

# 余剰コアを活用した OpenMP Task による In-situ 解析の実現

赤沢 龍哉<sup>1</sup> 埴 敏博<sup>1,2</sup> 三木 洋平<sup>2</sup>

**概要:** 近年, CPU 性能向上にはコア数の増加のみが寄与しており, HPC システムではメニーコア CPU が広く用いられている. しかし CPU 内全てのコアを使うのではなく, 意図的にコアを余らせた方が高い実効性能を得られる場合も多い. そこで, 主計算の性能向上に寄与しない「余剰コア」を副次的な処理に活用するフレームワーク UTHelper の実現を目指している. 本研究では, 主計算を補うプリポスト処理を効率よく実現する手法として, OpenMP task 構文を用いる方法を提案する. 主計算と解析が逐次処理で行われていた宇宙物理の  $N$  体計算コード GOTHIC に対して本手法を適用し, 簡便な修正で主計算と解析をオーバーラップして実行することで, シミュレーション全体の実行時間が 54% 程まで短縮された.

## 1. はじめに

近年のスーパーコンピュータの性能向上の主な要因は CPU に搭載されるコア数の増加であり, 60 コアを超えるものも登場している. このコア数の増加傾向は今後もしばらく続くと推測されており, CPU のメニーコア化が進んでいる. しかし性能を最大限発揮させようと CPU 内の全コアを稼働させると, 電力やメモリバンド幅の制約, 並列性の限界などの理由から, かえって性能低下を招くため, 意図的に使用コア数を制限し, 一部を未使用のまま残す方が良い場合がある. 我々は, このような主計算の性能向上に寄与しないために使われない「余剰コア」に主計算をサポートする処理を行わせることで, ユーザに価値を提供するフレームワーク “UTHelper” の実現を目標としている.

一方, データ活用においては, 演算性能の向上に I/O 性能の向上が追いついていないため, 出力データの全てを保存することは不可能である. この問題を回避するために近年, 積極的に研究開発されている「In-situ 可視化・解析」という技術がある. これを UTHelper の機能として提供することを目標としている.

本研究では, フレームワークとしての In-situ 可視化・解析の発展に向けて, ユーザが利用しやすい枠組みを設計し, その動作を確かめる. 次に既存の科学アプリケーションに対して適用し, その性能を評価する.

## 2. 背景

### 2.1 余剰コアの発生

スーパーコンピュータのピークパフォーマンスは過去 70 年間で約 500 億倍近く向上している. この性能向上を支えている要因を説明する. 図 1 はマイクロプロセッサが誕生してからの性能トレンドを示している. 2005 年程まではムーアの法則に従い, 論理コア数は 1 のままでも 1 スレッドあたりの性能が向上してきた. しかしそれ以降はクロック周波数やスレッド性能の向上が落ちてきた一方で, 論理コア数が飛躍的に増加している. このことからプロセッサの性能向上の要因が, スレッド性能向上などから論理コア数を増やすことにシフトしていることが分かる. このことは CPU の物理コア数の増加に反映されている. 例えば 2012 年に本稼働していた, スーパーコンピュータ「京」に使われた CPU のコア数は 8 コアであるが, 近年は 60 コアを超えるものも登場している.

コア数の増加によってピークパフォーマンスが向上したが, 実用上ある問題が発生している. それは全てのコアを効率的に活用しきれないという問題である. コアを全て稼働させたとしても理想的な性能は得られず, かえって性能が低下してしまう場合がある. ではなぜこのような問題が発生するのか理由を説明する.

まず 1 つ目は, CPU の演算性能にメモリバンド性能の向上が追いついておらず, データ移動時間がボトルネックになるからである. CPU はキャッシュ上に欲しいデータが存在しない場合, 各ソケットを通してメモリへアクセス

<sup>1</sup> 東京大学 工学系研究科

<sup>2</sup> 東京大学 情報基盤センター

しデータを取得する必要がある。プログラムのキャッシュヒット率が悪ければ悪いほど、その回数は増えるため、メモリバンド幅のボトルネックの影響が大きくなる。さらにHPC クラスタで行うような大規模科学技術計算は、大量のデータを取り扱うため、メモリアクセス頻度がさらに高くなる。つまりメニーコア CPU を最大限活用しようと、使用コア数を増やしたとしても、メモリバンド幅のボトルネックによって実行速度の向上が見込めない可能性がある。この時の最適なコア数は、アプリケーションの実行演算性能に対してデータの供給が追従できる状態を保てる数となる。

2つ目は、電力消費やそれに伴う熱問題である。プロセッサの計算性能向上の要因の1つは、クロック周波数を上げることである。回路はこの周波数に合わせてトランジスタのスイッチ切り替えを行うため、これが高ければ高いほど、計算速度が向上する。しかし、このことは熱や電力に関する問題を孕む。なぜならクロック周波数を上げるためには、大きな電力が必要となり、発熱を伴うからである。メニーコア化によってプロセッサへの搭載コア数が増加すると、全体の消費電力はさらに大きくなり、熱による故障の可能性を無視できない。これを防ぐために、CPUにはスロットリングという、CPUの発熱による故障を防ぐ機能がある。これはCPUに高負荷な処理をさせた際に生じる発熱によってパッケージ温度が高くなりすぎた場合に、コアのクロック周波数を自動的に下げることで熱暴走を防ぐという機能である。例えばユーザーがHPCアプリケーションの高速化を図るために、全コアを稼働させたとすると、各コアの電力消費に伴う発熱によって全体温度が上昇し、閾値を超える可能性がある。その時スロットリング機能によってクロック周波数が下げられると、理想的なパフォーマンスが得られないどころか、寧ろ低下することが考えられる。

3つ目は、プログラムそれぞれには、並列化可能部分と並列化できない部分が存在する。並列数を増すと、並列実行される部分の実行時間は短縮が見込まれるが、並列化できていない部分はその恩恵を受けない。したがって性能向上のために並列数を上げるべきか、もしくは1つのコアを一杯稼働させるべきかどうかはプログラムや実行アルゴリズムに左右される。

以上の理由から、HPC クラスタにおけるCPU内には、使われずに余っているコアが存在することになる。我々は、この余剰コアを科学技術計算のために効率的に活用しようと考えている。

## 2.2 I/O ボトルネック

大規模計算を行う上で欠かせないスーパーコンピュータの性能は年々向上していることは先に述べた。それに伴い科学技術シミュレーションでは大量のデータが生成される

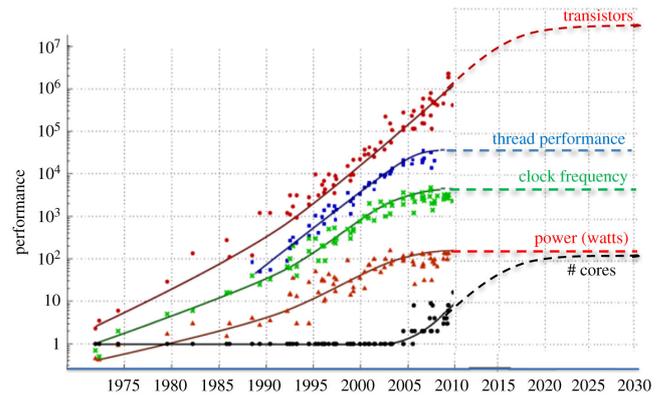


図 1: Microprocessor trend [1]

ようになっている一方で、ストレージ I/O 帯域幅とネットワーク帯域幅も増加しているが、演算性能の向上と大量に生成されるデータに対する相対的な伸びは大幅に下回っている。この演算性能と保存能力のギャップは、シミュレーションデータ出力後の可視化や分析の際のデータ移動、保存の大きな障害となる。これについて詳しく説明する。

スーパーコンピュータの演算性能が著しく向上していることは先に述べた。例えば、最新の TOP500 において1位を獲得した日本のスーパーコンピュータ「富岳」の性能を表1に示す。ピークパフォーマンスが倍精度浮動小数点で488 Petaflopsとなった。一方でメモリバンド幅は163 PB/s、ノード間のネットワーク幅は28 Gbps×2 lane×10 port、つまりは高々560 GB/sである。

この性能差によって、I/O ボトルネックが発生し、データ活用において重大な障害を引き起こす。極めて高い演算性能によって出力された大量のシミュレーションデータは、メモリに蓄えられる際、次にノード間を移動する際、そして最後にストレージに保存される際など、各ステップでその多くが損なわれることを避けられない。

従来、科学データ解析の有効な手段の1つである可視化のフローはシミュレーションの出力結果がストレージに書き込まれた後に、可視化ルーチンによってストレージから読み込まれ、可視化データへと加工されてから我々ユーザーの目に届くという順序であった。つまりこれまでの可視化は計算後に行われる後処理のタスクであった。しかしこれでは、先に説明したI/O ボトルネックにより、可視化できるデータは本来生成されたデータのほんの一部にすぎない。I/O ボトルネックという課題は今後も解消の目処が立っていない。したがって、より高度かつ複雑化しデータ量がさらに膨大になると考えられる将来の科学技術計算に対応するためには、I/O ボトルネックの解消を目指すだけでなく、それを回避するためのソフトウェアを考える必要がある。そこで現在注目を集めている技術が、In-situ 技術である。

表 1: Fugaku Specifications [2]

Peak Performance	488 Petaflops (FP64)
Total Memory	4.85 PiB
Bandwidth	163 PB/s
Interconnect	28 Gbps × 2 lane × 10 port

### 3. In-situ Visualization

近年、HPC 分野で In-situ 可視化は注目を集めており、積極的に研究されている。図 2 は従来の後処理の可視化と現在研究されている In-situ 可視化のフローそれぞれを示している。従来、科学技術計算におけるシミュレーションデータの可視化は図 2a のように後処理的なタスクとして行われてきた。つまり生成された膨大なデータの一部がメモリに蓄えられ、一杯になったらストレージに書き込まれる。こうして全てのシミュレーションが完了した後にデータがストレージから読み込まれ、可視化される。しかし、こうした手法では I/O ボトルネックにより生成データが損なわれるため、現在では図 2b に示されるフローの In-situ 可視化が採用されるようになってきている。簡潔にいうとシミュレーションと同時にリアルタイムで可視化して解析するという流れである。ここで図中の ParaView[3] とはオープンソースの科学計算向け可視化ソフトウェアで、Catalyst とは ParaView を用いて In-situ 可視化を行うためのライブラリである。出力データは ParaView のコプロセッシングアダプタに送られ、Catalyst に配置される。その後、データをその場で可視化するか、抽出して保存するかなどを選択的に決定する。可視化処理する際に、一度ストレージに保存するというステップを踏まないからこそ、出力データの全てにアプローチできる。

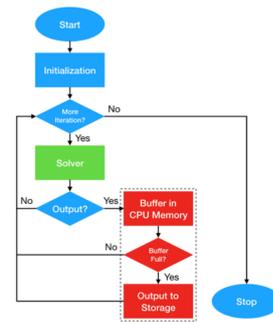
シミュレーションによっては、結果の生データが扱えずらく、人が読めるログ、統計、可視化データの方が望ましい場合がある。このような時にも In-situ 処理は有用な技術である。

### 4. 余剰コア活用フレームワーク UTHelper

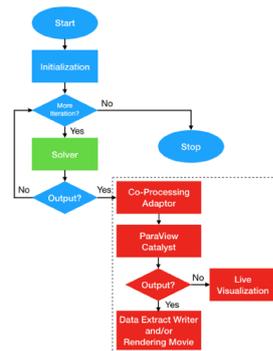
#### 4.1 概要

我々はプロセッサ内に存在する主計算の性能向上に寄与しないため、未使用のまま残されるコア「余剰コア」に、主計算をサポートするための処理を行わせることで、計算リソースを無駄なく使うだけでなくユーザーに更なる価値提供するフレームワーク“UTHelper”の実現を目指している。

余剰コアを活用した UTHelper が主計算をサポートするイメージを図 3 に示す。電力、並列性、メモリバンド幅などの制約によって主計算に割り当てる必要がない余剰コアが UTHelper に活用される様子が示されている。図 3 にお



(a) 後処理の可視化フロー



(b) In-situ 可視化のフロー

図 2: 後処理の可視化と In-situ 可視化のワークフロー [4]

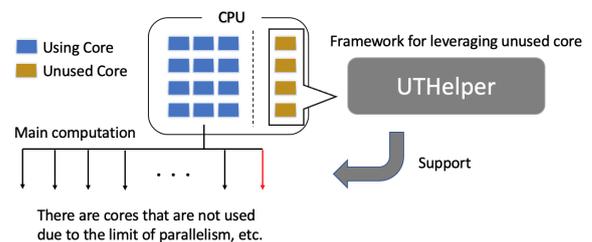


図 3: UTHelper の活用イメージ図

いては、主計算コア数より余剰コア数が少ない例が示されているが、その逆の可能性もある。したがって、その場合には In-situ 可視化などの比較的高負荷な処理も主計算に悪影響を小さくして、実行することが十分可能であると考えられる。

現時点で、UTHelper への搭載を予定している機能は以下のとおりである。これらは暫定的なものであり、フレームワークとして提供するべきだと考える技術は追加する予定である。尚、これらの機能は全てが独立しているわけではなく密接に関係しているものもある。例えば、並列数の自動調整には、実行中性能プロファイリングからフィードバックされた情報が使われる。このように各々の機能を連携させることによって価値が最大化するため、包括的な機能開発が重要となる。

- 実行中性能プロファイリング
- キャッシュプリフェッチ

- 並列数の自動調整
- コア割り当ての最適化
- In-situ でのデータ可視化・解析
- タスクの並列化

以上が現在、我々が着手している余剰コアを活用した主計算サポートフレームワーク“UTHelper”の概要である。

## 4.2 UTHelper の先行研究

工藤らの研究 [5], [6] によって実際に余剰コアを活用して実行中モニタリング、主計算の最適化を行うための仕組みづくりが行われた。

ここではフックという、プログラムの特定箇所に処理を追加する仕組みが根底にある。例えば、監視したい区間の前後にプロファイリングの開始と終了の指示する処理をフックによって挟むことで実行中モニタリングが実現できる。

関数のフックを実現するために採用されたのが SystemTap [7][8] というツールで、ユーザプログラムを改変することなく、様々な処理を挿入することができる。実際に使う際は `stap` コマンドを用い、引数にはユーザが挿入したい処理が書かれたスクリプトやオプションが入る。

また、フックした関数をプロファイルするためのツールとして PAPI が採用された。PAPI とは CPU のパフォーマンスカウンタからキャッシュミス率や実行クロック数などの情報をリアルタイムに取得できるツールである。

この研究では、SystemTap を用いて計測したい箇所の初めと終わり、それぞれに PAPI によるプロファイリングの開始と終了の指示を挿入し、様々なアプリケーションのプロファイルを行った。その結果、最適なコア数の検証や余剰コアの発生が確認された。また余剰コア上で、どれほど高負荷な処理を行えば、主計算に影響を及ぼすことになるのかについてを評価する実験が行われた。

## 4.3 余剰コアを活用した In-situ 可視化・解析の実現

In-situ 可視化の実装では、主計算と可視化ルーチンへの計算リソースの割り当て方が主に 2 つある。1 つ目は密結合である。計算コードと可視化コードを同じノード内に配置することでリソースを共有するため、同プロセスでそれぞれが実行される。2 つ目は疎結合である。計算コードと可視化コードを異なるノードに配置するため、計算ノードからシミュレーションデータが可視化専用ノードにネットワーク転送されてから可視化処理される。それぞれの配置イメージを図 4 に示す。図 4a が密結合、図 4b が疎結合のイメージを表している [9]。余剰コア上での In-situ 可視化は、主計算と同じリソース、つまりはシミュレーションコードと可視化コードが同一のノード内に存在することになるため、密結合となる。In-situ ルーチンとシミュレーションで直接リソースを共有するが故に、様々なメリット

とデメリットが存在する。それぞれについて説明する。

メリットとしてはまず、データアクセスのしやすさがある。メモリを共有しているためネットワークによるデータ転送を経る必要がないため、可視化コードはシミュレーションデータを容易に入手できる。次に、データの複製が容易であること。これも同様の理由である。In-situ 可視化によって、大量のデータを可視化するためには、データコピーによって新たに用意するのではなく、ゼロコピーによって I/O 帯域を節約する必要がある。このため、データ処理の容易さは肝心である。最後に、性能調整のしやすさがある。リソースの割り当て方が単純であるため、比較的性能チューニングが容易である。例えば、48 コアのプロセッサであれば、30 コアを主計算に、残りの 18 コアを可視化ルーチンにあてるというイメージである。また、ネットワーク転送の同期が不要なため、通信の影響を考慮する必要がない。

一方でデメリットもいくつか存在する。シミュレーションコードと可視化コードがメモリアクセスで競合する可能性があるため、精密な設計が求められる。メモリを共有しているからこそ、主計算データを保持する領域と可視化データを保持する領域をそれぞれ確保しなければならない。また、実行にあたってシミュレーションの各タイムステップ後に可視化完了を待たなければいけない。なぜなら同メモリを使用するため、可視化のための生データを保持しておくことができないからである。したがって非効率なチューニングや実装では、スレッド間の同期オーバーヘッドにより、急激な性能低下を招く可能性がある。

密結合はそのリソース割り当ての特性から、余剰コアを出さずにフルスケールでプロセッサを使用できる。またチューニングに関する問題も UTHelper で提供する予定の実行中性能プロファイリングによって最適化するような仕組みを用いれば対処できると考えている。したがって、余剰コアを活用した In-situ 可視化・解析と、その他のフレームワークの機能の相性は良いと言える。

実行中性能プロファイリングによる自動最適化機能のコアへの負荷は、In-situ 可視化・解析と比較すると小さいと考えられる。したがって、シミュレーションに対して常に動作させ、性能向上を目指すことが理想とする使われ方である。一方で、In-situ 可視化・解析機能を使うかどうかはユーザの選択によって決まる。余剰コアを使用するが、それでも発生する主計算への多少のボトルネックを加味したとしても、In-situ 可視化・解析を利用したいと考えた時に初めて利用される。ここでの UTHelper の価値は、In-situ 可視化・解析を誰もが容易に利用できる形で提供することにある。

## 4.4 設計にあたって

UTHelper の In-situ 可視化・解析機能の実装にあたって

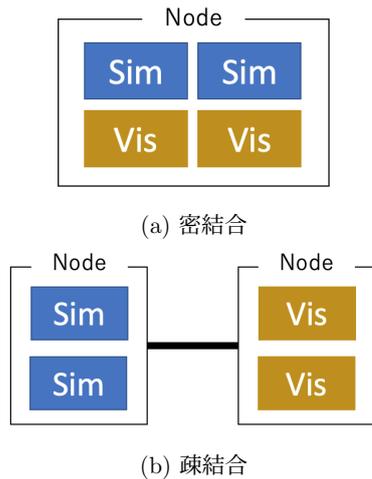


図 4: In-situ 可視化における密結合と疎結合のイメージ

の大前提として、

- (1) 主計算への悪影響を小さくする
  - (2) 余剰コアを使って柔軟に動作を変更できる
  - (3) 既存のプログラムに対して容易に機能追加できる
- という 3 つの条件は必ず充足されなければいけないと考えられる。

1 つ目は、主計算に悪影響を与えてはならないことである。余剰コア活用フレームワークの価値は、主計算の性能向上に貢献しないために未使用のまま残されているコアを活用することと、シミュレーションを行うにあたって、ユーザに様々なサポート機能を提供することの主に 2 つがある。フレームワークに割り当てられる計算リソースはあくまで余剰コアであるため、主計算リソースを奪ってまでフレームワークを動作させようとして、かえって性能を低下させてはならない。また計算リソースの分配が適切であったとしても、フレームワークが提供する機能のオーバーヘッドが主計算性能のボトルネックになる可能性も考えられる。

しかしどんなに理想的な設計であったとしても、機能によっては、ボトルネックをゼロにすることは極めて困難である。したがって、多少の性能低下を引き起こしたとしても、このデメリットを上回るメリット、つまりユーザが使いたいと考えられる機能を提供することに意味がある。

In-situ 可視化・解析機能は比較的、高負荷な処理が必要とされる。具体的な処理の流れとしては、主計算の各タイムステップに生成された生データを、余剰コアに割り当てられた可視化・解析ルーチンがメモリから読み取る。そして可視化・解析処理を行い、データを加工したのちユーザに届ける。各ステップ毎に処理を挟み込むため、実行時間が伸びることは避けられない。それでも主計算の負荷、可視化・解析の負荷、両方のバランスを考えて、ボトルネックを最小にするためには、パフォーマンスの監視を行いながら、それぞれに何コアずつ割り当てるのかを設定できる

必要がある。

2 つ目は、余剰コアを使って柔軟に動作を変更できることである。In-situ 可視化・解析において、全てのシミュレーションデータを可視化・解析処理することは、負荷の大きさを考慮すると得策ではない。例えば、科学計算において初めから解析したい時間範囲が決まっている場合は、In-situ 可視化・解析をシミュレーションの途中から開始、あるいは途中で終了してしまった方が良い。なぜならユーザにとって、見るべきデータが削減できたり、実行時間が短くなったりなどするからである。これを満たすためにはシミュレーション中に、主計算スレッドを監視しながら、ある時点でヘルパースレッドの中断する機能が必要となる。

3 つ目は、既存のプログラムに対して容易に機能追加できることである。フレームワークとして In-situ 可視化・解析機能を提供するにあたって、ユーザに利用をしやすいことが求められる。そのために、高いポータビリティによってユーザが元のプログラムを大幅に改変する必要のない仕組みをフレームワークで提供する。ユーザの工数はなるべく減らし、主に行うことはヘルパースレッドに解析したい変数を指定するなどの単純な作業のみに限定することを目指す。

これらを達成するために本研究では OpenMP task 構文を採用した。

## 5. OpenMP Task

本研究では、余剰コア上での In-situ 可視化・解析の実現に向けて OpenMP [10] の task 構文を採用した。これは、余剰コア上に様々なサポートタスクを割り当てる上で基本の動作モデルとなる。ここでは、task 構文の概要やその使い方、挙動を説明する。

### 5.1 Task 構文の概要

Task 構文とは、指示文を挿入することで使い、容易に for ループ以外のタスク、例えば関数や while ループ、再帰関数などを並列実行することができる。実際の使用例を Program 1 に示す。1 行目で 4 つのスレッドが生成され、そのうちの 1 つが 4 つのタスクを生成し、各タスクがタスクプールに置かれる。各スレッドはタスクプールから 1 つずつ取り出し、実行する。

```

1 #pragma omp parallel num_threads(4)
2 {
3     #pragma omp single
4     {
5         #pragma omp task
6         task_1();
7         #pragma omp task
8         task_2();
9         #pragma omp task
    
```

```

10     task_3();
11     #pragma omp task
12     task_4();
13 }
14 }
```

Program 1: Task 構文の使用例

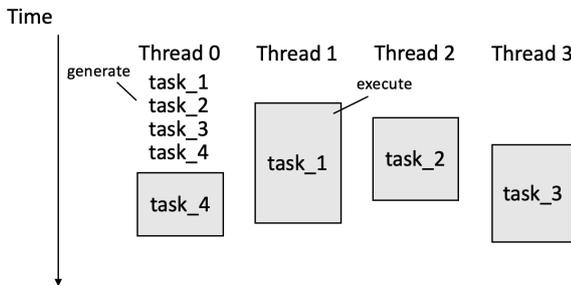


図 5: 挙動イメージ

## 5.2 task 構文の活用

task 構文とその他構文等を組み合わせて使うことで、先に説明した、3つの条件が達成できると考える。

主計算への悪影響を小さくするためには、OpenMP のスレッド数設定やコアアフィニティ機能が有用である。主計算と In-situ ルーチンそれぞれへのリソース割り当ての最適な調整を行うことができる。なお、先行研究では動的な並列数の変更が実現されている。

余剰コアを使って柔軟に動作変更を行うためには、その他構文や指示節を使う。今回は、その中でも taskwait 構文を紹介する。スレッドが taskwait 構文を検知すると、そのスレッドは、同じ階層の他の全てのタスクが終わるまでそこで待機することになる。使用例を Program 2 に示す。また図 6 に挙動イメージを示す。3 行目で single 領域に入ったスレッドは、5 行目でタスク 1 を生成し、13 行目の taskwait 構文を検知する。この時、タスク 3 には進まず、ここで待機する。そしてタスク 1 が完了したのを確認したら、タスク 3 に進む。

ここで注意しなければならないのは、タスク 3 はタスク 2 の完了を待ちはしない、ということである。タスク 2 は、13 行目の taskwait より、1 つ深い階層に定義されているからである。taskwait 構文を使うことによって、階層を利用したタスク並列をスケジューリングができる。さらに変数の依存関係を作るための task 構文 depend 節とも組み合わせて使うことによって、より柔軟な並列機構の設計が行えると考えられる。

```

1 #pragma omp parallel
2 {
3     #pragma omp single
```

```

4 {
5     #pragma omp task
6     {
7         task_1();
8         #pragma omp task
9         {
10            task_2();
11        }
12    }
13 #pragma omp taskwait
14 #pragma omp task
15 {
16     task_3();
17 }
18 }
19 }
```

Program 2: Taskwait の使用例

最後に、task 構文のタスクブロックは独立しており、並列実行したい処理の記述が混在することがないため比較的容易に機能を追加することが可能だと考えられる。

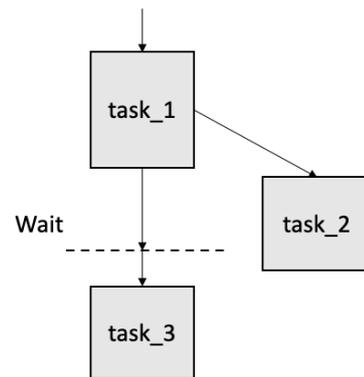


図 6: taskwait 構文の挙動イメージ

## 6. 実験と評価

今回は 2 つの実験を行った。1 つ目の実験では、ユーザが容易にタスクの並列化や In-situ 可視化・解析機能を使える仕組み発展に向けて、並列実行を行うための簡単な枠組みを設計し、その挙動を調べた。2 つ目では、設計した枠組みを既存アプリケーションに適用し、その性能を評価した。

### 6.1 実験環境

本研究での実験環境を表 2 に示す。本研究の全ての実験は同様の環境で行っている。

NVIDIA Nsight Systems [11] とは、NVIDIA から提供されている、CPU や GPU で実行されるアプリケーション

表 2: 実験環境

システム	Wisteria/BDEC-01(Aquarius)
CPU	Intel Xeon Platinum 8360Y
プロセッサ数 (コア数)	2 (36+36)
GPU	NVIDIA A100
コンパイラ	gcc-8.3.1
OpenMP	v4.5
NVIDIA Nsight Systems	v2020.3.4.32-52657a0

のパフォーマンス分析やトレース機能を備えた、プロファイラツールである。コマンドラインからプロファイル実行命令を出すのが、視覚化用 GUI も備えているため、ユーザーは出力結果を手元の PC で視覚的に分析することが可能である。

本実験で使った際の流れを簡単に説明する。Nsight Systems を用いて HPC クラスタ上で実行中のプログラムをプロファイリングし、その出力結果を手元の PC の視覚化用 GUI で分析を行った。尚、このツールには、用途に合わせて様々な実行コマンドが用意されているが、今回、主に使用したのは、nsys profile コマンドである。オプションによって、トレースの開始時間や、期間、収集する情報などを指定できる。またプログラムにマーカーを埋め込むことで、その箇所にどのスレッドが割り当てられているのか確認できる。

## 6.2 並列実行のための枠組み設計

Task 構文を用いてユーザーが容易に並列実行を行うための簡単な枠組みを設計し、その挙動を調べた。この枠組みを Program 3 に示す。10 行目の simulation() には、科学技術シミュレーションなどを想定しており、14 行目の data() には、主計算データを analysis() に渡すなどのデータ転送処理を想定している。最後に、analysis() には、主計算データを解析するための処理を想定している。

動作の流れを説明する。1 ループ目、シミュレーションループに入った単スレッドは、主計算タスクを生成する。13 行目には、taskwait 構文が置かれているので、主計算が終わるまでそこで待機する。主計算が終わると、データ転送部に進み、解析タスクの生成、実行を行う。解析はタスク並列で記述されているため、解析の実行と並行して、次のループの主計算タスクの生成、実行が始まる。そして taskwait 構文により、前のループの解析タスクと今のループの主計算タスクが完了するまで、そこで待機し、同期をとる。その後、またデータ転送部を経て、解析タスクと次のループの主計算タスクが実行されるという流れである。

以上の動作を実際に動かしてみて、トレースした結果を図 7 に示す。なお、今回の目的は枠組みが想定通りの挙動を示すかどうかをまず確認することであるため、各処理を sleep 関数で置き換えて実験した。トレース結果から想定し

た挙動を示していることがわかる。主計算 (simulation) と解析 (analysis) はオーバーラップされ、データ転送 (data) は逐次処理となっている。

```

1 #pragma omp parallel
2 {
3     #pragma omp single
4     {
5         // simulation loop start
6         while(1){
7             #pragma omp task
8             {
9                 // Main computation
10                simulation();
11            }
12            #pragma omp taskwait
13            // Not parallelizable
14            data();
15            #pragma omp task
16            {
17                // Analysis
18                analysis();
19            }
20        }
21        // simulation loop end
22    }
23 }
    
```

Program 3: タスク並列、In-situ 可視化・解析実装に向けた枠組み

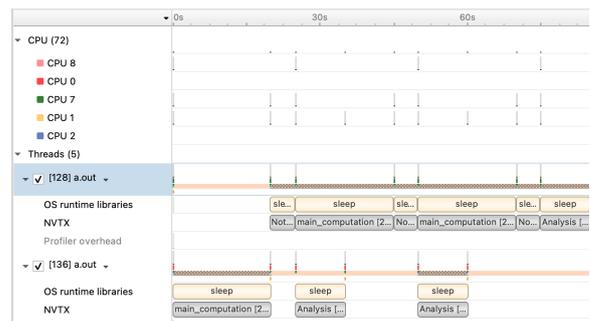


図 7: Program 3 に基づいて記述したコードのトレース結果

## 6.3 実アプリケーションによる評価

実アプリケーションにおける性能評価の対象として、重力ツリーコード GOTHIC (Gravitational Oct-Tree code accelerated by Hierarchical time step Controlling) [12], [13] を用いた銀河衝突シミュレーションを取りあげる。銀河衝突シミュレーションにおいては、 $N$  体計算による時間進

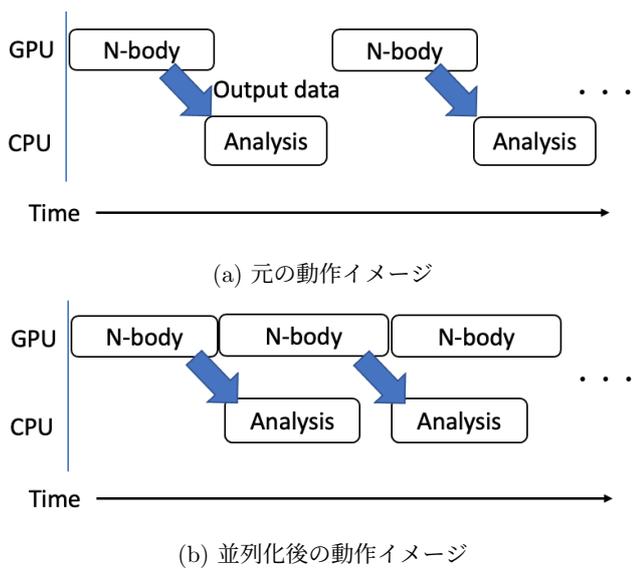


図 8: GOthic の並列化前後の動作イメージ

化の計算だけではなく、形成された構造と観測された構造とを比較する解析処理も必要である。多くの場合には、 $N$  体計算によって出力したスナップショットファイルを生成した後に、解析コードを用いてスナップショットを別途解析するという手順がとられる。しかし、観測されている構造をうまく再現できる衝突パラメータを探索するためには多数回の  $N$  体計算の実行が必要であるため、大量のファイル出力が生じてしまう。そこで  $N$  体計算を実行しながら同時に解析処理も実行することで、ファイルの出力量を削減しつつ研究全体のワークフローの効率化が図られている [14]。

GOthic においてはこうした  $N$  体計算と解析処理両方の処理を

- (1) 主計算である  $N$  体計算を GPU 上でを行い、一定時間の時間進化の計算後に
- (2) 計算結果を GPU から CPU へと転送し、CPU 上で解析処理を行う

という手順を繰り返すように実装されており、In-situ 解析の実装例である。この動作イメージは図 8a に示したとおりであり、GPU 上で主計算が行われている間には CPU は休んでおり、反対に CPU 上での解析処理中には GPU が休んでいる。つまり計算リソースを効率的に利用できおらず、余剰コアの活用による性能向上が期待できる。

本研究では、task 構文を用いて主計算と解析を並列化した。この時の動作イメージが図 8b であり、GPU 上での主計算の裏で、CPU は 1 つ前のループのデータ解析を行っている。GPU, CPU 共に休む時間が減るため、計算リソースの無駄を削減できる。

実際に GOthic に並列化を施して、1 ループあたりの実行時間の計測と Nsight Systems によるトレースを行い、

元のプログラムと比較した。なお、使用コア数は 18 コアとなっており、そのうち 17 コアが解析処理のために、残りの 1 コアは task 構文によるオーバーラップのために使われている。実験の結果を図 9 と図 10 に示す。図 9a は元のプログラムの実行においてスレッドをトレースした結果であり、main computation が GPU での  $N$  体計算（主計算）、analysis が CPU 上での解析処理を示している。図 8a のイメージどおり、それぞれが逐次処理であることが読み取れる。図 9b に示した task 構文による並列化後のトレース結果では、main computation と analysis が並列実行されていることが読み取れ、計算リソースを有効に活用できていることが確かめられた。

図 10 は、元のプログラムと並列化後の実行時間を示している。なお、主計算と解析のループが 5 回行われた際の実行時間の平均を取ることで 1 ループあたりの実行時間としている。1 は元のプログラムの 1 ループあたりの実行時間を示しており、主計算と解析の合計時間は 0.549s となった。2 は並列化後の 1 ループあたりの実行時間を示しており、0.296s となった。逐次処理の場合の実行時間と比較して、およそ 54% まで短くなった。

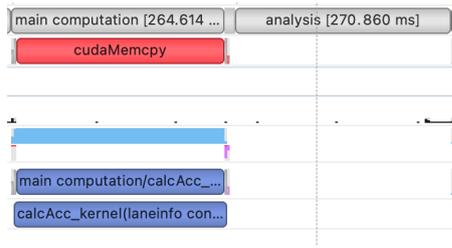
以上の性能評価によって、task 構文による主計算と解析処理の並列化による性能向上が確認できた。今回は、主計算と解析の実行時間に大きな差がなかったため、並列化によって実行時間がおよそ半分になった。しかし、実アプリケーションによっては、主計算または解析のどちらか一方の実行時間が他方よりも長い場合もある。そのような場合にも並列実行可能であることを確かめる必要がある。

図 11 は、主計算と解析の処理時間に差を作った場合の、逐次処理と並列処理の実行時間を示している。図 11a は主計算より解析の時間を長くした場合、図 11b は解析より主計算の時間を長くした場合の実行時間を示している。この図から、どちらのパターンでも並列化により実行時間が短縮されていることがわかる。逐次処理の時間と比較すると、図 11a では 79%、図 11b では 69% 程度となった。主計算、解析どちらか実行時間が長い方の処理がボトルネックとなり、並列化後の実行時間として現れていることがわかる。

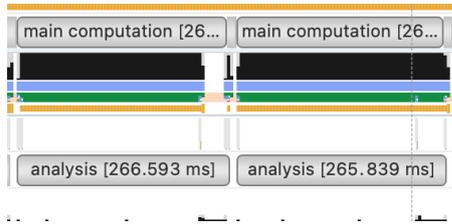
## 7. おわりに

本研究では、余剰コア活用フレームワーク “UTHelper” の搭載予定機能である、ユーザフレンドリな In-situ 可視化・解析の実現に向けて、OpenMP task 構文を用いた枠組みの設計と実アプリケーションによるその性能評価を行った。

今後は、まずは元々、主計算と解析・可視化などの主計算結果に対する処理が逐次的に行われる、様々な既存のコードに対して、設計した枠組みで容易に並列化可能かどうか確かめる必要がある。もしそれが可能だと判断できた



(a) 元のプログラムのトレース



(b) 並列実行のトレース

図 9: 並列化前後の実行トレース

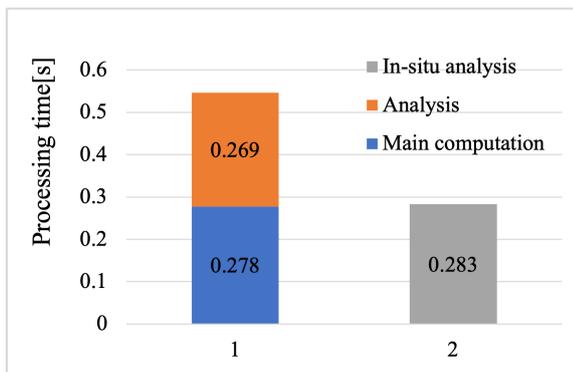


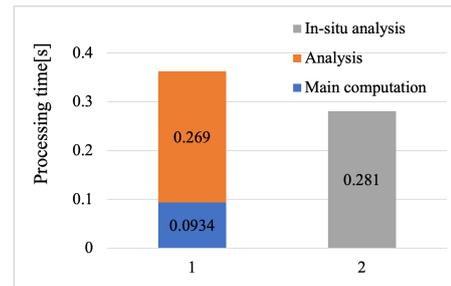
図 10: GOthic の計算ループを 5 回実行した際の平均実行時間

ら、今度はユーザが元のコードを改変する工程をなるべく減らし、最終的には自動で並列化できる仕組みが実現できればさらに利便性が高まる。

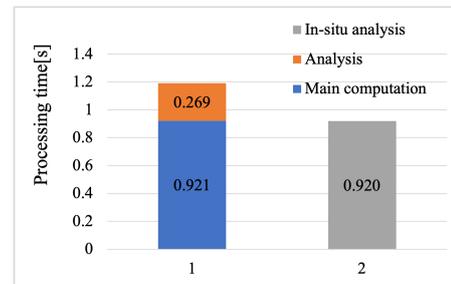
また、今回は主計算と解析の 2 つのタスク並列であったが、そこにファイルへの書き出しなどを含めた、3 つのタスク並列を考えることでさらにリソースを最大限使えるようになる。

元のプログラムと並列化後の性能評価指標として、今回は実行時間を選んだ。そこで電力や発熱について考えてみる。In-situ 解析により実行時間が短縮されたとしても、元の逐次処理の場合と比較すると、並列化によりコアが最大限活用されるため、時間あたりの CPU の負荷は増すと考えられる。したがって、この時の実行時間全体の電力消費や発熱量を並列化前後で比較した時、どちらが効率的にコアを使えていると言えるのかは、性能問題の観点から議論する必要がある。

最後に、既存の主計算と解析などを兼ね備えたコードに対する並列化が容易に行える仕組みづくりができて初め



(a) 解析が長い場合



(b) 主計算が長い場合

図 11: 主計算と解析の実行時間に大きな差があるパターンの実験結果

て、主計算のみのシミュレーションコードに対して、新たに In-situ 可視化・解析機能を組み込めるようにフレームワークとして成立させることに取り組むことができる。

謝辞 本研究の一部は、JSPS 科研費 JP20H00580 および JP20K14517 の助成を受けたものです。

## 参考文献

- [1] Igor L. Markov: The core software framework for the LHCb Upgrade, *Nature*, Vol. 514, pp. 147–154 (online), DOI: <https://doi.org/10.1038/nature13570> (2014).
- [2] Fujitsu Limited: Specifications, (online), available from (<https://www.fujitsu.com/jp/about/businesspolicy/tech/fugaku/specifications/>).
- [3] Kitware Inc. and Los Alamos National Laboratory.: ParaView: An End-User Tool for Large Data Visualization, (online), available from (<https://www.paraview.org/>).
- [4] Mu, D., Moran, J., Zhou, H., Cui, Y., Hawkins, R., Tatineni, M. and Campbell, S.: In-situ Analysis and Visualization of Earthquake Simulation, Practice and Experience in Advanced Research Computing (PEARC '19), New York, NY, USA, ACM, (online), available from (<https://doi.org/10.1145/1122445.1122456>) (2019).
- [5] Kudo, J.: 高性能計算システムに向けた余剰コア活用フレームワーク, Master's thesis, The University of Tokyo (2021).
- [6] 工藤 純, 埜 敏博: 余剰コアの活用に向けた実行中プロファイリング手法の検討, 研究報告ハイパフォーマンスコンピューティング (HPC), Vol. 2020-HPC-177, No. 7, pp. 1–8 (オンライン), 入手先 (<http://id.nii.ac.jp/1001/00208769/>) (2020).
- [7] Prasad, Vara and Cohen, William and Eigler, FC and Hunt, Martin and Keniston, Jim and Chen, Brad: Locating system problems using dynamic instrumentation,

Citeseer (2005).

- [8] Systemtap: <https://sourceware.org/systemtap/>.
- [9] James Kress: In Situ Visualization Techniques for High Performance Computing, Area exam, University of Oregon, Computer and Information Sciences Department (2017). Available at <https://www.cs.uoregon.edu/Reports/AREA-201703-Kress.pdf>.
- [10] Open Multi-Processing: <https://www.openmp.org/>.
- [11] NVIDIA Corporation: NVIDIA Nsight Systems, (online), available from (<https://docs.nvidia.com/nsight-systems/>).
- [12] Miki, Y. and Umemura, M.: GOthic: Gravitational oct-tree code accelerated by hierarchical time step controlling, *New Astronomy*, Vol. 52, pp. 65–81 (online), DOI: 10.1016/j.newast.2016.10.007 (2017).
- [13] Miki, Y.: Gravitational Octree Code Performance Evaluation on Volta GPU, *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, ACM, pp. 62:1–62:10 (online), DOI: 10.1145/3337821.3337845 (2019).
- [14] Miki, Y., Mori, M., Kawaguchi, T. and Saito, Y.: Hunting a Wandering Supermassive Black Hole in the M31 Halo Hermitage, *The Astrophysical Journal*, Vol. 783, p. 87 (online), DOI: 10.1088/0004-637X/783/2/87 (2014).