Vol.2020-HPC-177 No.7
2020/12/21

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

工藤 純[1]　塙 敏博[2,1]

概要：近年，CPU の世代が進むに従って，内蔵されるコアの数が増加している．しかし，全てのコアを使って必ずしも最大性能が得られるわけではなく，むしろ性能が低下することもあり，最大の性能が得られる適切なコア数で実行することが重要である．その際に，使われずに余る「余剰コア」の活用を検討している．そのためには，実行時のプロファイルを行いながら，適切な並列度に変更する機能が必要になる．本研究では，実際に System Tap を用いた動的なプロファイリングと並列度変更を実現した．実際に NAS Parallel Benchmarks の FT に適用した結果，プロファイルによって得た実行クロック数の値から余剰コアの発生を確認できた．

1. はじめに

近年，CPU に内蔵されるコアの数は，世代が進むごとに増加の一途を辿っている．これはコア性能の性能向上が頭打ちになっているためであり，コア数を増加させることで全体性能の向上を図っているためである．しかしながら，全てのコアを使えばいつでも高い性能が得られるわけではなく，一部のコアをあえて使わずに残した方が全体として性能が向上する場合も多い．

そこで我々は，今後の高性能計算システムを考えたときに，このような場合に余る「余剰コア」について，主計算を支援する役割を与えることで，システム全体の性能改善や，電力効率を高めるための省電力制御，付加機能を低オーバヘッドで実現することが可能ではないかと考え，ユーザレベルでこのような支援機能の実現を検討している．

そのためにはまず，実行中の主計算を動的にプロファイリングし，また適切な並列度で実行されるよう，動的に変更できるような機構が必要である．本研究では，並列度の動的制御，動的なプロファイリングを行う機構を試作し，このような機構が実現可能であることを示す．また実ベンチマークに対して，提案するプロファイリング手法を適用することで余剰コアの存在が認識可能であることを確認し，妥当な性能分析が可能であることを示す．

2. CPU のメニーコア化と余剰コア

2.1 CPU 内蔵コア数増加の背景

最近の CPU のパフォーマンス向上はチップ内のコア数を増加させることに支えられている．図 1で示すように

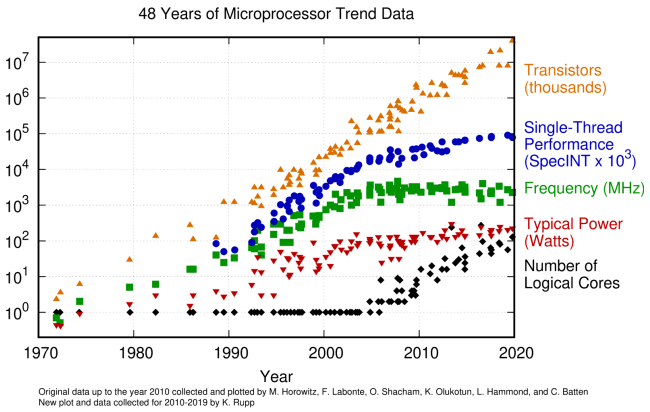

図 1　42 年間のマイクロプロセッサの傾向データ [1]

2000 年半ばまではムーアの法則に習い，1 スレッドあたりの向上が 3 年で 4 倍程度見られていた．しかし，それ以降は電力あたりの性能を最大化した結果，クロック周波数はほぼ一定のままでスレッドあたりの性能も微増となっている．一方で論理コア数が飛躍的に増加していることが読み取れる．これらが意味しているのは近年の物理コア数の増加傾向である．

CPU によっては 1 ソケットで 60 個以上の物理コアを備えている場合もあり，このような CPU は一般的なマルチコア CPU と対比して，メニーコア CPU と呼ばれる．メニーコア CPU は計算処理能力の向上に不可欠な要素となっており，一般的なマシンや HPC クラスタに多く取り入れられている．以下の表 1が HPC クラスタの物理コア数の例である．

しかし CPU のメニーコア化には、それに伴う性能問題がある．メニーコア CPU では全コアを稼働させると却ってパフォーマンスを低下させてしまうことがある．その場

1　東京大学大学院 工学系研究科
2　東京大学 情報基盤センター

表 1　メニーコアクラスタ搭載の CPU コア数

HPC クラスタ	プロセッサ	コア数
Fugaku (理研 R-CCS)	Fujitsu A64FX	52
Oakbridge-CX (東大)	Intel Xeon Cascade Lake	56
Oakforest-PACS (東大)	Intel Xeon Phi	68

合はパフォーマンス低下を防ぐために，多くのコアを敢えて稼働させない手法をとることで対処する。そのためメニーコア CPU の性能を最適に発揮するには一部のコアだけを稼働させるケースも珍しくない．そうすると稼働しないコアが発生する。今後の説明では、この意図的に稼働させないコアを「余剰コア」と呼ぶ。

また，一般的にメニーコア CPU を搭載していることが多い HPC クラスタで演算処理を行う場合はパフォーマンスチューニング等を行うことが多い．用いるノード・コア・スレッド数やアフィニティの設定等，計算処理のパフォーマンスに伴う設定を最適化することでさらなるパフォーマンス向上が図れる．

HPC クラスタ上でのアプリケーション実行における解析と最適化はヒューリスティックに行われる現状があり，またその際には多種多様なパフォーマンス分析ツールを用いる必要がある．分析や最適化の対象には電力・CPU・メモリ・ストレージ・ネットワーク・キャッシュと様々なスコープがあり，それを最適化する際のユーザの手間が大きい現状がある．そのため様々なパラメータを統一的に扱うユーザフレンドリなライブラリの需要がある．今回研究では余剰コアを活用したライブラリの実装を目指す。

2.2　利用コア数によって性能が制約される要因

コア数を過剰に割り当てた場合のプロセッサ性能低下の要因について，以下で述べる．

2.2.1　コアのクロック数低下

メニーコア CPU のパフォーマンス低下は，単一のプロセッサとして熱設計消費電力 (Thermal Design Power:TDP) を超えた動作ができないことに起因する．

近年は CPU がマルチコア化したことによって，全てのコアが同時に最大動作をした場合を考慮する必要が出てきた．そのため各コアの最大動作クロック周波数が旧来のシングルコア CPU の最大動作クロック周波数，すなわち定格クロック周波数から下げざるを得ず結果的に個々のコアの性能は低下する．

この一つのコアあたりのクロック周波数低減に対処するべく，各メーカはプロセッサにコアのクロック周波数を動的にコントロールする機能を搭載している．

- Intel 社 Turbo Boost Technology
- AMD 社 Turbo CORE Technology

その機能としては負荷が高いコアのみ動作クロックを定格クロック以上に上昇させるというものだ．基本的には高負荷状態のコア数ごとに最大値が細かく設定されている．Intel 社の CPU Xeon Platinum 8260L の例を図 2に示す．基本的には高負荷なコアの数が増えるほどクロック周波数が落ちる，Turbo Boost した際のクロック周波数が落ちる．

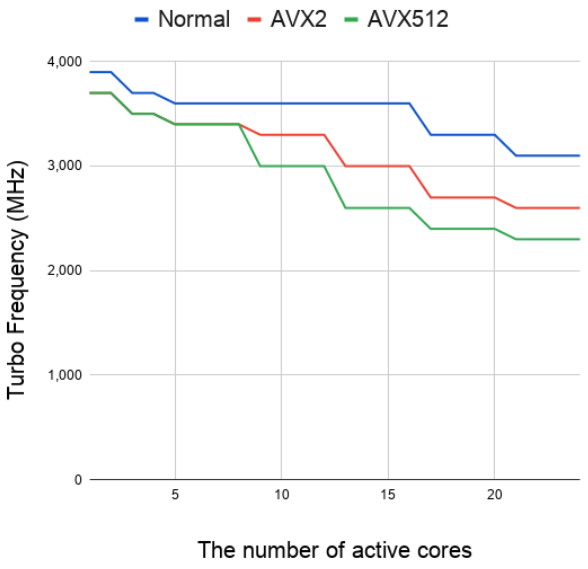

図 2　Intel 社の CPU Xeon Platinum 8260L の例におけるアクティブなコアの数とクロック周波数の関係 [2]

また，逆に CPU のパッケージ温度が上昇しすぎて閾値を超えた場合や，電力消費が激しすぎる場合・使用コア数が増えた場合等にスロットリングという機能が働き，自動的にコア自体のクロック周波数が下げられる場合がある．そのため全てのコアで高負荷な演算を行うと，結果的にコアのクロック周波数が低下しパフォーマンスの低下へとつながる場合がある．

注意してもらいたいのは，コアのクロック数低下は CPU の演算性能を最大まで発揮しようとする際に発生しやすいという点である．例えばデータセンタといった持続的にサービスを提供し続けるケースでは，常に演算をし続けているわけではないので演算性能を最大限発揮しているわけではない．すなわち熱設計消費電力問題が発生しづらい．

それに対して HPC クラスタは使い方として，大規模シミュレーションやデータ解析と高速・高負荷な計算をし続けるニーズが高い．つまり一定時間で演算性能を最大限発揮するユースケースがほとんどを占める．

また利用方法として借りるノード数と利用時間で料金が決まることから，ユーザ視点だと出来る限り短時間で演算

が終わるほうが都合が良い．そのためユーザが CPU 性能をフルに活用しようとコアをすべて稼働させることで熱設計消費電力問題が発生し，却って性能を落としてしまうケースが多く見られる．

逆に言えばコア数最大でコア単体のクロック数が低下し，プロセッサとしての性能が落ちた場合も，そのクロック数でも十分にタスク処理に余力が残っており，並列性が担保されている場合にはコア数を増やした方が良い場合も当然存在する．

2.2.2 メモリバンド幅のボトルネック

コア数をいくら増やしてもコアからのメモリアクセスがボトルネックとなることでも，パフォーマンスが向上しない・低下するケースがある．一般的にコア自体の演算性能に対してプロセッサのメモリバンド幅がボトルネックになることや，図 3から分かるようにメモリと CPU との性能差によってこの問題は発生する．

近年は CPU の性能向上が緩やかになったものの，まだこの性能差は解消されていない．そのため稼働コア数を減らした方が高いパフォーマンスを発揮することもある．いくらプログラムの並列性を担保できたとしても，メモリアクセスのボトルネックによってコア数を増やす利点が失われてしまう．

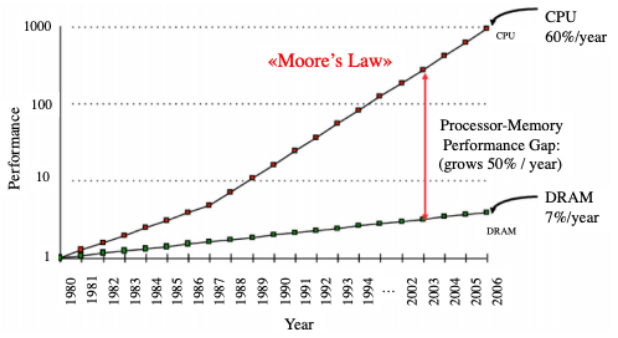

図 3 プロセッサとメモリの性能変化の傾向 [3]

2.2.3 並列性の限界

単純にプログラムに並列性に対してコアが多すぎる場合，大量のコアを使用しても効果を得ることは難しい．ただし，その際にはコアが大量に余ってしまう．またこの問題はソフトウェア側で解決されるべきもので，アプリケーションやアルゴリズムに依存する．

3. 余剰コア管理フレームワークの開発

3.1 概要

一般的なメニーコア CPU でも性能を最大限発揮するためには，余剰コアが発生する場合がある．これまで使われてこなかった休眠状態の余剰コアを適切に使用してプロファイルや性能向上を行うフレームワークの実現を目指すことが本研究の最終的な目的である．

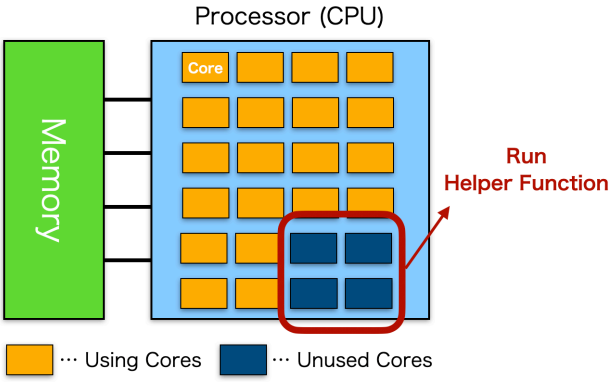

図 4 主計算のサポートを行う余剰コアのイメージ

フレームワークが動作するイメージを図 4に示す．余剰コアに主計算以外の補助的な機能を行わせる．高負荷な演算処理をおこなわけなければ，コアに対する負荷は基本的に低い．そのため低負荷の補助的な機能を余剰コアに任せることで全体のパフォーマンスを落とさずに，余剰コアを有効活用できる．フレームワークの機能としては主に以下のような機能を構想している．

- 実行中モニタリング
- コア割当の最適化
- 電力・性能バランスの向上
- In Situ プロファイリング・データ解析
- キャッシュプリフェッチに依る性能向上
- 非同期処理による通信・ファイル IO の隠蔽

今回はフレームワークの根幹の機能としてマシンの情報を取得・収集するプロファイリングの機能開発に着手した．この機能によって本当に余剰コアが発生しうるのか計測をすることができる．かつプロファイリング機能を活用することで，フレームワークリとして性能最適化の機能に発展させることができる．

3.2 関連研究

先行研究として，主計算の補助的な役割としてスレッドを用意する，ヘルパースレッドというアイデアがあり，マルチコアプロセッサが登場以後様々な形で行われている．これは，あらかじめ主計算を行うスレッドとは別スレッドを用意しておき，高速にコンテキストスイッチすることでキャッシュやページテーブルのプリフェッチ [4]・ファイル操作 [5] を実現する仕組みである．

これらは Simultaneous Multi-threading(SMT) を対象としており，余剰コアを使っているわけではなかった．SMT は Intel 社では Hyper-Theading という機能に当たる．SMT を用いた場合にはキャッシュのプリフェッチには適しているが，ヘルパースレッドと物理コアを共有する主計算スレッドでリソースの制約を受けるので性能劣化が起こりやすいと予想できる．

また，先行研究の多くは OS 等の低レベルで予めスレッドを割り当てておき，コンパイラによってキャッシュプリフェッチのためのコードを自動生成するものだった．なおかつモニタリングをヘルパースレッドに行わせる例は見られない．

3.3 フレームワークの設計に関して

最終的には自動的に動的最適化を行い，様々な付帯処理を効率よく行おうとするものを目標としている．動的最適化や付帯処理を効率よく行うためには，その時にどのようなコアがどのような処理をしているのかという情報をリアルタイムにモニタリングする必要がある．

例えば電力最適化であれば現在コアの消費している電力情報等が必要になると考えられる．そのために，本研究ではまず，余剰コアを活用した実行中モニタリングから機能を実現する．

フレームワークでは，動的に計算コア群から不要な余剰コアを切り出してヘルパースレッドとして使用したり，元通り計算に復帰させるなど，柔軟な制御を可能にしたいことから，実行段階でコアを指定することができる numactl [6] と同様のコア割り当て機能を実現する．

また動的最適化の際には Performance Application Programming Interface (PAPI) [7] [8] というパフォーマンスカウンタから値を取得できるインターフェイスを用いる．これを用いて取得した情報を解析し，解析を元に CPU 全体のコア動作を制御する．

また実際にヘルパースレッドが動作する際にはコア間のキャッシュ共有関係を元にしたスレッドの再割付が必要になる．そのためライブラリである Portable Hardware Locality (hwloc) [9] [10] などを用いてキャッシュ構成といったハードウェア情報を取得し，適切なコアに割付し直すような機構を設計・実装する．

そして動的最適化・プロファイルの実現方法として現在，SystemTap [11] [12] を主計算の監視役として余剰コアで動作させ，関数をフックする手法を検討している．以下に組み込む予定のライブラリの詳細を示す．

3.4 numactl

numactl は linux コマンドの一つで，プログラムの実行時に使用メモリ・使用コアや，メモリ配置ポリシ等を設定できる．メモリ配置ポリシとしては同一ソケット内のメモリに限る設定と，配置可能な全メモリに平均的に配置する設定があり，基本的には同一ノード内のメモリに限った配置ポリシのほうが良いパフォーマンスが出る．

これを利用することで，プログラムの実行時にコアや CPU・メモリへの割り付けが可能，実行段階で任意のリソースを稼働させることができる．実行段階でのスレッド割当最適化等に用いることを検討している．

HPC クラスタの使用時においても，numactl を利用することでプログラムの実行時に OS が稼働しているコアを，明示的に計算に使用するコアから除外しパフォーマンス向上を図るユースケースがある．

3.5 hwloc

hwloc はハードウェアの様々な情報の収集が可能である．複雑化する並列計算プラットフォームの情報を収集し，アプリケーションを支援することを目的としている．様々な OS をサポートしておりメモリバインディング API といった機能も提供しているが，主にコア数・ハードウェアスレッド数・キャッシュ構成といったプロセッサのアーキテクチャを把握するために用いることを検討している．

3.6 PAPI

PAPI は CPU に備わっているパフォーマンスカウンタから，キャッシュのヒット率や命令実行数などの情報をリアルタイムに取得することができる．様々なコンポーネントを組み込むことが可能で，他のライブラリと組み合わせることで消費電力等の取得するインターフェイスも備えることが出来る．

リアルタイムに情報を取得できる点を利用して，アプリケーションの実行パフォーマンスを動的取得への活用を検討している．そうすることで実際のコアの動作状況をリアルタイムにモニタし解析・最適化に活かすことが出来る．

3.7 SystemTap

SystemTap は実行中のシステムの情報収集をするために開発されたツールである．Linux システムの環境下で動的トレースを実現できるツールである．

3.7.1 SystemTap の特徴

通常，対象のプログラムに対してプロファイリングを行う際には，対象のプログラムそのものに手を加えなければいけないケースが多い．例えば Scalasca [13] [14] では，対象のプログラムと一緒にコンパイルするためにリコンパイルを必要とされる．一方で Intel 社の vtune [15] [16] においても，希望の区間をプロファイルするにはユーザ自身が対象のプログラムにコードを追記する必要がある．

それに対して SystemTap はユーザのプログラムに手を加えることなく動的トレースを実現できるため，コードを追記してリコンパイルといった手を加える作業が必要にならない．動的トレース対象には関数や，場合によってプログラムの特定行数の指定が可能で，専用のスクリプトをトレース対象のプログラムとは別に記述する．カーネルや関数を対象として開始時と終了時に処理をフックをすることで，特定の関数に絞った区間の正確なプロファイルが実現される．なおライブラリ関数のフックやカーネル関数に対するフックも可能である．

また SystemTap は情報収集の利用だけでなく，任意の処理を挟み込むことが可能である．例えばまたプロファイルに限らず，関数の開始時に最適化処理を挟み込むといったことができる．今回行った実験でも，OpenMP から提供されている API をフック時に呼び出すことで実行するスレッドの並列数を動的に変更している．また，別の例としてはカーネルの動的トレースもできることからコンテキストスイッチや I/O の監視といったことに活かすことができる．またフック対象の関数においてローカル変数やグローバル変数を書き換えるといったことも可能なので他にも多種多様な活用方法が考えられる．

当初はカーネルの情報を収集するために開発されたが，現在はユーザ空間のアプリケーションのトレースにも対応している．またツールを用いた際に発生するオーバーヘッドに関しては年々アップデートを重ね改善が行われている．

3.7.2 SystemTap の使用例

SystemTap では専用のスクリプトを記述することでフック対象を指定することができる．ただ専用のスクリプトに C 言語の記述を埋め込むことが可能となっている．以下で対象のプログラムの関数 FUNC_NAME が呼ばれた際に，OpenMP の API を使って並列数を変更する記述例について示す．%{ %} で囲うことで C のコードを埋め込むことができる．また対象のプログラムと関数名を明記することでフック対象を指定する．

```
%{
#include <omp.h>
}%

function _set_omp(num) %{
  omp_set_dynamic(0);
  omp_set_num_threads(STAP_ARG_num);
%}

\\ フックの定義部分
probe process("PROGRAM_PATH").function("
    FUNC_NAME").call{
  _set_omp(20);
}
```

4. 予備実験

フレームワークを設計していくにあたって，二つの実験を行った．1 つめの実験では，実際に関数のフックを行い，プロファイリングのオーバーヘッドを計測し，実行時パフォーマンスへの影響を確認した．また 2 つめの実験では，実際にプロファイリングを行い，その結果が妥当なものかどうかを確認するための実験を行った．

4.1 実験環境

実験環境を表 2に示す．また両実験で用いたライブラリのバージョンは以下の通りである．

表 2 実験で用いたハードウェア環境

CPU	Intel(R) Xeon(R) Platinum 8260L
コア数	24 x 2
基本動作周波数	2.4GHz
メモリ	DDR4-2933 16GB x 12 (192GB)
OS	CentOS v7.7 + Linux Kernel v5.5

- SystemTap v4.0
- PAPI v5.2

4.2 SystemTap 使用時のオーバーヘッド

プログラムに対して SystemTap を用いたフックを行い，そのオーバーヘッドを計測した．その際には Sustainable Memory Bandwidth in High Performance Computers (STREAM) [17] [18] のベンチマークを用いた．

STREAM はメモリバンド幅を計測するベンチマークツールである．プロセッサのキャッシュの効果を反映させないように，非常に大きな配列間のロード・ストアの性能を測定する．つまり持続可能なメモリの帯域幅を測定するベンチマークである．巨大なサイズの配列をメモリから CPU にデータを読み込むことと，CPU からメモリに書き込むことによってメモリバンド幅を計測している．

実験では GNU Compiler Collection v9.3.0 を使用した．ベンチマークの実行関数に対して，PAPI でその関数区間のパフォーマンスカウンタの値を取得するといった動作を行う．それを 1 万回連続で実行させた．その終了までにかかった時間を計測し，プログラムでオーバーヘッドを調査した．

その際に，PAPI の処理を追記したパターン二通りと，何もせずパターン一通りで，以下の計三通りのパターンで実験を行った．それぞれのパターンで 10 回ずつ繰り返し平均と標準偏差を算出した．なお，この実験に限り使用するソケット数を一つに制限している．

(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 でフックを行う

以上の内容で行った実験結果を 3と図 5 で示す．

表 3 それぞれの実行パターンによる実行時間の平均・標準偏差

何も行わない	93.3589366 ± 0.1410698721 sec
プログラムに直接追記	99.4300237 ± 0.1272547429 sec
フックを実行	99.5868690 ± 0.2165288874 sec

何も行わない場合に対して，残りの二つの実行時間からオーバーヘッドが読み取れる．ただしプログラムに直接処理を追記して動作させた場合と，関数のフックによって処理を行った場合で実行時間にあまり差がないことから，これは SystemTap によるフックではないことが判断できる．

ここで顕著に現れているオーバーヘッドは PAPI を用い

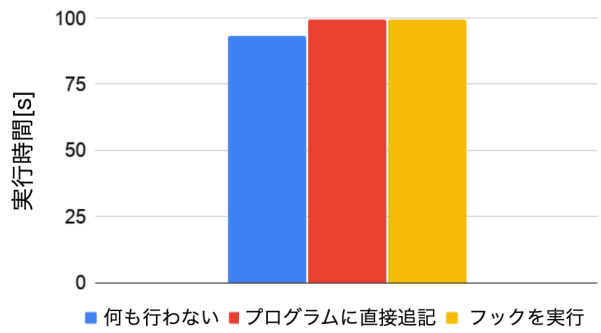

図 5 それぞれのパターンによる STREAM を実行した時間平均値

た際によるものである．一方でプログラムに直接記述した場合と関数のフックの場合の差が実際の SystemTap のフックのオーバーヘッドを示している．

以上から SystemTap にフックを行う手法そのものに関してはプログラムに直接処理を追記した場合とほぼ変わらず，オーバーヘッドはかなり低いものだと評価できる．

4.3 SystemTap によるプロファイル

SystemTap を利用した関数のフックによるプロファイルで妥当な結果が得られるか実験を行った．実験には NAS Parallel Benchmarks [19] [20] を利用した．

このベンチマークは高度に並列化されている高性能計算機の性能評価を目的にしたものである．様々な種類のベンチマークを内包しており，今回はその中の一つである高速フーリエ変換 (FFT) を用いて三次元偏微分方程式を解くベンチマークである FT を実験に用いた．また NAS Parallel Benchmarks にはクラスという設定があり，このクラスを選択することで，予めクラスごとに定められている問題サイズや反復回数でベンチマークを実行できる．今回はそのクラスの中で比較的大きい問題サイズであるクラス D を用いた．

ベンチマークプログラムのコンパイルには Intel C++ Compiler v19.0.5.281 を使用し，コンパイルオプションにて -O3 -xCORE-AVX512 を指定した．NAS Parallel Benchmarks v3.4.1 の FT を実行し，各コアごとのパフォーマンスカウンタの値を PAPI で取得を行った．主に下記のようなカウンタ値を取得した．

- 各レベルのキャッシュミス率
- 実行クロック数
- 参照クロック数 (基準のクロック周波数でカウンタが回った数)

以上の試行を主計算に使用するコア数を 2 から 48 まで 2 ずつ変化させて実行し，パフォーマンスカウンタの値変化を観察した．並列数は SystemTap のフックによって設定し、使用するコアは numactl によって設定した．

なおコア数・スレッド数 48 の場合は全コアが動作しているためヘルパースレッドが余剰コアで動作していない．それ以外のケースでは主計算で使用していないコアにヘルパースレッドを割り当てている．

また，今回のハードウェアは二つソケットがあるため，それぞれのソケットで主計算に使用するコア数を同一の数にしている．例えば合計スレッド数 40 で実行する場合はそれぞれのソケットに対して 20 のスレッドが割り当てられる．

上記の内容で行った実験結果を以下に示す．まず各コアごとの NAS Parallel Benchmarks の FT の結果が図 6と図 7 である．まずこの結果からコア数最大ではなくコア数 42 が一番良いパフォーマンスを出していることが分かる．この事からコアが余ることがわかる．

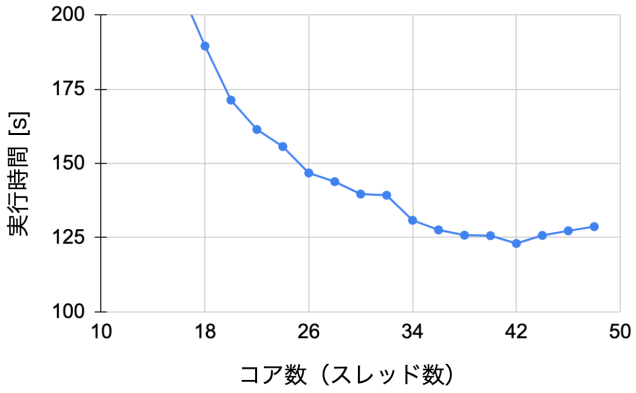

図 6 NAS Parallel Bechmarks FT (クラス D) におけるコア数ごとの実行時間

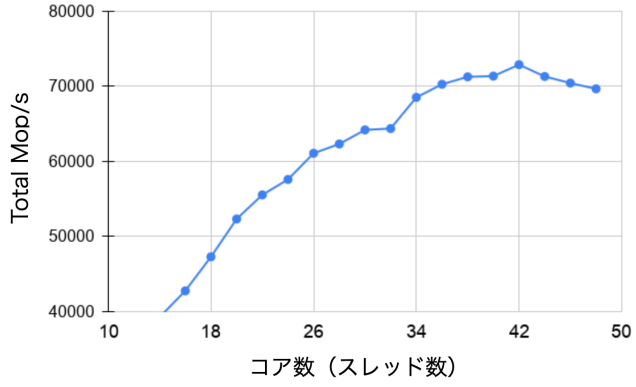

図 7 NAS Parallel Bechmarks FT (クラス D) におけるコア数ごとの Total Mop/s

次に PAPI からフックによって得た実行クロック数を図 8に示す．カウンタ値を示す際は各実験結果におけるそれぞれのコアのカウンタ値のうち最大値を選択して表示している．全コアの中の最大値を用いることによってボトルネックとなっているコアの情報を抽出できる．この図からも分かるように，実際 PAPI から得られた実行クロック数

を見ることでもコア数 42 のときの値が最小になっている．これはベンチマークの出力結果の傾向とも一致しており妥当な結果が得られていると言える．

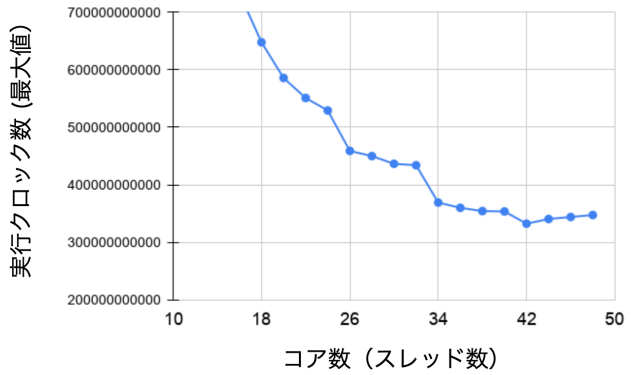

図 8 NAS Parallel Bechmarks FT (クラス D) におけるコア数ごとの実行クロック数の最大値

図 6で示している実行時間や図 7の示す Total Mop/s の傾向と同様に PAPI によるフックの計測でも全コアにおける実行クロック数の最大値が最小となったのは並列数 42 のケースで，ベンチマークの出力と合致していることが確認できる．

また，コア毎の Level 3 キャッシュミス数を取得した．それぞれの実行コア数で，最大値算出した結果が図 9である．傾向を見てみるとコア数 40 42 でキャッシュミス数が最低値になっていることがわかるが，この原因については現在調査中である．

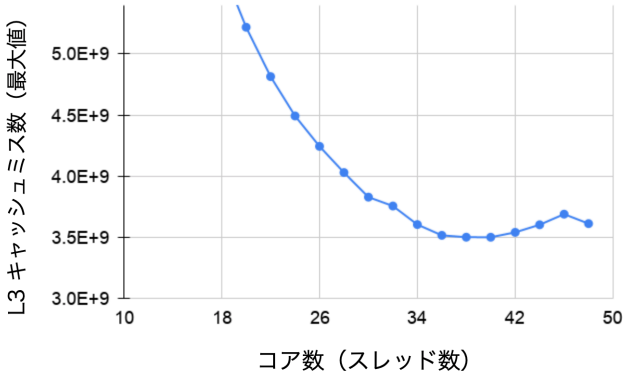

図 9 それぞれのコア数における L3 キャッシュミス数の最大値

取得したカウンタ値のうち，実行クロック数を参照クロック数で割った結果の最大値を図 10で示す．グラフの傾向を読み取ると，コア数が増えるほどに実行クロック数と参照クロックの比の値が小さくなっていることが分かる．また図 2 における AVX512 のパターンの形と類似しており，ターボブースト機能による性能変化に正しく追従して検知できていると言える．

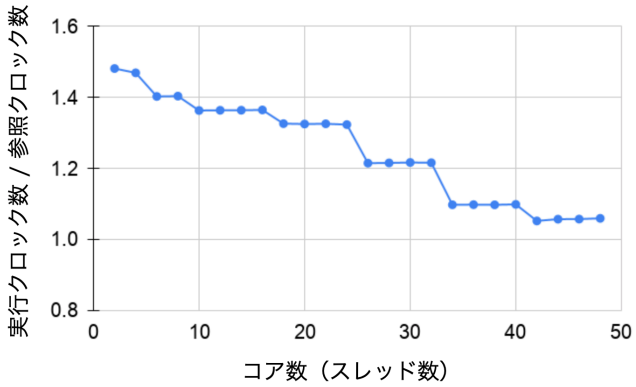

図 10 それぞれのコア数における実行クロック数と参照クロック数の比率

5. おわりに

今回は余剰コア管理フレームワークの開発に向けて，余剰コア上で SystemTap を動作させることで主計算の動作を監視し，フックを行わせることで実行中プロファイリングと並列数の変更を実現した．そして今回の手法がパフォーマンスに及ぼす影響についてオーバーヘッドを評価し，フックから得られたプロファイリング結果の妥当性を示した．

現状の課題としては，OpenMP におけるフックによる並列数変更を既に実現させている一方で，フックによるコアの割付設定の変更が実現できていない．これは OpenMP の仕様が関連しており，コアの割付設定ポリシなどは起動時に環境変数を加味して確定され，かつそのポリシを書き換えるための API が現状提供されていないためである．

今後は，プロファイルによって様々な環境でボトルネックを検出できるように，より有意な情報が取得・抽出しやすいようにより強化をすすめ，プロファイラとしての機能を完成させる．また一方でプロファイルだけでなく特にフック時のコア割付の再設定や MPI への対応といった実行最適化の機能開発をすすめていく．

謝辞 本研究の一部は，JSPS 科研費 20H00580 の助成を受けたものです．

参考文献

[1] Karl Rupp. 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. `https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`. Online; accessed 04 July 2020.

[2] Xeon platinum 8260l - intel - wikichip. `https://en.wikichip.org/wiki/intel/xeon_platinum/8260l`. Online; accessed 04 July 2020.

[3] Concezio Bozzi, Sébastien Ponce, and Stefan Roiser. The core software framework for the lhcb upgrade. *EPJ Web of Conferences*, Vol. 214, p. 05040, 01 2019.

[4] D. Kim, S. S. . Liao, P. H. Wang, J. del Cuvillo, X. Tian,

X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 27–38, 2004.

[5] Benoît Dinechin, Pierre Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, Vol. 18, pp. 1654–1663, 12 2013.

[6] numactl(8) - linux man page. `https://linux.die.net/man/8/numactl`. Online; accessed 04 July 2020.

[7] Performance application programming interface. `https://icl.utk.edu/papi/`. Online; accessed 05 July 2020.

[8] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In Matthias S.M¥"uller, Michael M. Resch, Alexander Schulz, Wolfgang E. Nagel（編）, *Tools for High Performance Computing 2009*, pp. 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[9] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 180–186, Feb 2010.

[10] Portable Hardware Locality (hwloc). `https://www.open-mpi.org/projects/hwloc/`. Online; accessed 04 July 2020.

[11] Systemtap. `https://sourceware.org/systemtap/`. Online; accessed 08 November 2020.

[12] Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. Citeseer, 2005.

[13] Scalasca. `https://www.scalasca.org/`, Online; accessed 18 November 2020.

[14] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, Vol. 22, No. 6, pp. 702–719, 2010.

[15] インテル VTune プロファイラー. `https://www.xlsoft.com/jp/products/intel/vtune/index.html`, Online; accessed 18 November 2020.

[16] James Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.

[17] Sustainable Memory Bandwidth in High Performance Computers. `http://www.streambench.org/`. Online; accessed 04 July 2020.

[18] John D McCalpin. Sustainable memory bandwidth in current high performance computers. *Silicon Graphics Inc*, 1995.

[19] NAS Parallel Benchmarks. `https://www.nas.nasa.gov/publications/npb.html`, Online; accessed 18 November 2020.

[20] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, Vol. 5, No. 3, pp. 63–73, 1991.