修士論文

余剰コアを活用したOpenMP Taskによる
In-situ解析の実現

令和4年1月27日 提出

指導教員　塙 敏博　教授

東京大学大学院
工学系研究科　電気系工学専攻
37-206453　　赤沢 龍哉

要旨

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より，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した．実際に，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の短縮を確認した．

Abstract

In recent CPUs, only the increasing number of cores has contributed to improving the performance, and many-core CPUs are widely used in HPC systems. However, higher effective performance can be obtained by intentionally leaving unused cores than by using all the cores in the CPU in many cases. Therefore, we aim to realize “UTHelper”, a framework that utilizes the “unused cores” that do not contribute to the performance improvement of the main computation for the sub-task. In this study, we proposed utilizing the OpenMP task pragma as an efficient method for pre-post processing to supplement the main computation. In practice, we applied this method to GOTHIC, an N-body calculation code for astrophysics, in which the main computation and analysis were performed sequentially, and confirmed that the execution time of the entire simulation was reduced by overlapping the main computation and analysis with a simple modification.

目 次

図 目 次

表 目 次

第1章　序論

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演算性能は，著しく向上している．近年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，昨年，世界 1 位を達成したスーパーコンピュータ「富岳」には，1CPU あたり 48 コア搭載されている．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，マルチコアからメニーコアへとが移り変わっている．

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ことになる．したがって，意図的に使用コア数を制限し，一部のコアを未使用のまま残すことがある．

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．

またスーパーコンピュータのデータ活用においては，演算性能の向上に I/O 帯域幅の性能向上が追いついていないため，I/O ボトルネックにより，出力されるデータの全てを保存し解析することはできないという問題が発生している．

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．しかし実装が複雑であるため，簡単に利用することが難しい．そこで我々は，既存のコードへの適用を容易にするフレームワークを提案する．この技術を UTHelper の機能として提供することで，ユーザフレンドリな In-situ 計算を実現しようと考えている．

本研究では既存の科学アプリケーションに対してパイプライン処理を適切に表現することで，In-situ 解析を行い，その性能を評価する．また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．

以降，2 章では HPC システムの性能トレンドや性能諸問題を述べ，3 章では関連研究について紹介する．そして 4 章では，余剰コア活用フレームワーク”UTHelper”の概要や余剰コア上での In-situ 可視化・解析の方針について述べる．5 章では，使用するツールやその使い方を説明する．6 章では，実験と評価を行う．最後に 7 章で結論と今後の課題について述べる．

第2章　背景:システムとメニーコアCPU

本章ではI/Oボトルネックによる性能問題や，多数のコアを使う際に生じる性能低下について説明する．

2.1　CPUのメニーコア化

図2.1に1990年代から現在までのスーパーコンピュータのピークパフォーマンスのトレンドを示す．過去70年間で約500億倍近く，計算性能が向上していることが読み取れる．この性能向上を支えている要因について説明する．

図2.2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着いてきた一方で，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．デジタルコンピューティング用の論理ゲートの実装には実用上の限界が訪れると考えられており，ITRS (International Technology Roadmap for Semiconductors) によると2021年以降もその壁は乗り越えられないと言われている[8]．過去50年間ムーアの法則に従ってきた技術開発はそれに追従できず，図2.2によると2025年までに平坦化すると予想されている．

ムーアの法則に追従できない中で，今後，HPCクラスタ，CPUの性能を最大限発揮するためには，コンピュータアーキテクチャとソフトウェアの親和性を高めることが重要である．具体的には，CPU内のコアを効率的に扱うことやI/Oボトルネックを回避したソフトウェア設計が要求される．

2.2　余剰コアの発生

図2.2から分かるように今，CPU内のコア数は増加傾向にある．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，近年のスーパーコンピュータ「富岳」や「Wisteria/BDEC-01」で使われているCPUの演算コア数は48となり，マルチコアからメニーコアへと移り変わっている．表2.1に他のCPUのコア数を示す．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ていないという問題だ．コ

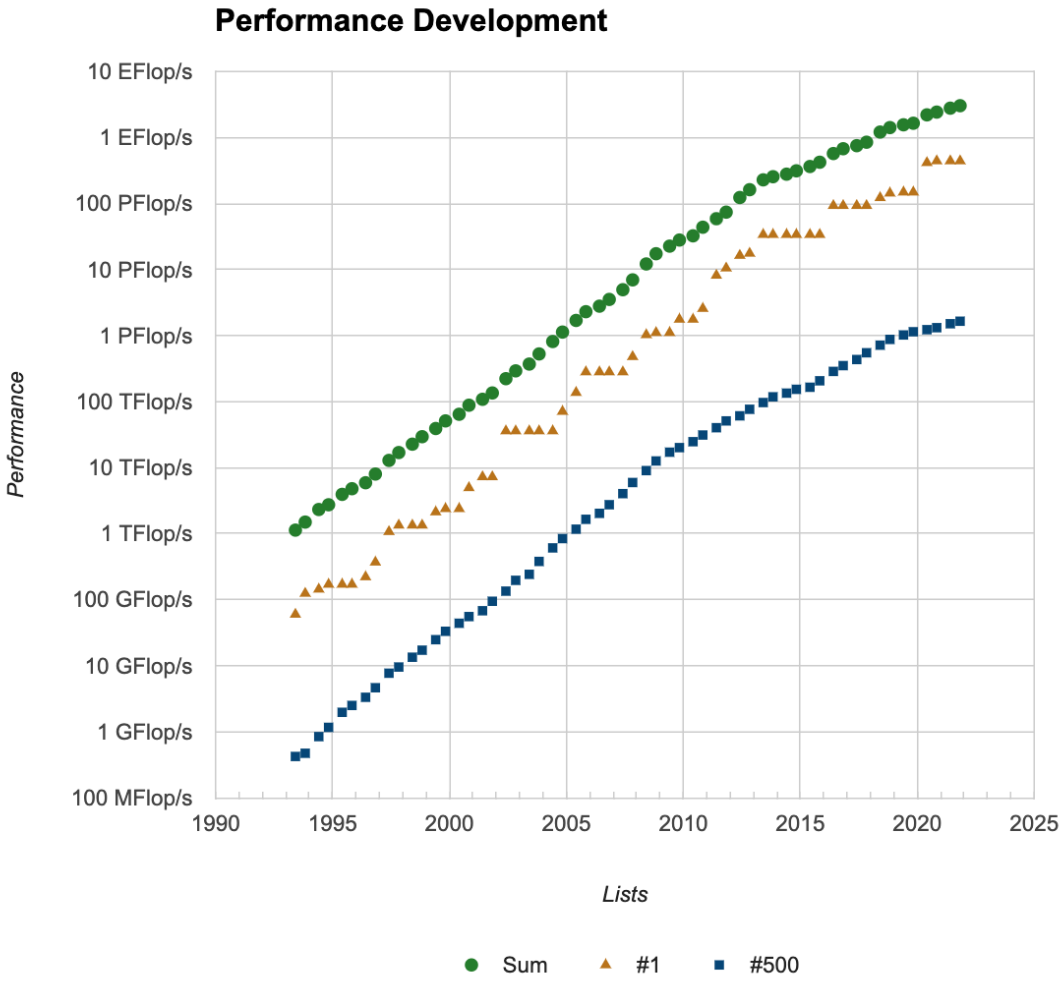


図 2.1　HPC Performance[1]

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もある．

表 2.1　スーパーコンピュータに使われたプロセッサの仕様

プロセッサ	コア数/ソケット	ピーク性能 [FLOPS](FP64)	メモリ帯域 [GB/
Fujitsu A64FX(2019)	48	3.3792 テラ	1024
Intel Xeon Platinum 8260L(2019)	24	1766.4 ギガ	140.8
SPARC64 VIIIfx(2010)	8	128 ギガ	64

ではなぜ全てのコアを活用しきれないのかについて説明する．

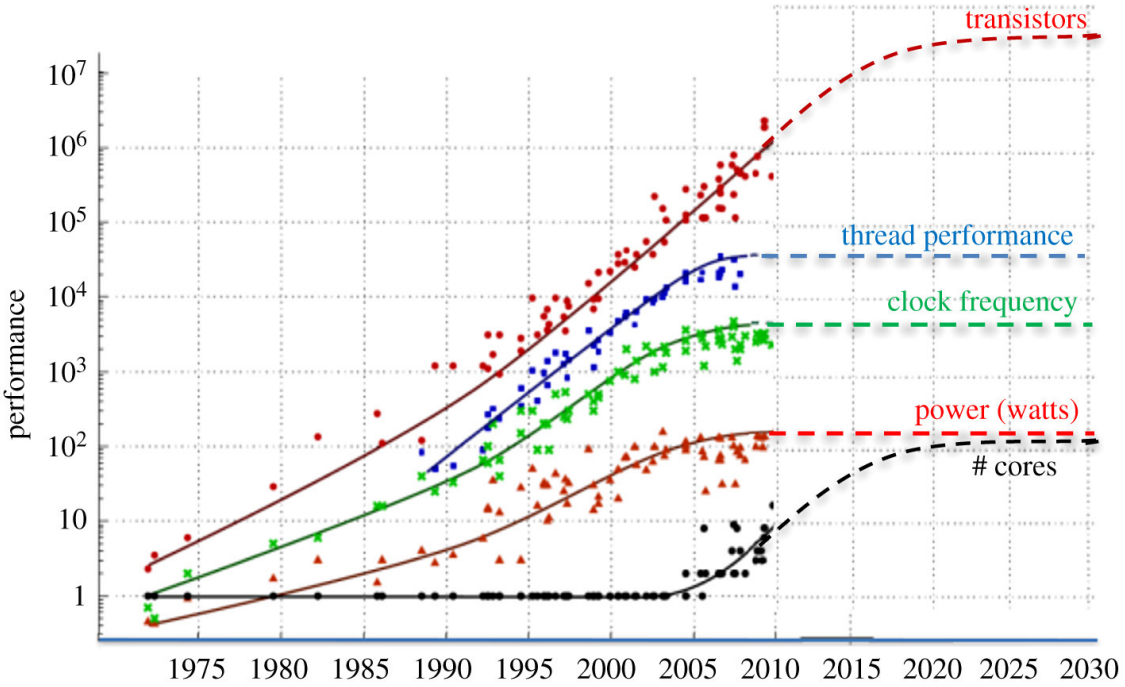


図 2.2　Microprocessor trends[2]

1．並列性の限界

1 つ目は並列性の限界である．プログラムそれぞれには，並列可能部分と，そうではない直列部分が存在する．つまり並列数を増すと並列可能部分の実行時間は短縮が見込まれるが，直列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 つのコアを目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．この関係性を定量的に示す，“ アムダールの法則 ” というものがある．アムダールの法則によると処理効率とはそれを構成する個々の平均の効率ではなく，ある非効率部分によって律速されると言われている．それは関係式で表現できる．N を並列台数，S を直列部割合，P を並列部割合とした時，速度向上比は以下の式で与えられる．

$$\text{速度向上比} = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

図 2.3 は並列化する前とした後の処理の流れのイメージ図である．並列化前は実行にかかる時間は S+P で表されるが，並列化後は S+P/N とかける．つまり並列部割合 P が小さいときは，並列台数をいくら増大しても，速度向上は頭打ちとなる．すなわちアプリケーションによっては全コアを使う必要がない可能性がある．

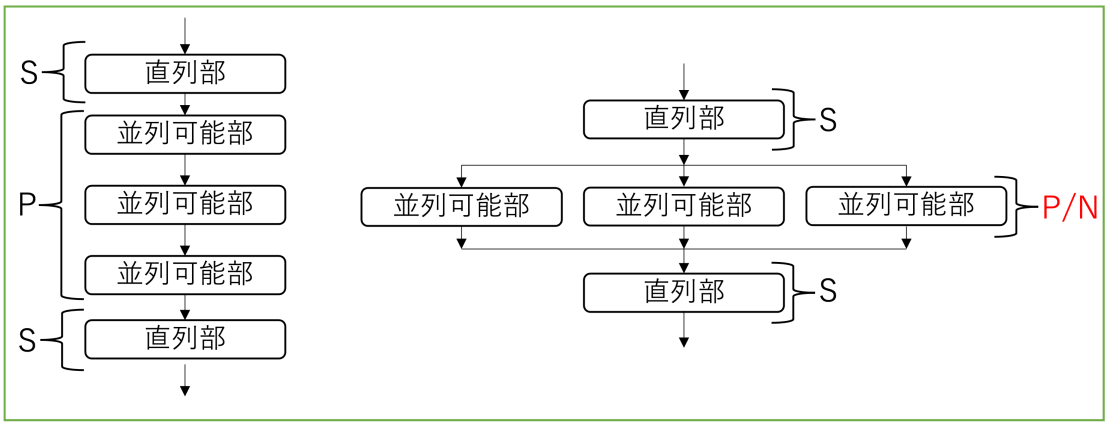


図 2.3　並列化前後の処理イメージ

2. メモリバンド幅の制約

2 つ目は，CPU の実行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という問題である．最近は，メニーコア化によりプロセッサの演算性能が著しく向上しているため，この問題は重大となっている．

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．

3. 熱や電力の問題

3 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．その時に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．

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計算のために効率的に活用しようと考えている．これについては 3 章の関連研究で詳しく説明する．

2.3　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．それに伴い科学シミュレーションでは大量のデータが生成されるようになっている一方，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，計算能力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．しかしこの計算能力と保存能力のギャップは、エクサスケールでの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．

表 2.2　Fugaku Specifications[7].

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

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，今年日本が開発したスーパーコンピュータ富岳の性能が世界で 1 位を獲得した，富岳の性能を表 2.2 に示す．ピークパフォーマンスが倍精度浮動小数点で 488Petaflops となった．ペタスケールからエクサスケールへとまさに移行してきている．一方で，トータルメモリは 4.85PiB，メモリバンド幅は 163PB/s，ノード間のネットワーク幅は 28 Gbps x 2 lane x 10 port，つまりは高々560GB/s である．ピークパフォーマンスとトータルメモリの性能を比較するとおよそ 100 倍もの差があり，メモリバンド幅，ネットワーク幅についてもその差は歴然である．この性能差は，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．これを I/O ボトルネックという．

実際の科学シミュレーションを例に取る．磁気を閉じ込めた核融合装置におけるプラズマ乱流をシミュレーションする XGC1 コード [9] は，ペタスケールでは平均 100 ステップ

毎にデータを保存できていた．しかし，エクサスケールへ適用するとその保存頻度は 1000〜10000 ステップ毎になってしまい，保存の間隙の貴重な科学情報等が失われる可能性が大きくなる．当然，このことは他の科学計算コードにも該当し，多くの科学者がデータ解析する上で悪影響を及ぼす．

従来，科学データ解析の有効な手段の一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．しかしこれだと，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は立っていない．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 可視化である．これについては 3 章の関連研究で詳細に取り上げる．

第3章　関連研究

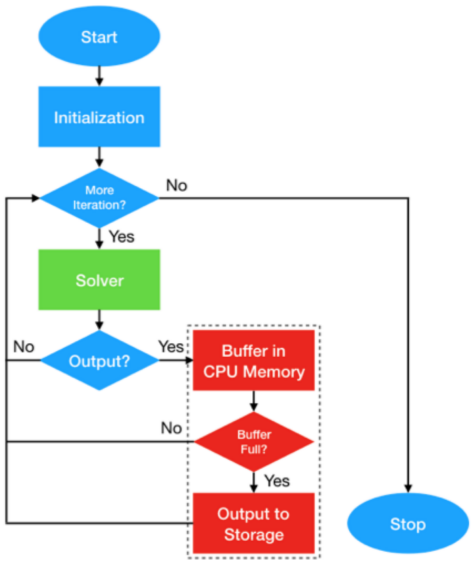
本章では，2 章で述べた余剰コアを活用するため，I/O ボトルネックを回避してデータを可視化するための技術 In-situ 可視化についての関連研究を紹介する．

3.1　In-situ Visualization

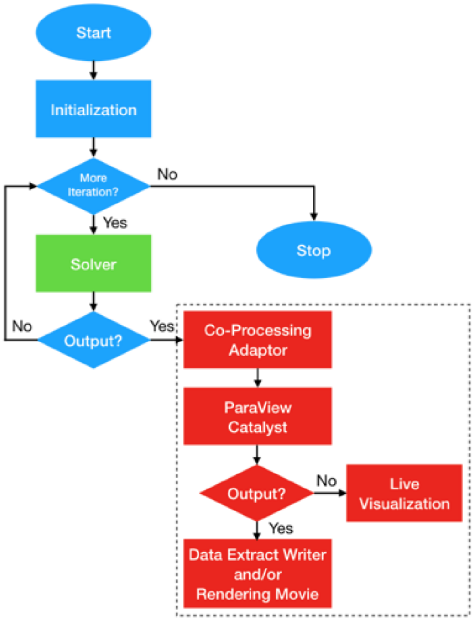
3.1.1　概要

In-situ 可視化は，近年，HPC 分野で注目を集めており，積極的に研究されている．図 3.1 は従来の後処理の可視化と現在研究されている In-situ での可視化のフローそれぞれを示している．従来，科学計算におけるシミュレーションデータの可視化は図 3.1a 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．しかしこれだと生成データが損なわれるため現在は，図 **??**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[10] 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，Catalyst によってインタラクティブにシミュレーションにアクセスし，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．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，結果の生のデータが扱いづらく，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 可視化は有用な技術である．In-situ 可視化の概要については説明した．次に In-situ 可視化ためのツールについて紹介する．



(a) 後処理の可視化フロー



(b) In-situ 可視化のフロー

図 3.1　後処理の可視化と In-situ 可視化のワークフロー [3]

3.1.2　In-situ 可視化ツール

米国のエネルギー省 (DOE) が資金提供をして開発が進められてきた「SENSEI[?]」というフレームワークがある．SENSEI は In-situ でのデータ分析のためのフレームワークであり，大きく 4 つのコンポーネントで構成されている．1 つ目はデータモデルである．解析側が可視化のためのデータ表現をできるように VTK を採用している．2 つ目は，データアダプタである．シミュレーション側のデータ構造と VTK のデータモデル間のデータマッピングのためにある．3 つ目は，解析アダプタである．SENSEI のインターフェースが ParaView Catalyst などのさまざまな In-situ インフラストラクチャと接続するためのメカニズムである．四つ目は In-situ ブリッジである．これはアプリケーションの実行中に SENSEI にデータ移行や制御命令ために，アプリケーション開発者が実装する API である．ブリッジの簡単な実装イメージ例は、シミュレーションの初期化フェーズでデータアダプタと解析アダプタを初期化し、各タイムステップで現在のシミュレーションデータ配列とその他のメタデータをデータアダプタに渡し、解析アダプタの実行を呼び出すというものである．

では実際に In-situ 可視化を行ったものをみせる．

3.1.3　VisIt LibSim を用いた In-situ Visualization

VisIt[11] は、オープンソースの相互的かつスケーラブルな可視化・アニメーション・解析ツールであり，さまざまな規模のデータに対応している．2 次元および 3 次元の構造化メッシュ、適応メッシュ、非構造化メッシュで定義されたスカラーおよびベクトル場を含むさまざまなデータを表示できる豊富な機能を備えている。なお，HPC クラスタのような大規模な分散メモリを備えたシステムで効率的に動作するように設計されている．さらに LibSim という VisIt で In-situ 可視化を実行するためのライブラリが存在する．

今回は，In-situ での可視化のデモとして「CloverLeaf3D[12]」というミニアプリのシミュレーション中に LibSim で，In-situ 可視化を行った．その様子を示しているのが，図 3.2 である．左のウィンドウがシミュレーション結果を映しており，右のウィンドウが VisIt 上で可視化している様子を示している．

このように In-situ 可視化は技術的には実用的なものとなってきているが，ParaView や VisIt を用いての In-situ 可視化は課題を抱えている．元々これらのソフトウェアは後処理の可視化のために開発されており，完全に In-situ 向けではない．そのため，実装の複雑さや多くの他のソフトウェア，コンポーネントに依存しており，ユーザ，科学者にとって敷居が高い．さらに近年は CPU と GPU をのせたスーパーコンピュータが登場しているが，それらで効率良く使用するための機能を備えていない．したがってユーザーにとって使用が容易かつ柔軟な設計の In-situ のインフラストラクチャが必要となる．このような背景から現在，DOE で研究開発が進められているプロジェクトがある．それについて紹介する．

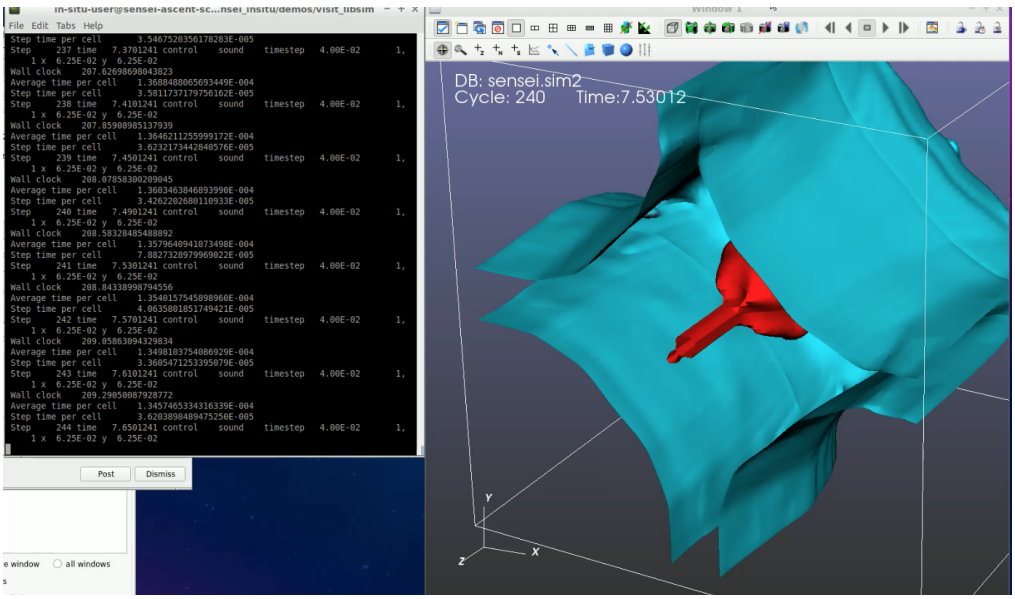


図 3.2　Visit LibSim を用いた In-situ Visualization

3.1.4　The ALPINE Ascent

ALPINE というエクサスケールコンピューティングプロジェクトがある．これはエクサスケールシステムに移行するにあたって，DOE が資金提供をし，複数の研究機関によって取り組まれている．これにより，従来の In-situ 可視化ライブラリの課題を克服するために新たに開発されたのが，“ Ascent ”である．Ascent は主に次の 3 つの特徴がある．

1. 最先端のスーパーコンピューティングアーキテクチャをサポート
2. 軽量なインフラストラクチャ
3. 他のソフトウェアとの相互運用性

Ascent は最新のスーパーコンピュータを念頭において設計されている．つまり GPU のサポート，さらには分散メモリをサポートしており，ノード内の共有メモリの並列処理，ノード間の分散メモリでの並列処理のハイブリッドな実行ができる．これは，Ascent がハードウェアに依存しない VTK-m という可視化ライブラリに MPI を組み合わせた VTK-h が使われているからである．

ユーザが比較的簡単に使用可能，シミュレーションコードに負担を与えない，他のパッケージに 非依存，処理中のデータコピーとメモリ使用に関してのオーバーヘッドを最小限にする，これらの利点により，ユーザー使用，ソフトウェアの両方の意味で軽量なインフラストラクチャを実現可能にしている．

そして VTK-m は Ascent 内で重要な可視化ツールの役割をしているが，Ascent ランタイムが新たにライブラリを追加するために設計されている．Ascent は実行を管理するため，

具体的には，VTK-h のフィルターなどを効率的に作成，実行するための“ Flow ”というデータフローライブラリを利用する．しかし，データ型に囚われない設計のため，フローライブラリは VTK-h に依存しない．これにより，他のデータモデルや API のデータを処理できるフィルターを柔軟に作成できる．

実際の Ascent のシステムは図 3.3 のようになっている．シミュレーションの生成データを Ascent API で可視化できるようメッシュデータに変換する．その後の処理機能としては主にデータをフィルタを通して変換したり，レンダリング法を設定できる Pipeline，データを描画し，画像として保存する Scenes，欲しいデータを抽出する Extracts という 3 つがある．これについては後で実際に使った様子を示す．続いて，Ascent の依存関係を図 3.4 に示す．幾つかのコンポーネントがあるが，ここでは主に使用される“ Conduit ”,“ Flow ”,“ VTK-h ”の 3 つについて説明する．

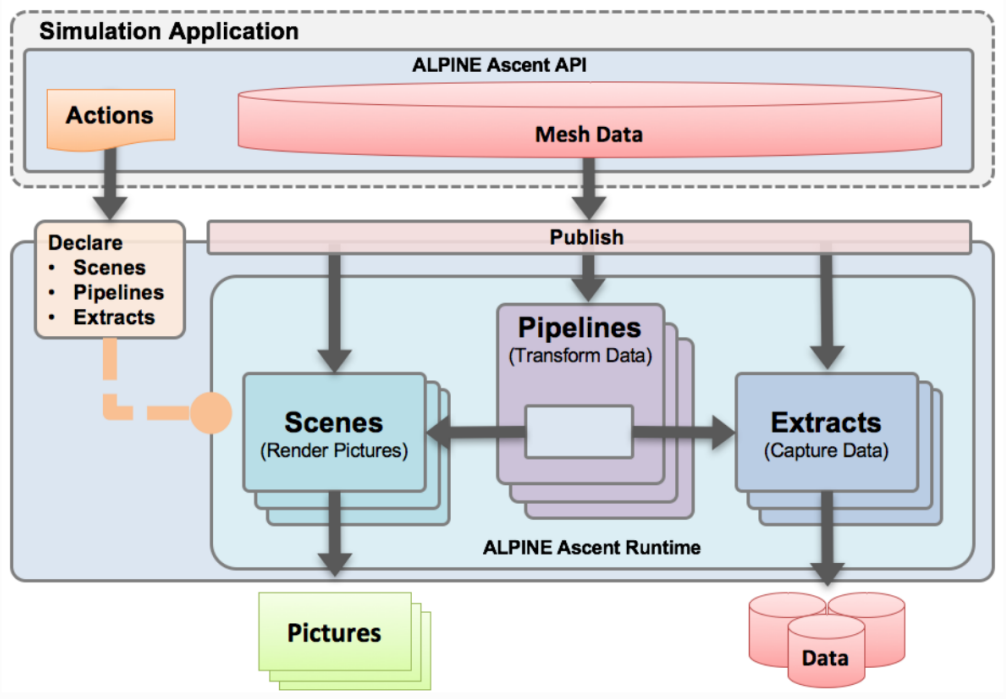


図 3.3　Ascent のシステム概要 [4]

- Conduit

Conduit とはローレンス・リバモア国立研究所が開発したオープンソースのプロジェクトであり，C++，C， Fortran，および Python の複数言語で階層的に科学データを記述できる直感的なモデルを提供している．図 3.5 に Conduit の使用例を示す．初めに conduit.Node() で Conduit のオブジェクト n を生成する．次に一階層 my に data を定義している．次に階層/a/b/c の下に d という階層を生成している．尚，スラッシュを使わずとも多次元配列の

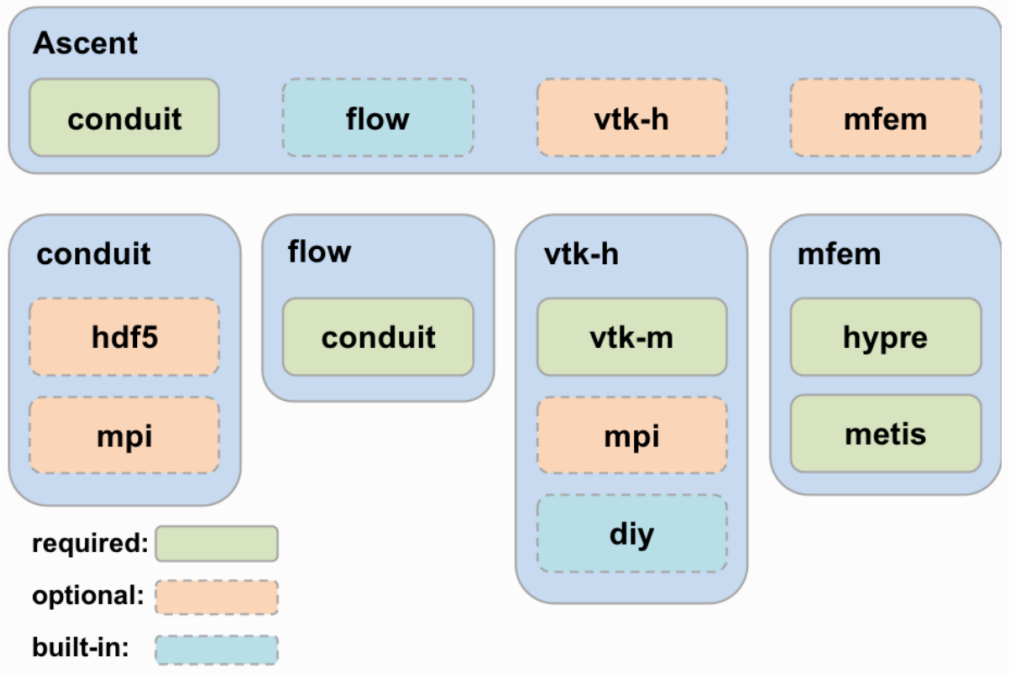


図 3.4　Ascent の依存関係 [4]

ように階層を記述することもできることが示されている．n の出力を見ると階層的データ表現ができていることがわかる．

```
>>> import conduit
>>> n = conduit.Node()
>>> n["my"] = "data";
>>> n["a/b/c"] = "d";
>>> n["a"]["b"]["e"] = 64.0;
>>> print(n)

my: "data"
a:
  b:
    c: "d"
    e: 64.0
```

図 3.5　conduit の使用例

- VTK-h

VTK-h は，共有メモリの並列処理に重点を置いた VTK-m ライブラリの上に分散メモリ層を実装するスタンドアロンライブラリであり，実行モデルは含まれない．VTK-h の設計は，VTK-m アルゴリズムのラッピングを容易にして，ALPINE Ascent，ParaView，VisIt などの他の視覚化ツールの実行モデルに含めることができるようにすることを目的としている．また MPI を使用した分散メモリでの並列処理に対応している．

- Flow

VTK-h は VTK-m をラッピングしたものであり，独自の実行モデルを提供していないことを述べた．これにより，VTK-h API が簡素化され，ParaView および VisIt の既存のフル機能の実行モデル内で VTK-h を簡単に活用できるようになる．ALPINE Ascent は ParaView または VisIt のインフラストラクチャを活用 しないため、ユーザーの要求されたアクションを実行するための VTK-h アルゴリズムの使用をサポートするための基本的な実行モデルが必要である．したがって，Ascent は Flow というデータフローライブラリを使うことで VTK-h のフィルタを作成，実行することができる．

そして最後に Ascent の機能を実際に使った様子を紹介する．

- Pipeline と Scenes

パイプラインを使用すると、ユーザーは入力データを新しいメッシュに変換するフィルターを作成できる．ここで，ユーザーは一般的な幾何学的変換 (クリッピングやスライス)，フィールドベースの変換 (しきい値や輪郭) を指定する．各パイプラインからの結果データは，Scenes または Extracts への入力として使用できる．図 3.6 は実際に，データに等高線とカラーの設定をするパイプラインを構築し Scenes によって画像を出力した際のパラメータ設定である．action:”add_pipelines”を宣言し，パイプラインを構築してからフィルターの型やパラメータを設定する．ここでは field:”braid”で頂点中心スカラー場を形成し，iso_values で等値面の値を設定している．そして画像として保存するために，action:”add_scenes”を宣言し，画像を出力するためのフィルターを用意する．type:”pseudocolor”で擬似カラー，pipeline:”pl1”で構築したパイプラインの使用などを設定している．Scenes は，scenes[”s1/plots/p1/type ”] = ”pseudocolor ”; のように記述し，画像を生成するためのための情報をカプセル化する．Scenes を定義するために，ユーザーはプロットのコレクション (ボリュームや面のレンダリング) と，カメラの定義，ライトの位置などの複数のパラメータを指定する．少なくともプロットの記述には，プロットタイプとスカラー場の名称が必要で，この場合の Scenes は，シミュレーションによって生成された全てのデータを，デフォルトの設定で描画する．実際に描画すると図 3.7 のような画像が出力される．

以上が最先端の In-situ 可視化のインフラストラクチャである Ascent の紹介である．このプロジェクトで研究されている Ascent は ParaView などのこれまでの In-situ 可視化の，ユーザ利用には敷居が高いなどの課題を克服した．しかし，可視化や出力などのパラメータは前もってユーザーが自ら設定する必要があるため，予め出力データの情報がある程度分からなければ，適切な設定が行えず新たな発見することが困難である．

```
action: "add_pipelines"
pipelines:
  pl1:
    f1:
      type: "contour"
      params:
        field: "braid"
        iso_values: [0.200000002980232, 0.400000005960464]

action: "add_scenes"
scenes:
  s1:
    plots:
      p1:
        type: "pseudocolor"
        pipeline: "pl1"
        field: "braid"
    image_name: "out_pipeline_ex1_contour"
```

図 3.6　Pipelines と Scenes のパラメータ設定

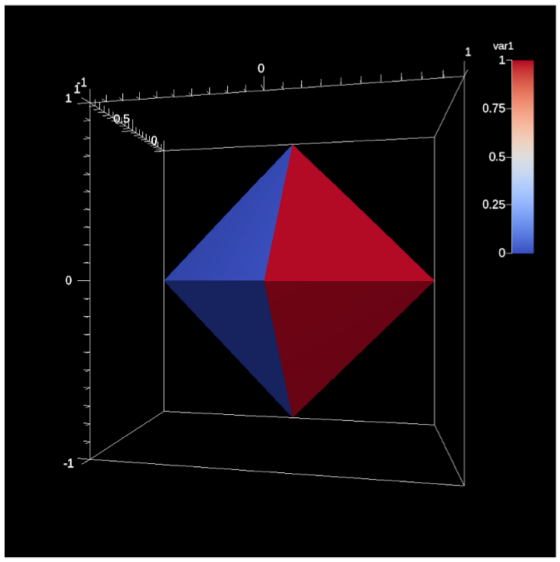


図 3.7　Scenes による画像出力

そして最後に，In-situ 可視化には性能の最適化という大きな課題がある．各コンポーネントは互いにデータをやり取りし続けるため，相互作用しながらプロセッサやネットワーク帯域幅などの計算リソースを使い合う．つまりプロセッサコアなどをどのように割り当てるかによって全体の性能が大きく変化する．極端な例でいうと，シミュレーション側にほとんどのリソースを割けば，In-situ 可視化の処理がボトルネックとなり，大幅に速度低下しうる．したがって最大性能に近づけ，シミュレーションの妨げにならないよう，設定しなければならない．しかし実行中に，最適なチューニングを行うのは難しい．なぜならば，設定すべきパラメータは各コンポーネントが作用し合うため，アプリケーション単体

のそれよりも倍数的に増加し膨大となり手がつかなくなるからである．

3.1.5　In-situ ワークフローのオートチューニング

Tong Shu らの研究 [5] は，ブートストラップ法を用い，ML ベースのと ACM ベースのホワイトボックスモデリングとブラックボックスモデリングを組み合わせ，In-situ ワークフローのオートチューニングアルゴリズム CEAL に実装して，パラメータのオートチューニングを行うものである．オートチューナーの設計の要はモデリングアルゴリズムである．例えばニューラルネットワークだと，学習コストがかかりすぎるため In-situ ワークフロー使用できない．したがってフェーズ 1 のホワイトボックスモデリングとフェーズ 2 のブラックボックスモデリングの 2 つのフェーズで行う．図 3.8 にアルゴリズムの概要を示す．

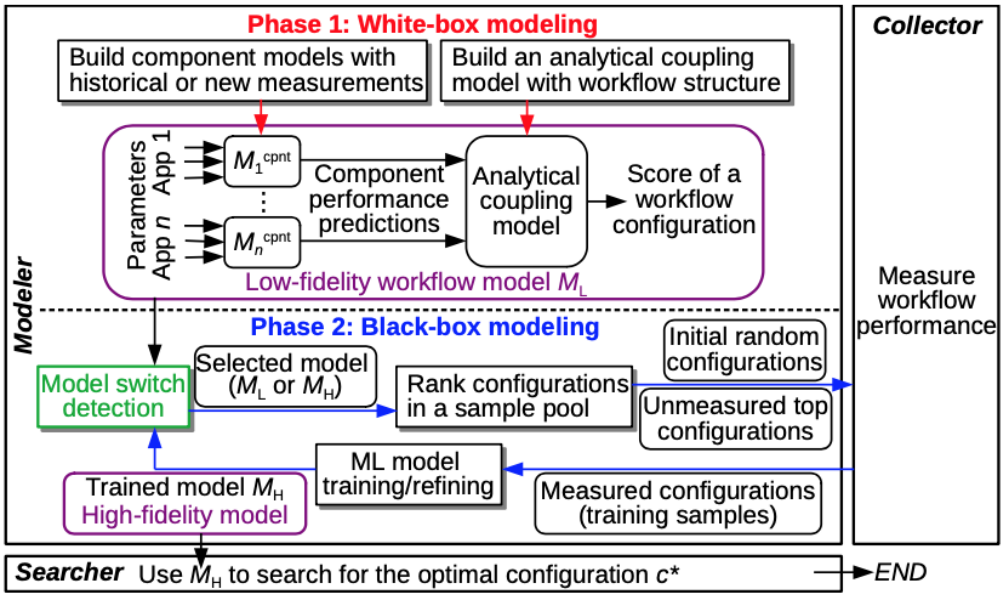


図 3.8　ブートストラップ法を用いた In-situ ワークフローのオートチューニング [5]

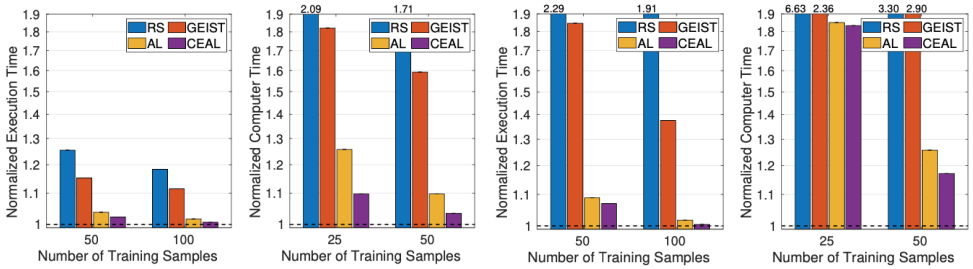
フェーズ 1 では，各コンポーネントが元々独立であるという特性を利用して，個々の性能モデルを構築する．パラメータはそれぞれが相互作用するワークフローよりも大幅に少ないため，低コストで構築でき，更には過去にアプリ単独で使用した際の値を再利用できるという利点がある．これらの各コンポーネントモデルを組み合わせて，低品質のワークフローモデルを構築する．このモデルは In-situ ワークフローの全体モデルのパラメータ選定の指針となる．

しかし，このモデルはコンポーネントを単純な組み合わせにより構築されているため，相互作用が考慮されておらず，近似的なモデルとなっている．したがって，フェーズ 2 ではこの近似モデルのスコアリングを元に厳選された標本を使用して，より高精度なモデルを構築する．高精度モデルは構築初めは，精度が高くないが繰り返し学習することで向上し，

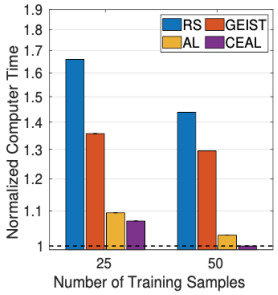
近似モデルよりも高い忠実度を示すようになる可能性がある．そこで近似モデルと高精度モデルを監視し，より良い方を選択する．このアプローチは，パラメータ空間を小さくするのではなく，空間の全範囲から部分的に選択され，In-situ ワークフローの性能が低い領域からは少ない標本数にし，高い領域からは多くを選択することで精度の高いモデルを実現する．

こうして実装された CEAL アルゴリズムを用いて HPC 向けアプリケーションの性能を測定した実験結果が図 3.9 である．比較対象となるのは，ランダムサンプリングによる学習 **RS** や他 **AL**，**GEIST**，そして CEAL でのコンポーネント結合にブラックボックスモデリングを用いた **ALpH** である．そして次に実際に適用したアプリケーションを述べる．*LV* は，分子動力学シミュレータ LAMMPS[13] とボロノイ型テッセレータ Voro++[14] の 2 つのコンポーネントを結合したアプリケーションであり，粒子シミュレーションや可視化における多くのケースのモデルとなっている．*HS* は，熱伝導シミュレーション Heat Transfer[15] と解析アプリケーションである Stage Write の 2 つのコンポーネントを結合しており，PDE 計算や I/O バッファリングおよびフォワーディングにおけるケースモデルとなっている．*GP* は，次の 4 つのコンポーネントの組み合わせである．反応拡散系 Gray-Scott 方程式シミュレーション，その出力解析アプリケーション PDF calculator，同様に可視化解析アプリケーション G-Plot，最後に PDF calculator の出力に適用される可視化アプリ P-plot，以上である．

CEAL での LV，HS，GP の実行時間，計算時間で正規化した際の，各アルゴリズムについての値が図 3.9 にプロットしてある．CEAL が他のアルゴリズムの性能を上回っていることが読み取れる．



(a) *LV*　(b) *HS*



(c) *GP*

図 3.9　The best configuration auto-tuned w/o historical measurements (dashed lines: the best configuration in the test set)[5]

この研究では今後 In-situ 可視化を行う上で必要な性能最適化について取り扱った．機能としては確立してもそれを実用する上では繊細なパラメータ設定が必要となることが説明された．そこでオートチューニングを開発し，その有効性が示された．

我々が余剰コアを活用しようと考えていることはすでに述べた．その活用法の 1 つとして In-situ 可視化を想定している．我々がそれを実現する上で，ここまで紹介した研究，ソフトウェアは密接に関係しており，指針となるものである．

3.2　余剰コアを活用した実行中モニタリングの研究

本研究室，工藤純らの研究では，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を行った．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する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 というツールで，任意の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 を用いることで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い，次に余剰コア上で，どれほどの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を行った．

この研究は，本研究でも紹介する余剰コア活用フレームワークに関わる研究であるため，4 章で詳しく説明する．

第4章　余剰コア活用フレームワーク UTHelper

2.2 節では,「余剰コア」というものを説明した．2.3 節や 3 章では，高い演算性能により大量にデータが生成されたとしても，I/O ボトルネックにより有効活用できるのはその一部に過ぎないため In-situ 可視化技術が鍵になることを述べた．

本章では，我々が考え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 ”についての紹介し，その概要や機能について説明する．

4.1　背景

スーパーコンピュータはあらゆる科学分野の数値シミュレーションを支えるものであり，必要不可欠である．しかし近年の科学計算は複雑化しており，生成されるデータが大量であるため I/O ボトルネックを回避する必要があったり，生データ自体が解析困難である可能性がある．

そこで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．

本研究では，UTHelper への搭載予定の機能である，In-situ 可視化や非同期解析の実装に使用予定である，OpenMP の task 構文を用いて，In-situ での解析を行った．

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 4.1 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 4.1 においては，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及ぼさずに，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なく密接に関係している．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．

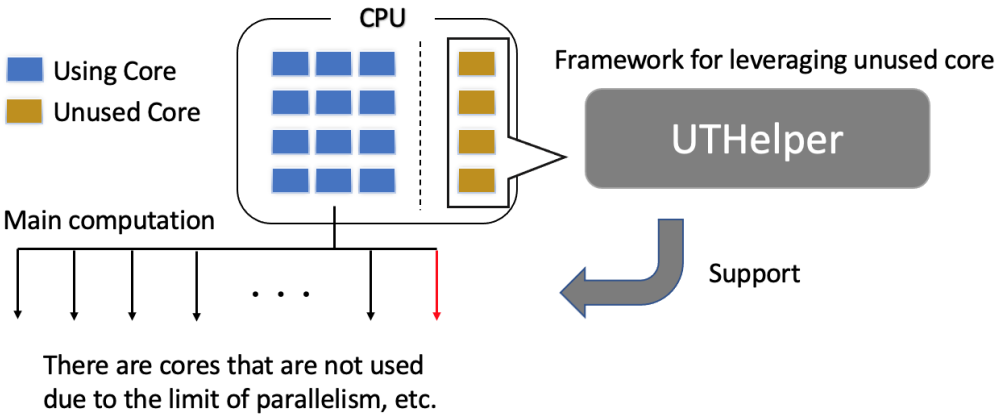


図 4.1　UTHelper の活用イメージ図

- 実行中性能プロファイリング
- In-situ でのデータ可視化・解析
- 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
- 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

3 章の関連研究で述べた工藤純らの研究 [6] では，フレームワークの機能である実行中性能プロファイリングや並列数の自動調整の実現に向けた基盤実装が進められた．そこでは，SystemTap の実際の活用例が示されており，そのスクリプトを示す．Program 4.1 は，処理を挿入されるプログラムであり，Program 4.2 は，フック処理が書かれた Systemtap のスクリプトである．これらについて簡単に解説する．Program 4.1 では 2 行目に OpenMP による並列実行の宣言がされている．この並列数を SystemTap によって決定しているのが Program 4.2 である．具体的には関数 func(); が実行される前に並列実行数を 20 に設定している．

```
1 void func(){
2 #pragma omp parallel for
3   ...
```

```
4 }
5
6 int main(){
7    func();
8    ...
9 }
```

Program 4.1　フック対象のプログラム [6]

```
1  %{
2  #include <omp.h>
3  }%
4
5  function _set_omp(num) %{
6    omp_set_dynamic(0);
7    omp_set_num_threads(STAP_ARG_num);
8  %}
9
10 probe process("PROGRAM_PATH").function("func").call{
11   _set_omp(20);
12 }
```

Program 4.2　OpenMP の並列数を変更する SystemTap スクリプト [6]

SystemTap を使用が主計算に対する大きなオーバーヘッドになると，フレームワーク基盤としての実用は難しい．そこで STREAM ベンチマーク [16] のプログラムに対して，SystemTap を用いて関数のフックを行い，PAPI によってパフォーマンスカウンタを取得した際のオーバーヘッドを評価する実験が行われた．その際の PAPI の計測指示をどのように行うかについて，以下の 3 パターンの実行時間が計測された．

1. 何も行わない
2. プログラムに直接処理を追記しリコンパイル
3. SystemTap でフックを実行

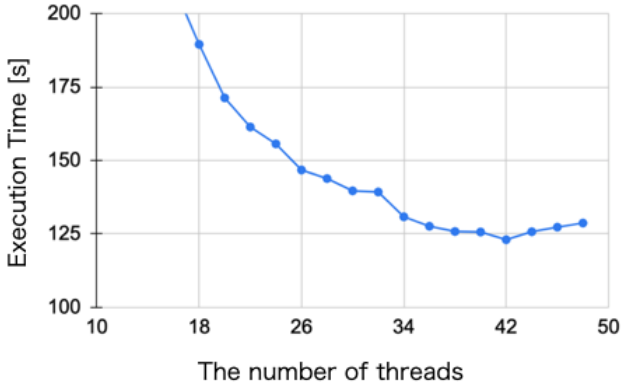
STREAM は，Copy・Scale・Add・Triad の順で実行され，そのループ回数は指定することができる．今回は，1 万回に指定され，ループ全体の前後で PAPI による計測開始と終了を指示した．こうしてベンチマークの実行開始から実行終了までにかかった時間を計測する実験をそれぞれのパターンに対して 10 回行った．表 4.1 に実験結果を示す．SystemTap によるオーバーヘッドは，プログラムに直接追記した場合と SystemTap によるフックを実行した場合との比較で得られ，その差はおよそ 0.15%ほどであり，低オーバーヘッドとなった．

続いて，SystemTap を使用したプロファイリングの実験が行われ，余剰コアの存在が確認された．プロファイルの対象となったのは，NAS Parallel Benchmarks の FT である．FT とは 3 次元 FFT を解くベンチマークである．問題サイズとしては D クラスが設定された．

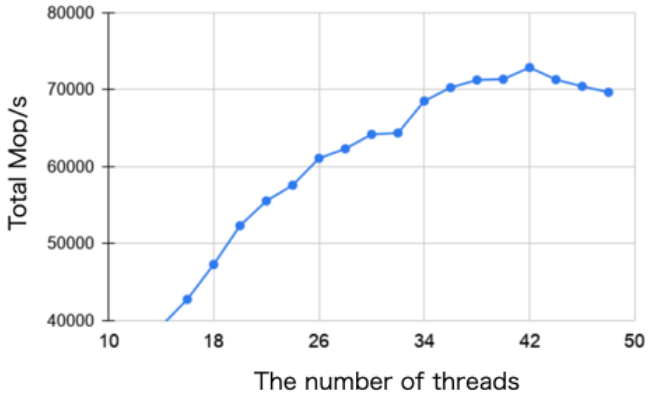
表 4.1　それぞれの実行パターンによる 10 回実行した際の平均・最小・最大時間 [6]

実験パターン	平均実行時間 [s]	最小実行時間 [s]	最大実行時間 [s]	平均オーバーヘッド
何も行わない	93.359	93.076	93.611	
プログラムに直接追記	99.430	99.391	99.737	+ 6.5030%
SystemTap によるフックを実行	99.587	99.167	99.858	+ 6.6710%

FT 実行中に，PAPI で各コアごとのパフォーマンスカウンタ，具体的には，キャッシュミス率や実行クロック数，参照クロック数などが取得された．その際，使用コア数は 2 から 48 まで 2 ずつ変化させながら実験された．上記の内容の実験結果が図 4.2 である．結果を見ると，図 4.2a ではコア数 42 が最小となっており，4.2b でも 42 の時が最大となっている．つまりコア数 42 が本実験での最適なコア数となる．つまりこの時，(48-42=)6 コアは余っている状態と言え，余剰コアが観測された．



(a) ベンチマークの実行出力から得た実行時間



(b) ベンチマークの実行出力から得た Total Mop/s

図 4.2　稼働コア数 (並列数) ごとの NAS Parallel Bechmarks FT (クラス D) の実験結果 [6]

この研究は，フレームワークの機能である実行中性能プロファイリングや並列数の自動最適化の実現に向けて，プロトタイプの実装やその実験が行われた．本研究はフレームワークの別の機能である In-situ 可視化・解析やタスクの並列化の実現に向けて，進められたという位置付けである．

4.3　余剰コアを活用した In-situ 可視化・解析

本研究では，UTHelper の機能の 1 つとして、In-situ 可視化・解析機能を提供することを目指している。そこで，余剰コア上での In-situ 可視化・解析を実装方針やどのようなシチュエーションでの利用を想定しているのかについて検討したことを述べる．

4.3.1　余剰コアを活用した In-situ 可視化・解析

例えば，In-situ 可視化の実装では，主計算側と可視化ルーチン側への計算リソースの割り当て方法として，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 4.3 に示す．図 4.3a が密結合，図 4.3b が疎結合のイメージを表している．

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ので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．

4.3.2　In-situ 可視化・解析の利用

本節では，ユーザがどのようなシチュエーションに置かれた場合に，UTHelper の In-situ 可視化・解析を利用すべきかや想定している利用の流れについて述べる．

UTHelper は余剰コアを活用するフレームワークであるが，そもそもユーザはシミュレーションを実行するにあたって，余剰コアの存在を予め把握していたり，わざわざ調べたりするケースはあまり考えられない．

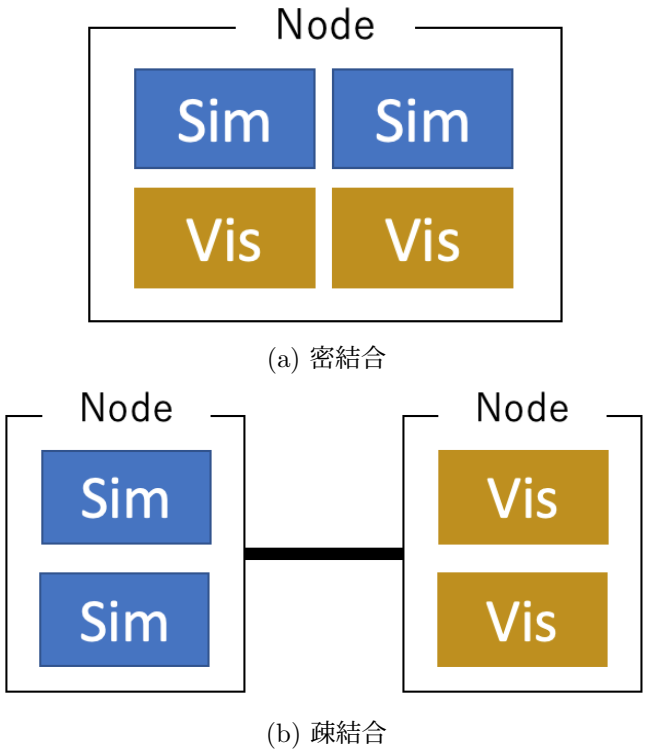
実行中性能プロファイリングによる自動最適化機能のコアへの負荷は，In-situ 可視化・解析と比較すると小さいと考えられる．したがって，シミュレーションに対して常に動作させ，性能向上を目指すことが理想とする使われ方である．

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．

したがって，ここでの UTHelper の価値は，まずユーザにとって In-situ 可視化・解析を利用すべきだという気づきを与えることや誰もが容易に利用できる形で提供することにある．

実際に使う流れを説明する．ユーザは元のアプリケーションに大幅な改変をすることなく，フレームワークを組み込む形となる．In-situ 可視化・解析を利用する際は，ユーザは解析したい変数や解析頻度などのパラメータを決定する．そうして実行することで出力結果は，生データではなく，読みやすいデータに変換される．

以上がユーザ利用の説明となる．これを踏まえて，5 章では，設計する際に具体的にどのような要件が必要となるのかを述べる．



(a) 密結合

(b) 疎結合

図 4.3　In-situ 可視化における密結合と疎結合のイメージ

4.3.3　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，大前提，次の 3 つの条件は必ず充足されなければいけないと考えた．

1. 主計算に悪影響を与えないこと
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

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くすることが求められる．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．

第5章　OpenMP Task

本章では，OpenMP Task 構文について紹介し，本研究でどのように使っているのかを説明する．

5.1　OpenMP の概要

初めに簡単に，OpenMP について説明する．OpenMP(Open Multi-Processing)[17] とは，共有メモリ型計算機環境において，並列プログラミングを可能にする API であり，Fortran, C, C++のプログラミング言語に対応している．特徴としては，以下のようなことが挙げられる．

- 既存のシリアルなプログラムを並列化
- 単純な指示文を挿入するだけで並列実行可能
- スレッドの待機や同期などの制御が可能
- スレッド数やコアアフィニティが設定しやすい
- 環境変数が用意

元々シリアルなプログラムであっても，指示文を挿入するだけで並列化が行える．また，並列実行の際，各スレッド間での変数の共有，占有やスレッドの同期を指示することができる．さらに，環境変数が用意されており，コアのアフィニティや並列数の設定が行える．現在も研究開発が進められている，ツールである．OpenMP の使用例である Program 5.1 を考える．2 行目の関数 func1() は，単スレッドのみの実行になる．3 行目には`#pragma omp parallel`指示文が挿入されている．したがってスレッドはここで fork され，ブロック内の関数 func2() は複数スレッドによって並列実行される．func2() を実行後，ブロックの終わりまでくると全てのスレッドの終了を待ち，スレッドは join される．その後，func3() が単スレッドで実行される．この様子を示したのが図 5.1 である．

```
1 int main(){
2   func1();
3   #pragma omp parallel
4   {
5     func2();
6   }
```

```
7     func3();
8 }
```

Program 5.1　OpenMP を用いた並列実行

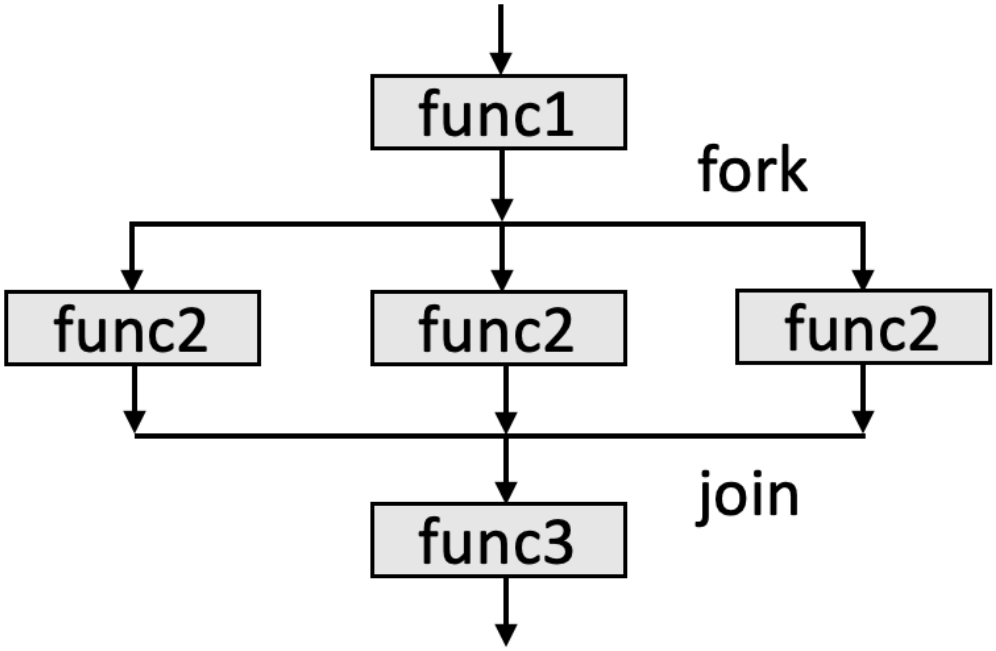


図 5.1　Program 5.1 の実行イメージ

OpenMP には，ワークシェアリングやデータ属性を指示する構文などが用意されている．そのうちの 1 つに OpenMP task 構文がある．

5.2　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 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．

5.2.1　task 構文の概要

task 構文とは，バージョン 3.0 で新たに追加された OpenMP で用いられるメカニズムであり，指示文を挿入することで使える．これを使うことにより，簡単に，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．

本節では，task 構文の基本的な使い方とその挙動についてを説明する．Program 5.2 が task 構文の記述例であり，図 5.2 はその挙動イメージである．

```
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 5.2　OpenMP task 構文の基本的な使用例

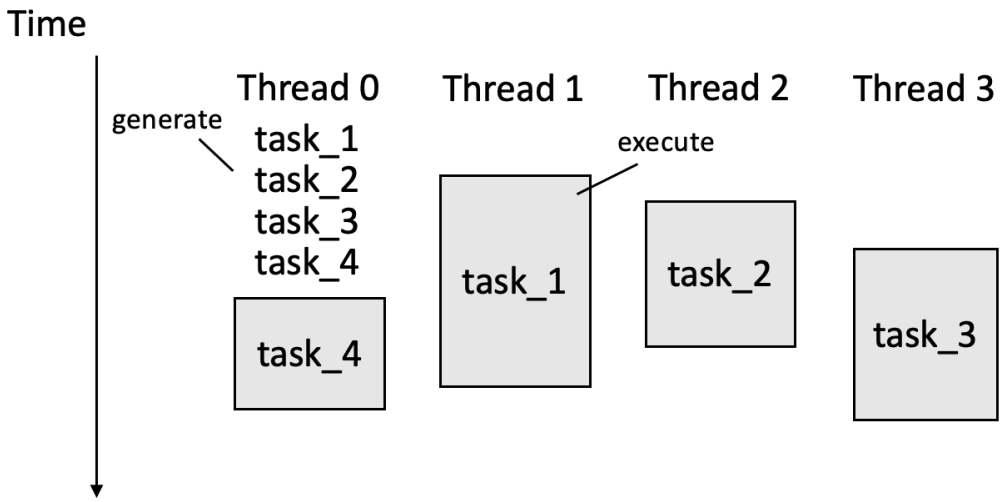


図 5.2　Program 5.2 の挙動イメージ

Program 5.2 では，まず 1 行目で`#pragma omp parallel` の指示文によって fork され，4 つのスレッドが生成される．3 行目には`#pragma omp single` が書かれている．したがってこのブロック内は単一のスレッドのみが実行することになる．このスレッドは 5 行目，7 行目，9 行目，11 行目で task 指示文を検知し，それぞれのタスクを生成して並列領域に関連づけられたタスクプールに置く．その後，タスクプールに置かれた 4 つのタスクのうちの 1 つを実行することになる．また，1 行目で生成され，待機している残り 3 つのスレッドも，タスクプールにタスクが置かれると，それらを実行する．図 5.2 で，これら一連の流れのイメージを示している．

ここで注意しなければいけないことがいくつかある．もし 1 行目`#pragma omp parallel`で生成されるスレッド数が，task 指示文の数より少なければ，あるスレッドはタスクプールに置かれた 2 つ以上のタスクを実行することになるということだ．極端な例をいうと，もし 1 行目で 2 つのスレッドしか生成されなかったとする．その場合，2 スレッドはそれぞれタスクプールから 1 つずつ取り出して実行し，それが終えたらまた取り出して実行するという処理を，タスクプールがゼロになるまで繰り返すことになる．また，もし 3 行目`#pragma omp single`がなければ，task 指示文によるタスクの生成が複数スレッドで行われることになり，予期しない挙動の要因になる．

5.2.2　task 構文の活用

ここまでで，task 構文の基本的な使い方と挙動を説明した．しかし，これだけだと依存関係のないタスクしか並列実行の恩恵がないことやスケジューリングが設定できないなどの制約があるが，In-situ 可視化・解析を実現するためには，これらが必要だと考える．実は task 構文には，データ属性，依存関係の指定やスレッド制御，そしてスケジューリング機能などの活用方法が存在する．今後の UTHelper の開発を進めるにあたって，役立つ機能と考えられるので，本節ではそれらの task 構文の活用について説明する．

まずは，指示節について説明する．記述方法としては`#pragma omp task` 指示節のように書く．task 構文に限らず OpenMP のワークシェアリング構文 (`#pragma omp for`) などにも共通するものもある．今回はその中から代表的なものを列挙する．

- private(list)
- firstprivate(list)
- shared(list)
- reduction(opetator:list)
- depend(type:list)
- priority(value)
- affinity(A[i])

private(list) を使うと，list に指定された変数がスレッドそれぞれに対してプライベートに用意することができる．例えば private(i) とすると，変数 i は各スレッドそれぞれに用意されるため，別々の値を持つことができる．これを用いることで他スレッドからの変数 i の上書きなどを防ぐことができる．

shared(list) を使うと，list に指定された変数はスレッド間で共有される，つまりは共有変数となる．private などを使わない，デフォルトの場合，各変数は共有変数となっている．

depend 節は，データ間の依存関係を作り，task の実行順序を決定することができる有用な機能である．したがって記述例とその挙動イメージを詳しく説明する．Program 5.3 は depend 節を用いた task 構文の記述例である．図 5.3 に Program 5.3 の挙動のイメージを示す.

```
1  int a, b, c;
2  #pragma omp parallel num_threads(4)
3  {
4    #pragma omp single
5    {
6      #pragma omp task depend(out:a)
7      task_1();
8      #pragma omp task depend(out:b)
9      task_2();
10     #pragma omp task depend(in:a,b) depend(out:c)
11     task_3();
12     #pragma omp task depend(in:c)
13     task_4();
14   }
15 }
```

Program 5.3　depend 節を用いた依存関係

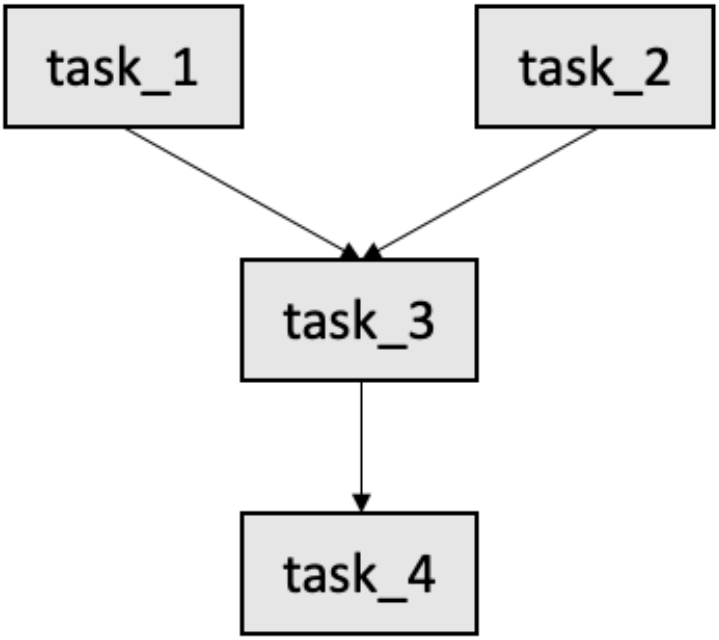


図 5.3　depend 節の挙動イメージ

depend 節は depend(type:list) で，type にタスク間の依存関係の方向を，list で依存関係

を持たせたい変数を指定する．Program 5.3 でいうと，タスク 3 は変数 a,b に依存 (in:a,b) するように指示されている．したがって，タスク 3 はタスク 1 とタスク 2 に従属する，つまりはタスク 1 とタスク 2 の実行が終わってから，タスク 3 が実行される．またタスク 4 は変数 c に依存 (in:c) する．したがって，タスク 4 はタスク 3 が終わってから実行される．

また，priority 節もタスクの実行優先順位を指定できる．具体的には，タスクプールに置かれた生成済みのタスクの中から，priority(value) の value の値が大きいものから先に実行する．

何も指示節がない場合，生成されたタスクの実行順序は不定であるが，depend 節や priority 節によってワークフローを作ることができる．これにより単純な並列化が難しいコレスキー分解などの並列実行にも応用できる．

最後に affinity 指示節である．例えば，`#pragma omp task affinity(A[i])` の場合，そのタスクはデータ A[i] に近いメモリ位置で実行されることになる．

ここまでで，task 構文に付随した指示節の機能の説明を行った．さらに OpenMP の構文の中には，task 構文と組み合わせて使うことで，スレッド制御をしたりなどの柔軟な実行ワークフローを表現することができるものがある．

次に以下に示した構文の説明を行う．

- `#pragma omp taskwait`
- `#pragma omp taskgroup`
- `#pragma omp taskyield`
- `#pragma omp flush`

1. taskwait

構文`#pragma omp taskwait` について説明する．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 5.4 に示す．また図 5.4 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いけないのは，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．なぜならば，8 行目からのタスク 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 task 構文の 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 5.4　taskwait 構文の使用例

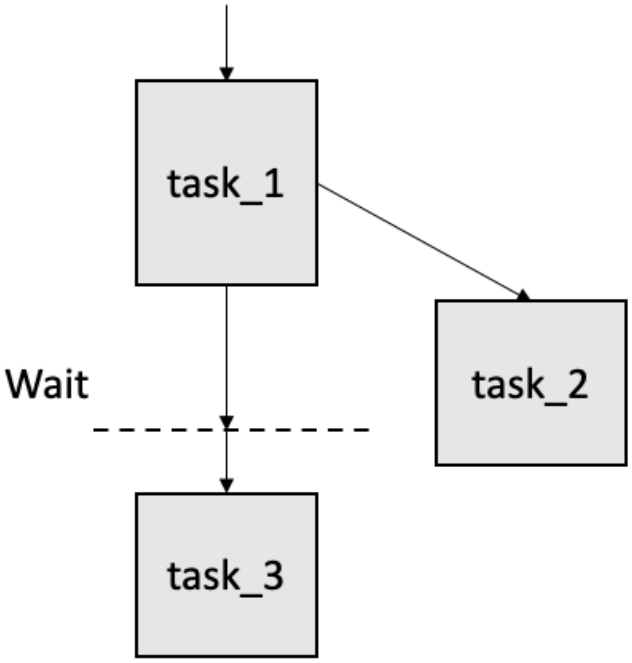


図 5.4　taskwait 構文の挙動イメージ

2. taskgroup

構文`#pragma omp taskgroup`について説明する．この構文は，taskgroup の領域内のタスクが終了するまで，先に進まないという機能を持つ．Program 5.5 に使用例を示す．また図 5.5 にその挙動イメージを示す．5 行目から taskgroup が始まり，そのブロックは 15 行目まで続いている．このプログラムの挙動を簡単に説明する．taskgroup 内のタスクが全て実行されてから，処理が先に進む．具体的にいうと，タスク 1 とタスク 2 が実行完了を待つ．その後，タスク 3 が実行される．taskwait 構文との違いは，taskgroup は，階層に関係なくブロック内の全てのタスクが終わるのを待つことである．

```
1  #pragma omp parallel
2  {
3    #pragma omp single
4    {
5      #pragma omp taskgroup
6      {
7        #pragma omp task
8        {
9          task_1();
10           #pragma omp task
11           {
12             task_2();
13           }
14       }
15     }
16     #pragma omp task
17     task_3();
18   }
19 }
```

Program 5.5　taskgroup 構文の使用例

3. taskyield

構文`#pragma omp taskyield`は，現在のタスクの実行を中断し，他のタスクの実行をするように指示する構文である．ただし，使用する際に慎重にプログラムを記述しなければ，デッドロックが発生する可能性があるため注意が必要である．

4. flush

スレッド間で，変数や配列を共有していても，値が一致しないことがある．これは，値がレジスタに保持されている状態であり，メモリに書き出していないからである．構文`#pragma omp flush`を用いることで，スレッドが参照する共有変数の値を更新することができる．`#pragma omp flush(list)`の list に明示することで，書き込む変数を指定することもできる．使用するとメモリアクセスが発生し，負荷がかかるため，頻繁な使用には注意しなければならない．

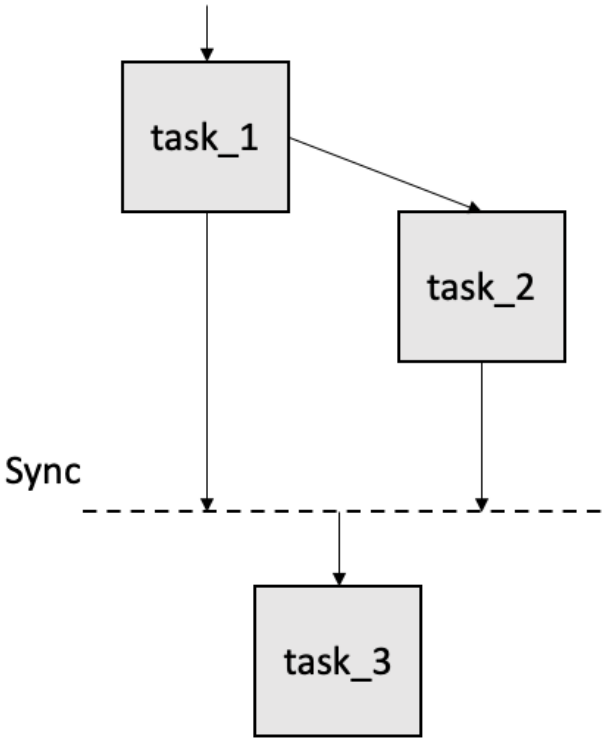


図 5.5　taskgroup 構文の挙動イメージ

5.2.3　In-situ 可視化・解析への適用

本節では，紹介した構文を場面に応じてどのように In-situ 可視化・解析に適用するかを，実際にユーザが機能を利用する流れに沿って説明する．

- In-situ 可視化・解析機能の組み込み

ユーザは利用するにあたって，まずフレームワークの In-situ 可視化・解析機能を自分の環境に移植する必要がある．task 構文で，2 つのタスクを用いることで，主計算と In-situ タスクを並列実行させることができる．次にユーザはシミュレーション実行前に，主計算結果のどのデータ，変数を In-situ タスクで読み取り，解析するかを決定しタスク間で共有できるようにする．この場合には shared 指示節や flush 構文が用いられる．また，各タイムステップ毎に In-situ タスクがメモリからデータを読む際には，メモリバンド幅によるボトルネックが発生するため，affinity 指示節で変数が保持されている場所に近い位置で In-situ タスクを実行することで，ボトルネックを小さくすることも考えられる．

- シミュレーション実行

シミュレーションの実行にあたって，主計算と In-situ タスクは，どちらか一方のループばかりが先に進んではいけないため，タイムステップ毎に同期を取らなければいけない．こ

のために用いるのが，taskwait 構文である．また場合によっては，In-situ 解析を行っている最中に，途中で終了したい，もしくは可視化処理に変更したいタイミングが現れるかもしれない．その切り替えは，実行アプリケーションとユーザの双方的なやり取りが実現できれば，taskyield 構文によって可能となる．

第6章　実アプリケーションによる評価とフレームワークの提案

今回は，“ GOTHIC ”というシミュレーションコードに対して OpenMP task 構文による並列化を施すことで性能向上を図る実験を行った．GOTHIC とは，衝突のない N 体シミュレーションを行うコードであり，階層的なタイムステップとすることで高速化を図ったものである．解析処理の分量を柔軟に設定できるなどの特徴もある．また，将来のユーザフレンドリな In-situ 可視化・解析実装に向けて，ユーザが簡単に非同期処理が行える枠組みを設計し，その動作を確認した．

6.1　実験環境

本研究での実験環境を表 6.1 に示す．本研究の全ての実験は同様の環境で行っている．

表 6.1　実験環境

システム	Wisteria/BDEC-01 Aquarius 1 ノード
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

使用ライブラリについて説明する．OpenMP については 5 章で紹介したとおりである．

NVIDIA Nsight Systems[18] 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が，視覚化用 GUI も備えているため，ユーザ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　実アプリケーションによる評価

ユーザが容易にタスク並列や In-situ 可視化・解析機能を使える状態にするためには，実際に並列化作業を行い，どう枠組みを作るべきかを考えたりや実際に性能向上するかを確認したりする必要がある．そこで今回は，既存の科学シミュレーションに対して，task 構文による並列化を施す実験を行った．

6.2.1　task 構文による GOTHIC の並列化

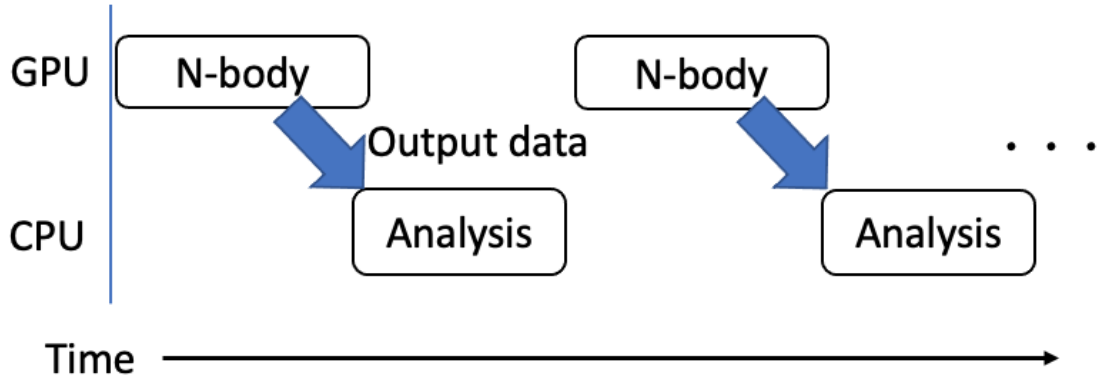
並列化を施した科学シミュレーションコードは，GOTHIC:Gravitational oct-tree code accelerated by hierarchical time step controlling[19] である．宇宙物理の N 体計算を行うコードであり，以下のような特徴がある．

- 主計算と解析が逐次処理
- 主計算は GPU で行われる

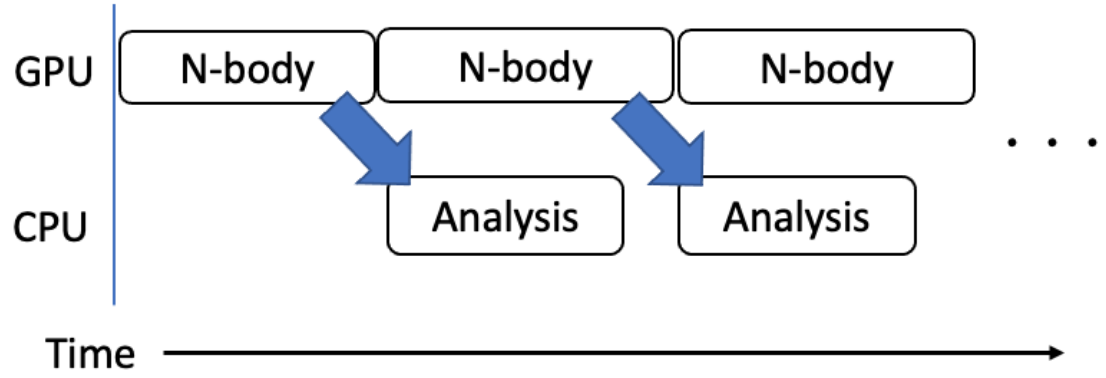
シミュレーションにおけるループの流れを説明する．初めに N 体計算が GPU 上で行われる．次に計算結果が GPU から CPU に転送される．そして最後に転送されたデータが CPU 上で解析され，終わり次第，次のタイムステップの N 体計算が始まる，という流れである．このコードの動作イメージを図 6.1a に示す．

この図から GPU 上で主計算が行われている間，CPU は休んでいることが読み取れる．余剰コアという観点でいうと，この場合は GPU 上での主計算の性能向上に，CPU は寄与しないため，コアが余っているということもできる．計算リソースの効率的な利用とは言えない．

この問題を解決するために，task 構文による主計算と解析の並列化を考える．この時の動作イメージを図 6.1b に示す．GPU 上での主計算の裏で，CPU では 1 つ前のループのデータ解析を行う．GPU，CPU 共に休む時間が減るため，計算リソースを無駄を削減できる．



(a) 元の動作イメージ



(b) 並列化後の動作イメージ

図 6.1　GOTHIC の並列化前後の動作イメージ

スレッド間の同期方法の 1 つとして，共通変数を利用したフラグを使う方法がある．今回，各タイムステップでの主計算スレッドと解析スレッドの同期にはこの方法を使った．フラグを用いた使用例を Program 6.1 に示す．タスク 2 は task_1() の実行が完了し，タスク 1 で flg が更新されるまで，while ループを実行する．flg の更新を検知したら，while ループを出て，task_2() が実行される．

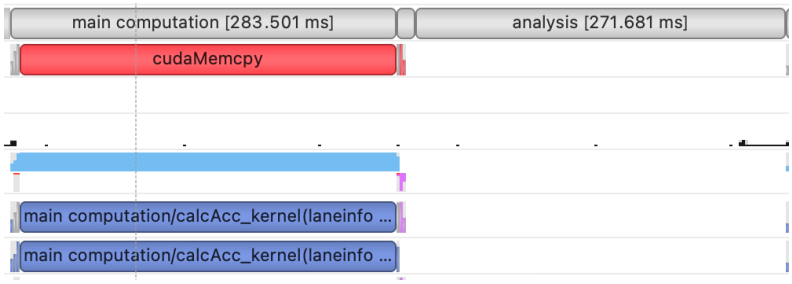
```
1  #pragma omp task
2  {
3    task_1();
4    flg=1;
5  }
6  #pragma omp task
7  {
8    while(1){
9      #pragma omp flush
10     if(flg==1){
```

```
11          break;
12        }
13      }
14      task_2();
15  }
```

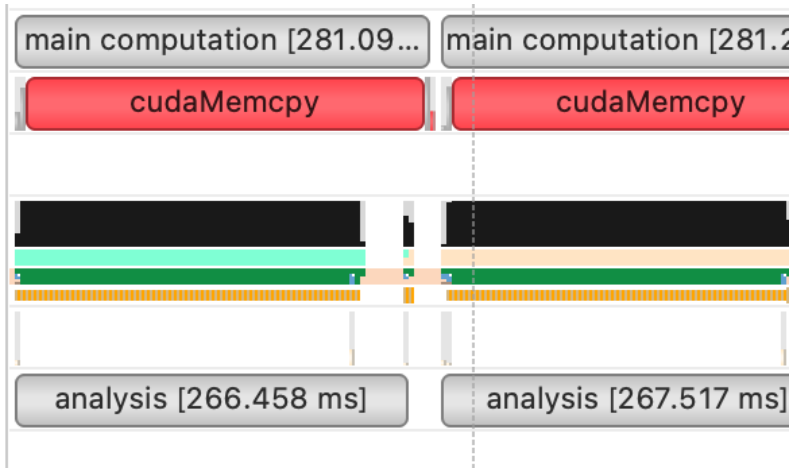
Program 6.1　flag を用いたスレッド間の同期例

6.2.2　結果

6.2.1 節で説明した方法を用いて 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．実験の結果を図 6.2 と図 6.3 に示す．



(a) 元のプログラムのトレース



(b) 並列実行のトレース

図 6.2　並列化前後の実行トレース

図 6.2a 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る．main computation が GPU での N 体計算，つまり主計算であり，analysis が CPU 上での解析処

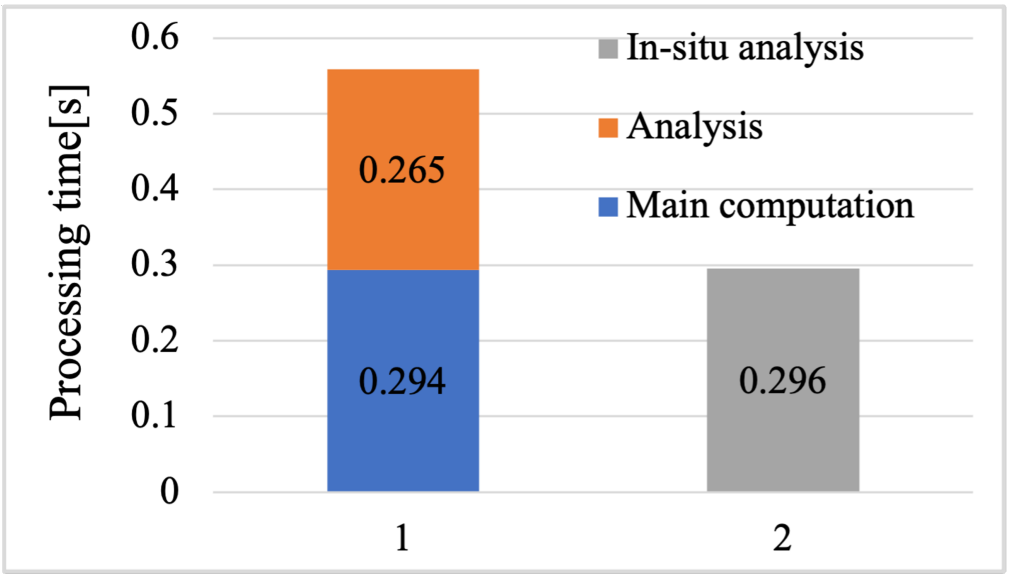


図 6.3　GOTHIC の計算ループを 5 回実行した際の平均実行時間

理を示している．図 6.1a 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．そして，task 構文による並列化後のトレース結果を図 6.2b に示す．main computation と analysis が並列実行されていることが読み取れる．

図 6.3 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．

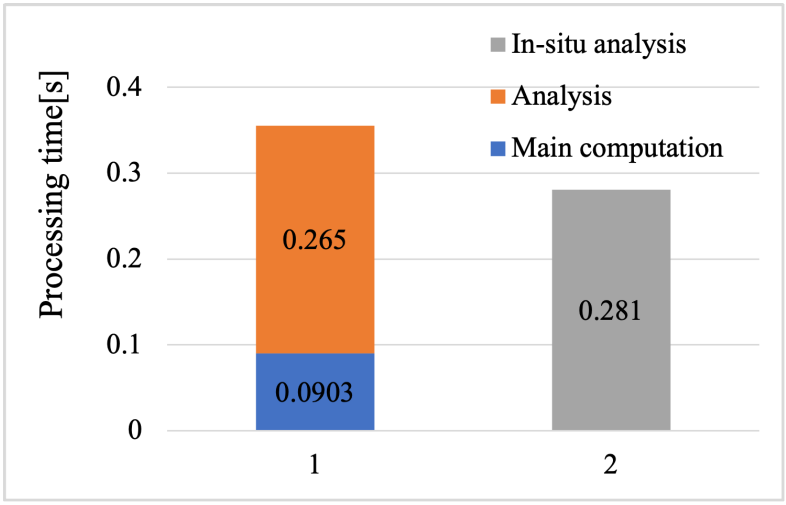
1 は元のプログラムの 1 ループあたりの実行時間を示しており，主計算と解析の合計時間は 0.549[s] となった．2 は並列化後の 1 ループあたりの実行時間を示しており，0.296[s] となった．逐次処理の場合の実行時間と比較して，およそ 54%まで短くなった．

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見られたと言える．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，科学シミュレーションによっては，主計算又は解析のどちらか一方の実行時間が他方よりも長い，もしくは短い可能性がある．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．

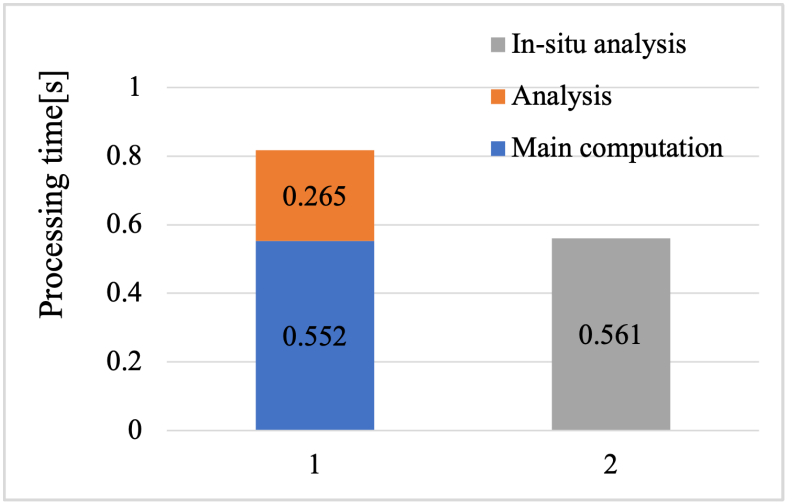
図 6.4 は，主計算と解析の処理時間に差を作った場合の，逐次処理と並列処理の実行時間を示している．図 6.4a は主計算より解析の時間を長くした場合，図 6.4b は解析より主計算の時間を長くした場合の実行時間を示している．この図から，どちらのパターンでも並列化により実行時間が短縮されていることがわかる．逐次処理の時間と比較すると，図 6.4a では 79%，6.4b では 69%程度となった．以上から，主計算と解析の処理時間が近い方が並列化がより有効となることがわかる．

また，6.4b の並列化の実行では，解析をするスレッドは解析を終えると，主計算の実行が完了するまでしばらく待機する必要があるが，while ループによる待機では，コアに高負

荷がかかる．電力や発熱について考慮すると，この待機方法は適切ではない．



(a) 解析が長い場合



(b) 主計算が長い場合

図 6.4　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

この実験で並列化作業を行った際に，タスク並列や In-situ 可視化・解析の適用する際には，アプリケーションの主要部分を，主計算部，シミュレーションデータを解析側に運ぶためのデータ転送部，そして解析部の大きく 3 つに分けて考える必要があるとわかった．主計算と解析は並列化可能である．しかし，データ転送部は，GPU と CPU 間，もしくは CPU と CPU 間などで，メモリを使ってデータをやり取りする必要があるため，並列化することはできない．

以上を踏まえて，フレームワークとしてのタスク並列，In-situ 可視化・解析実装に向け

ての枠組みを設計することを目指す.

6.3　並列実行のための枠組み設計

今回は，ユーザ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．なお，先で述べように，シミュレーションループの中に並列化可能な主計算部と解析部，そして並列化できないデータ転送部があると想定している．task 構文を用いて記述した枠組みのプログラムを Program 6.2 に示す.

5 行目が，シミュレーションループ開始位置である．7 行目から 11 行目までが主計算タスクブロックである．13 行目では，taskwait 構文が使われている．15 行目には，データ転送を想定した，並列化ができない処理を想定している．17 行目から 21 行目が解析タスクブロックである.

次に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 6.5 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，主計算，データ転送，解析の領域にある各 task_A, task_B， task_C それぞれを sleep 関数で置き換えて実行している．それぞれの sleep 時間は，20 秒，5 秒，10 秒である.

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．今回は，主計算タスクより解析タスクの実行時間が短いため，解析スレッドには，解析を終えてから主計算タスクが完了するまでの間，待機している時間がある．ここで，図中の CPU という項目は，使用されたコアに負荷がかかっているかどうかを示しており，負荷がかかっている時間帯は，黒い色を示す．ここを見ると解析スレッドが待機している間は，どのコアも白いままであり負荷がかかっていないことがわかる．したがって，その時間の電力消費，発熱については，フラグを用いた while ループでの待機の場合よりも抑えられていると推測される.

今回は sleep 関数を用いたが，科学シミュレーションコードに対して，この枠組みを適用する際は，task_A() にはシミュレーションループにおける主計算を記述する．task_B には，task_A の出力結果を task_C から参照できるようにするための処理を書く．GPU や CPU のデータ転送やスレッド間のデータ同期のための`#pragma omp flush` などが該当する．最後に，task_C には，task_B によって運ばれたデータを解析するための処理を記述する．以上が使い方の説明である.

```
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         task_A();
11       }
12
13       #pragma omp taskwait
14       // Not parallelizable
15       task_B();
16
17       #pragma omp task
18       {
19         // Analysis
20         task_C();
21       }
22     }
23     // simulation loop end
24   }
25 }
```

Program 6.2　タスク並列，In-situ 可視化・解析実装に向けた枠組み

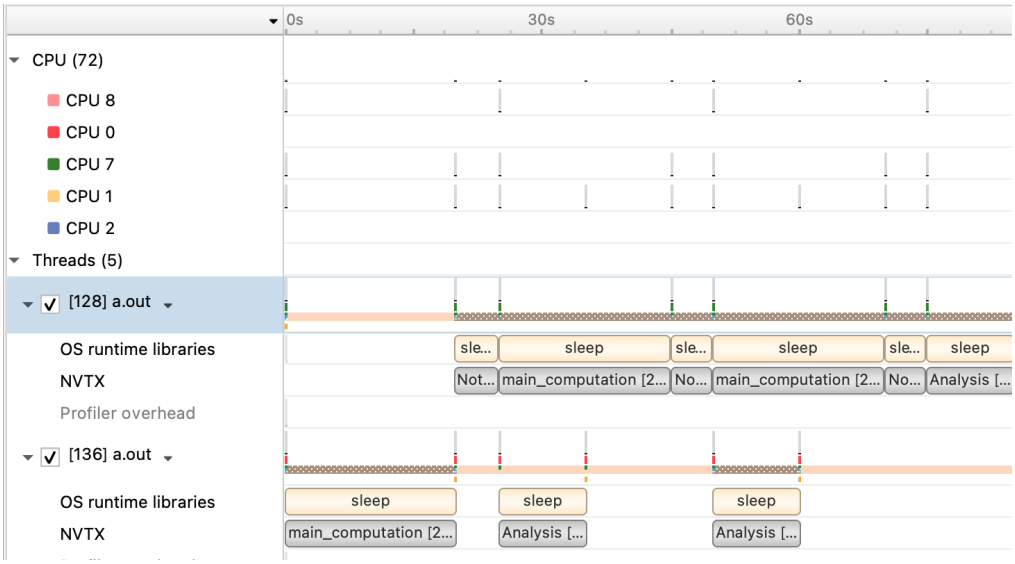


図 6.5　Program 6.2 に基づいて記述したコードのトレース結果

第7章　結論

7.1　結論

本研究は，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・解析の実現を最終的な目標としている．今回は，まず既存の N 体計算コード GOTHIC に対して，OpenMP task 構文を用いた並列化を適用することで In-situ 解析を行った．

その GOTHIC の実験では，元々は主計算と解析が逐次処理であり，計算リソースを効率的に使えていなかったため，task 構文により並列化し，In-situ 解析を行うことでそれを解消できると考えた．そして実際に並列化後の実行時間は，元のプログラムと比較して実行時間がおよそ 54%という結果が得られ，性能向上を確認できた．

この実験を通じて，スレッドがコアに負荷をかけずに待機するため方法を考える必要があることや，In-situ 可視化・解析を多くのコードに対して適用可能かつユーザが複雑さを意識せずに使える枠組みの設計にあたって，まずは科学計算コードに共通する大まかな構成を見出す必要があることを発見した．

そして，以上を踏まえて，OpenMP task 構文によるタスク並列のための枠組みを設計し，実際に動かして，その挙動をトレースした．主計算と解析の同期のためのスレッドの待機は，taskwait 構文を用いることで，頻繁なメモリアクセスやコアへの大きな負荷をかけずに実現できた．また，設計にあたって並列化可能な主計算部と解析部，そして並列化不可能なデータ転送部というふうに明示的に領域を分けたことで，将来にユーザが既存のコードに対して task 構文による並列化を行いやすい機能作りに役立てることができると考える．

7.2　今後の課題

今回は，既存のシミュレーションコード GOTHIC に task 構文による並列化を適用して In-situ 解析を行った．そして今後 In-situ 可視化・解析機能を利用しやすい形で，ユーザに提供するための出発点として，簡単なトイプログラムを書いた．

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，あらゆる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．GOTHIC や他の様々なアプリケーションに対して適用して枠組みを作り込んでいく．もしそれが可能だと判断できたら，今度はユーザが元のコードを改変しなくとも，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 20H00580 の助成を受けています.
本研究を進めるにあたり，お忙しい中，親身にご指導を賜りました塙敏博教授に心より感謝申し上げます.
発表練習に協力してくれた研究室の Tianya WU さん，Yijie YU さんに感謝を意を表します.

学会発表

1. 赤沢 龍哉，塙 敏博，三木洋平“In-situ analysis with OpenMP task for leveraging unused core.”HPC Asia 2022 (2022)
2. 第 183 回 HPC 研究会 2022 年 3 月（発表予定）

参考文献

[1] Performance development. https://www.top500.org/statistics/perfdevel/, Online; accessed 10 July 2021.

[2] Karl Rupp. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/. Original data up to the year 2010 collected and plotted by M. Horowitz, F.Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2017, Online; accessed 17 January 2022.

[3] Dawei Mu, Jon Moran, Hui Zhou, Yifeng Cui, Ronald Hawkins, Mahidhar Tatineni, and Steve Campbell. In-situ analysis and visualization of earthquake simulation. Practice and Experience in Advanced Research Computing(PEARC '19), New York, NY, USA, 2019. ACM.

[4] Ascent. https://ascent.readthedocs.io/en/latest/, Online; accessed 22 January 2022.

[5] Tong Shu, Yanfei Guo, Justin Wozniak, Xiaoning Ding, Ian Foster, and Tahsin Kurc. In-situ workflow auto-tuning via combining performance models of component applications, 2020.

[6] Jun Kudo. 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.

[7] Specifications. https://www.fujitsu.com/jp/about/businesspolicy/tech/fugaku/specifications/, Online; accessed 15 July 2021.

[8] John Shalf. The future of computing beyond moore ' s law. *Phil. Trans. R. Soc. A*, Vol. 378, , 01 2020.

[9] Xgc1. https://docs.nersc.gov/performance/case-studies/xgc1/, Online; accessed 23 January 2022.

[10] Paraview: An end-user tool for large data visualization. https://www.paraview.org/, Online; accessed 15 July 2021.

[11] Visit is an open source, interactive, scalable, visualization, animation and analysis tool. https://visit-dav.github.io/visit-website/, Online; accessed 15 July 2021.

[12] A 3d lagrangian-eulerian hydrodynamics benchmark. https://uk-mac.github.io/, Online; accessed 15 July 2021.

[13] Lammps molecular dynamics simulator. https://www.lammps.org/, Online; accessed 22 January 2022.

[14] Voro++, a 3d cell-based voronoi library. http://math.lbl.gov/voro++/, Online; accessed 22 January 2022.

[15] Heat transfer. https://github.com/CODARcode/Example-Heat_Transfer/blob/master/README.adoc, Online; accessed 22 January 2022.

[16] Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/. Online; accessed 25 January 2022.

[17] Open multi-processing. https://www.openmp.org/, Online; accessed 15 July 2021.

[18] Nvidia nsight systems. https://docs.nvidia.com/nsight-systems/, Online; accessed 23 January 2022.

[19] Yohei Miki and Masayuki Uemura. Gothic: Gravitational oct-tree code accelerated by hierarchical time step controlling. *New Astronomy*, Vol. 52, pp. 65–81, 2017.

付録 A 実験に使用したスクリプトとNsight Systemsの使用例

本研究で用いたスクリプトとプロファイラツール NVIDIA Nsight Systems の使用例を示す.

```
#!/bin/bash
#PJM -L rscgrp=share
#PJM -L gpu=1
#PJM --mpi proc=1
#PJM --omp thread=3
#PJM -L elapse=3:00:00

# module set
module load cuda/11.1

nsys profile --duration=90 --sample=cpu --trace=openmp,osrt,nvtx -
   o filename ./a.out
```

Program 7.1 6.3 節の実験で用いたシェルスクリプト

```
#include <stdio.h>
#include <omp.h>
#include "../NVTX/c/include/nvtx3/nvToolsExt.h"

int main() {
  #pragma omp parallel
  {
    #pragma omp single
    {
      while(1){
        #pragma omp task
        {
          // Main computation
          nvtxRangePushA("Main_computation");
          sleep(20);
          nvtxRangePop();
        }

        #pragma omp taskwait
        // Not parallelizable
        nvtxRangePushA("Not_parallelizable");
```

```
22          sleep(5);
23          nvtxRangePop();
24
25          #pragma omp task
26          {
27            // Analysis
28            nvtxRangePushA("Analysis");
29            sleep(10);
30            nvtxRangePop();
31          }
32        }
33      }
34    }
35  }
```

Program 7.2 6.3 節の実験で用いたスクリプト