卒業論文

電力制約下における蓄電池を用いた高性能計算システムの性能向上

03-120601　**酒井 崇至**

指導教員　中村 宏 教授

2014 **年** 2 **月**

東京大学工学部計数工学科システム情報工学コース

概要

近年、コンピュータの消費電力の増大が大きな問題となっており、コンピュータの性能の指標として単なる実行速度だけではなく、消費電力あたりの実行速度（電力対性能）が重要視されるようになってきている。特にスーパーコンピュータのような今日の高性能計算システムでは数メガワットもの電力を消費しており、物理的制約からこれ以上の電力供給力の向上は困難である。このような背景により、予め決められた消費電力の制約下での実行速度の最大化が、今後の高性能計算システムの性能向上の鍵となっている。

そこで、本論文では蓄電池を用いた高性能計算システムの性能を向上手法を提案する。現在の高性能計算システムには、停電時にもシステムへの電力供給を続けられるように UPS（無停電電源装置）が搭載されている。それを非停電時にも積極的に充放電を行い、アプリケーションの中の電力対性能が上がりにくい部分から上がりやすい部分へ時間方向に電力を融通することによって、電力制約下における性能を向上させることができる。今回はこの手法を CPU-GPU ハイブリッド構成の計算ノードを用いて 3 種類のベンチマークで性能評価実験を行い、この手法を用いない場合に比べて平均？ % の性能向上が実現できることを示し、その有用性を確認した。

目次

第 1 章

序論

現代社会においてコンピュータの担う役割はかつてないほど大きくなっており、我々の生活に欠くことのできない存在となっている。より高性能なコンピュータを作るべく、これまで多くの研究者がコンピュータ技術の発展に貢献し、Moore の法則 [1] の示す通りチップの集積度が指数関数的に増すと共にコンピュータの性能も向上し続けている。

近年、コンピュータの性能向上の妨げとなっている要因の一つが消費電力の増大である。一般に、高速な演算を行うためには大きな電力を消費しなければならず、数年前までは性能向上と共に消費電力も増加し続けてきた。ところがスーパーコンピュータなどの HPC 領域においては既に供給できる限界に近い電力を消費しており、物理的な電力供給能力によってコンピュータの性能が制限されている。そのため、与えられた電力制約の中でいかに処理能力を向上させるかが現在の大きな課題となっている。

この課題を解決するため、プロセッサやメモリの動作速度を動的に制御する DVFS という技術が開発され、現在の多くのコンピュータに搭載されている。この技術は性能のクリティカルパス上にないモジュールの動作速度を落とすことにより、性能低下を防ぎつつ消費電力を下げるというものであり、この技術を HPC 領域に応用することによる、電力制約下での性能向上が期待されている。

また、現在のデータセンターやスーパーコンピュータなどの大規模高性能計算システムにおいては、BCM(事業継続マネジメント) の観点から、地震や火事などの災害による停電時にも継続してコンピュータを稼働させられるように自家発電設備や蓄電池が搭載されているケースが多くなってきた。ただ、現状ではそれらの設備はあくまで緊急時のための予備電源としてのみ見なされており、平常時には使用されていない。そのため、それらの新たな電力資源を有効活用して電力対性能を向上させることができると提案されている [2] が、まだこの可能性が示唆されてから日が浅く、未開拓の領域が多く残されている。

そこで本論文では、蓄電池が搭載された高性能計算システムにおいて非停電時にも積極的に蓄電池の充放電を行うことによって、電力制約下での性能向上手法を提案する。HPC 領域において蓄電池を用いた電力制約下における性能向上手法はいまだ提案されておらず、本稿において初めての試みである。

本手法では、Tapasya Patki らの研究 [3] の対象となっているような、厳しい電力制約のた

めに全てのモジュールを常に最高動作速度で動作させることはできないようなシステムを対象とする。まずアプリケーションのテスト実行時のプロファイルデータからアプリケーションの電力対性能グラフの時間推移を予測する。そして消費電力を減らしても性能が下がりにくい部分を見つけて充電し、逆に消費電力を増やすと大きく性能が上がる部分で放電することにより、電力制約下における性能向上を目指す。

以降、2 章では本論文に関する技術や研究を紹介し、3 章では解くべき問題の定義と、提案手法の核となる論理を説明する。4 章では 3 章での手法の有用性を確認するための実験方法について述べる。5 章で実験結果を示し、6 章でその結果について考察した後、7 章で結論と今後の課題を述べる。

第 2 章

研究の背景

本章ではまず提案手法の核となる技術である DVFS、及び DVFS を用いた既存の電力削減手法について説明する。そして、対象とする蓄電池を含んだシステムの電力供給システムについて説明した後、蓄電池と DVFS の両方を用いた電力削減手法の関連研究を紹介する。

2.1 DVFS

2.1.1 DVFS とは

基本的に、プロセッサやメモリはある一定の周波数で動作するように設計されている。動作周波数が高いほど処理能力も高くなるが、同様に消費電力も大きくなる。そのため、プロセッサやメモリを省電力化する最も単純な手法の一つとして、動作周波数を低くするというものがある。かつてのプロセッサやメモリは設計時に決められた一つの動作周波数でしか動作することはできなかったが、現在では一つのプロセッサやメモリが複数の動作周波数をサポートしており、演算中であっても瞬時に動作周波数を切り替えられるようになった。この技術を用いて動的に動作周波数を切り替え、処理速度と消費電力を変化させることによって省電力化を行う手法を DVFS(Dynamic Voltage and Frequency Scaling) と呼ぶ。

図 2.1 に、あるサーバプロセッサにおいて DVFS を用いたときの電力削減のグラフを示す [4]。動作周波数を低くすることにより処理できる最大負荷 (Computer load) は下がるが、電力を削減することもできている。つまり、処理できる負荷であれば低い動作周波数の方が消費電力を少なくすることができる。図 2.1 の DVS savings の推移を見れば分かるように、この例では DVFS を用いることにより最大で 20% ほどの電力削減が行えることになる。

2.1.2 DVFS を用いたコンピュータの既存の省電力化手法

プログラム実行時、メモリやネットワークなどのプロセッサ以外のモジュールがボトルネックとなっているときには、プロセッサはビジーループとなり、処理を行わず電力だけを消費している時間の割合が高くなる。そのためこのような状況ではプロセッサ自体の処理能力を落としてもシステム全体の処理能力はあまり下がらないため、プロセッサを低い動作周波数に切り

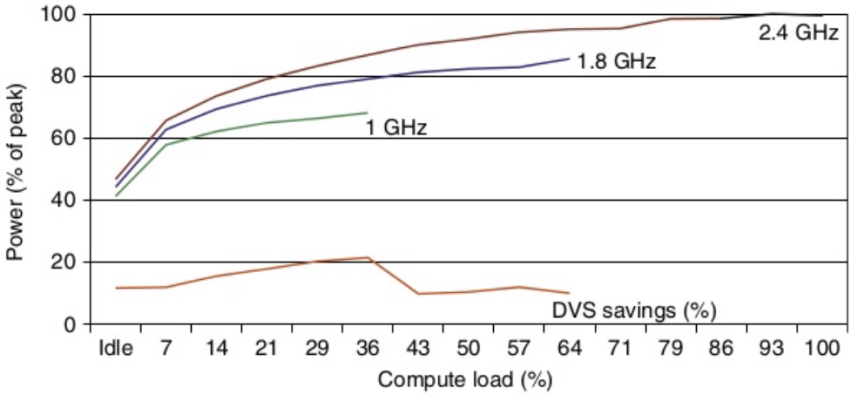

図 2.1. DVFS による電力削減 (AMD Opetron microprocessor) 文献 [4] Figure 1.12 より

替えることで性能低下を防ぎつつ省電力化を行ってきた。

同様に、メモリがボトルネックとなっていない状態ではメモリの動作周波数を落とすことで電力を削減することができる [5]。

また、近年では複数のプロセッサを搭載したマルチプロセッサシステムが増えてきた。マルチプロセッサシステムは複数のプロセッサで並列に処理を行うことで高速化をはかっている。しかし、ひとつずつ順番に処理を行うことが必要なプログラムではひとつのプロセッサのみが処理を行っており、その他のプロセッサはほとんど処理を行っておらず、無駄な消費電力が発生していた。そのような状況では、処理を行っているひとつのプロセッサのみを高い周波数で動作させ、その他のプロセッサの動作周波数を落とすことで消費電力を削減している。

DVFS という技術は登場してからまだ日が浅く、DVFS を用いた電力削減や電力対性能向上の研究は現在盛んに行われている。例えば、2008 年の研究では、複数プロセッサの組み込みシステムにおいてナノ秒単位で DVFS 制御を行うことにより既存の DVFS 制御からさらに 20% もの電力削減が行えるとされている [6]。2011 年の研究によると、メモリのバンド幅の使用率を用いてメモリの DVFS 制御を行うことにより、システム全体のエネルギーの 2.4% を削減できる [5]。これ以外にも多くの研究がなされているが、それでもまだまだ多くの課題が残されているのが現状である。

2.1.3　プロセッサとメモリの DVFS と組み合わせた電力削減の関連研究

一般に、プログラム実行時はプロセッサかメモリのどちらかの処理能力がシステム全体のボトルネックとなっていることが多く、このときボトルネックとなっていないモジュールでは処理能力が必要以上に高い状態となっており、電力が無駄に消費されている。そのため、それぞれのモジュール間での処理能力の差をなくすことが、無駄な電力消費を減らす上で重要である。

この問題を解決するため、プロセッサとメモリの DVFS を同時に用いることによって、そ

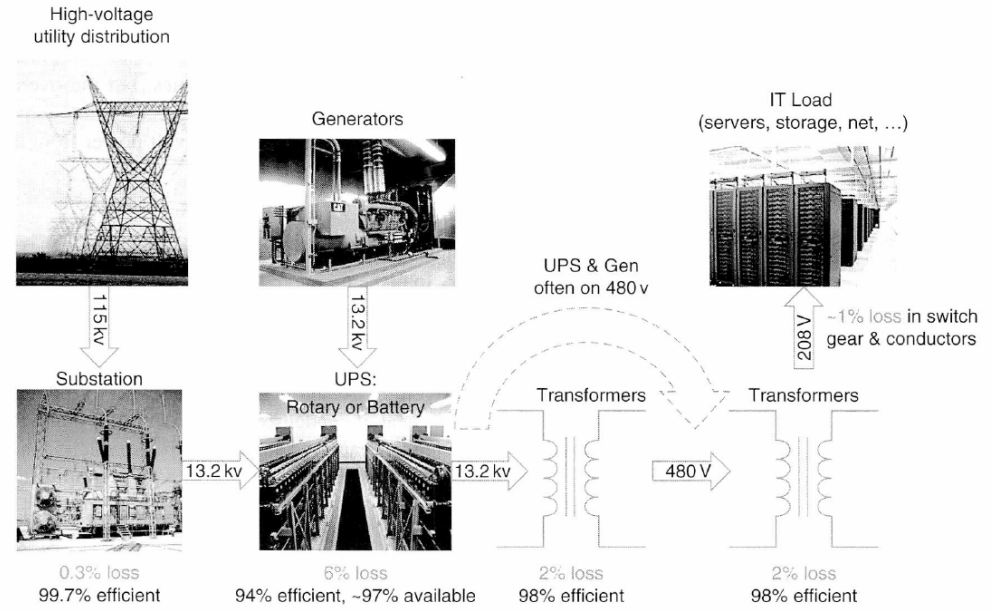

図 2.2. データセンタにおける電源設備 文献 [4] Figure 6.9 より

れぞれの DVFS を別々に行う場合よりもさらに電力対性能の向上を目指した手法が存在する [7]。この手法では、5 ミリ秒おきにプロセッサとメモリの処理能力の両方を監視して、一方のモジュールの処理能力が足りないときにはそのモジュールに電力を融通することによって処理能力の偏りをなくし、与えられた性能制約を満たしつつ省電力化を行っている。

2.2 蓄電池を含む電力供給システム

スーパーコンピュータやデータセンタなどの大規模高性能計算システムにおいては、高い信頼性が要求されるため、一瞬たりとも電圧低下や電力供給停止は許されない。そのため、停電や機器の故障によって電力会社からの電力供給が受けられない時にも、コンピュータへの電力供給を継続するためにいくつかの冗長電源設備が用意されている。現在のデータセンタの一般的な電源設備は図 2.2 のようになっている。

電力会社からの電力供給が停止した場合には、UPS(無停電電源装置) が電力供給を行い、同時に自家発電設備が起動する。数分後、自家発電設備が完全に起動して電力供給が可能になると、自家発電設備から電力供給が行われるようになる。電力会社からの電力供給が再開すると自家発電設備は停止し、電力会社からの電力を使用するようになる。

ここで UPS は 3 つの役割を担っている。一つ目は、コンピュータへの供給電圧を安定させること。二つ目は、停電時に自家発電設備からの電力供給が始まるまでの間、電力を供給すること。三つ目は、停電復帰後に自家発電設備から電力会社に電力供給元を切り替えるとき、一時的に電力供給を行うことである。

UPS 単体がシステム全体に電力を供給し続けられる時間は数分〜30 分程度である場合が多

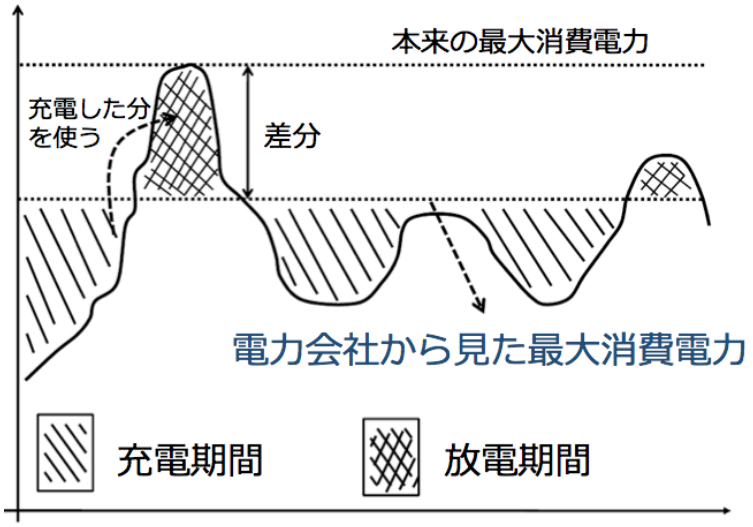

図 2.3. 蓄電池を用いた電力ピークカット手法

い。現在の多くの UPS では電源として蓄電池が使用されているが、充放電が行われるのは基本的に停電時のみであり、今のところ平常時に積極的に充放電を行うような使い方はなされていない。

また、Google や Facebook のデータセンタにおいては一つのラックやサーバごとに UPS が搭載されている [8]。これは電力架線からサーバまでの電力供給の間に行われる D/A 変換の回数を減らして、全体での電力変換効率を向上させるためである。電力効率を重視する上ではこれからのスタンダードになっていくと考えられている。

2.3　データセンタにおける蓄電池を用いたピーク電力削減手法

前節 2.2 で述べたように、今までは平常時に積極的に UPS の蓄電池から充放電を行うことはなかったが、2011 年に発表された論文 [2] において、UPS からの充放電を用いたデータセンタの電力ピークカット手法が提案された。本稿の提案手法と大きく関わる内容であるので、ここで詳しく紹介する。

データセンタにおいてはコンピュータでの消費電力や冷却にかかる電力コストは全体の運用コストの 10〜30% に上り、サービス向上のために電力コストの削減が必要とされている。データセンタを建設するときの初期投資、及び電力会社との契約料金はピーク時の電力に大きく影響される。そのためピーク電力を削減すべく、この論文では UPS の中の蓄電池を用いた電力ピークカット手法を提案している。

データセンタの 1 日の電力需要の推移は、統計や過去の研究によってある程度予測ができるようになっている。その電力需要曲線から最適な蓄電池の充放電計画を立て、電力会社から引き込む電力の最大値を低く抑えることがこの紹介論文の主旨である (図 2.3)。

紹介論文において解くべき対象としている問題を言葉で表現すると以下のようにまとめら

れる。

- 目的
  - minimize (一日の最大消費電力)
- 与えられる情報
  - 一日の電力推移グラフ
- 制御対象
  - バッテリーをいつ、どれだけ充放電するか
- 制約条件
  - 一日の放電時間・回数
  - バッテリーの残量

この紹介論文の研究以前にも、電力会社からのピーク電力を削減するためにプロセッサの動作速度を変更する手法 [9, 10, 11, 12, 13] や、負荷を時間的もしくは空間的に分散させる手法 [14, 15] が提案されてきた。しかし、これらの手法を適用すると処理速度の低下が必ず起こってしまうことが問題であった。紹介論文における提案手法は、UPS に含まれるバッテリーという既存設備を用いることで、この性能低下を起こさずに電力ピークカットを実現できることを示している。

この手法で実際に用いられているアルゴリズムは図で表現すると図 2.4 のようになり、言葉で表現すると以下のようになる。

1. 一番高いピークが、二番目に高いピークと同じ高さになるように放電を計画（制約条件を満たせば次のステップへ）
2. 二番目のピークより高いピーク全てが、三番目に高いピークと同じ高さになるように放電を計画（制約条件を満たせば次のステップへ）
3. ・・・(制約条件を満たさなくなるまで繰り返し)

この紹介論文は他にもバッテリーの電力を使用することによる停電時の信頼性低下や、充放電頻度に対するバッテリーの寿命低下も考慮に入れて充放電計画を立てることによって、データセンタの事業継続性を保ちつつ電力コストを削減できるとしている。

この紹介論文では性能制約を守った上でどれだけ省電力化を行えるのかという問題であった。一方で、決められた電力制約を超えないように制御を行う Power Capping という手法についての先行研究も存在する [16]。Power Capping を実現するためには様々なアプローチがあるが、定期的に使用電力を監視して電力制約に近づくとプロセッサの周波数を下げる・実行するタスクを減らす、といったような手法が提案されている。

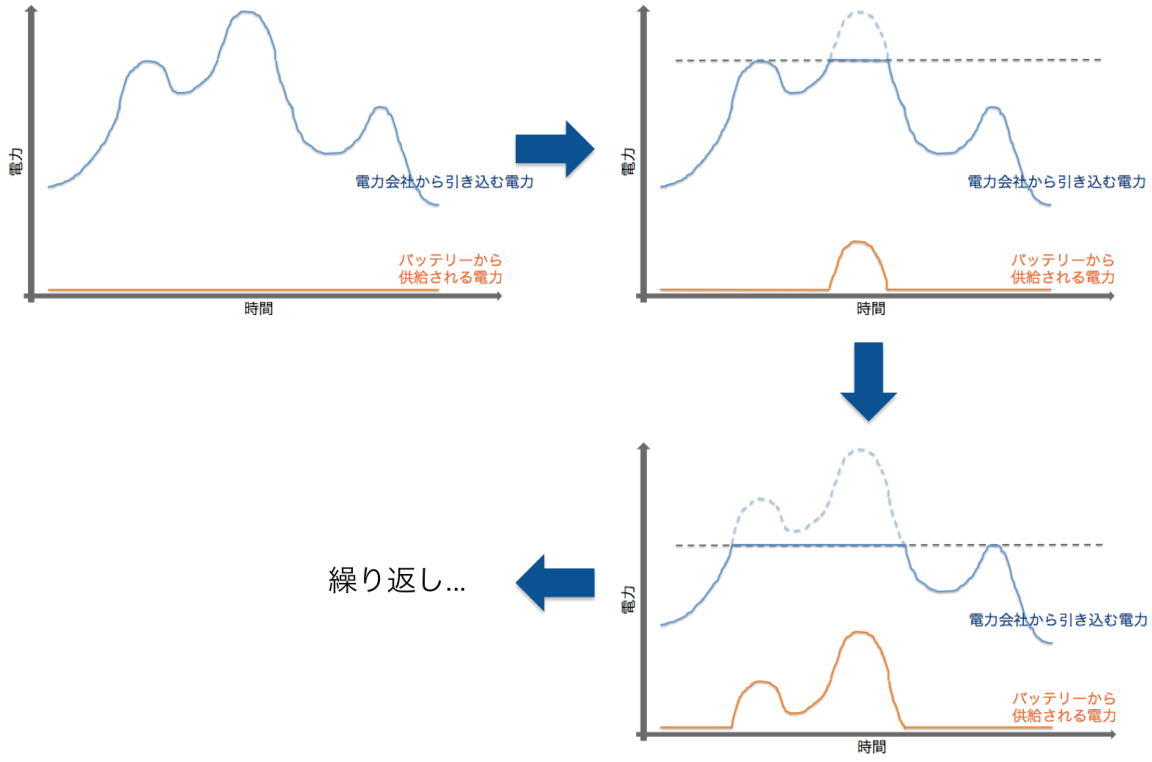

図 2.4. 紹介論文 [2] における UPS を用いた電力ピークカットアルゴリズム

第 3 章

蓄電池を用いた高速化手法

前章でデータセンタにおける電力コスト削減要請についての背景および関連研究を紹介した。実は電力削減はデータセンタだけに限らず、コンピュータアーキテクチャ全体の共通の課題である。その中でも HPC（High Perfomance Computing）領域においては消費される電力は物理的制約による供給可能電力に達しつつあり、近い将来スーパーコンピュータの性能は電力供給能力によって頭打ちになると予想されている。そのため電力対性能の高いシステムの構築が必要とされている。本章ではその実現手法のひとつとして、既存設備に含まれるバッテリーを用いた電力対性能の向上手法を提案する。

3.1 フェーズ間の電力融通手法の提案

スーパーコンピュータ上で走るアプリケーションは実行時間が長く、実行が進むにつれて処理の特性が大きく変化するものも多い。CPU による演算中心の処理やデータの読み書きなどの I/O が中心の処理、プロセス間の通信が中心の処理など、異なる処理は基本的に異なる特性を持っている。また同じ処理であっても演算の並列度などの他の多くの要因に処理の特性は影響される。一般に処理の特性が異なると、その処理にかける電力と処理を終えるまでの実行時間の関係を表した電力ー実行時間曲線は異なったものになる。異なる電力ー実行時間曲線において、かける電力を変化させたときの実行時間時間の変動の大きさは異なる（図 3.1）。

本手法では処理の特性の違いによる電力ー実行時間曲線の違いに着目する。ひとつのアプリケーションを異なった処理の特性を持った時間的に連続する複数の区間に分割し、かける電力を減らしても実行時間があまり短くならない区間から、電力を多くかけると実行時間が大きく短くなる区間へ蓄電池を用いて時間方向へ電力を融通することにより実行時間を短縮する。

また、現在のプロセッサは離散的な有限の数の周波数でしか動作することはできない。そのため、電力制約によって最高周波数で動作できない場合であっても、使い切れていない電力というものが存在する。例えば、消費電力 80W および 60W の 2 種類の動作周波数をサポートしているプロセッサに対して 70W の電力制約をかけた場合、プロセッサは消費電力 60W の周波数でしか動作することができないので、10W の電力を使い切れないことになる。蓄電池を用いればこの余った電力も時間方向に融通することで活用することができるので、その分実

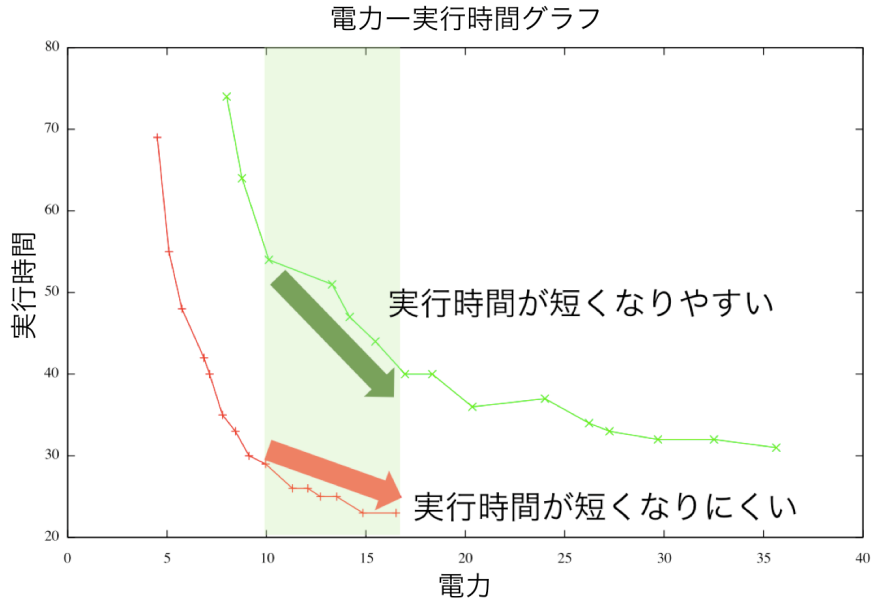

図 3.1. 電力ー実行時間曲線の違いによって、かける電力の増加による短縮される実行時間が異なる様子

行時間の短縮に繋がると考えられる。

以上の実行区間ごとの電力ー実行時間曲線の違いを利用した性能向上と、動作周波数が離散的であることによる余剰電力を利用した性能向上が本論文における提案手法である。次節以降では、アプリケーションの区切り方および電力融通をどう行うかについての各論を述べる。

3.2　フェーズの要件

節 3.1 で述べたように、本手法では処理ごとの電力ー実行時間曲線の違いを利用して性能を向上させる。そのため、電力ー実行時間曲線の異なる区間をそれぞれ別のフェーズとして定義する。フェーズの区切り方が細かいほど電力融通の機会は増えることになるので、理想的なバッテリーを用いる場合には、アプリケーション内の電力ー実行時間曲線が異なる区間全てを別のフェーズとして区切る場合が理想的なフェーズの区切り方となる。

しかし、実際のバッテリーはあまり高頻度に充放電を行えない・充放電の速度に限界があるなど様々な物理的制約がある。また、フェーズの数が多くなるほど節 3.4 で述べる電力融通問題を解くことが困難になる。さらに、入力データが異なる場合には処理の順序が異なるので、どの時刻にどの処理が行われているかが分かりにくく、アプリケーション内の電力ー実行時間曲線が異なる部分を全て求めること自体も難しい問題である。そのため、現実の問題を扱う場合には電力ー実行時間曲線が異なる全ての部分ではなく、もっと粗い粒度でフェーズを区切ることになる。

3.3 フェーズの求め方

HPC 領域において、アプリケーションの性質や演算装置の特徴に応じてソースコードに手を加えることは珍しいことではない。ソースコードに手を加えるときに手間となるのはソースコードを書き直すことである。以下のコードは CPU 並列化プラットフォームの OpenMP のコードであるが、このようにいくらかのコードを書き足す程度であればプログラマの大きな負担にはならない。

OpenMP のソースコード

```
int main(int argc, char *argv[])
{
    int i;
#pragma omp parallel for        //性能向上のために追加される唯一の行
    for(i = 0; i <= 10000; i++)
    {
        //（並列処理させたいプログラム）
    }
}
```

本手法では、上の例のようにユーザプログラマにいくらかのコードを足してもらうことによりフェーズを区切る。具体的には以下のようになる。

本手法で想定するフェーズの指定方法

```
int main(int argc, char *argv[])
{
#phase start A    //フェーズ A の始まりを示す
    //フェーズ A の処理
#phase end A      //フェーズ A の終わりを示す
    //何らかの処理（ない場合もある）
#phase start B    //フェーズ B の始まりを示す
    //フェーズ B の処理
#phase end B      //フェーズ B の終わりを示す
}
```

この区切りを示すコードは、時系列的に異なる処理の部分に配置さえされていれば、以下のように繰り返し文の中に入っていたり、他の関数にまたがっていたりしても構わない。

繰り返し文における例

```
int main(int argc, char *argv[])
{
for(i = 0; i <= 10000; i++)
{
    #phase start A    //フェーズ A の始まりを示す
        //フェーズ A の処理
    #phase end A      //フェーズ A の終わりを示す
    #phase start B    //フェーズ B の始まりを示す
        //フェーズ B の処理
    #phase end B      //フェーズ B の終わりを示す
    }
}
```

関数にまたがっている場合の例

```
functionA()
{
#phase start A    //フェーズ A の始まりを示す
    //フェーズ A の処理
#phase end A      //フェーズ A の終わりを示す
}

functionB()
{
#phase start B    //フェーズ B の始まりを示す
    //フェーズ B の処理
#phase end B      //フェーズ B の終わりを示す
}

int main(int argc, char *argv[])
{
    functionA();
    functionB();
}
```

3.4　電力融通問題の定式化

節 3.3 の手法によってアプリケーションが n 個のフェーズに区切られていて、それぞれのフェーズが 1 から n までの番号を一意に割り振られている状況を考える。フェーズ i（ただし $1 \leq i \leq n$）における電力ー実行時間曲線を $T_i(p)$ と定義する。$T_i(p)$ はフェーズ i にかける電力 p に対して、フェーズ i を終えるのにかかる実行時間を返す関数である。$T_i(p)$ はフェーズ分割が行われていれば、それぞれの DVFS パターンについてテスト実行を行うことで得ることができる。この点については節 4.2 で詳しく述べる。

本論文での目的は、与えられた電力制約下においてアプリケーション全体の実行時間を最小化することである。そこで、与えられる電力制約を p_{max} とする。そして本手法ではフェーズごとに蓄電池を用いて電力を融通するため、フェーズ i において蓄電池から供給される電力を Δp_i とする。Δp_i はマイナスのときは蓄電池に充電することを意味する。

以上の変数を用いて最適化問題として定式化すると、以下のようになる。

$$\min \quad \sum_{i=1}^{n} T_i(p_{max} + \Delta p_i) \tag{3.1}$$

$$\text{s.t.} \sum_{i=1}^{n} \Delta p_i T_i(p_{max} + \Delta p_i) \leq 0 \tag{3.2}$$

式 (3.1) は実行時間の最小化を意味する。式 (3.2) の左辺は、アプリーケション実行の全体を通して、蓄電池から供給されるエネルギを意味している。蓄電池はあくまで電力を時間方向に融通しているだけであり、エネルギを増やすことはできない。そのため、式 (3.2) はエネルギ保存制約式となる。

また、$T_i(p)$ はおおまかに近似をとると式 (3.3) で表される直角双曲線になるため、式 (3.1)、(3.2) で定式化される最適化問題は非線形計画問題となる。

$$T = a_0 + \frac{a_1}{p - a_2} \tag{3.3}$$

ただし、ここで対象としている蓄電池は以下のような特徴をもつ理想的な蓄電池である。

1. 電池容量無限
2. 充放電速度無限

3.5 電力融通問題の解法

非線形計画問題を解くアルゴリズムはいくつも研究されている。しかし、節 3.1 でも述べたように現実のプロセッサは有限の数の周波数でしか動作することはできない。そのため、式 (3.1)、(3.2) の中の Δp_i は有限のパターンしか存在しない。本論文では Δp_i が有限個の値しか取れないことを利用して、全てのフェーズにおいて取りうる全ての Δp_i を実際に代入してアプリケーション全体の実行時間を計算することにより、最適な $\Delta p_i (1 \leq i \leq n)$ を求める。

第 4 章

実験

4.1 実験の目的

節 3.1 で述べたような、電力ー実行時間曲線の違いによって生じる電力を増減させたときの実行時間の変動幅の違いを利用した性能向上手法は今のところ先行研究が存在しない。そのため、本実験において蓄電池を用いて理想的な電力融通を行うことができた場合にどれほどの効果があるのかを見積もることが第一の目的である。また同じく節 3.1 で述べたように、プロセッサの周波数が離散的な値しか取ることができないことによって生じる余剰電力を利用した効果もあると予想されるため、その効果もできる限り個別に評価する。これが第二の目的である。

以下、これら二つの目的を達成するための実験方法及びその結果・考察について述べる。

4.2 実験方法

本手法を用いた性能向上を実現するためには以下の 3 つの段階を踏む必要がある。

1. ユーザがアプリケーションのソースコードに埋め込んだフェーズ区切り文を読み取り、アプリケーションを分割する。
2. 分割されたフェーズそれぞれについて電力ー実行時間曲線を求める
3. 分割されたフェーズに対する電力融通問題を解く

まず、1 段階目について述べる。実際に節 3.3 のように書かれたソースコードからフェーズ区切り文を読み取るのはコンパイラレベルでの実装が必要となり困難である。そのため、本実験ではそのフェーズに入ったもしくは出た時刻をログとして出力する自作関数をソースコードに埋め込むことによって、その代わりとした。また、今回実験に利用したアプリケーションは自作のものではないので、ソースコードを読んでどの部分がフェーズの区切りであるかを完全に知ることは困難であった。そこで簡単のため、並列処理部分と逐次処理部分のみをソースコードから判別してフェーズ分割を行った。そして、プロセッサの取りうる全ての DVFS パターンについてアプリケーションを実行し、ログファイルを得た。ただし表 4.3 に示したアプ

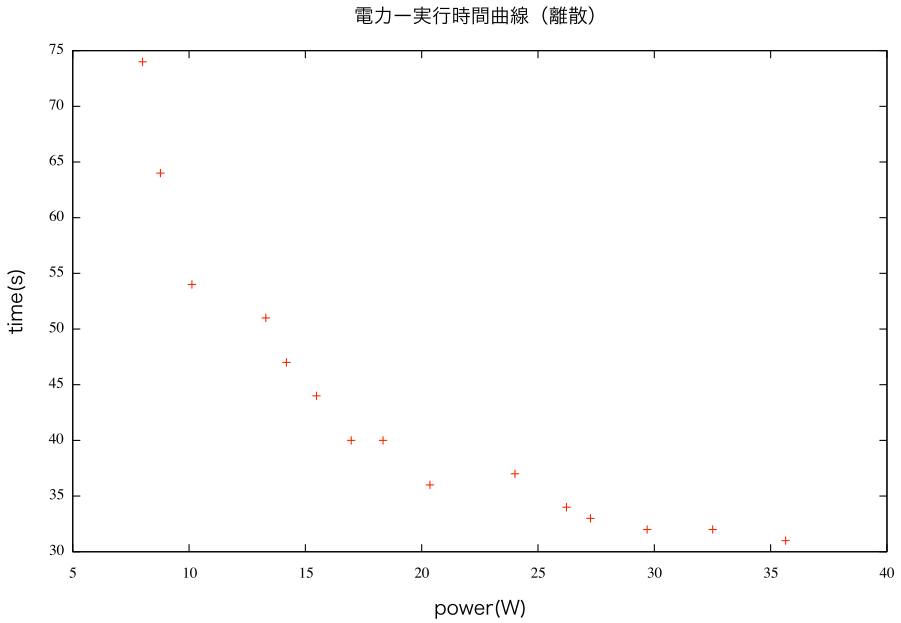

図 4.1. 離散的な周波数しか取れない場合の電力ー実行時間曲線

リケーションのうち、QT Clustering のみは並列実行の中に 2 つのフェーズがあったため、全体で並列処理 1・並列処理 2・逐次処理の 3 フェーズに分割した。

次に 2 段階目である。ここでは 1 段階目で得たログファイルから、それぞれのフェーズについて全ての DVFS パターンにおける平均電力と実行時間を得ることができる。それを用いてそれぞれのフェーズの電力ー実行時間曲線を計算した。ただし実際には DVFS パターンは有限であるので、電力ー実行時間曲線は図 4.1 のようになった。

最後に 3 段階目は、節 3.5 で述べたように得られた電力ー実行時間曲線から総当たりで最適な DVFS 設定値を見つけた。

また節 4.1 で述べたように、本実験の目的は電力ー実行時間曲線の違いを利用した性能向上と、プロセッサの動作周波数が離散的であることを利用した性能向上のそれぞれを評価することである。ここまでの実験方法では両者の影響が入り交じった結果のみしか得られない。そこで、2 段階目で得られた電力ー実行時間グラフを式 3.3 に合うように補間することによって、連続な電力ー実行時間グラフを得た（図 4.2）。ただし完全に連続であると電力融通問題を総当たりで解くことができなくなるため、実際にはある程度の細かい間隔で補間することで連続的な曲線への近似とした。そして、3 段階目では同様に総当たりで最適な DVFS の設定値を見つけ、周波数が離散的な場合の結果と比較することによって、電力ー実行時間曲線の違いによる効果とプロセッサが離散的であることによって生じる余剰電力による効果のそれぞれを見積もった。

用いたベンチマークアプリケーションは表 4.3 の通りである。CPU 並列のアプリケーションでは CPU の電力のみ、GPU 並列のアプリケーションであれば GPU の電力のみを測定した。

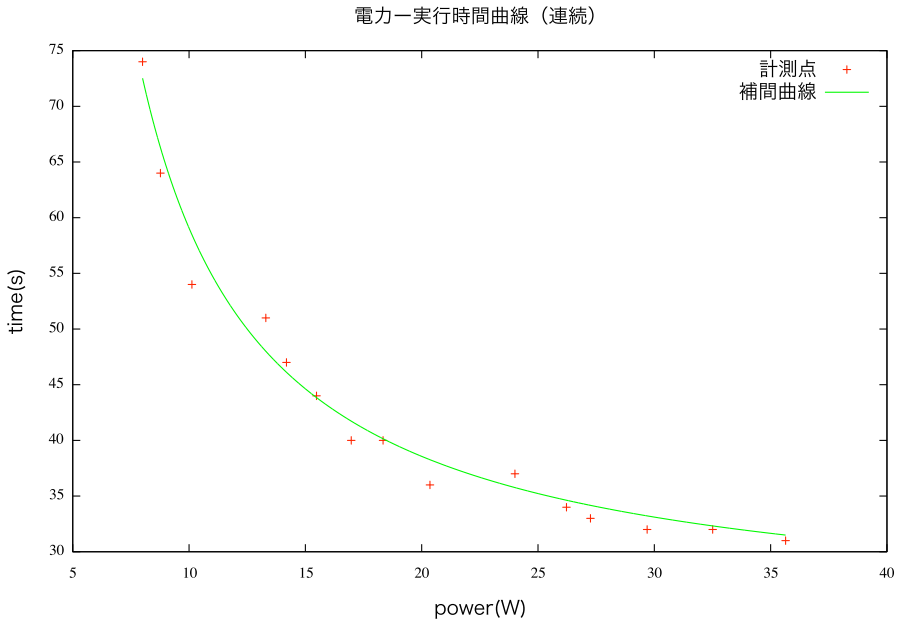

図 4.2. 連続補間した場合の電力ー実行時間曲線

ワークロード名		説明
PARSEC Canneal	CPU 並列	SA アルゴリズムを用いてルーティングコストが最小となる chip を設計する.
PARSEC Stream Cluster	CPU 並列	ストリーミングされる点列のオンラインクラスタリングを行う.
Rodinia mummergpu	GPU 並列	深さ優先探索により DNA 塩基配列を求める
SHOC QT Clustering	GPU 並列	クラスタメンバ間の相関が指定されたカットオフ値より高いことを保証するクラスタリング

表 4.1. 使用したベンチマークアプリケーションの詳細

4.3　結果

それぞれのアプリケーションの電力ー実行時間曲線を図 4.3、4.4、4.5、4.6 に、スピードアップ曲線を図 4.7、4.8、4.9、4.10 に示す。ただし、スピードアップは以下のように定義される。

$$\text{スピードアップ} = 1 - \frac{\text{UPS ありの場合のアプリケーション全体の実行時間}}{\text{UPS なしの場合のアプリケーション全体の実行時間}} \tag{4.1}$$

Canneal の電力ー実行時間グラフは並列処理と逐次処理の電力消費の違いをよく表しており、並列処理の方が逐次処理より大きい電力を消費していることが分かる。Stream Cluster も同様の傾向は見て取れるが、Canneal とは違って逐次処理部分の実行時間が非常に短いことが

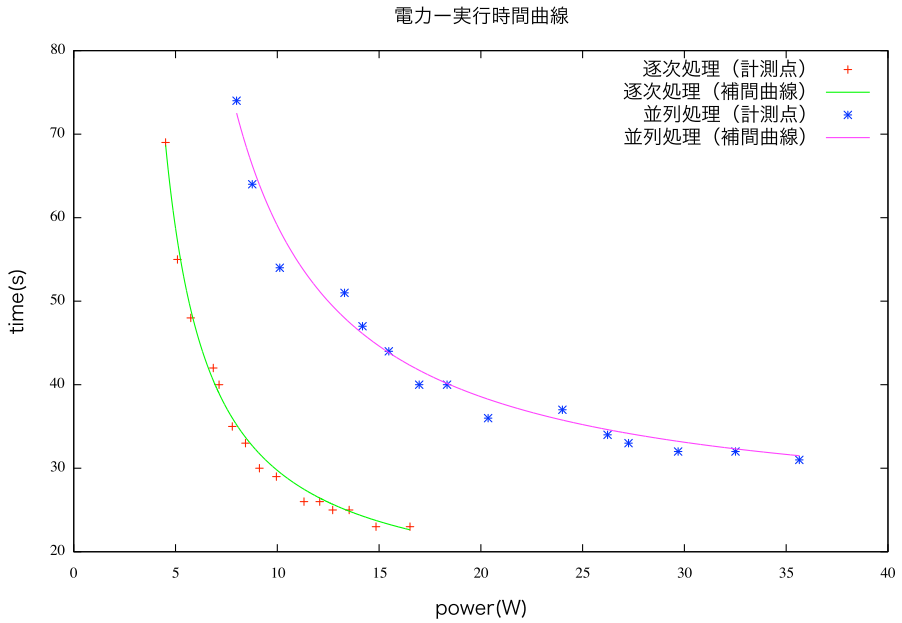

図 4.3. Canneal 電力ー実行時間曲線

ワークロード名	スピードアップ平均
PARSEC Canneal	4.537573714787333%
PARSEC Stream Cluster	0.5633360322900735%
Rodinia mummergpu	18.23325679997816%
SHOC QT Clustering	15.8912119519231%

表 4.2. スピードアップの算術平均（周波数が離散の場合）

特徴である。mummergpu のグラフを見ると、どの DVFS 設定においても逐次処理部分の実行時間が変わらないことが分かる。これは節 4.2 で述べたように GPU 並列アプリケーションは GPU の電圧のみを測定しているため、CPU で演算をしている逐次処理部分はの実行時間は GPU の周波数に影響を受けないためである。QT Clustering では 3 つのフェーズがあり、mummergpu 同様に逐次処理部分は DVFS の設定によらずほぼ同じ実行時間になっている。

次にスピードアップのグラフについてであるが、全体的に離散的な周波数しか取れない場合の方が、連続的に周波数を変化できる場合に比べて効果は大きかった。また離散的な場合は電力制約ごとにスピードアップの変動が大きく、大きくスピードアップする場合もあれば全くスピードアップしない場合もある。連続の場合では、離散的な場合のようにスピードアップの値が大きく変動することはなかったが、やはり電力制約に応じてある程度の変動は見て取れる。

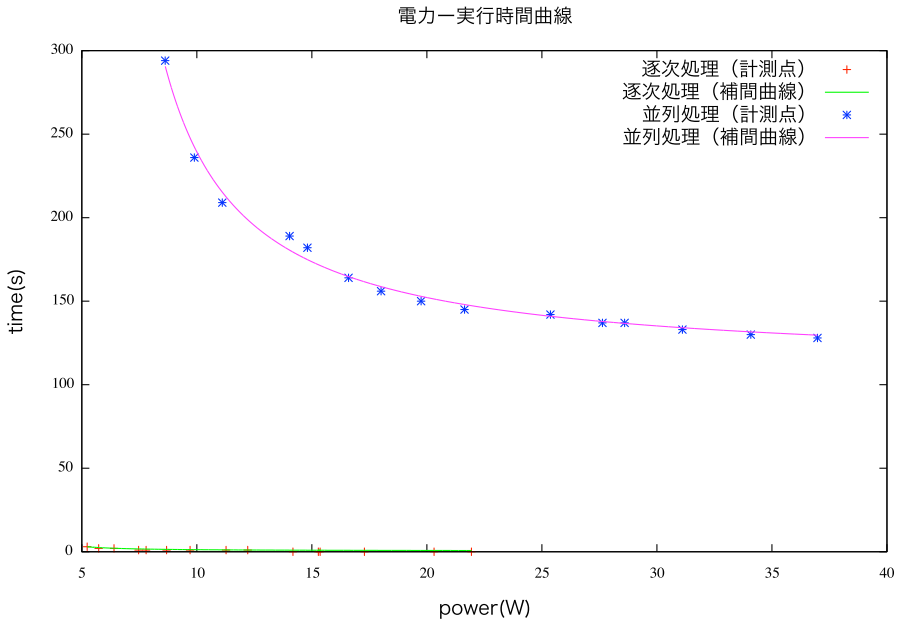

図 4.4. Stream Cluster 電力ー実行時間曲線

ワークロード名	スピードアップ平均
PARSEC Canneal	2.2228332480274293%
PARSEC Stream Cluster	0.018911137755933463%
Rodinia mummergpu	10.4436532563669 93%
SHOC QT Clustering	2.5042232710450536%

表 4.3. スピードアップの算術平均（周波数が連続の場合）

4.4　考察

4 つのアプリケーションの中で、Stream Cluster のみは蓄電池を用いた電力融通にスピードアップがとても小さかった。これは、逐次処理部分の実行時間が並列処理部分の実行時間と比べて非常に短かったため、蓄電池が充放電する時間がほとんどなく、電力融通を上手く行うことができなかったためであると考えられる。

4.4.1　周波数が連続的な値を取れる場合の考察

連続的に周波数が変更できる場合にどのような周波数が最適解となっているかについて考察する。ここでは比較的説明のしやすい、Canneal の連続補間した場合の結果について説明する。

図 4.11、4.12 はそれぞれ、周波数が連続的に変えられる場合の Canneal の電力ー実行時間

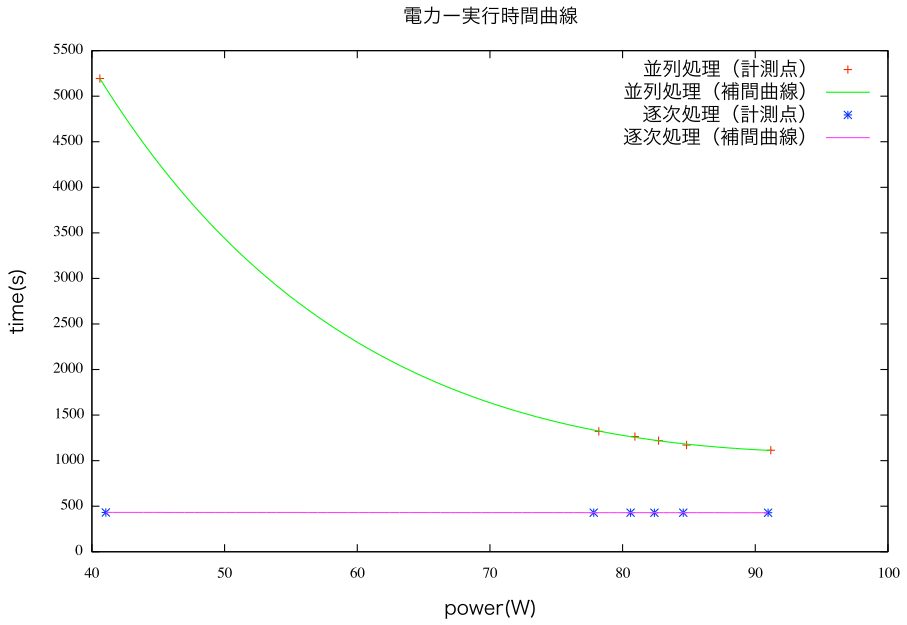

図 4.5. mummergpu 電力ー実行時間曲線

曲線とスピードアップグラフである。図 4.11 を見ると、電力制約が 17W の部分からスピードアップが増加し始め、28W 付近でピークに達しその後は減少していくのが分かる。

まず、電力制約が 17W より小さいときの最適周波数について考える。図 4.12 を見ると分かるように、この範囲においては、逐次処理の周波数を下げて並列処理の周波数を上げている。つまり、逐次処理部分は電力を上げても実行時間があまり短縮されないフェーズであり、逆に並列処理部分は電力を上げることで実行時間が大きく短縮されるフェーズであったということである。ただしここで注意すべきは、蓄電池によって融通しているものは電力ではなくエネルギであるという点である。フェーズ A で 10W で充電できたからといって、フェーズ B で 10W で放電できるとは限らない。これはそれぞれのフェーズの実行時間が異なるため、充電できるエネルギに差が生じるためである。しかし、節 3.4 で述べたような理想的な蓄電池の場合にはフェーズ A で 10J 充電できた場合はフェーズ B で必ず 10J 放電することができる。つまり、それぞれのフェーズにかけるエネルギを微小変化させたときの、実行時間の短縮の大きさを見てやればよいことになる。これは、実行時間を実行エネルギで微分ものを正負逆転させたものに他ならない。これをエネルギ実行時間短縮率と呼ぶ。エネルギ実行時間短縮率は式 3.3 を用いて以下のように求められる。

$$E = p * T(p) \tag{4.2}$$

$$-\frac{dT}{dE} = -\frac{dT}{dp} \Big/ \frac{dE}{dp} \tag{4.3}$$

$$= -\frac{1}{a_2 - \frac{a_0}{a_1}(p - a_2)^2} \tag{4.4}$$

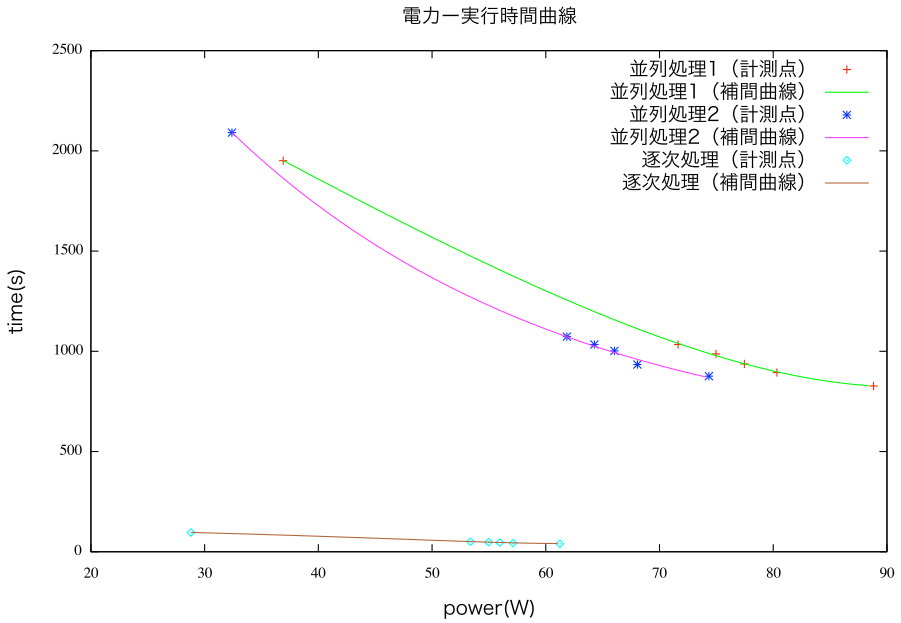

図 4.6. QT Clustering 電力ー実行時間曲線

実際に 17W 以下の部分については、式 4.4 の値がほぼ一致していた。つまりこの部分については、それぞれのフェーズのエネルギ実行時間短縮率の一致する部分が最適解となっている。また、この部分におけるスピードアップは 0.5% 未満とかなり小さい。

次に 17-28W の部分について考える。図 4.12 を見ると分かるように、17W のときに、電力融通をしない場合の逐次処理部分での動作周波数が頭打ちになり、実行時間が短くならなくなる。一方で電力融通を行う場合には頭打ちになるのがもう少し遅く、頭打ちになってもその部分で余った電力を並列部分へ融通して周波数を上げることができるので、実行時間は短くなり続ける。そのため実行時間の差は開き続け、スピードアップが大きくなり続ける。

この電力制約における最適周波数での式 4.4 の値を見ると、逐次処理の値の方が小さい。これは逐次処理の方がエネルギ実行時間短縮率が大きいことを意味しているため、もし逐次処理の周波数をさらに上げることができれば、並列処理部分での動作周波数を下げ、代わりに逐次処理部分では動作周波数を上げていたと予想される。

最後に 28W 以上の部分について考える。この部分についての説明は容易である。図 4.12 から、28W のときに電力融通を行っている場合には逐次処理・並列処理のどちらのフェーズも動作周波数が頭打ちになる。従って、電力融通をしている場合にはこれ以上実行時間を短縮する余地がない。一方で電力融通していない場合には、並列処理部分の動作周波数をまだ上げることができるため、実行時間が短縮されていく。結果としてスピードアップは減少していく。

以上それぞれのフェーズにおける考察より、プロセッサが連続的な周波数で動作できる場合にはエネルギ実行時間短縮率が高いフェーズから低いフェーズへできる限り電力を融通できるような充放電計画が最適となることが分かる。また、17W 以下の部分のスピードアップを見

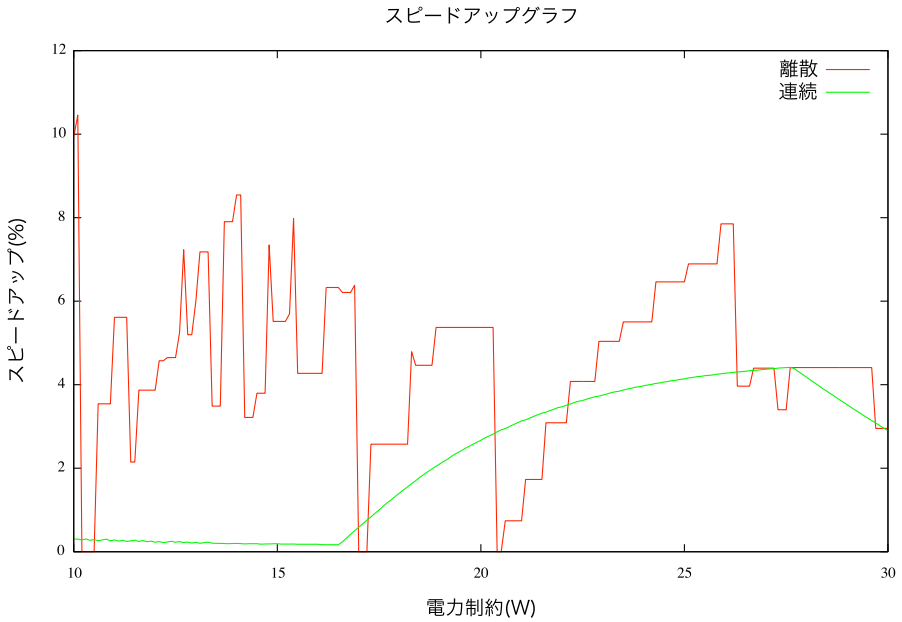

図 4.7. Canneal スピードアップグラフ

れば分かるように、電力ー実行時間曲線の特性の違いによって生じるエネルギ実行時間短縮率の違いを利用した電力融通だけでは効果は小さい。一方で 17W 以上の部分のように、一方のフェーズにおけるプロセッサの性能が頭打ちになることによる余剰電力などがあれば大きな性能向上が見込める。

4.4.2 周波数が離散的な値のみしかとれない場合の考察

離散の場合には電力制約によってはところどころ針が立ったように大きくスピードアップしている部分がある。これはもう少し電力制約が大きくなればもう一つ上の周波数に上げることができる、という状況で起こるものである。逐次処理でわずかに蓄えたエネルギを並列処理部分に融通することで、周波数を一段階上げることができたのである。周波数が連続的に変えられる場合には、周波数を数段階上げる程度では大した時間短縮にはならないため、このようなことは起こらない。

全てのアプリケーションを通して、離散的な周波数しか取れない場合の方が連続的に周波数を変化させられる場合に比べて効果は大きかった。これは節 3.1 で述べたように、周波数が離散的であるために生じる余剰電力を蓄電池を用いて融通することによる性能向上効果のためであると考えられる。

全体的に、離散的な周波数しか取れない場合には電力制約ごとにスピードアップの値が大きく異なる。これまでに述べたように離散的であると余剰電力は多く生じるのだが、周波数を一段階上げるために必要なエネルギ融通量も大きくなる。また、周波数を一段階上げたときに短縮される実行時間も大きい。そのため、上手く周波数を上げられるような電力制約の場合は大

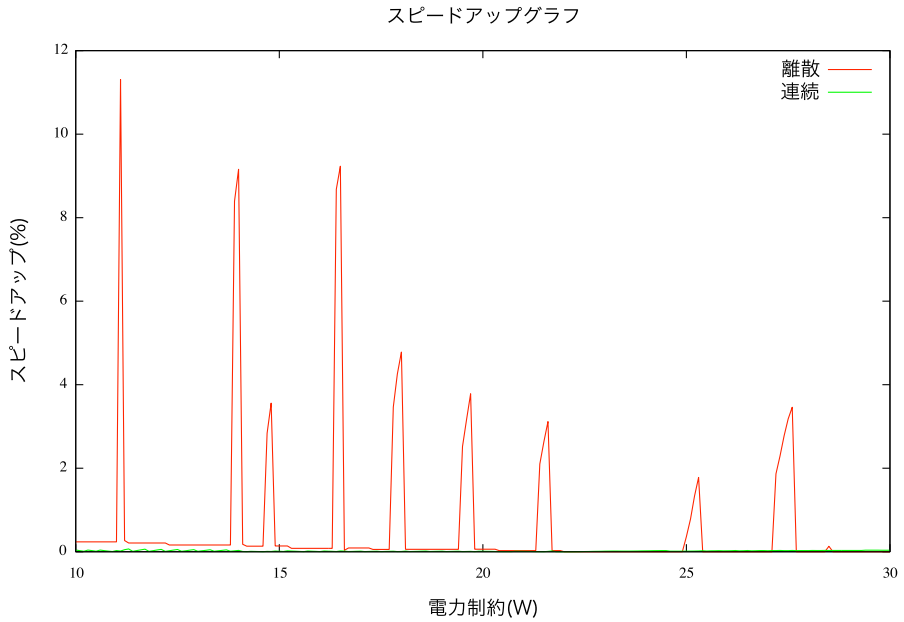

図 4.8. Stream Cluster スピードアップグラフ

きくスピードアップする一方で、逆に周波数を上手く上げられないような電力制約ではほとんどスピードアップさせることができないのである。

以上のことをまとめると、周波数が離散的である場合に効果があるのは、電力制約がサポートされているいずれかの動作周波数よりわずかに小さいような場合である。この場合には他のフェーズの余剰電力を蓄電池で融通することで周波数を上げることができる可能性が高いため、実行時間が大きく短縮される。逆に効果が薄いのは電力制約がサポートされているいずれかの動作周波数と同じか、わずかに大きい場合である。この場合には次の周波数まで大きなギャップがある上、余剰電力もあまり生じないため電力融通を上手く行うことができない。

今回の実験では電力融通問題を解くために全探索で最適解を求めた。n 個のフェーズがあるアプリケーションを、m 個の動作周波数をサポートしているプロセッサ上で走らせた場合の最適解を全探索で求めると、その計算量は $\mathrm{O}(m^n)$ となる。これは非常に大きな計算量であるため、新たなアルゴリズムの構築が求められる。節 4.4.1 で述べたようなエネルギ実行時間短縮率を指標とした計算量の小さなアルゴリズムの考案が課題となるであろう。また今回は理想的な蓄電池を想定したが、蓄電池の電池容量や充放電速度などの物理的制約を考慮に入れたアルゴリズムの構築も課題である。

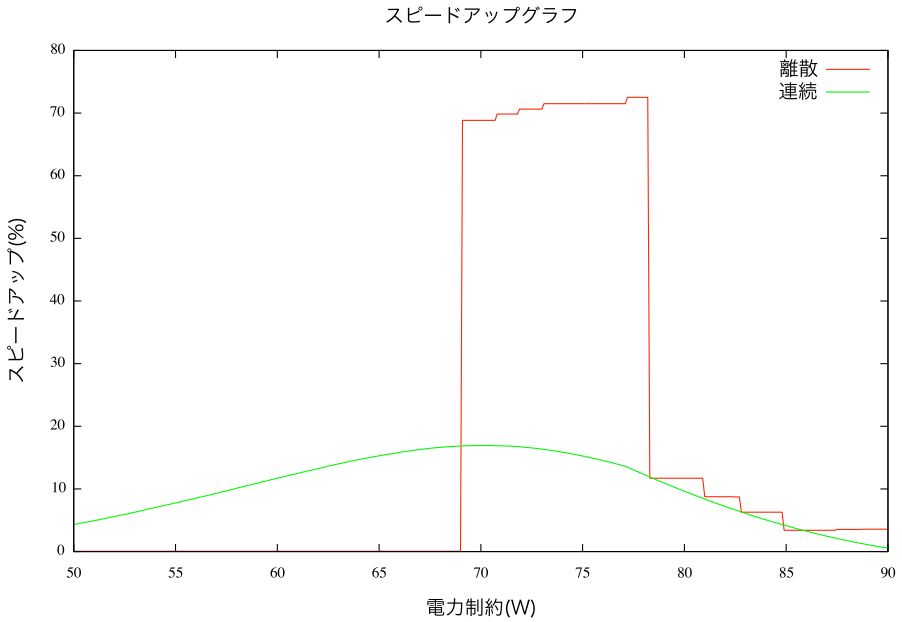

図 4.9. mummergpu スピードアップグラフ

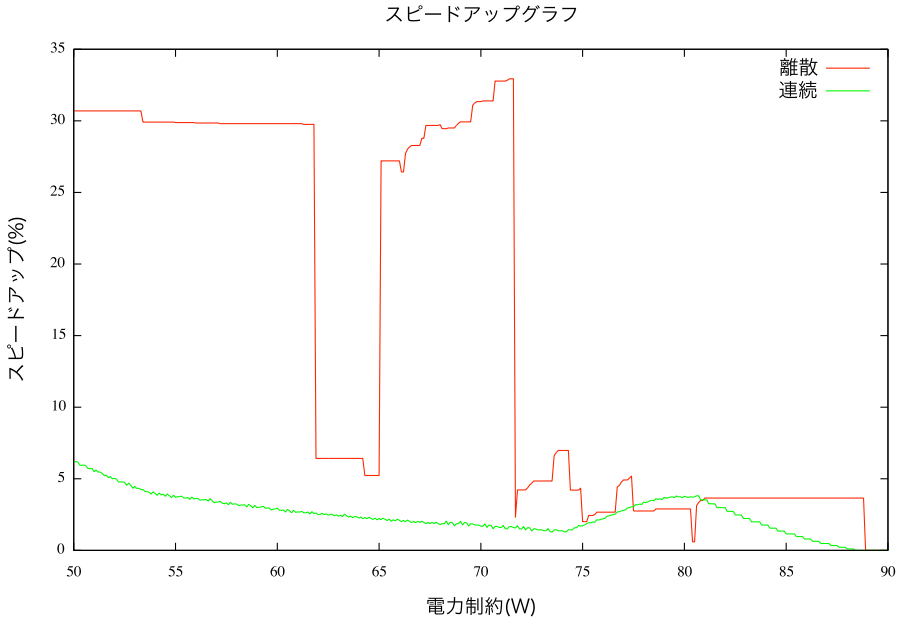

図 4.10. QT Clustering スピードアップグラフ

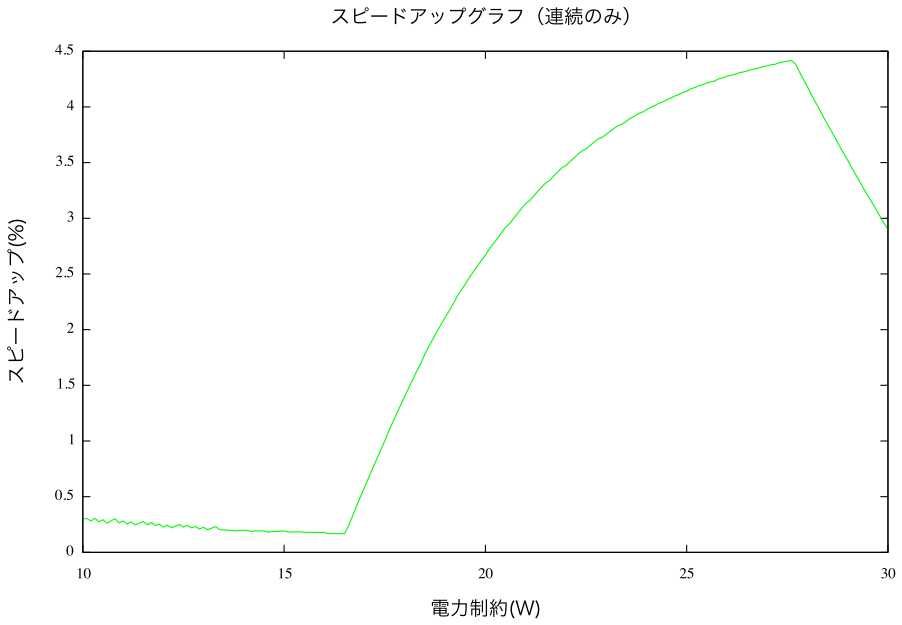

図 4.11. Canneal スピードアップグラフ（連続の場合のみ）

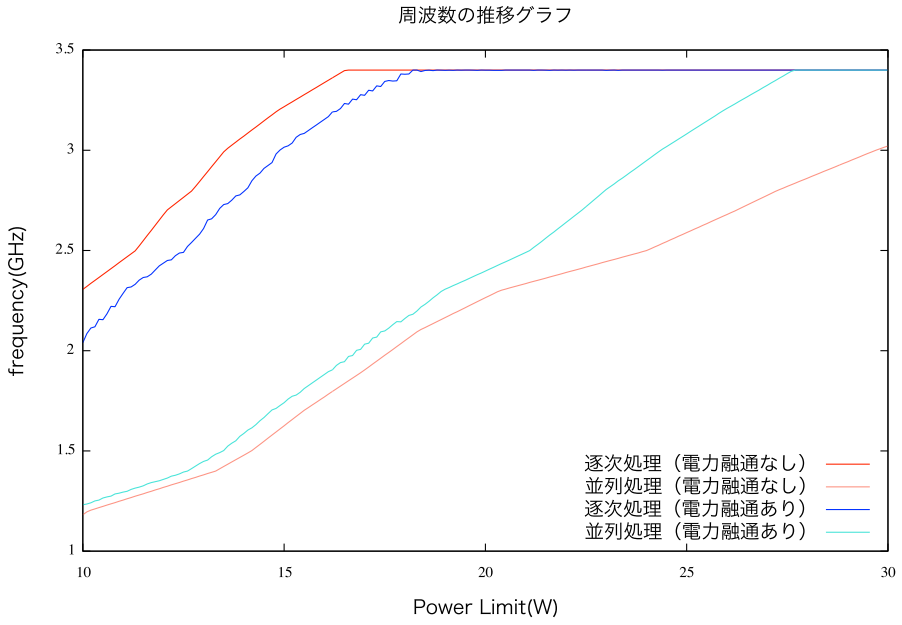

図 4.12. Canneal 周波数の推移の様子

第 5 章

結論

本研究では一つのノードを占有して実行されるアプリケーションを想定した。ユーザにアプリケーションの異なる処理を行っている区間の区切りを示すコードを埋め込ませることで、アプリケーションを電力ー実行時間曲線が異なる複数の区間に分割した。そして、分割された区間の中で電力をかけても実行時間があまり短縮されない区間から電力をかけると実行時間が大きく短縮される区間へ、蓄電池を用いて電力が融通できるように充放電計画を立てることにより、電力制約を守りながら実行時間が短縮することを提案した。本提案手法の適用により、同程度の実行時間であるフェーズを複数持つような CPU 並列のアプリケーションでは平均 4.5%、GPU 並列アプリケーションでは平均 15% 程度の実行時間の短縮がなされることを確認した。

本提案手法では、電池容量・充放電速度が無限大の理想的な蓄電池を想定している。また、充放電計画の決定アルゴリズムとして全探索を用いており、上記の結果は理想的に電力融通がなされた場合の結果と言える。しかしアプリケーションを分割する部分では、第三者である著者がソースコードを読むことによってベンチマークアプリケーションの異なる処理部分を分割しているため、より良くアプリケーションの性質を理解しているアプリケーション作成者自身が分割した方が効果が高いと考えられる。そのため、フェーズ分割の精度向上によるさらなる性能向上の余地が残されている。

本手法を実際のシステムに適用するまでには、蓄電池の電池容量・充放電速度・最小放電間隔や寿命などの物理的制約を含めた、本手法よりも計算量の小さなアルゴリズムの構築が必要である。この実現のためには 4 章で定義したようなエネルギ実行時間短縮率などの評価関数を用いることが有用であると考えられる。

謝辞

本研究を進めるにあたり、ご指導を頂きました中村研究室のスタッフ・先輩方に深く感謝致します。お忙しい中、ミーティングの中で日々の卒業研究へのご指導を頂きました中村宏教授に心から感謝致します。また、理論的な考察や手法についてや研究の方向性への様々なアドバイスを下さった三輪忍助教、些細な質問にも丁寧に答えて頂き、日々の相談にものって下さった中田尚特任助教両名に深くお礼申し上げます。

薦田先輩には博士論文や審査でお忙しい中、毎日のように私の研究にとても多くの時間を割いてお世話頂き本当に感謝しております。薦田先輩なしでは今の私の卒業論文はなかったでしょう。有間先輩には数学的な手法や理論についてアドバイス頂きまして、大きな助けになりました。會田先輩にはプログラム関連の有用なアドバイスを頂きまして、本当に助かりました。皆様に心から感謝致します。

参考文献

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electonics*, pages 114–117, April 1965.

[2] Sriram Govindan, Anand Sivasubramaniam, and Bhuvan Urgaonkar. Benefits and limitations of tapping into stored energy for datacenters. *SIGARCH Comput. Archit. News*, 39(3):341–352, June 2011.

[3] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 173–182, New York, NY, USA, 2013. ACM.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[5] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 31–40, New York, NY, USA, 2011. ACM.

[6] Wonyoung Kim, M.S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134, 2008.

[7] Qingyuan Deng, D. Meisner, A. Bhattacharjee, T.F. Wenisch, and R. Bianchini. Coscale: Coordinating cpu and memory system dvfs in server systems. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 143–154, 2012.

[8] Efficient Data Center Summit 2009. `https://www.google.com/about/datacenters/efficiency/external/2009-summit.html`.

[9] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting cen-

ters. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 303–314, New York, NY, USA, 2005. ACM.

[10] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: Coordinated multi-level power management for the data center. *SIGARCH Comput. Archit. News*, 36(1):48–59, March 2008.

[12] L. Ramos and R. Bianchini. C-oracle: Predictive thermal management for data centers. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 111–122, Feb 2008.

[13] Xiaorui Wang and Ming Chen. Cluster-level feedback power control for performance optimization. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 101–110, Feb 2008.

[14] L. Ganesh, J. Liu, S. Nath, G. Reeves, and F. Zhao. Unleash stranded power in data centers with rackpacker. In *Proceedings of the Workshop on Energy-Effcient Design (WEED)*, 2009.

[15] Justin Moore, Jeff Chase, Parthasarathy Ranganathan, and Ratnesh Sharma. Making scheduling "cool": Temperature-aware workload placement in data centers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 5–5, Berkeley, CA, USA, 2005. USENIX Association.

[16] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. *SIGARCH Comput. Archit. News*, 35(2):13–23, June 2007.

発表文献

[1] 組込み研究会

付録 A