Vol.2013-HPC-140 No.34
2013/8/2

メニーコア・プロセッサにおける動的スレッド数切替え手法

福本 尚人[1,a)]　中島 耕太[1]　成瀬 彰[1]

概要：本稿では，プログラムの特徴に応じて稼働させるコア数とコアに割当てるスレッド数を変更する手法の実装，および，性能と消費電力の評価について述べる．現在，SMT (Simultaneous MultiThreading) をサポートするマルチコア・プロセッサが主流となっている．このようなプロセッサで並列アプリケーションを実行する場合，最大コア数の活用が最も高い性能を達成できるとは限らない．そこで，プログラムの特徴に応じてスレッド数を切替える手法を提案する．本稿では，実行コア数だけでなく，コアに割当てるスレッド数によって性能が大きく変わることを示した上で，提案手法の実現に必要な割当てスレッド数の切替え手法を実装し，Intel 社の Xeon Phi で評価した．その結果，オーバヘッドは十分に小さく，NAS parallel benchmarks において手法適用により消費電力を増加させることなく，平均 12%，最大 41%の性能向上が得られることが分かった．

キーワード：メニーコア・プロセッサ，Simultaneous multithreading, 並列プログラム，スレッド数制御

Dynamic Thread Assignment for SMT Many-core Processors

Abstract: We implement dynamic thread assignment method on SMT (Simultaneous MultiThreading) multicore processors. Our method changes the number of executing cores and the number of threads per core with low overhead. Recent years, SMT multi-core processors have been mainstream in high-end CPU area. Our method changes the number of execution cores and the number of threads per core depending on application characteristics on these processors. In our evaluation, it is observed that our approach can achieve 41% performance improvement in the best case and 12% performance improvement on average without increasing power consumption compared to conventional execution with all cores.

Keywords: Many-core processors, SMT, multithreaded programs, concurrency throttling

1. はじめに

近年，複数スレッドを同時に処理できるコアを搭載したマルチコア・プロセッサが主流となっている．このようなプロセッサでは，コア内において，複数のスレッドから発行可能な命令を抽出できるため，コア内リソースの利用効率が向上する．また，多数のコアで並列処理することで性能を高められる．微細化技術の進歩とともに，チップに搭載されるコア数は増加する傾向にあり，数十コアが搭載されるメニーコア・プロセッサも販売されている [1], [2]．よって，並列処理の高性能化は今後ますます重要となる．

多くのアプリケーションでは搭載された全てのコアで並列処理することで高性能化を達成できる．しかしながら，コア間の同期が多いプログラムやメモリバンド幅ネックのプログラムでは，一部のコアを停止させるほうが性能が高いことがある [3]．こういった場合，最大コア数での並列処理は性能の低下のみならず，消費電力量の増加を招くといった問題が生じる．

この解決策として，プログラムの特徴に応じて稼働コア数を調整する手法がある [3], [4], [5]．この手法では，適切な稼動コア数をプログラム実行中に予測し，当該コア数で並列処理する．これにより性能向上および消費電力量削減を同時に達成する．しかしながら，この手法はコア数のみに着目しており，コアに割当てるスレッド数を考慮していない．我々が調査した結果，コアに割当てるスレッド数は性能へ大きな影響を与えることが分かった．よって，稼動コア数に加えてコアに割当てるスレッド数を適切に決定することで，従来方式より高い性能を得ることができる可能

[1] 株式会社 富士通研究所
[a)] fukumoto.naoto@jp.fujitsu.com

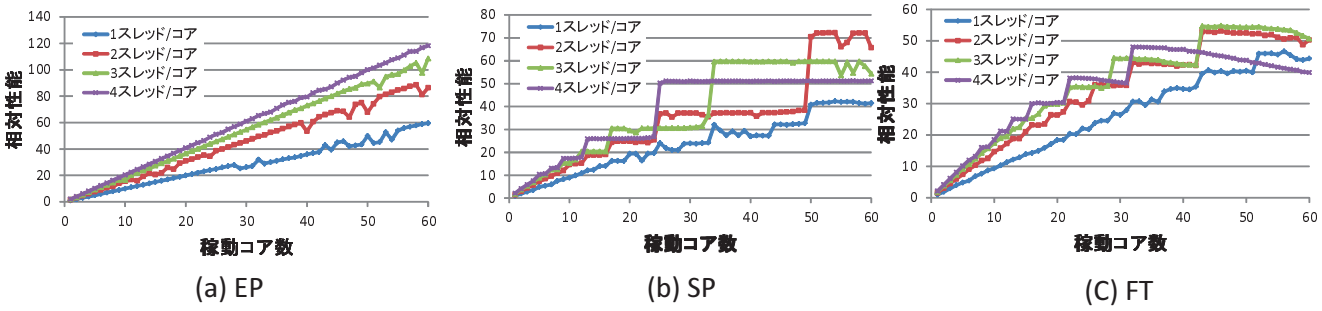

図 1 稼動コア数とコアあたりスレッド数変更による性能変化

性が高い．

そこで，稼動コア数とコアに割当てるスレッド数を適切に変更する手法を提案する．本手法では高性能を達成する稼動コア数とコアに割当てるスレッド数をプログラム実行中に予測し，当該割当てへ切替える．本稿では，まず，稼動させるコア数だけでなく，コアに割当てるスレッド数を変更することで性能が向上することを示す．次に，本手法で必要となるスレッド数切替え手法のプロトタイプを実装する．ここでは，稼動コア数をスレッドの生成・消滅で制御するのではなく，スレッドの状態変更で制御することでオーバヘッドの削減を狙う．評価した結果，実装したスレッド数切替え手法は従来の方法より，オーバヘッドが小さいことを確認した．また，手法の適用により，消費電力を増加させず，平均 12%，最大 44% の性能向上を達成した．

本稿の構成は以下の通りである．まず，第 2 章で研究の動機を説明する．ここでは，稼動コア数，および，1 コアあたりのスレッド数を変更した場合の性能を評価し，スレッド数切替えによる性能向上の見込みが大きいことを示す．次に第 3 章で，プログラム実行中に低オーバヘッドで割当てスレッド数を切替える手法を提案する．さらに第 4 章で，スレッド数切替え手法のオーバヘッド，および，ベンチマークプログラムに手法を適用した場合の性能と消費電力を評価する．そして第 5 章で関連研究について述べ，第 6 章でまとめる．

2. 稼動コア数とコアあたりスレッド数と性能の関係

本章では，稼動コア数とコアに割当てるスレッド数を変更した場合の性能を複数プログラムで評価する．また，プログラム内部においても同様の評価を行い，稼動コア数だけでなくコアに割当てるスレッド数をプログラム実行中に変更することの重要性を示す．

表 1 システム構成

プロセッサ	Intel Xeon Phi 5110P
動作周波数	1.05GHz
コア数	60
スレッド数/コア	4
L1 命令/データキャッシュ	32kB, 8way
L2 非共有キャッシュ	512kB, 8way
メインメモリ	8GB (GDDR5)
システム・ソフトウェア	MPSS 2.1.4982

2.1 評価環境

表 1 に評価に使用した環境を示す．このプロセッサでは，1 つのコアで 4 スレッドまで同時に実行することができる．ベンチマーク・プログラムとして，NAS Parallel Benchmarks 3.3 OpenMP 版 [6] の EP, SP, FT を選択した．入力サイズは ClassB である．

2.2 プログラム間における最適スレッド数の調査

図 1 に，稼動コア数およびコアに割当てるスレッド数を変更した場合の性能を示す．3 つのグラフはそれぞれ異なるプログラムの結果を表す．横軸はプログラムを実行するコア数，縦軸はコアあたりのスレッド数 1，稼動コア数 1 を基準とした相対性能である．凡例は，1 つのコアに割当てられているスレッド数を示す．

図 1(a) の EP では，並列処理に参加するコア数が多いほど性能が高い．また，コアに割当てるスレッド数が多いほど高性能である．こういったプログラムで高性能を達成するには，多数のコアを活用し，コアに割当てるスレッド数を増やせばよい．一方，図 1(b)(c) の SP や FT では稼動コア数やコアあたりのスレッド数が多いほど性能が高いわけではない．SP では一定コア数から性能向上が飽和し，FT では性能が低下する．最も性能が高い構成は SP ではコア数 54，2 スレッド/ コア，FT ではコア数 44，3 スレッド/ コアである．こういったプログラムでは，適切な稼動コア数，コアあたりのスレッド数で実行することで，消費電力を増加させることなく性能を向上させることができる．

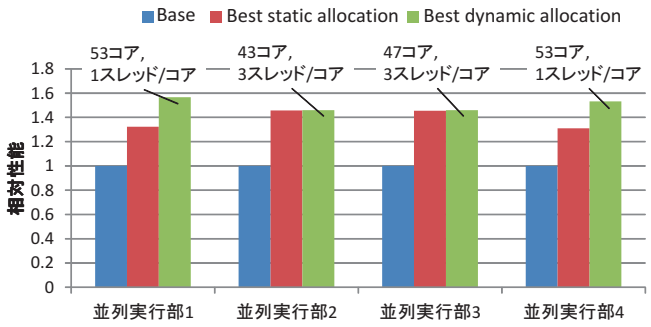

図 2 各並列実行部の性能

2.3 プログラム内部における最適スレッド数の調査

図 2 に FT の主要な並列実行部の性能を示す．横軸が並列実行部の順番を，縦軸が 60 コア, 4 スレッド/コアの性能を 1 とした場合の相対性能を表す．凡例は，“BASE” が 60 コア，4 スレッド/コアで実行した場合，“Best static allocation” がプログラムの実行開始から終了まで同一のスレッド割当てで実行するという条件で，最適な割当てを決定した場合の結果である．“Best dynamic allocation” は並列実行部ごとに最も高性能であるコア数，コアあたりのスレッド数を決定している．また，バーの上方に”Best dynamic allocation” の場合に選択したコア数，コアあたりのスレッド数を載せている．

図 2 によると，並列実行部 1, 4 では”Best static allocation” より”Best dynamic allocation” のほうが性能が高く，並列実行部 2, 3 では両者の性能はほぼ変わらない．この結果より，プログラム内部においても適切な稼動コア数とコア当たりのスレッド数が変わり，性能差があることが分かる．よって，プログラム実行中においても稼動コア数やコアあたりのスレッド数を変更することにより，性能改善効果が得られるといえる．

2.4 考察と課題

本章では，稼働コア数やコアあたりのスレッド数を変更することで性能が改善する見込みが大きいことを示した．プログラム実行中に稼動コア数やコアあたりのスレッド数を切替えるには，動的に適切なスレッド割当てを予測し，予測したスレッド割当てに切替える必要がある．プログラム実行中にスレッド数を切替えるため，スレッド数切替えのオーバヘッドが性能へ与える影響は大きい．スレッド数切替え時間が長い場合，予測方法によらず性能改善効果を得ることができない．そこで本稿では，まず，低オーバヘッドの割当てスレッド数切替えの手法の実装に取り組む．

3. 割当てスレッド数切替え方法

3.1 概要

プログラムの特徴に応じて，低オーバヘッドで稼動コア数とコアあたりのスレッド数を切替えたい．一般にプログラムの並列実行部はループで実現されていることが多く，各イタレーションを複数コアで分担する．ループでは同一コードが繰り返し実行されるため，適切なスレッド割当てが変わらない可能性が高い．そこで並列実行部ごとにスレッドの割当てを変える．

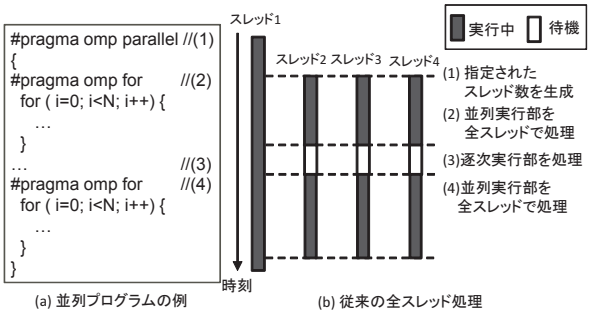

図 3 従来の並列処理の流れ

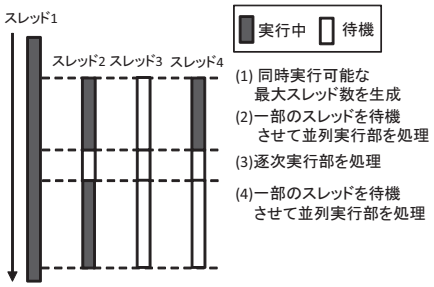

図 4 スレッド切替えの流れ

図 3 に 2 つの並列実行部を持つプログラムを従来方式で実行する手順を示す．具体的には以下の手順で実行する．

(1) 指定された数のスレッドを生成
(2) 並列実行部を生成された全てのスレッドで分担して実行
(3) 逐次部分では，一つのスレッド以外は待機
(4) 並列実行部を生成された全てのスレッドで分担して実行

これに対して，スレッド数切替えを適用した場合の並列処理手順を図 4 に示す．以下の手順でスレッド数を切替える．

(1) 同時実行可能な最大数のスレッドを生成
(2) 必要に応じて一部のスレッドを待機させ，残りのスレッドでイタレーションを分担して並列実行（図 4 では 1 スレッド待機，3 スレッド実行）
(3) 逐次部分では，一つのスレッド以外待機
(4) 必要に応じて一部のスレッドを待機させ，残りのスレッドでイタレーションを分担して並列実行（図 4 では 2 スレッド待機，2 スレッド実行）

本稿で提案する割当てスレッド数切替えでは，不要なスレッドを待機させ，必要に応じてスレッドを待機状態から

```
#pragma omp parallel
{
set_flag();
#pragma omp barrier
if( wait_thread(thread_id) ) { //  (1)
  wait();
}
else {
   /* parallel section*/
    … //  (2)
}
reset_flag();
}
```

図 5 スレッド数切替えの実装

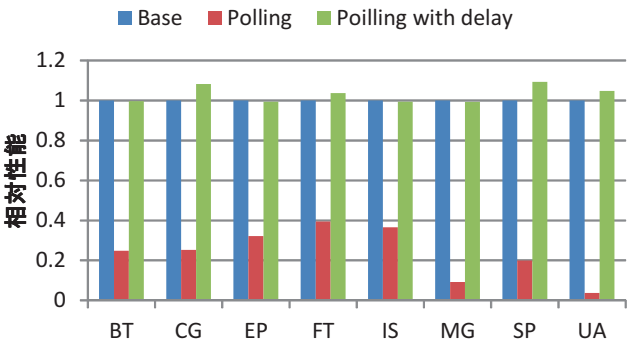

図 6 待機状態のスレッドが他のスレッドの性能へ与える影響

活性化させる．スレッド待機時に他のスレッドの性能へ悪影響を与えず，待機状態から活性状態に高速に遷移させることが出来れば，スレッド切替えによる性能への悪影響は小さい．また，スレッド数切替えを実現するためにスレッド数の増減は行わない．スレッドの生成時間が長いためである．

3.2 実装

アプリケーション開発者の負担を小さくすることを目的として，スレッド切替え手法を OpenMP ライブラリに実装する予定である．ただし，本稿ではプロトタイプとしてアプリケーションのソースコードに手を加える事で割当てスレッド数切替えを実現し評価する．具体的な実装は図 5 の通りである．並列実行部の前で，待機用のフラグをセットし，バリア同期をとる．その後，(1) で待機するスレッドであるか否かを判断する．待機するスレッドはフラグをチェックしながら待機し，その他のスレッドは (2) で並列実行部を実行する．ここでは，実行スレッド数に応じて仕事の分配を適切に行う．並列実行部の実行完了後にフラグをリセットし，待機していたスレッドを通常状態に戻す．最後に必要であればバリア同期を行う．

スレッド数切替え方式の実装で重要なポイントは，スレッドの待機方法である．他のスレッドの性能へ悪影響を与えず，即座に待機状態から復帰できる方法で実装することが望ましい．本稿では，Xeon Phi 特有の Delay 命令を活用することでこれを実現する．Delay 命令は指定した時間だけ実行しているスレッドの命令フェッチを止める．これを活用し，一定時間 Delay 命令で待機した後，フラグのチェックを行う．これを繰り返すことでスレッドを待機させる．

4. 評価

本章では，まず，切替え手法適用によるオーバヘッドを見積もる．次に，適切なスレッド数を予測できたと仮定したときのスレッド切替え時の性能および消費電力を評価する．

4.1 評価環境

第 2 章の表 1 に評価に使用したシステムを示す．評価プログラムは，NAS Parallel Benchmarks 3.3 OpenMP 版の ClassB を使用した．ただし，IS, MG は実行時間が短いため，ClassC を用いて評価している．コンパイラは Intel compiler 2013.2.146 を用いた．全てのスレッドは環境変数 KMP_AFFINITY を用いてコアにバインドした．消費電力は Xeon Phi のモニタリング用コマンドである micsmc を一定間隔で実行し，その平均をとった．

4.2 オーバヘッド

本章ではスレッド数切換えのオーバヘッドを評価する．スレッド数切換えのオーバヘッドは，手法適用による性能向上効果を決定するため非常に重要である．本稿で実装したスレッド数切替え手法は，スレッドを待機・活性させることで実行するスレッド数を変更する．そこで，まず待機しているスレッドが性能へ与える影響を評価する．その後，一回のスレッド数切替えに要する時間を測る．

まず，待機するスレッドが他のスレッドの性能へ与える影響を調べる．稼動コア数 60 で，コアあたりに 1 スレッドを割当てたときの性能を図 6 に示す．横軸はベンチマーク・プログラム名，縦軸は相対性能である．全ての凡例において，60 個の論理コアにおいてベンチマーク・プログラムを実行している．ただし，“Polling” では残りの論理コアでポーリングを行い，“Polling with Delay” では Delay 命令を利用したポーリングを行う．ディレイ命令で待機する時間は 1000 クロックサイクルとしているため，およそ 1usec ごとにフラグをチェックする．図 6 によると，通常のポーリングを行う場合は大きな性能低下が発生している．一方，ディレイ命令を活用したポーリングを行う場合は性能低下が非常に小さい．この結果より，ディレイ命令を活用することによって待機状態のスレッドが性能へ与える影響を小さく抑えることができていることが分かる．

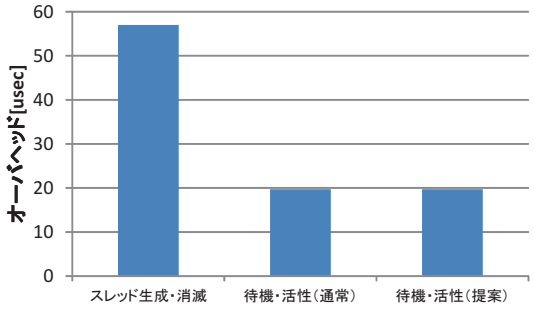

図 7 スレッド切替えに要する時間

図 7 に一回のスレッド切替えに要する時間を評価した結果を示す．このオーバヘッドは，全てのコアで 4 スレッドずつ実行している状態から 3 スレッド実行に切換え，その後 4 スレッド実行に戻すまでの時間である．縦軸は，スレッド数切換えに要した時間であり，横軸はスレッド数切換えの制御方法を示す．“スレッド生成・消滅” は，スレッドを生成・消滅させることで実行コア数を変更する．“待機・活性” では，一部のスレッドを待機・活性させることで実行コア数を変更する．“通常” では単純なポーリングで，“提案” ではディレイ命令を活用したポーリングで待機している．図 7 によると切換えオーバヘッドは，スレッドを生成・消滅させるより待機・活性させるほうが小さい．また，ディレイ命令の活用によるオーバヘッドの増加は小さい．

図 6, 7 の結果より，本稿で実装したスレッド数切替え手法のオーバヘッドが従来の方法より小さいことが分かる．このオーバヘッドは，実行時間がある程度長い並列実行部であれば，許容できる．例えば，NAS parallel benchmarks の CG の場合，並列実行部が 7,800 回呼び出され，実行時間は約 15 秒である．この場合のオーバヘッドは 0.17 秒となり実行時間の約 1%である．実行時間の短い並列実行部はオーバヘッドの影響が大きくなるため，スレッド数切替えを適用しないなどの工夫が必要となる．こういった制御は，並列実行部のイタレーション数と 1 イタレーションの実行時間を計測することで行うことが可能である．

4.3 性能

図 8 にスレッド切替え手法を適用した場合の性能を示す．ただし，本評価では実行時間の短い並列実行部に対してスレッド数切替え手法を適用していない．横軸はベンチマーク・プログラム名，縦軸は全コア実行を基準とした相対性能を表す．多くのプログラムで性能が向上している．例えば，FT や SP では，それぞれ 31%，21%の性能改善が得られている．これらのプログラムでは，第 2 章の図 1 を見れば分かる通り，あるコア数から稼動コア数増加により性能が改善しなくなり，1 つのコアに 4 スレッドを割当てることが性能改善に効果的ではない．こういったプログラムでは，稼動コア数とコアあたりのスレッド数を適切に切

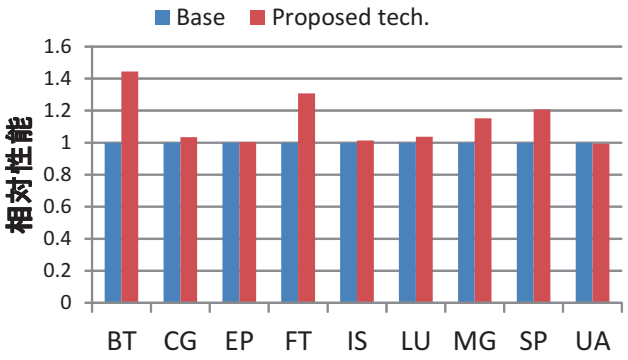

図 8 全コア稼動時と比較した性能向上

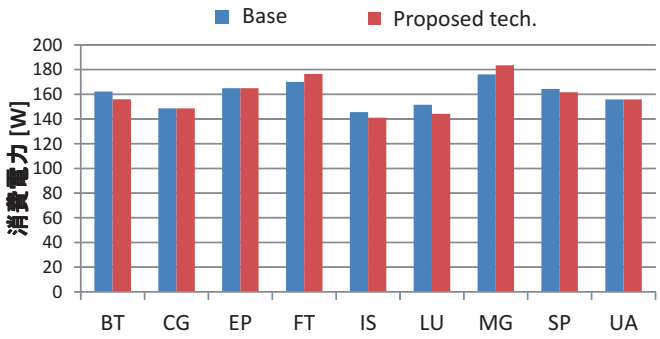

図 9 全コア稼動時と比較した消費電力

替えることで高性能化を達成できる．一方，CG, EP, UA といったプログラムではスレッド数切替えによる高性能化を達成できていない．これらのプログラムでは，全てのコアを稼働させる場合が最も性能が高い．

4.4 消費電力

図 8 に各ベンチマーク・プログラム実行時の消費電力を示す．基本的には使用しているコア数は提案方式の方が全コア実行時より少ないにも関わらず，消費電力は全コア実行と提案方式でほぼ等しい．これは以下の様な理由によると予想している．本手法では，稼動させるコア数を少なくすることで性能を向上させる．性能が向上するということは，チップに搭載されている資源の利用率は増加するはずである．その結果，従来の全コア実行と提案方式でチップ内資源の利用率がほぼ等しくなり，消費電力も拮抗したと考えている．また，おそらく Xeon Phi ではコア単位での電圧・周波数制御が行われていないため，消費電力削減効果が小さいということも考えられる．

5. 関連研究

マルチコア・プロセッサでの並列処理において，プログラムの特徴に応じて稼動コア数を適切に決定することで電力あたりの性能を向上させる手法が提案されている．

Suleman ら [3] は，並列実行部の最初の 1 イタレーションを実行し，ハードウェアモニタリング・カウンタの値を取得し，単純なモデルを用いて適切な稼働コア数を予測する．稼動コア数の変更は OpenMP ライブラリを修正し，スレッド数を増減させることで実現している．Cochran ら [4] は，電力の上限が決められている条件で稼動コア数と動作周波数を適切に決定する方式を提案した．この手法では，事前に機械学習で作成したモデルに対して，プログラム実行時に取得したハードウェアモニタリング・カウンタの値を代入することで適切な稼動コア数と動作周波数を決定する．稼動コア数の変更はスレッド割当て対象のコアを変更することで実現している．

本稿で提案するスレッド切替え手法では，稼動コア数だけでなく，コアあたりのスレッド数を変更する．図 1 で示したように，コアあたりのスレッド数は Xeon Phi では性能へ与える影響が大きいため，適切に決定することで大きな性能向上が得られる．

6. おわりに

本稿では，プログラムの特徴に応じて稼動コア数とコアあたりのスレッド数を切替える方式を提案した．特に，実行コア数のみならず，コアに割当てるスレッド数を変更することで性能が大きく変わることを示した．また，本手法を実現するにあたり，必要となる低オーバヘッドのスレッド数切替え手法を実装した．評価した結果，オーバヘッドは十分に小さく，手法適用により消費電力を増加させることなく平均 12%の性能改善を達成することが分かった．稼働コア数を制御する手法は，コア数に見合う性能が得られない場合に効果がある．微細化技術の進歩とともに，チップに搭載されるコア数は増加しているため，コア数に見合う性能向上を達成することがますます難しくなる．したがって，今後さらにプログラムの特徴に応じた稼動コア数の制御手法が重要になる．

今後の課題はスレッド数切替え手法のライブラリ化である．OpenMP などの並列化ライブラリでは並列実行部の前後で実行される関数がある．これらの関数を修正することでスレッドを制御する．これにより，リンクするライブラリを変更することで本手法を適用できるようになる．また，イタレーション数などの情報を活用してオーバヘッドの削減にも取り組む．さらに，適切な稼動コア数とコアあたりのスレッド数の予測手法も行う予定である．

参考文献

[1] G. Chrysos and S. P. Engineer. Intel® Xeon Phi™ coprocessor (codename knights corner). In *Proceedings of the 24th Hot Chips Symposium*, 2012.

[2] C. Ramey. Tile-Gx100 manycore processor: Acceleration interfaces and architecture. In *Proceedings of the 23rd Hot Chips Symposium*, 2011.

[3] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pp. 277–286, 2008.

[4] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pp. 175–185, 2011.

[5] M. Curtis-Maury, K. Singh, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. De Supinski, and M. Schulz. Identifying energy-efficient concurrency levels using machine learning. In *Cluster Computing, 2007 IEEE International Conference on*, pp. 488–495, 2007.

[6] H. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, 1999.