つのみであったため時間を要していたからであるが，逆に考えれば，for 文が多いほど処理時間がかかるため並列化をする必要があり，並列化の効果が得られやすいと考えられる．実験 2 の結果を見ると，特に，ループ回数が多くなるにつれ，並列化の効果が出ていない，逆に時間がかかってしまっていることがわかる．このことから，フィルタ処理においては CPU による並列化による処理時間短縮には限界があると考えられ，GPU による並列化を考える必要があると実験結果から推察できる．

# 6　まとめ

本研究では，大規模並列計算アーキテクチャに基づく画像処理プログラムの高速化について考察した．

本研究の成果として，OpenMP による並列化はコンテンツ内容，ループ回数の多さ，画像処理内容の複雑さによっては，処理時間が思うように短縮できなかったりすることがあることが把握できた．

今後の展望として，GPU による並列計算アーキテクチャの検討，より効果的な並列計算アルゴリズムの活用，量子計算への応用を考えていく．

## 参考文献


[1] OpenMP, https://www.openmp.org/, accessed June 13, 2025.

[2] S. Bak et al., “OpenMP application experiences: Porting to accelerated nodes,” *Parallel Comput.*, vol.109, 102856, 2022.

[3] S. N. Agathos et al., “Compiler-assisted, adaptive runtime system for the support of OpenMP in embedded multicores,” *Parallel Comput.*, vol.110, 102895, 2022.

[4] J. Sun et al., “ompTG: From OpenMP Programs to Task Graphs,” *J. Syst. Archit.*, vol.126, 102470, 2022.

[5] A. Mohammed et al., “Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP,” *IEEE Trans. Parallel Distrib. Syst.*, Vol.33, No.12, pp.4383–4394, December 2022.

[6] H. Cho et al., “Memory-efficient flow accumulation using a look-around approach and its OpenMP parallelization,” *Environ. Model. Softw.*, vol.167, 105771, 2023.

[7] Y. Shen et al., “A machine learning method to variable classification in OpenMP,” *Future Gener. Comput. Syst.*, vol.140, pp.67–78, 2023.

[8] W. Wang et al., “Anthropomorphic diagnosis of runtime hidden behavior in OpenMP multi-threaded applications,” *J. Parallel Distrib. Comput.*, vol.177, pp.17–27, 2023.

[9] P. Nookala et al., “X-OpenMP – eXtreme fine-grained tasking using lock-less work stealing,” *Future Gener. Comput. Syst.*, vol.159, pp.444–458, 2024.

[10] D. Miao et al., “An automated OpenMP mutation testing framework for performance optimization,” *Parallel Comput.*, vol.121, 103097, 2024.

[11] A. Munera et al., “Fine-grained adaptive parallelism for automotive systems through AMALTHEA and OpenMP,” *J. Syst. Archit.*, vol.146, 103034, 2024.

[12] Y. Li et al., “MPI/OpenMP-Based Parallel Solver for Imprint Forming Simulation,” *Comput. Model. Eng. Sci.*, vol.140, no.1, pp.461–483, 2024.

[13] A. Althiban et al., “Automated Models for Predicting Software Defects in Hybrid Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) Parallel Programs Using Deep Learning,” *IEEE Access*, Vol.13, pp.65939–65954, 2025.

[14] K. Stec at al., “Performance portability of sparse matrix-vector multiplication implemented using OpenMP, OpenACC and SYCL,” *Future Gener. Comput. Syst.*, vol.170, 107825, 2025.

[15] J. McKevitt et al., “Accelerating Fortran coded: A method for integrating Coarray Fortran with CUDA Fortran and OpenMP,” *J. Parallel Distrib. Comput.*, vol.195, 104977, 2025.

[16] H. Cho et al., “Avoid backtracking and burn your inputs: CONUS-scale watershed delineation using OpenMP,” *Environ. Model. Softw.*, vol.183, 106244, 2025.

[17] E. Yildirim et al., “Parallel Versus Distributed data Access for Gigapixel-resolution Histology Images: Challenged and Opportunities,” *IEEE J. Biomed. Health Inform.*, Vol.21, No.4, pp.1049–1055, July 2017.

[18] J. Fan et al., “Rasterization Computing-Based Parallel Vector Polygon Overlay Analysis Algorithms Using OpenMP and MPI,” *IEEE Access*, vol.6, pp.21427–21441, 2018.

[19] I. M. Spiliotis et al., “Parallel implementation of the Image Block Representation using OpenMP,” *J. Parallel Distrib. Comput.*, vol.137, pp.134–147, 2020.

[20] T. Yang et al., “Distributed Real-Time Image Processing of Formation Flying SAR Based on Embedded GPUs,” *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.*, vol.15, pp.6495–6505, 2022.

[21] V. Kelefouras et al., “Design and Implementation of 2D Convolution on x86/x64 Processors,” *IEEE Trans. Parallel Distrib. Syst.*, Vol.33, No.12, pp.3800–3815, December 2022.

[22] E. Torti et al., “Spatial-Spectral Feature Extraction With Local Covariance Matrix From Hyperspectral Images Through Hybrid Parallelization,” *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.*, Vol.16, pp.7412–7421, 2023.

[23] Y. Kang et al., “Multisegment Parallel Coregistration of Sentinel-1 SAR Time-Series Images by Combining OpenMP With MPI,” *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.*, vol.18, pp.1656–1669, 2025.

[24] Z. Zhou et al., “Efficient OpenMP Based Z-curve Encoding and Decoding Algorithms,” *Comput. Mater. Contin.*, vol.82, no.1, pp.1313–1327, 2025.

[25] B. Akdemir et al., “From Technical Prerequisites to Improved Care: Distributed Edge AI for Tomographic Imaging,” *IEEE Access*, vol.13, pp.14317–14343, 2025.

[26] J. J. Williams et al., “Accelerating Particle-in-Cell Monte Carlo simulations with MPI, OpenMP/OpenACC and Asynchronous Multi-GPU Programming,” *J. Comput. Sci.*, vol.88, 102590, 2025.

[27] P.K. Yeung et al., “GPU-enabled extreme-scale turbulence simulations: Fourier pseudo-spectral algorithms at the exascale using OpenMP offloading,” *Comput. Phys. Commun.*, vol.306, 109364, 2025.

[28] 片桐孝洋，“並列プログラミングのツボ”，東京大学出版，2024 年 4 月.

[29] NICT, http://www.nict.go.jp/, accessed June 13, 2025.

[30] Autodesk, http://www.autodesk.co.jp/, accessed June 13, 2025.