Vol.2014-HPC-143 No.23
2014/3/4

ロードバランスを考慮した 電力制約下におけるCPUのDVFS制御

會田 翔[1,a)]　三輪 忍[1]　中村 宏[1]

概要：現在の大規模計算環境においては，実質的に利用可能な電力の限界が見えてきたことで電力制約が性能向上の妨げとなっている．近年そういった環境下において必要となる性能対電力のトレードオフの調整を行う技術が発達してきた．本稿では，電力効率改善のため，CPU 間のロードバランス改善のためのランタイム制御手法を提案し，最大 21.3%の性能向上を達成した．

キーワード：省電力, 電力制約, ロードインバランス

1. 序論

近年の HPC 環境において，電力の問題は最大のテーマである．京コンピュータは 12.66MW の電力容量で 10.51PFLOPS を達成している [11] が，この値は既に一つのデータセンタに供給可能な電力の現実的な限界に近づいており，今後も 20MW 程度までしか増やせないという見積もりの上で開発が行われている [4]．この厳しい制約下で Exa-Scale を達成するには電力あたりの性能を数十倍向上させることが必須となる．そのためには数多くのモジュールについて高効率化，省電力化が求められる．また，電力制約によってデータセンタに搭載されるモジュール全てを同時に動かすことができなくなると，各コンポーネントに対して電力を割り当て，その電力の範囲内で動作させることとなり，これをパワーキャッピングと呼ぶ．さらにパワーキャッピングを行いつつ電力の動的な変更，すなわちアプリケーション及びアプリケーション内の実行フェーズに応じて，その都度性能向上に有益なモジュールにのみ電力を集中的に投入するというパワーシフティングという考え方が出てくる．

本稿ではパワーシフティングの一環として DVFS を用いた CPU 間のロードバランス調整に着目する．ロードインバランス状態では同期待ちの計算資源が発生し，実効的な処理を行わない上にスピンウェイトによって無駄な電力消費が発生してしまうため，ロードバランスをとることは電力効率の観点から見て重要である．また，電力制約下においては全 CPU を最高周波数で動かせない環境が想定される．そこで，ロードインバランスを検知してプロセッサ間のパワーシフティングを行い，一部の処理量の多い CPU に高い周波数を割り当てて性能を向上させる．本稿の構成は以下の様になっている．2 章では，提案手法の開発方針について述べる．3 章では，DVFS を行う際の指標について述べる．4 章では，3 章で述べた指標から周波数を決定するアルゴリズムについて述べる．5 章では，制御ランタイムシステムの内容を述べる．6 章で評価を行い，7 章で関連研究について紹介し，8 章で結論を述べる．

2. 開発方針

本研究の最終目標は，制約条件として CPU 全体の電力制約が与えられたときに，実行するアプリケーションの性能最大化，すなわち実行時間の最小化を実現する CPU の DVFS 制御を与えることである．各ノードに電力予算を配分し，そこからさらに各 CPU に電力予算を配分する．

開発目標として，既存のハードウェアで実現可能であること，ユーザビリティを考慮しソースコード改変を禁止することを掲げた．前者において問題となるのは，各 CPU 内でコア間の周波数が同一でなければならないという制約がある CPU の存在である．以下では，周波数変更を行う際には全てのコアが影響を受けるという環境を想定して記述する．そうでない場合は，コア数分の CPU がありCPU 内は 1 コアであると仮定して計算を行うことにより同等の解を得ることができる．後者の制約により，アプリケーションの詳細な状態遷移を把握することができない．代わりにアプリケーションのリンク時にライブラリのリンク追加及び実行時のランタイムシステムの各ノード 1 プロセス追

[1] 東京大学
The University of Tokyo
[a)] aita@hal.ipc.i.u-tokyo.ac.jp

加，テスト実行によるデータ収集というアプローチを取った．このアプローチではランタイムの動作により発生するオーバヘッドの問題からランタイム処理を軽量化することが求められ，テスト実行はできるだけ短くて済むことが求められる．提案手法では，アプリケーション毎に必要なテスト実行をフェーズ推定 (後述) 用の数秒間にとどめた．

3. 処理量の見積もり

3.1 ロードバランスとクリティカルパスとの関係

実行のクリティカルパスに乗っていて実行のボトルネックとなっているノードを判定し，優先的に電力予算を与えて DVFS により周波数を上昇させることで，効率的に性能が向上する．よって，周波数制御によってロードバランスをとるためには，ある CPU で行われている処理がクリティカルパスに乗っているかどうかを判定したい．それを判定するための指標が以下で定義するクリティカリティである．

3.2 クリティカリティの導入

3.2.1 コアの稼動状態

時刻 t におけるコアの稼働状態 $w(t)$ とは 2 値関数であり，対象アプリケーション実行中かつ同期待ちや他プロセスとの送受信でない状態の時に 1，それ以外の場合 0 をとる．つまり，$w(t) = 0$ の時はコアの周波数が低くてもアプリケーションの性能に影響を及ぼさないことを意味する．$w(t) = 1$ の時を Active 状態，$w(t) = 0$ の時を Wait 状態と呼ぶ [*1]．コアの稼働状態はブロッキング通信や同期が発生する時だけ変化するので，通信関数をフックすることでイベント駆動型で効率良く状態変化を検知できる．

3.2.2 タスク量

次にタスク量という概念を導入する．スレッド i のタスク量とはスレッド i が割り当てられているコア j の周波数，稼働状態の積の時間積分である．あるコア j の時刻 $t_0 \leq t \leq t_1$ における周波数変動が $f_j(t)$ で与えられ，コア j の稼働状態が $w_j(t)$ で与えられる場合，t_0 から t_1 のタスク量 $T_i(t_0, t_1)$ を求める式は

$$T_i(t_0, t_1) = \int_{t_0}^{t_1} f_j(t) w_j(t) dt \tag{1}$$

となる．特にコア j が周波数 $f_j(t) \equiv f_0$ 一定で常に Active 状態である場合，

$$T_i(t_0, t_1) = (t_1 - t_0) f_0 \tag{2}$$

と簡潔に表現できる．

[*1] 上記の定義で $w(t) = 1$ となる場所でもメモリアクセスの部分では CPU の周波数が低くても問題ない．今後の課題として，キャッシュアクセスタイミングの参照などによってこの部分の切り分けを行うことでメモリインテンシブアプリケーションに対してもより良い制御を行うことが挙げられる．

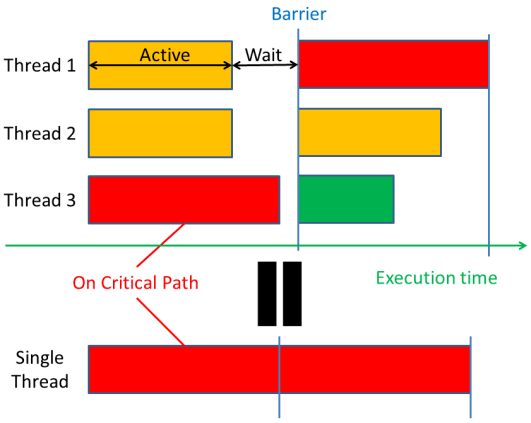

図 1 並列処理のクリティカリティ

3.2.3 逐次処理に対するクリティカリティ

逐次処理 i に対するクリティカリティ C_i とは，並列処理のうちのある 1 つの処理の流れに着目し [*2] その処理が割り当てられたコア j を最低周波数 f_{min} で動かし，かつ他の処理に対する同期待ちや OS スケジューリングによる他のプロセスの同一コアへの割り当てが無い場合の逐次処理 i を実行するのにかかる時間である．これは i の全タスク量 T_i が既知 ($= T_{K_i}$ とする) の場合，

$$C_i = \min_t \{t | T_i(0, t) = T_{K_i}\} \tag{3}$$
$$= T_{K_i} / f_{min} \tag{4}$$

という式を導くことができる．式 (3) は全タスクが終わった直後の時刻を min として表現しており，式 (4) はコアが周波数一定かつ実行が終了するまで $w(t) = 1$ であることから式 (2) を適用した．式 (4) から，システムの最低周波数 f_{min} が定数として決定してしまえばクリティカリティとタスク量とは本質的に同じ量をあらわすことが示された．

3.2.4 並列処理に対するクリティカリティ

並列処理におけるクリティカリティとは，対象となっている全並列処理が別々のコアに割り当てられている前提で全コアを最低周波数 f_{min} で動かし，対象以外の処理の同期待ちや OS スケジューリングによる他の処理の同一コアへの割り当てが無い場合の全並列処理の実行が終了するのにかかる時間である．図 1 に概念図を示す．このように，並列処理間で同期待ちがある場合は複数処理の中で最も遅い処理を繋ぎ合わせた逐次処理に対するクリティカリティと同等のクリティカリティを持つ．対象外の並列処理との同期があった場合 [*3] は実際の実行ではどのスレッドもクリティカルパスに乗っていなかったとしても他の並列処理との同期待ちが無い場合という定義から対象内で最も遅い処理がクリティカルパスに乗った結果となる．図 1 の様な

[*2] 例えばマルチスレッド処理ならばそのうちの 1 スレッドを対象とすることに等しい．
[*3] 例えば 1 つの CPU 内をクリティカリティ計算対象としている時に，他ノードの CPU の処理とも同期待ちを行っている場合など．

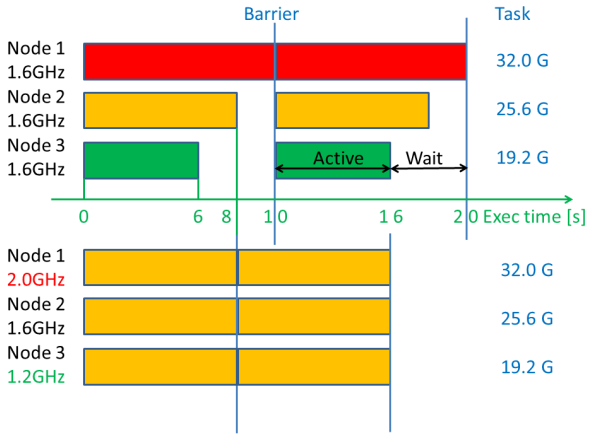

図 2　クリティカリティに基づく周波数制御による性能向上例（実行時間がタスク量に比例し周波数に反比例する理想的な場合）

クリティカリティを求めるために，積分的なアプローチ，すなわち細かい時間単位で分割した上で対象の全並列処理のタスク量を求め，各時間単位の最大値を対象の全時間単位について合計することによって求める．これにより同期タイミングが不明であったり，全計算資源が一斉に同期を取らない隣接通信のような場合にもクリティカリティを求めることが可能となる．

3.2.5　クリティカリティとクリティカルパスとの関係

タスク量ないしクリティカリティのプロセス間の比はプロセスが割り当てられたコアで実行された実効的な処理量の比を表していると考えられる．クリティカリティの高いプロセス程，処理がプログラム中のクリティカルパスに乗っている確率が高いと考えられるため，より高い周波数を割り当てる．**図 2** はクリティカリティを基に周波数割り当てを効率的に行える理想的な例である．このように，実効的な処理量の比が常に変化しない CPU インテンシブな処理の場合，タスク量の比の通りに周波数を割り当てればロードインバランスが解消される．

4.　周波数制御アルゴリズム

4.1　周波数決定の方針

基本的には電力予算内でノードのクリティカリティ比とノードに割り当てる周波数比とが等しくなるように配分したい．ただし，CPU の周波数が離散的であったり，最大最小周波数の制約からクリティカリティの値をそのまま周波数にあてはめることはできない．そこで，**Algorithm 1** に示した貪欲法によって周波数割り当てを導く．

4.2　周波数決定の過程で導入した近似

最初に周波数決定を単純化するため 2 つの仮定を置いた．一点目は周波数と電力との関係についてである．CPU の周波数として設定できる値が $f_{min} \leq f \leq f_{max}$ の間で Δf 刻み [*4] で離散的であるものとした．現実に存在する CPU では周波数間隔が 0.1GHz 単位等と設定できるので特に問題とならない．連続的に設定できる場合でも適宜離散値を定めれば適用可能である．二点目は，アプリケーションの性質についてであり，Active 状態の性能はノードの周波数に比例することである．これは計算インテンシブなアプリケーションならば正しい．

Algorithm 1 各ノードの電力予算決定アルゴリズム

N :: Number of CPUs
f_{min} :: minimum frequency for CPU
f_{max} :: maximum frequency for CPU
Δf :: frequency step to increase
$P_f(f)$ :: power consumption of a node with f
P_{Budget} :: power budget in the cluster
$C[]$ :: (INPUT) array of criticality for each node
$P[]$ :: (OUTPUT) array of power consumption for each node
P_{sum} :: sum of power conumption
d :: data consists of $\{ C, f, i \}$
Q :: priority queue of data (top shows maximum d)
for $i = 1$ **to** N **do**
  $P[i] \leftarrow P_f(f_{min})$
  $d \leftarrow \{C[i], f_{min}, i\}$
  push(Q, d)
end for
$P_{sum} \leftarrow N * P_f(f_{min})$
while hasElement(Q) **do**
  $d \leftarrow$ top(Q)
  pop(Q)
  $f_{next} \leftarrow d.f + \Delta f$
  $P_{next} \leftarrow w_f(f_{next})$
  if $P_{sum} - P[d.i] + P_{next} > P_{Budget}$ **then**
    CONTINUE
  end if
  $P_{sum} \leftarrow P_{sum} - P[d.i] + P_{next}$
  $P[d.i] \leftarrow P_{next}$
  if $f_{next} \geq f_{max}$ **then**
    CONTINUE
  end if
  $d \leftarrow \{C[d.i] * f_{min}/f_{next}, f_{next}, d.i\}$
  push(Q, d)
end while
for $i = 1$ **to** N **do**
  $P[i] \leftarrow P[i] + (P_{Budget} - P_{sum})/N$
end for
return $P[]$

4.3　貪欲法

電力を定めた時に動作可能な最大周波数 $f_P(P)$，周波数を定めた時の電力 $P_f(f)$ は予め周波数固定，100%Active 状態時の実測によって求めておき，クリティカリティを入力として Algorithm1 を適用する．周波数制御アルゴリズムと表記した場合，Algorithm1 の出力 P_{out} に対し与えられた電力予算以内で動作する最大の周波数 $f_P(P_{out})$ を計

[*4] 離散的だが等間隔で無い場合でも，Δf を周波数の関数とみなすことでアルゴリズムを適用することが可能である．

算し出力することを指す．プライオリティキュー Q の取り出しでは以下のデータのうち最大値をとるものが対象となる．保持するデータ構造 d_i は，タスク実行にかかると予想される時間 $t_i = \frac{C_i * f_{min}}{f(P_i)}$，周波数 $f_i = f(P_i)$，CPU 番号 i から構成され，順序関係 $d_i < d_j$ は $t_i = t_j$ の時は，$f_i > f_j$ [*5] それ以外の場合は $t_i < t_j$ として定義される．

このアルゴリズムの計算量は，周波数段階数を M とすると，キューからの取り出しは $O(\log N)$ であり，キューに挿入されるのは各 CPU 高々 M 回であるから，最悪計算量は $O(MN \log N)$ となる．計算コストが最大となるのは全 CPU が f_{max} をとるパターンである．

5. 制御ランタイムシステム

5.1 制御システムの構成

制御システムの全体構成，ノード内構成を**図 3** に示す．各ノード上で対象アプリケーションとは別に動くランタイム（マスタ/スレーヴデーモン）を実行し情報収集及び周波数制御を行い，全体の情報をマスタに集約することでノードをまたいだ情報管理を行う．さらに規模に応じノードをクラスタリングし，クラスタ間の電力融通を行うという階層構造をとることでマスタの負荷分散を行う．3.2.4 節で示した並列処理に対するクリティカリティ計算を行うため，固定時間周期（フレームと呼ぶ）のサンプリングを行う．フレーム長は短い程正確なクリティカリティを求められる一方で，計測頻度が上がるためにオーバヘッドが発生するというトレードオフの関係にあるのだが，本稿では数ミリ秒程度を想定する．ノード間のクリティカリティ集計及び周波数決定はそれよりも長い周期（フェーズと呼ぶ）で行う．

5.2 フェーズ間の流れ

本手法ではアプリケーションの典型的なモデルとして，同期待ちを含むループの反復の存在を仮定し，各 CPU の DVFS は前回の反復との間で並列処理に対するクリティカリティが変化しなかった場合に最適となるように行う．すなわち，前のフェーズから得られたクリティカリティを基に 4 章で示した周波数制御を行い複数ノード間のロードバランス改善を行う．周波数制御アルゴリズムを実行するのは，マスタが稼働しているノードであり，スレーヴはマスタにフェーズ単位のクリティカリティ情報を CPU の数だけ送信する．マスタでは得られたクラスタのクリティカリティを入力として Algorithm1 を実行し，計算して得られた各 CPU の電力予算を通信で伝達する．最後に各ノード内で電力予算から割り当て可能な最大周波数を求める．

5.3 フェーズ長推定

同期待ちを含むループの反復の存在を仮定した以上，反復の時間を推定する必要がある．ある 1 つのノードに対し，周波数一定下で一定数のフレーム分 Active 時間の系列を保持し，これに対する自己相関 Corr(τ) （τ:フレームシフト）を計算，ピーク群を検出し，最大の自己相関値をとるシフト長 τ_{max} を求め，フェーズ長とする．ピークから選ぶ際には最低限マスタ-スレーヴ間の通信オーバヘッドが無視できる程度の長さをとる．Active 時間の系列はテスト実行時に数秒間分得る．現状の実装ではテスト実行後オフライン処理を行っているが，今後この部分は本番実行の初期段階などに導入することでオンライン化を行い，テスト実行を不要にすることを目指している．

frameID	アプリケーション開始から何フレーム目か
modeLast	このフレーム中の情報が最後に記録された時以降の状態（Active または Wait）
t_A, t_W	このフレーム中の情報が最後に記録された時までの Active, Wait 時間の合計 [μs]
t_F	フレーム長（全データ共通の定数）

表 1 フレームのデータ構造

存在するフレーム	modeLast	Active 時間	Wait 時間
対象	Active	$t_F - t_W$	t_W
対象	Wait	t_A	$t_F - t_A$
対象以前のみ	Active	t_F	0
対象以前のみ	Wait	0	t_F

表 2 共有メモリのデータから Active, Wait 時間への変換

5.4 システムの実装

5.4.1 データの取得方法

MPI 等の通信ライブラリでは通信関数をユーザ側が再定義できる．これを用いて通信呼び出しが起こる度に同期，非同期の種類及び開始終了時刻をフックし，**表 1** に示すデータを共有メモリに書き込み，デーモンに伝達する．

5.4.2 アプリケーションとデーモンとのデータ交換

共有メモリの運用に際して，ロックによる性能低下を防ぐため，以下のポリシーを適用しロックレスなアクセスを行う．まず，アプリケーション側の各プロセスは別々の領域に書き込みを行い，デーモンはアプリケーション実行前の初期化処理以外読み取り専用かつアクセスするのは 1 フレーム以上過去の情報に限定する．同一ノード内であるから時刻同期は容易であると考えられる．デーモンが表 1 のデータから Active, Wait 時間を得る際は**表 2** のような変換により得る．フレーム情報を含めた共有メモリのデータ構造を**表 3** に示す．フレームはリングバッファとなっており，デーモンの集計間で一周しないように十分に大きな値をとる．フックプロセスがフレーム情報を追加する際はキューの要領で最後尾に追加し，データが一杯の時は最古のデータを捨てる．デーモンが Active 時間を求めるためにアクセスする際は frameID を指定して frameID 以下で最大の値を持つ frame を探索する．これは重複する値を持

[*5] 周波数の低い方が電力増分が小さく周波数増加比率は高いため優先的に上げる．

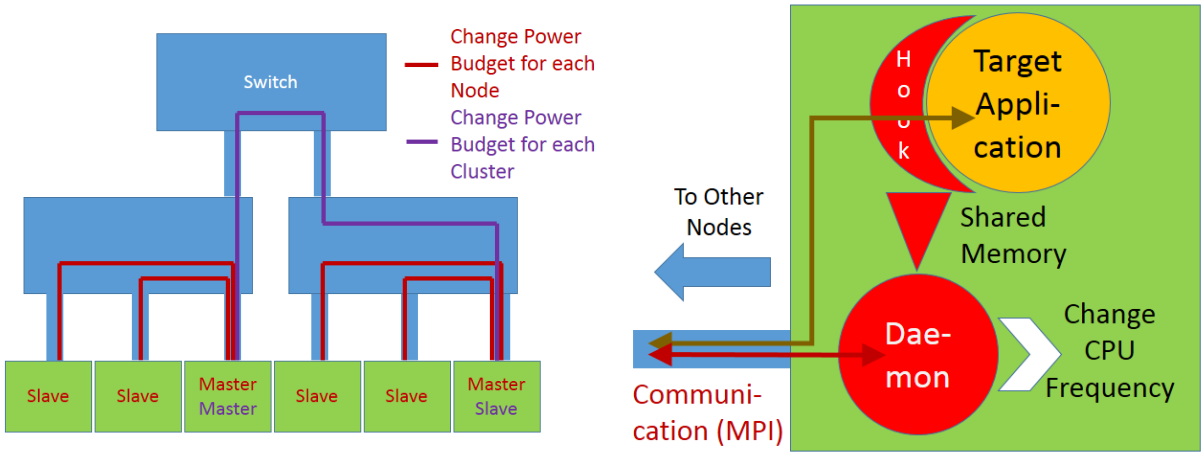

図 3　制御システムの全体構成及びノード内構成

isRunning	ここに実行中プロセスが割り当てられているか
t0	アプリケーションの実行開始時刻
begin, end	フレームリングバッファの先頭，（終端+1）
frame0	バッファ内に存在しない場合のデフォルト値
frame[ ]	フレーム配列

表 3　共有メモリのプロセス単位のデータ構造

たないソート済み配列へのアクセスであるので，2 分探索が行える．探索を行う際，古い部分は更新された変更途中のデータを見てしまう可能性があるため，配列全体を探索せずに一部分だけ（例えば最新 3/4）に限定して探索を行い，その中になければ frame0 を返すといった処理を行う．デーモンは毎フレーム共有メモリにアクセスし，他プロセスの重い処理が挟まって時間的に遅れた場合も遡ってアクセスするので，毎回 2 分探索を行うのはやや非効率的である．そこで直前の呼び出しでヒットしたインデックスをヒントとして与え，近いインデックスを先に探索する．

5.4.3　クリティカリティ収集と電力割り当て

デーモンは毎フレーム全プロセスのタスク量を計算し最大値を加算する．フェーズ間隔でデーモン間通信を行い，マスタならば Algorithm1 を実行する．マスタ計算には数フレームかかるため，送受信間にタイムラグを設ける．

6.　評価

6.1　システム構成

実験環境を表 4 に示す．周波数変更はノード単位でしか変更できない．トポロジは図 4 に示すような Fat-tree を用いた．スイッチ間は 10Gbit イーサネットを 4 本ずつ接続した．4 本という値は，本環境においてリンク本数を変化させたときの実行時間変化を実アプリケーション別に調べた際に，最も多い本数で変化した SPEC MPI 2007 の 130.socorro の結果 (図 5) を基にしている．

6.2　比較対象

even とは全ノード周波数一定で動かした場合である．

Node	Dell PowerEdge r620
CPU	Intel®Xeon®CPU E5-2630L
frequency	1.2 to 2.0 GHz (frequency step : 0.1GHz)
frequency driver	acpi-cpufreq
Number of Cores	6-core 6-thread per node
Number of Nodes	16(8 × 2)
Switch	Dell PowerConnect 8132
Ethernet	ELECOM LD-TWS/BM05,3 (10GbE)
compiler	mpich2-1.4.1p1 gcc-4.8.0
benchmark	synthetic application NPB3.3-BT-MZ class C em2
compile option	-O2 -funroll-loops

表 4　実験環境

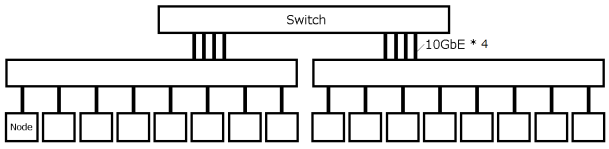

図 4　ネットワーク構成

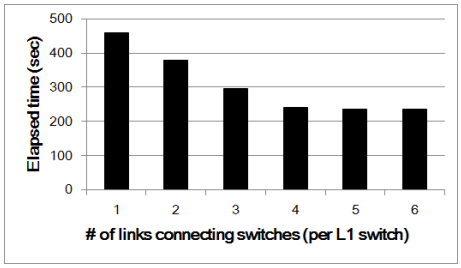

図 5　130.socorro におけるスイッチ間のリンク本数対実行時間

detect とはフェーズ長推定時の状態で，通信関数のフック，共有メモリへの書き込み，及びフレーム単位のデータ収集を行いつつ，全ノード周波数一定で動かした場合である．static とは，本番実行時のプログラム全体のクリティカリティ比を用いて実行前に予め周波数を設定し，実行中は周波数を変化させない手法である．[1] との違いは，ク

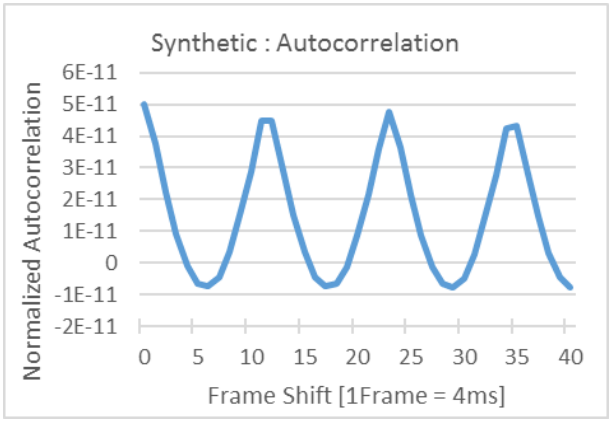

図 6　フェーズ長推定用の自己相関

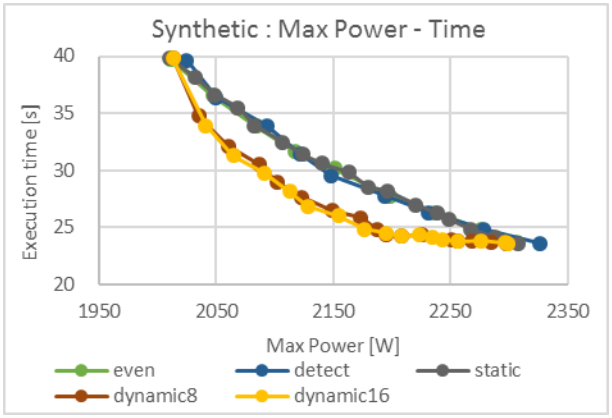

図 7　電力対実行時間

リティカリティを求める際に通信関数のフックを利用している点のみである．dynamic8, dynamic16 が今回提案するランタイムシステムによる周波数制御であり，dynamic16 は全ノードを 1 ノードで集中制御した場合，dynamic8 は 8 台毎にクラスタリングを行い，合計周波数を 2 等分しクラスタ間パワーシフティングをしない場合で評価を行った．

6.3　提案手法に関するパラメータ

フレーム長は 4ms とした．特に断らない場合のフェーズ長は 2.0GHz での実行時におけるに得たデータを用いたピーク推定結果を仮の推定値とし，実行時は平均周波数を f[GHz] とした時に推定値の $2.0/f$ 倍したものを用いた．複数ピーク候補からの選定は 24 フレーム以上のシフト長の中からグラフを目視して選んだ．初期周波数はテスト実行時のクリティカリティを基に行った．

6.4　シンセティックアプリケーションに対する評価

6.4.1　ベンチマークの概要

ノード間で処理量に最大 2 倍の差がありかつ定期的にランダムにシャッフルされるハイブリッド並列プログラムである．実験中乱数の種は固定している．

6.4.2　結果

自己相関は図 6 のように平均 2.0GHz の時に 12 フレーム毎にピークが現れたため，フェーズ長は 24Frame，96ms となるように設定した．図 7 は電力対実行時間の結果である．左上が平均 1.2GHz，右下が平均 2.0GHz で even, detect は 0.1GHz 刻み，他は 0.05GHz 刻みである．detect において，ランタイムが動いていることによる性能低下は見受けられなかった．static では全体のクリティカリティしか追えていないため変動についていけず even とほぼ同等の結果となっている．dynamic8 では平均 1.65GHz で最大 14.1%，dynamic16 では平均 1.6GHz で最大 13.3%の性能向上が達成できた．クラスタリング有無の差もほぼ無いが，同じ平均周波数において 8 ノード単位の方が多少実行時間が遅く最大消費電力は低いというトレードオフの関係

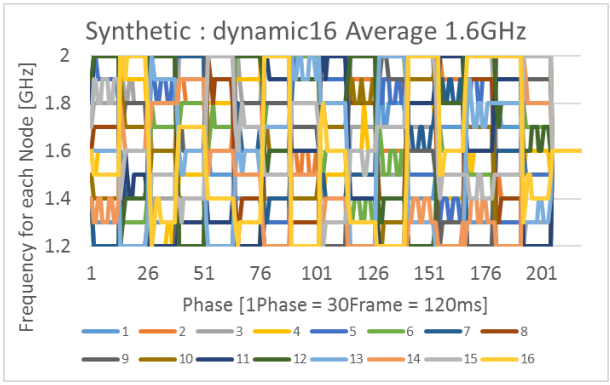

図 8　フェーズ単位の周波数推移

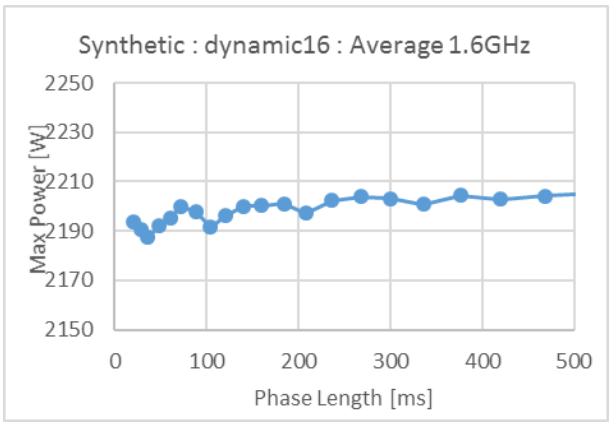

図 9　フェーズ長変化に対する最大電力

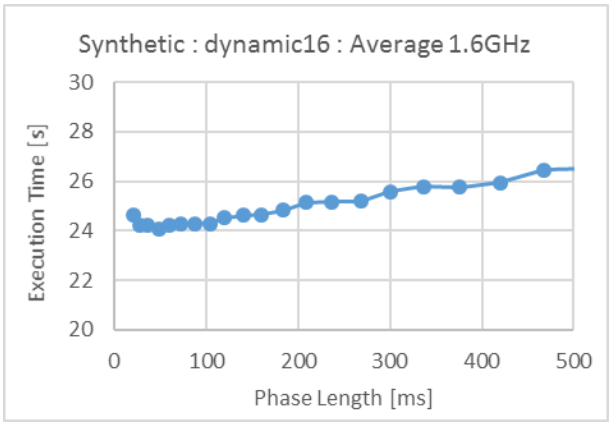

図 10　フェーズ長変化に対する実行時間

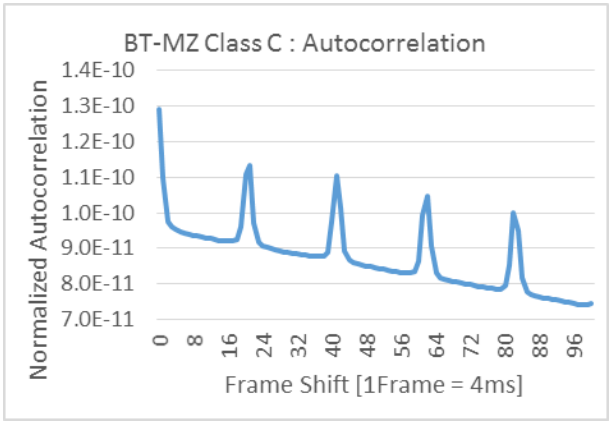

図 11　フェーズ長推定用の自己相関

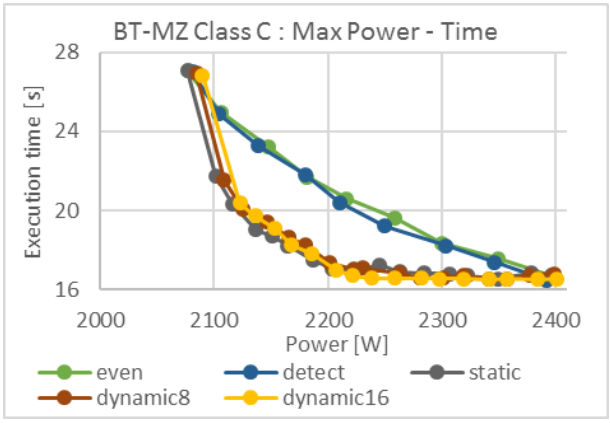

図 12　電力対実行時間

が見られた．図 8 はフェーズ間の周波数推移である．このように，定期的にノード毎の周波数が変化していることが分かる．図 9，図 10 はフェーズ長を変化させた時の電力，実行時間の変化結果で，最短 5 フレームからプロットしてある．最短時間を出したのは 12 フレーム時の 24.04 秒で，平均周波数 1.6GHz でも周波数制御による高速化で 2.0GHz 時の 1 周期分が最善となった．周波数制御後にどれだけの周波数相当の周期に達するかは現状判断できていないため，フレーム推定のオンライン化とともに今後の課題とする．5, 7 フレーム時の性能オーバヘッドは 2.4%, 0.8%程度だった．7 フレームあれば 16 ノードの周波数アルゴリズムのオーバヘッドが無視できるようになったといえる．また，フェーズ長を長く取るとクリティカルパスの変動に遅れをとって実行時間が長くなることが観測された．

6.5　BT-MZ に対する評価

6.5.1　ベンチマークの概要

BT-MZ は NAS Parallel Benchmark[6] の Multi Zone ベンチマーク [7] の一つである．BT は離散化版 3 次元 Navier-Stokes 方程式を解くプログラムであり，それを 2 次元方向に領域を分割して別個に解くのが BT-MZ である．プログラムの反復内では，計算フェーズ及び隣接通信フェーズに分かれており，領域間のロードインバランスにより計算が早く終わってアイドルとなるコアが発生しやすい．

6.5.2　実験条件

問題サイズは CLASS C を用い，プロセス数は 96，スレッド数は 1 とした．

6.5.3　結果

フェーズ長は図 11 から平均 2.0GHz の時に 42Frame, 168ms となるように設定した．図 12 は，最大電力対実行時間を示している．even-detect の比較では，ランタイムの実行による性能の差は見られなかった．static，dynamic8, dynamic16 それぞれ平均 1.35, 1.35, 1.5GHz のとき最大で 22.3%, 19.8%, 21.3% の性能向上が達成された．図 13 は各フェーズにおける各ノードの周波数割り当て結果である．各ノード終始ほぼ一定の周波数が割り当てられており，static とほぼ変わらない結果となったことを裏付けている．このようにクリティカリティが変化しないアプリケーションの場合，フェーズ長はある程度より長くとれば問題ない．

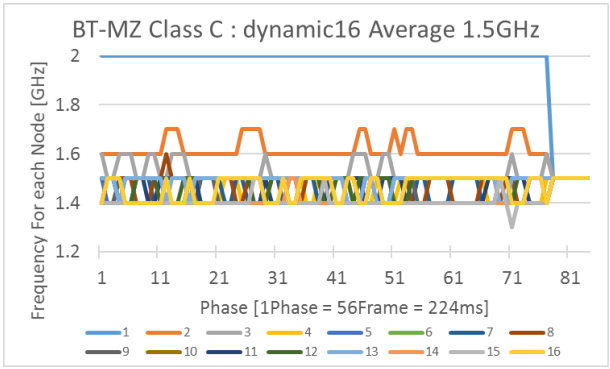

図 13　フェーズ単位の周波数推移

6.6　em2 に対する評価

6.6.1　ベンチマークの概要

em2 は, 名古屋大学太陽地球環境研究所の梅田隆行助教に提供していただいた, 2.5 次元の電磁気に関するアプリケーションである．粒子単位で計算を行っており, 並列化は空間領域単位で行われ, 各領域に存在する粒子数の違いによってロード・インバランスが発生する．

6.6.2　実験条件

16 プロセス ($2 \times 4 \times 2$) 6 スレッドにて実験を行い, 最外ループ数は 16, 空間格子サイズは XY ともに 400, 粒子数は合計 40,000,000 とした．コンパイル時の最適化の影響なのか原因不明だが, フックライブラリをリンクしなかったバイナリではメモリアクセス違反が起きてうまく実行できなかったため, detect との比較を行った．

6.6.3　結果

フェーズ長は図 14 の結果から，2.0GHz において 392Frame，1568ms と推定された．この値は最外ループ数 1 回分に対応している．図 15 は電力対実行時間の結果

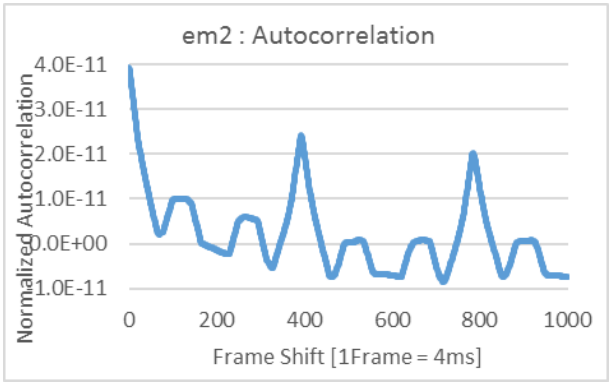

図 14　フェーズ長推定用の自己相関

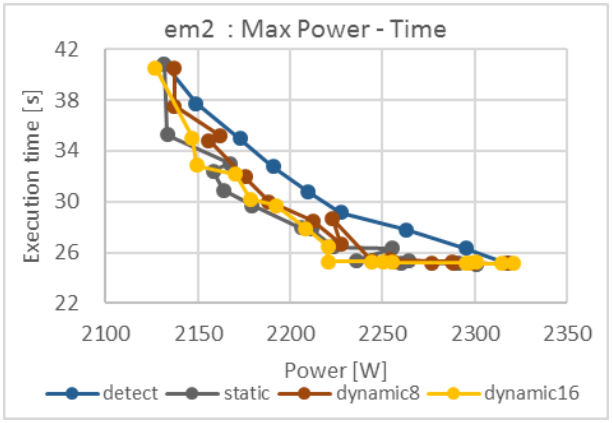

図 15　電力対実行時間の全結果

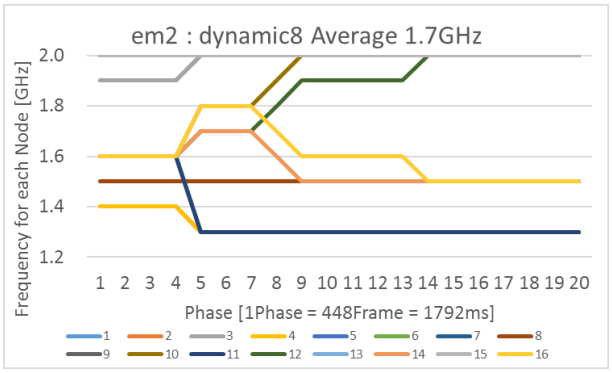

図 16　フェーズ単位の周波数推移

である．detect と比べて，static, dynamic8, dynamic16 それぞれ平均 1.35, 1.75, 1.6GHz のとき最大で 13.3%, 9.9%, 15.4% の性能向上が達成された． **図 16** は，長期的周波数制御によるフェーズ単位の周波数変化である．最外ループ数が 16 なのに 20 フェーズまであるのは，最初の 4 フェーズが初期化処理を行っているためである．BT-MZ と同様に，各ノード毎にばらつきはあるものの，フェーズ毎のクリティカルパスの変化はあまり無いアプリケーションであることが分かる．

7. 関連研究

CPU の DVFS によるロードバランス調整について，以下のような関連研究がある．Ge ら [2] は大規模計算環境において CPU の周波数変更時の性能と電力を複数のアプリケーションに対し評価している．Hsu ら [3] はアプリケーションが遅れるのを防ぎつつも CPU の周波数を自動的に制御するランタイムシステムを提案している．Springer ら [10] は大規模計算環境において電力性能モデルを用いて CPU 周波数制御の最適化を行っている．Rountree ら [8] はロードインバランスなアプリケーションに対する実行モデルを提案し，CPU 周波数制御に適用している．Sarood ら [9] Kappiah ら [5] が各ノードのスラックに基づく CPU 周波数の動的制御システムを提案している．この手法は目標が性能を維持した状態での省電力化にあり，電力制約化における性能最大化という目標とはやや異なる．

8. 結論

本稿では，CPU の DVFS によるロードバランス改善に着目し，電力制約下での性能向上のためのランタイムシステム制御を提案した．実験により，クリティカルパスが動的に変化するアプリケーションに対する性能向上を確認し，全ノードのプロセッサ周波数一定時の性能と比べて BT-MZ で最大 21.3%，em2 で最大 15.4%の性能向上が達成された．今後の課題としては，与えた制御が電力制約を満たせているかという評価，実アプリケーションにおいてクリティカルパスの変化する場合の評価，フェーズ推定のオンライン化，Active 状態時の CPU インテンシブ，メモリインテンシブの違いの区別等が挙げられる．

謝辞　本研究の一部は文部科学省「将来の HPCI システムのあり方の調査研究」事業，及び JST CREST による．

参考文献

[1] 會田 翔, 三輪 忍, 中村 宏, 電力制約下における CPU とネットワークの電力制御協調手法, SWoPP 2013
[2] R. Ge et al., Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters, SC'05,2005
[3] C. H. Hsu and W. C. Feng, A Power-Aware Run-Time System for High-Performance Computing, SC'05,2005
[4] HPCI 計画推進委員会: 今後の HPCI 技術開発に関する報告書 (2012)
[5] N. Kappiah et al., Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs, SC'05, 2005
[6] http://www.nas.nasa.gov/publications/npb.html
[7] Rob F. Van der Wijngaart et al., NAS Parallel Benchmarks, Multi-Zone Versions, NAS Technical Report NAS-03-010, 2003
[8] B. Rountree et al., Bounding Energy Consumption in Large-Scale MPI Programs, SC'07, 2007
[9] O. Sarood et al., A 'Cool' Load Balancer for Parallel Applications SC'11, 2011
[10] R. Springer et al., Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster, PPoPP'06, 2006
[11] http://www.top500.org/