jh240072

メニーコア CPU，GPU の最適なリソース割り当てに関する研究

河合 直聡（東北大学）

概要

本研究では、メニーコア CPU および GPU のリソースを、アプリケーションおよび使用するシステムの特性に併せて適切に割り当てるソフトウェアの開発を目的とする。アプリケーションの並列化では並列単位での負荷の均衡化が並列性能を得るために必須となるが、場合によってはこれが困難となる。また、システムによっては並列化時に全リソースを使用せず、一部だけを使うことで、消費電力だけでなく、計算時間も削減できる場合がある。さらに、使わないリソースをノード間通信やファイル IO に割り当てれば、システムの利用効率を最大化できる。本研究では、このようなアプリケーションやシステムの特性を加味した、最適なリソース割り当てを提供する基盤ソフトウェアを開発、アプリケーションのさらなる高性能化、低消費電力化を実現する。

1. 共同研究に関する情報

(1) 共同利用・共同研究を実施している拠点名

東北大学 サイバーサイエンスセンター

東京大学 情報基盤センター

名古屋大学 情報基盤センター

京都大学 学術情報メディアセンター

九州大学 情報基盤研究開発センター

mdx

(2) 課題分野

大規模計算科学課題分野

(3) 参加研究者一覧と役割分担

・河合 直聡（課題代表者、DCB 研究、DCB・UT-Helper 統合）

・塙 敏博（課題副代表者、UT-Helper 研究）

・伊田 明弘（アプリケーション提供、評価）

・星野 哲也（DCB の GPU 対応）

・大島 聡（DCB の GPU 対応）

・三木 洋平（アプリケーション提供、評価）

・椋木 大地（DCB の GPU 対応）

2. 研究の目的と意義

本研究では、メニーコア CPU および GPU リソースを、アプリケーションおよび使用するシステムの特性に併せて適切に割り当てる基盤ソフトウェアの開発を目的とする。並列化されたアプリケーションの性能阻害要因は多数あり、並列単位での負荷の不均衡や、アプリケーションの特性によって不足するハードウェアリソースなどが挙げられる。現在の大規模クラスタでの一般的な MPI/OpenMP による並列化では、プロセスレベルおよびスレッドレベルの両方で負荷の均一化が並列性能を得るために必須となる。しかし、アプリケーションによってはこの要件を満たすのが困難な場合がある。また、メモリバンド幅律速なアプリケーションでは、使用するコア数をある一定以上に増やしても、メモリの 性能上限から、性能向上が得られず、電力の浪費に繋がっている。このような

背景から、アプリケーションのプロセス毎の負荷やシステムの特性を加味し、最適なリソース割り当てを提供する基盤ソフトウェアが必要となっている。本研究では動的なコア割付を実現する Dynamic Core Binding(DCB)ライブラリおよび最適なリソース割り当てを提供し、また余剰リソース上に Helper スレッドを生成、バックグランドでファイル IO やノード間通信を実行する UT-Helpe を併用した基盤ソフトウェアを開発する。また、基盤ソフトウェアの開発により、既存 のアプリケーションの性能を改善するだけ でなく、消費電力も同時に削減し、システム 利用効率の最大化を図る。

2024 年度では DCB+UT-Helper を併用した基盤ソフトウェアの構築および簡易評価を目標としていた。

3. 当拠点の公募型共同研究として実施した意義

本研究でのソフトウェア開発にあたっては、様々なアーキテクチャ、システム、構成での検証が必要不可欠である。JHPCN 制度では、各大学センターの計算機が利用可 能であり、本研究の遂行に適していた。ま た、本研究で得られた知見、開発したソフトウェアの公開は、今後のスーパーコンピュータの運用効率化に資するものである。 近年のスーパーコンピュータでは、世界的な電力事情の逼迫などを背景に、アプリケーション実行に必要な電力量削減が重要視されている。一般的にアプリケーションの高速化は計算時間の短縮によって電力量削減に寄与するが、それ以外の手法での電力量削減も試行錯誤されている。本研究で提案する消費電力削減手法は簡易なアイデアに基づいており、また特殊な権限等も必要ないため、直ちに様々なシステムで実用化しやすく、HPCI で提供されている計算資源の全てを効率化する物である。

4. 前年度までに得られた研究成果の概要

前年度まででは、DCB の改良、評価を行うとともに、UT-Helper との連携準備、DCB の GPU 対応方法検討を実施した。

昨年度のプロジェクト開始前には DCB の基本的な実装が完了しており、計算時間を短縮するためのポリシー(RC ポリシー)と消費電力を削減するためのポリシー(PA ポリシー)の評価が完了していた。昨年度はさらに量ポリシーを併用した Hybrid ポリシーを提案、評価した。DCB での動的負荷分散の効果は、ノード内に限定される問題があるが、Hybrid ポリシーはこの問題を消費電力削減の方向で解消したポリシーとなる。

DCB では MPI/OpenMP で並列化されているが、負荷の不均衡があるアプリケーションを対象としている。RC ポリシーでは、プロセス毎の負荷の不均衡に基づいて、スレッド(コア)毎の負荷が均一になるように、同一ノードに割り当てられたプロセスに異なる数のコアを割り当てる。これにより、負荷の大きなプロセスには多くのコアが、小さなプロセスには少ないコアが割り当てられ、コアレベルで負荷が均一化、計算時間短縮が可能となる。また、PA ポリシーでは、負荷の大きなプロセスに割り当てるコア数はそのままに、それ以外のプロセスに割り当てるコア数を削減、同様にコアレベルで負荷を均一化する。RC ポリシーではすべてのコアを使っているが、PA ポリシーでは負荷の小さなプロセスに割り当てるコア数を減らす形で使わないコアを発生させるため、消費電力が小さくなり、アプリケーションの実行時間を延ばすことなく、電力量削減を実現できる。DCB の実装完了時の評価ではこれらのポリシーの期待通りの効果が確認できていた。一方で、RC ポリシーの問題点としてノード内の負

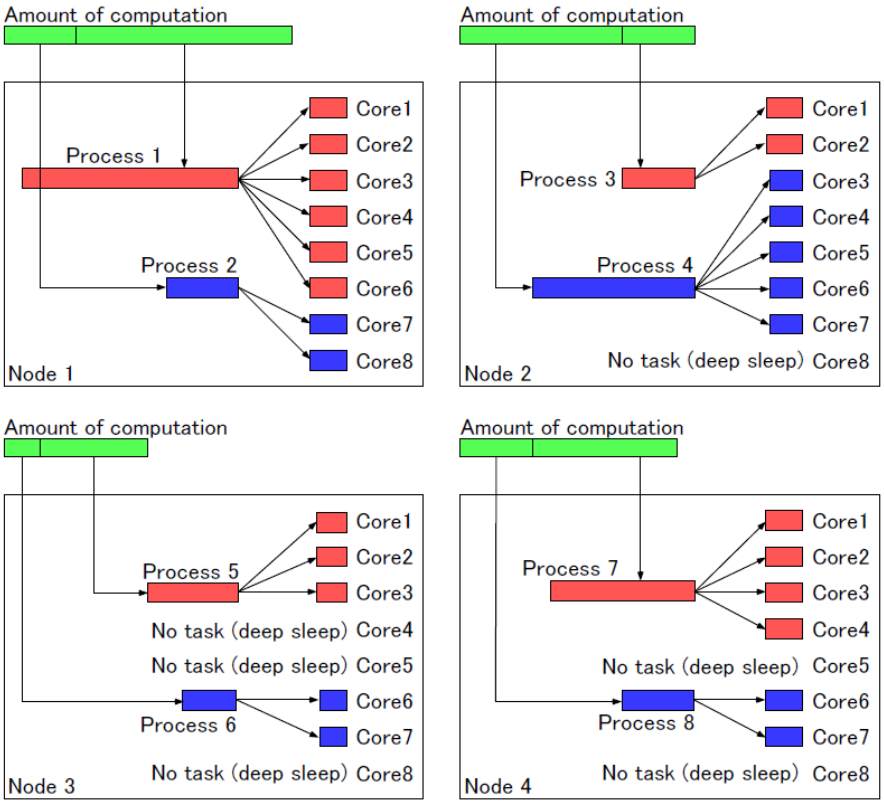


図 1 Hybrid ポリシー使用時のコア割り付け例

荷の不均衡改善しか対応できず、ノード間の負荷の均一化には異なる対応が必要となっていた。本年度に提案した Hybrid なポリシーでは RC ポリシーの問題点を、PA ポリシーとの併用による計算時間と消費電力削減を同時に実現する形で解決している。RC ポリシーに対して PA ポリシーではノード毎の負荷の不均衡は問題にならず、単純に最も負荷の大きいプロセスに合わせてそれ以外のプロセスに割り当てるプロセスにコア数を減らせばよい。Hybrid なポリシーではこの点を利用し、図 1 に示すように負荷の大きいノードでは RC ポリシーを使用して計算時間短縮に努め、負荷の小さいノードでは PA ポリシーを適用して消費電力削減、計算時間削減効果以上の電力量削減を達成している。図 2 は東京大学の Wisteria/BDEC-01 Odyssey(WO)で、図 3 は京都大学の Camphor3(C3)での評価結果を示している。評価に使ったアプリケーションは階層行列とベクトルの積であり、プロセス毎の負荷の不均衡は、演算量換算で最大 10 倍以上となっている。いずれの図でも Hybrid ポリシーの計算時間、電力量削減効果が確認できる。WO での評価では Hybrid ポリシーの計算時間は RC ポリシーと同程度、電力量は PA ポリシー以下となっており、期待通りの結果である。Hybrid ポリシーの電力量削減効果は、RC ポリシーの適用による計算時間削減と PA ポリシー適用による消費電力削減の両効果が表れるため、PA ポリシーと同程度かそれ以上と期待でき、実際に 4,096 プロセス(1,024 ノード)で 60%の削減を達成している。C3 での評価は Hybrid の効果だけでなく、PA ポリシーの適用による計算時間削減(64 プロセス(8 ノード)で約 30%)が目立っている。これは PA ポリシーの適用で全体の使用コア

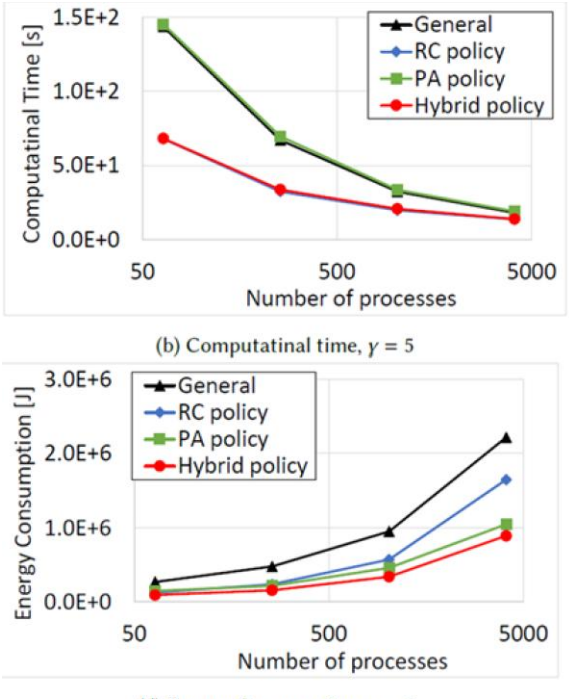
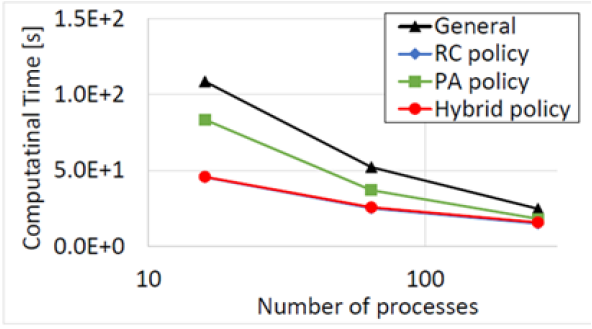
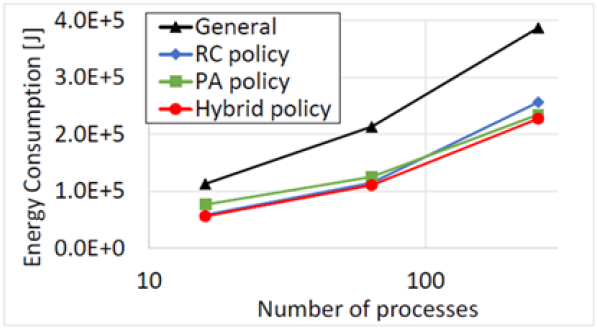


図 2 Wisteria/BDEC-01 Odyssey での Hybrid ポリシーの効果



(b) Computatinal time, $\gamma = 5$



(d) Energy Consumption, $\gamma = 5$

図 3 Camphor3 での Hybrid ポリシーの効果

数が 4 割程度にまで削減された効果であり、CPU のベースクロック 1.8～1.9GHz に対して PA ポリシー適用時は 3.2～3.5GHz と Turbo Boost が働き、負荷の大きなプロセスの計算時間が短縮されたためである。本結果から、DCB の効果を確認するとともに、DCB でアプリケーションの性能を最大化するためには、演算量などの単純なパラメータだけでなく、より高度なコア割り付け決定手法が必要と判明した。2024 年度にはこの点への対応についても対応を進めている。

昨年度の成果である DCB+UT-Helper 連携プラットフォームの開発や DCB の GPU 対応のための検討に関しては、今年度実際に進めているため、次節で述べる。

5. 今年度の研究成果の詳細

今年度は DCB の GPU 対応とともに、本研究の目的であるシステムの利用効率を最大化するための、DCB と UT-Helper を統合した基盤ソフトエアの実装を主に行った。実装した基盤ソフトエアでは以下の機能を有する、あるいは有する予定であり、それぞれの進捗について述べる。

① DCB の動的な CPU または GPU リソース割り付けによるアプリケーションの性能、実行効率改善

② DCB の動作の結果発生する余剰コアへの Helper スレッドの生成（UT-Helper）

③ DCB で割り付けるハードウェアリソースを動的に決めるための自動チューニング（AT）機構（実装中）

④ OpenMP Tools の利用による DCB ライブラリ呼び出しのブラックボックス化（実装予定）

順に実装内容について述べる。①に関して、DCB の CPU での評価は昨年度までに実施できているため、本年度の主な成果は GPU 対応となる。プロセス毎に割り付ける CPU リソースを変えることで、コアレベルで負荷を均一化させる考え方は GPU コンピューティングにも応用可能である。DCB の GPU 対応では MPI/OpenMP/OpenACC の 3 段階で並列化されたアプリケーションを想定する。ここで、OpenACC は各 GPU での演算を、OpenMP は GPU 毎の演算範囲を指定するために使用する。MPI/OpenACC での並列化は広く利用されているが、MPI/OpenMP/OpenACC の 3 段階並列はほとんど用いられない。しかし、リソース割り付けを変更可能なのは OpenMP のレイヤーのみであり、MPI/OpenACC に OpenMP を挿入するコストはさほど大きくない（kernels ディレクティブの外側に挿入）と想定し、3 段階並列化を前提としている。プロセス毎の異なる GPU リソース量の割り付けとしては、プロセス毎に異なる数の GPU を割り当てるか、単一 GPU に複数プロセスが演算をオフロードする方法が考えられる。一般的にノード当たりの GPU 数は 8 以下であり、CPU 数に比べると少ないため、プロセス毎に異なる数の GPU を割り当てる方法は不適である。ここで、MIG を

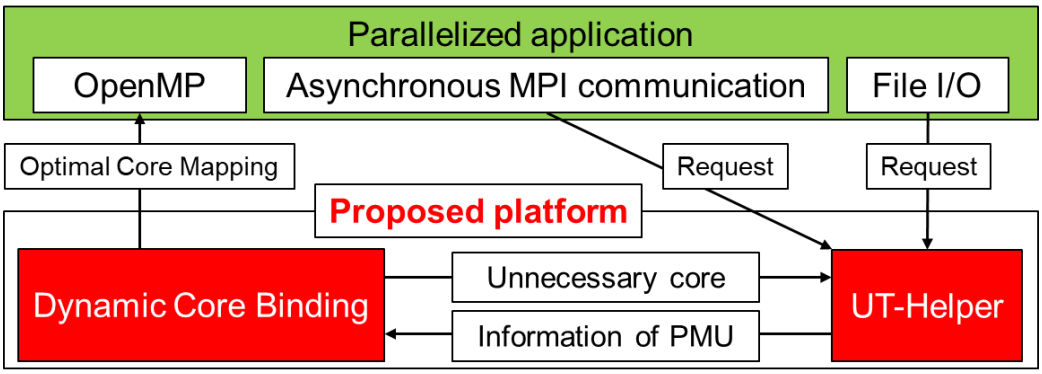
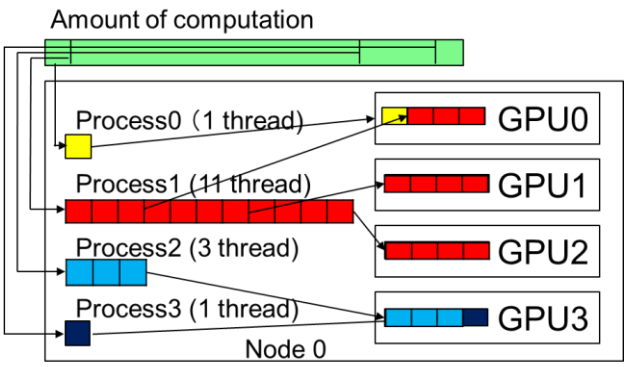


図 4 提案プラットフォーム 概念図

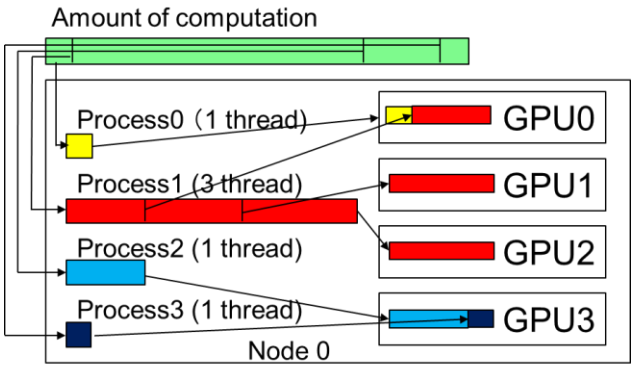
使用して GPU を 7 等分し、見かけ上の GPU 数を増やす方法も検討した。OpenMP ディレクティブでのスレッド数を各プロセスに割る当てる GPU 数と同じにし、スレッド事に ACC_SET_DEVICE_NUM()を呼び出し、スレッドが異なる MIG インスタンスに計算をオフロードする。この方法で問題となったのは CUDA ランタイムライブラリの制限であり、MIG 使用時はプロセス毎の MIG インスタンスは 1 つと限定されていた。結果、プロセス毎に異なる GPU リソースを割り当てる方法は断念した。

1 つの GPU に複数プロセスが演算をオフロードする方法では 2 つの実装を用意した(図 5)。1 つは 負荷が最小のプロセスに合わせてスレッドを生成、複数のスレッドが同一の GPU に演算をオフロードする実装(図 5, (a))、もう 1 つはプロセスが演算をオフロードする GPU 数と同じだけのスレッドを生成する実装(図 5, (b))である。DCB の実装概念から、スレッド毎の負荷は均一が前提となっており、図 1、(a)の方が好ましい。しかし、多数のスレッドで単一 GPU へのオフロードを行った場合のオーバーヘッドが懸念点として出てくるため、図 1、(b)の実装を別途用意した。図 1、(b)の実装では DCB ライブラリからの返り値に基づいてスレッド毎に異なる演算量を割り当てる実装が必要となる。このような余分な実装が必要となるが、1 つの GPU に対して 1 つのプロセスからオフロードされる演算も 1 つと限定でき、オーバーヘッドを最小化できる。本年度は実装及び簡単なコードでの実行が確認できており、現在は CPU 時同様の H 行列ベクトル積での評価を行っている。

②の DCB+UT-Helper 連携に関しても実装が完了しており、現在は時間ステップを含むステンシル計算をベンチマークとして用意、袖領域の通信およびスナップショットのファイル出力などを Helper スレッドで担当させ、



(a) 最小の演算負荷に合わせてスレッド数を割り当てる実装



(b) 割り当てる GPU と同じスレッドを生成する実装

図 5 DCB の GPU 対応 概念図

主計さんによる期待通りの隠蔽が可能か評価を実施している。DCB の PA や Hybrid モードで述べたように、DCB の動作の結果、不要なコアが発生する。UT-Helper ではこの不要なコア上に Helper スレッド(内部実装的には p-thread)を生成する。Helper スレッドに担当させる処理は関数ポインタで指定する実装を取っている。関数ポインタへは 1 つの引数を許可しており、構造体を使えば必要な数の変数を扱える。Helper スレッド生成時は DCB の内部変数を参照して余剰コア上に Helper スレッドが割り付けられるよう、pthread_create 呼び出し後、ユーザー関数を呼び出す前に pthread_setaffinity_np を呼び出している。そのほかに、Helper スレッドとの非同期な処理を実現するための同期やスレッド終了待ちのための関数を用意している。具体的な内容は図 6 に示す。なお、Helper スレッド自体の実装は C 言語で記述しているが、Fortran Interface も用意している。

③の AT 機構の実装に関しては現在準備中である。AT 機構実現のためには CPU や GPU の動作状況を監視する必要があり、Helper スレッドでこれを行う実装を想定している。図 3 に示した C3 での評価結果から、DCB の割り当てによるシステムの利用効率最適化のためには演算量や動作クロックだけでなく、Retire した命令数やメモリ、キャッシュへのアクセス量などの履歴も必要と考えており、これらは Helper スレッドで PMU から定期的に情報を収集、自動チューニング機構を実現する予定である。

④の DCB の DCB ライブラリ呼び出しの完全ブラックボックス化は自動チューニング機構の導入後に実現可能となるため、こちらは準備中である。現在は DCB でのコア割り付け決定にユーザーから提供される無次元化されたパラメータに依存しており、このパラメータ受け渡しのために DCB ライブラリの呼び出しが必要となっている。AT 機構導入後はこのパラメータ渡しが不要となるため、DCB の呼び出しをブラックボックス化可能となる。ブラックボックス化実現のためには OpenMP Tools を使用する。これは OpenMP で用意された First party tools を呼び出すための API であり、OpenMP ディレクティブをトリガーにできる。parallel ディレクティブへの到達をトリガーにして DCB ライブラリを呼び出せば、Thread の Fork 事に最適なリソース割り付けが提供可能となる。また、ブラックボックス化完了後にはユーザーはプログラムの改良が不要になり、コンパイル時に DCB ライブラリをリンクするだけでよくなる。

```
//uth_ctl_t 構造体：Helper スレッドの管理に利用

//uth_ctl_t の初期化
int uth_init(struct uth_ctl_t *uth_ctl, MPI_Comm const whole_comm);

//Helper スレッドの生成
int uth_create_thread(void (*func)(void *arg), void *arg, int const type_thread, struct uth_ctl_t *uth_ctl);

//スレッドの状態チェック（終了済み or 実行中)
int uth_check_thread(struct uth_ctl_t *uth_ctl);

//スレッドの終了待ち
int uth_join_thread(struct uth_ctl_t *uth_ctl);

//スレッドのキャンセル要請
int uth_cancel_thread(struct uth_ctl_t *uth_ctl);

//スレッドのキャンセルポイント設定
void uth_set_cancelpoint();
```

図 6 UT-Helper Interaface (C 言語版)

6. 進捗状況の自己評価と今後の展望

本年度のプロジェクト開始時には DCB+UT-

Helper 統合プラットフォームの実装完了および簡易的な評価までを目標としており、遅延が発生している状況である。現在も実装および評価を進めており、2025 年度には論文執筆などを実施できる予定となっている。今後は統合プラットフォームの評価や評価の結果必要となったアップデートを加え、外部公開・宣伝を行っていく予定である。さらに、実際に電力キャップが必要なシステム(導入設備的に供給可能な消費電力以上のシステム)を想定し、電力キャップ下でのシステム利用効率最大化を検討していく予定である。これが完遂されれば、近年ひっ迫している電力上を加味しながらも、より大規模・高性能なシステム導入が実現できる。