Vol.2016-SE-192 No.10
Vol.2016-EMB-41 No.10
2016/6/3

永久磁石同期モータ電流制御系のための予測制御アルゴリズム並列化

竹松 慎弥[1,a)]　道木 慎二[2]　嶋岡 雅浩[2]　枝廣 正人[1]

概要：近年, 制御システムは大規模・複雑化が進んでおり, マルチ・メニーコアを用いた並列処理の適用が注目されている. しかし, 並列化にはデータ依存関係や負荷分散などを意識した高度なプログラミングが必要とされる. そこで本研究では, 永久磁石同期モータ電流制御系を題材とし, そこで用いられるモデル予測制御アルゴリズムの高速化と並列化を行った. まず, 解空間削減やソートアルゴリズムの改善といった最適化により, 逐次実行を 6 倍高速化した. さらに, モデルの特性を利用した並列分枝限定法を OpenMP を用いて実現し,12 コアで 9 倍の高速化を達成した.

1. はじめに

近年の制御システムは動的かつ不規則に状態が変化するような制御対象を適切に制御するために大規模・複雑化している．また, 自動運転などに用いられる制御などでは状況に応じた素早い判断・対応が必要であり，リアルタイム性も求められている. そのため, 大規模・複雑化している制御システムを高速に処理することが近年求められるようになってきている. これを実現する方法としてマルチ・メニーコアを用いた並列処理の適用が注目されている.

ところが，単にマルチ・メニーコア上で制御プログラムを動かしても性能向上は見込めない．高並列性を実現するには負荷分散やデータ依存関係，ハードウェアに依存する部分などを考慮した高度なプログラミングが必要である．大規模・複雑化する制御システムに対して適用することが求められているが，容易なことではない．

本研究では小型・高効率・高出力を得られるとして近年注目されている永久磁石同期モータ電流制御系を題材とし，ハイエンド制御に対して有効とされているモデル予測制御アルゴリズムの高速化と並列化を図る．まず，事前に逐次実行での処理をできる限り高速化するためにプログラムの性能解析を行い，最適化する．そして，モデル予測制御アルゴリズムの並列化を行う．並列化には共有変数へのアクセスが簡単かつ高速である共有メモリ型並列化を支援する OpenMP を用いる．

[1] 名古屋大学大学院情報科学研究科
[2] 名古屋大学大学院工学研究科
[a)] ts1413@ertl.jp

2. 永久磁石同期モータの制御

本章では，本研究の並列化対象とする永久磁石同期モータおよびその制御アルゴリズムについて説明する [1][2][3].

2.1 変数の定義

永久磁石同期モータの制御についての説明を行ううえで必要な変数をここで定義する．なお，各座標系については 2.3 節を参照いただきたい.

$i_{uvw} = [i_u\ i_v\ i_w]^{\mathrm{T}}$: u-v-w 座標系の固定子電流ベクトル
$i_{\alpha\beta} = [i_\alpha\ i_\beta]^{\mathrm{T}}$: α-β 座標系の固定子電流ベクトル
$i_{dq} = [i_d\ i_q]^{\mathrm{T}}$: d-q 座標系の固定子電流ベクトル
$v_{dq} = [v_d\ v_q]^{\mathrm{T}}$: d-q 座標系の固定子電圧ベクトル
$e_{dq} = [e_d\ e_q]^{\mathrm{T}}$: d-q 座標系の誘起電圧ベクトル
R : 固定子巻線抵抗
L_d : d 軸方向の固定子巻線インダクタンス
L_q : q 軸方向の固定子巻線インダクタンス
K_E : 誘起電圧定数
P_n : 極対数
θ_{re} : 電気角における回転子位置
$\omega_{re} = P_n\omega_{rm}$: 電気角における回転角速度

2.2 モデル予測制御を用いた永久磁石同期モータ電流制御

本論文が並列化する制御システムとして，モデル予測制御を用いた永久磁石同期モータ電流制御系（以下，本モデルと表記）を用いる．本モデルではベクトル制御に基づ

Vol.2016-SE-192 No.10
Vol.2016-EMB-41 No.10
2016/6/3

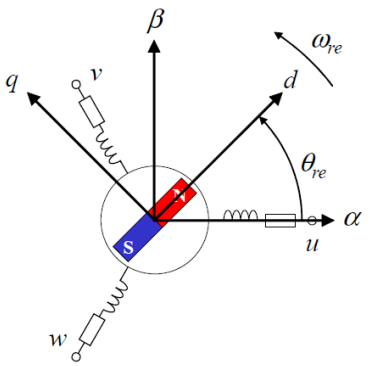

図 1　PMSM の物理モデルおよび座標系（文献 [2] より引用）

き，3 相 PWM インバータのスイッチングを切り替えることで，理想のトルクを得られるように電流 i_d,i_q を制御している．理想的な i_d,i_q を得られるような PWM インバータの最適なスイッチングタイミングの決定にはモデル予測制御を用いており，モデル予測制御の高速化のために分枝限定アルゴリズムを用いている．ベクトル制御，PWM インバータ，モデル予測制御，分枝限定法については以降の節で説明する．

2.3　ベクトル制御

永久磁石同期モータの電流制御には一般的にベクトル制御が用いられる [1][2][3]．ベクトル制御で用いられる座標系は図 1 に示されたとおりである．ベクトル制御は回転子方向に d 軸，それと直行方向に q 軸をとる直行二軸回転座標系を用いて，固定子電流を d，q 成分に分解し，各軸の電流を独立に制御する方法である．d-q 軸を用いることでモータの定常状態における三相交流電流を静止したベクトルとして表現できる．これにより，電圧，電流，磁束の取り扱いが簡単になり，制御システムの設計も容易になる．

図 1 において α-β 軸は u-v-w 軸を d-q 軸に変換するため座標系であり，u-v-w 軸上の電圧，α-β 軸上の電圧,d-q 軸上の電圧には以下の関係がある．また，これらの関係式は電流，磁束においても成り立つ．

$$v_{\alpha\beta} = \sqrt{\frac{2}{3}} \begin{bmatrix} 1 & -\frac{1}{3} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} v_{uvw} \tag{1}$$

$$v_{dq} = \begin{bmatrix} \cos\theta_{re} & \sin\theta_{re} \\ -\sin\theta_{re} & \cos\theta_{re} \end{bmatrix} v_{\alpha\beta} \tag{2}$$

次に，永久磁石同期モータを d-q 軸上の数学モデルとして表現する．永久磁石同期モータの電流を状態量とする状態方程式を離散化したものは次式となる．

$$i_{dq}(n+1) = e^{\boldsymbol{A}\Delta t} i_{dq} + [e^{\boldsymbol{A}\Delta t} - \boldsymbol{I}]\boldsymbol{A}^{-1}\boldsymbol{B}(v_{dq}(n) - e_{dq}(n)) \tag{3}$$

ただし，

$$\boldsymbol{A} = \begin{bmatrix} -\frac{R}{L_d} & \omega_{re}\frac{L_q}{L_d} \\ -\omega_{re}\frac{L_d}{L_q} & -\frac{R}{L_q} \end{bmatrix} \quad \boldsymbol{B} = \begin{bmatrix} \frac{1}{L_d} & 0 \\ 0 & \frac{1}{L_q} \end{bmatrix}$$

$$\boldsymbol{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad e_{dq} = \begin{bmatrix} 0 \\ \omega_{re} K_E \end{bmatrix}$$

また，永久磁石同期モータの発生トルク τ は式 (4) で表される．

$$\tau = P_n K_E i_q + (L_d - L_q) i_d i_q \tag{4}$$

式 (4) より，トルクは電流 i_d および i_q に従い，決定することがわかる．また，式 (3) より，電流 i_d,i_q は電圧 v_d,v_q によって制御できることがわかる. よって，期待するトルクを得られるように電圧で電流を制御する方式がベクトル制御である．

2.4　PWM インバータ

永久磁石同期モータの回転速度は三相交流が作る回転磁界の回転速度，すなわち，電源の周波数に依存する．また，式 (3)，(4) より電圧 v_d,v_q でトルクを制御できることがわかる. よって，固定子巻線に印加する電圧の振幅および周波数を自由に変化させることで，任意の回転速度およびトルクで永久磁石同期モータを回転させることができる．これを実現可能とするのが PWM インバータである [3]．PWM インバータは直流電源の on/off を切り替え，そのパルス幅（デューティ比）を変えることで，擬似的に任意の波形を作り出すことができる．三相交流においては各相に PWM インバータを配置する．

本モデルにおいて 3 相 PWM インバータのスイッチングタイミングを永久磁石同期モータに対する操作と定義し，制御器で最適な操作パターンを算出する．PWM インバータのスイッチングタイミングとしては以下の制約を設ける．

- 制御区間 Tc を設け，1 制御区間内に各相のインバータは 1 回だけスイッチングを行わなければならない．
- 制御区間 Tc は Nc 分割した区間 Ts を設定し，スイッチングは区間 Ts が切り替わるいずれかのタイミングで行われる．

例えば，Nc=20 とし，u，v，w 各相のスイッチングタイミングを 3,18,7 とすると図 2 の電圧スイッチングを意味する．

2.5　モデル予測制御

電流 i_d,i_q を制御するための最適な PWM インバータのスイッチングタイミングをモデル予測制御に基づいて計算する．モデル予測制御は考えうる全ての挙動を予測し，その中から各種制約を満たし，かつ最も良いと思われる操作を選択するという, 最適化計算を制御周期（操作を与えるタイミング）毎に行うというものである [3]．つまり，現在

Vol.2016-SE-192 No.10
Vol.2016-EMB-41 No.10
2016/6/3

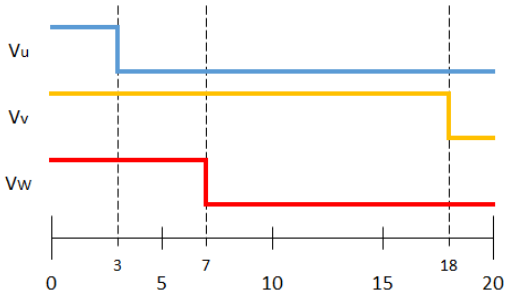

図 2　スイッチング例

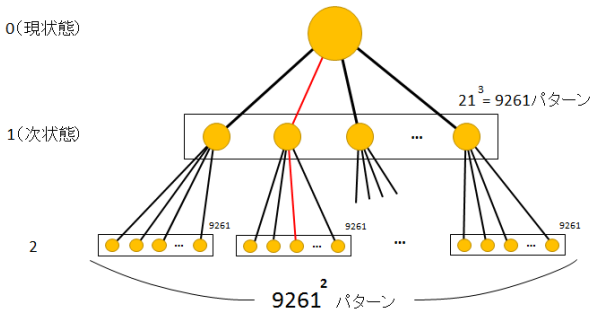

図 3　予測木

の状態を観測・推定し，考えうる全ての挙動を予測，その中から最適な操作を選択，操作を与えるという手順を繰り返す．制約を満たすような操作の中から最適なものを選択する時には，評価関数を用いる．

本モデルにおいて u 相，v 相，w 相の各電圧にそれぞれ Nc+1 のスイッチングタイミングがあるため，1 制御区間で $(Nc+1)^3$ スイッチングパターンが考えられる．さらに Np ステップ先まで予測するので，全パターンとしては $(Nc+1)^{3^{Np}}$ となる．Nc，Np の値としてそれぞれ 20，2 を用いるとすると，図 3 のような木構造の葉ノードから評価値が最適なものをひとつ選ぶという，最適化問題に置き換えて考えることができる．

また，評価関数を次式のように定める．

$$J = J_0 + Wid \times |id\_err| + Wiq \times |iq\_err| \qquad (5)$$

J は電流 i_d,i_q の累積絶対誤差を意味し，J が最小のものを最適解とする．ただし，J_0 をそれまでの評価値，電流 i_d,i_q の誤差をそれぞれ id_err,iq_err とし,id_err および iq_err に対する重みをそれぞれ Wid,Wiq とする．

2.6　分枝限定法

分枝限定法は組み合わせ最適化の最適解を高速に求めるアルゴリズムである [4]．主に次の 2 操作を用いて，問題を解く．

- 分枝操作
  一部の変数の値を固定することで部分問題に分解する操作．部分問題全てを解くことにより，間接的にもとの問題を解くことができる．
- 限定操作
  その部分問題を解いても最適解が得られないことがわかった場合に，その部分問題に対しては分枝操作を行わないこと．

分枝操作によって，部分問題に分けていき，限定操作による枝刈りを行う．これにより，探索数を減らし探索を高速にする．

本モデルにおいて評価値として累積絶対誤差を用いている．そのため，暫定解の評価値よりも計算中のパターンの評価値が大きくなった時点で，それ以上部分問題を解いても最適解が得られないことがわかるため，限定操作を行うことができる．

分枝限定法は並列化しても高速にならない場合がある．逐次実行であれば先に小さい評価値が得られているはずが，並列実行であると順番が入れ替わり，小さい評価値がまだ得られておらず，本来枝刈りされるべきノードを無駄に探索してしまうことがあるからである．並列化にあたり，探索ノード数をどれだけ抑えられるかが非常に重要である．

2.7　最適操作探索手順

最適なスイッチングパターンの探索は以下の手順で行う．

(1) 評価関数計算およびソート
次に考えられる全ての子ノード（スイッチングパターン）の評価関数を計算する．1 パターンの評価値計算が行われる度に構造体連結リストに評価値，パターン番号，電流値を保存する．全ての計算が終わった段階で，クイックソートが実行され，リストのデータは評価値が小さい順にソートされる．評価関数計算途中で評価値が最小値 Jmin を超えた場合，リストに追加しない (分枝限定法の限定操作)．

(2) 子ノード探索
子ノードのひとつを選択し，そのノードに対して再び手順 1 から実行する (分枝限定法の分枝操作)．子ノードの選択はリスト順に行われる．また，Jmin が更新されている可能性があるため，子ノード探索前に再び評価値と Jmin の比較を行い，探索する必要があるかを判定する．葉ノードの場合は次の手順に移る．

(3) 暫定解更新
評価値と Jmin を比較し，評価値が小さい場合，Jmin を更新する．また，そのスイッチングパターンを暫定解とする．

以上の処理が手順 2 の子ノード探索により，全ノードが探索されるまで再帰的に実行される．全ての探索が完了した段階で暫定解となっているパターンが最適解となる．出力としては次の操作だけが必要であるため，最適解にたどり着くために実行する必要がある最初のスイッチングパターンだけが出力される．

手順 1 にてリストを用いてソートを実行する理由としては，子ノードの探索順を評価関数が小さいパターンから順

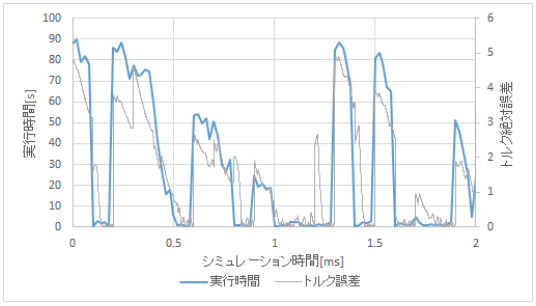

図 4　実行時間の推移

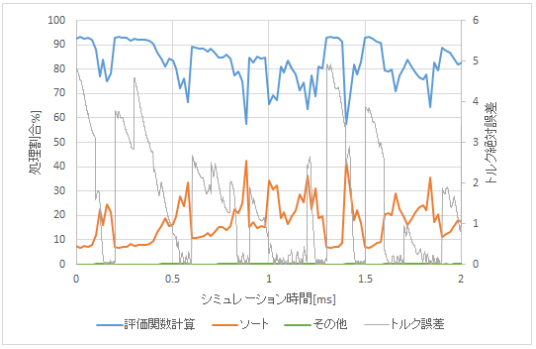

図 5　処理割合

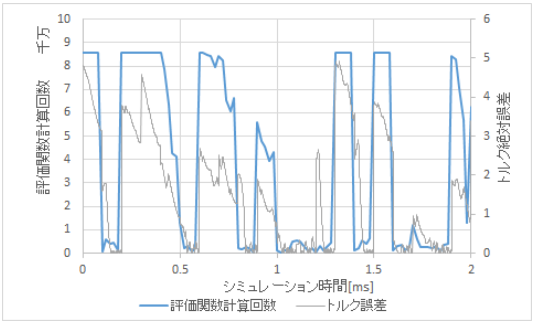

図 6　評価関数計算回数の推移

に探索するためである．つまり，深さ優先探索 + 最良優先探索の探索アルゴリズムとなっている．

3. 性能解析と最適化

本章では性能解析の結果および実施した最適化について述べる．

3.1 性能解析

以下の設定の下，シミュレーションを実行した．

シミュレーション時間 sim_time：2ms
制御周期 Tc：0.02ms
制御区間分割数 Nc:20
予測区間 Np：2
トルク指令値 torque_ref：ランダム
回転速度 wrm：10740（一定）

シミュレーション時間/制御周期=100 なので，100 回操作を与えることになる．よって，100 回最適操作計算が行われることになる．乱数の上限を 7.424，下限を 0 とし，乱数は 0.1ms ごと，つまり 5 回の最適操作計算ごとに乱数が生成される．

まず，プログラム実行時間の推移とトルクの絶対誤差を図 4 に示す．図 4 より，2 つの波形がほぼ同じ形をしていることから，トルク誤差が大きくなっている時にプログラム実行時間が長くなっていると言える．次にトルク誤差が大きい場合にプログラム実行時間が長くなる原因を特定する．処理を評価関数計算，ソート，その他の 3 つに分けた際にそれぞれの処理時間が全体に対してどれほどの割合を占めるかを調べた．その結果を図 5 に示す．図 5 にはトルクの絶対誤差の推移も記載した．図 5 より，大部分を評価関数計算が占めており，トルクの誤差が大きいほど，割合が大きくなっている．また，評価関数計算回数とトルク誤差には図 6 に示すようにトルクの誤差が大きくなるほど評価関数を計算する回数が増える傾向にあることもわかった．以上のことから，誤差が大きい場合に実行時間が増える原因として次のように考えられる．式 (4) より，トルクは i_d,i_q に依存する．トルク誤差が大きいということは i_d,i_q の誤差が大きいということであり，式 (5) により計算される評価値 J が大きくなる．J の値が大きいため，枝刈りが行われる回数が少なくなる．すると多くのパターンを探索することとなり，評価関数の計算回数が増え，処理に時間がかかるようになる．

トルク誤差の増大による処理量増加を防ぐには以下の方法が考えられる．

- 1 回にかかる評価関数計算時間を削減する．
- 評価関数計算回数を減らす．

1 つ目の方法であるが，1 回にかかる計算時間を測定したところ，平均してわずか 1μs であった．これをさらに削減しようとすると計算式自体を見直す必要がある．計算式はベクトル制御に基づくものであるため，さらに高速な計算式となると新たな制御手法を提案することとなり，本研究の目的とは異なる．よって，1 つ目の方法による高速化は断念する．

評価関数の計算回数を減らすことは探索数を減らすことを意味する．探索数を減らす方法としては分枝限定法に基づく枝刈り数を増やすことと解空間を狭める方法が考えられる．まず，枝刈りを増やす方法を考える．枝刈りは早く最適解を見つけられるほど多く行われる．そこで，最初の暫定解が最終的に得られた最適解とどれほど近い評価値であるかを調べた．図 7 にその結果を示す．図 7 では初期暫定解と最適解の差が最適解の評価値にとってどのくらいの割合であるかを「評価値差の割合」として示している．高速化すべき最適解の評価値が大きくなっている部分では，評価値差の割合は 5%にも満たない差となっている．つまり，最初に得られた暫定解と最適解は非常に近い評価値で

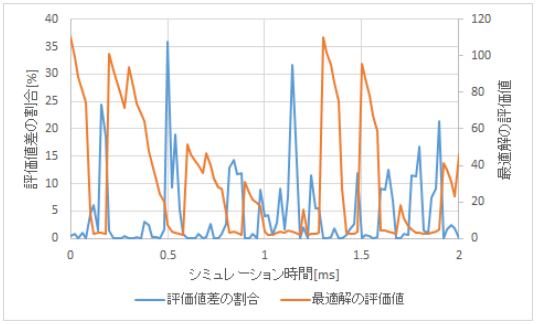

図 7　初期暫定解と最適解の差

あるということである．よって，枝刈りの回数はほぼ最大量に達しており，さらに枝刈りを増やすことで高速化を図るのは難しい．そのため，解空間を狭める方法による高速化を図ることとする．

一方で，図 5 からソートの処理割合も無視できるものではない．そのためソートを高速化する必要がある．

以上より，行うべき最適化として解空間の削減，高速なソートアルゴリズムへの変更とする．

3.2　最適化

性能解析により，行うべき最適化として解空間の削減，高速なソートアルゴリズムへの変更であると結論付けた．本節ではそれらに対して実施したことを説明する．

3.2.1　解空間の削減

本モデルの制御器として出力すべきは次の 3 相 PWM インバータのスイッチングパターンである．各相のスイッチングタイミングはそれぞれ 0~Nc の整数をとるため，1 つのスイッチングパターンを三次元空間 u-v-w 上の点としても表現できる．この表現を用い，最適解として選ばれている点の特性を調べ，解空間の削減を行う．

最適解の分布を三次元空間上で表現したものを図 8 に示す．3 次元空間表現のため，奥行きがわかりにくいので，3 つの視点からの図を掲載している．図中,u 軸,v 軸,w 軸はそれぞれ u 相,v 相,w 相の切り替えタイミングを意味しており，各視点の位置関係把握のために原点 (0,0,0) を赤色の点として示している．また，トルクの誤差が大きい場合のみを色付けしたものを図 9 に示す．図 9 より，トルク誤差が大きい場合は図のような位置に最適解が現れる特性があると考えられる．よって，図 10 に示す 2 つの立方体内の領域（u,v,w 全てが 15 未満または全てが 5 より大きい領域）を探索不要とみなす．この領域の削減により，1 制御区間のパターン数が 1/3 となる．本論文では Np=2 としているため，この解空間の削減により，総パターン数としては 1/9 となる．これにより，トルク誤差が大きい場合の処理は最大で 9 倍高速になる．

3.2.2　ソートの高速化

本モデルではクイックソートを用いている．クイックソートはソートしたいリストより適当な値（ピボット）を一つ選択し，それより小さい値をピボットより前方に，大きい値をピボットより後方に移動させる．ピボットにより，2 分割されたそれぞれのリストに対して，さらにクイックソートを行う．ピボットがデータ総数の中央値である場合，リストは偏りなく 2 等分される．このとき，ソートの再起実行回数は最も少なくなり，クイックソートは最短で完了する．そのため，できる限りピボットをデータ総数の中央値に近い値を選択したい．そこで，リストに追加されたデータの中で最小の評価値と最大の評価値の中間値をピボットとして選択し，できる限り中央値に近いピボットを選択されるように変更したことでより高速なクイックソートを実現した．

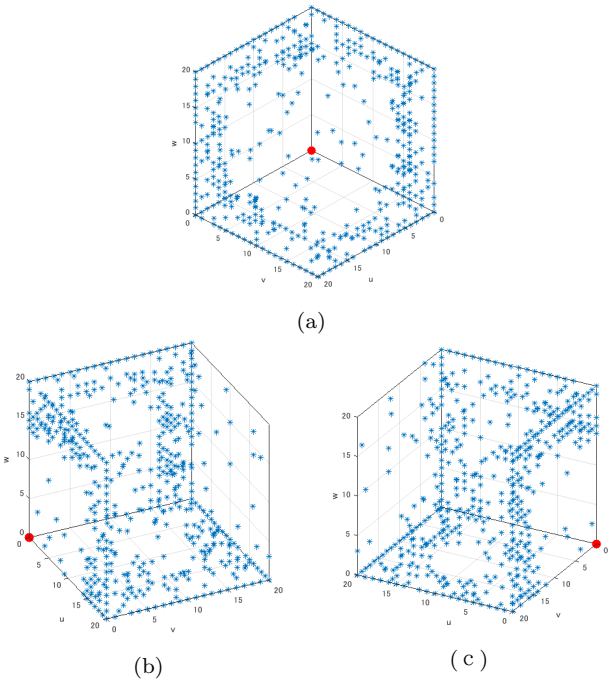

図 8　最適解分布

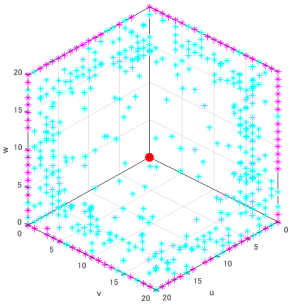

図 9　トルク誤差が多い場合の最適解分布

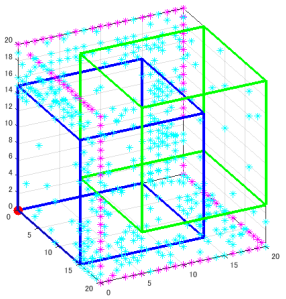

図 10　探索不要領域

4.　並列化

本章ではモデル予測制御を用いた制御プログラムに対して，実施した並列化を述べる．並列化には共有変数へのアクセスが簡単かつ高速である共有メモリ型並列化を支援す

る OpenMP[5] を用いた．

4.1 評価値計算の並列化

ある状態において次に考えられる操作はスイッチングパターン数 Nc^3=9261 だけある．評価値計算はその数だけ行われる．静的に計算回数が決まっているため，評価関数計算は for ループで実現されている．また，各スイッチングパターンにおける評価値計算は互いに関与しあわないため，データ並列性がある．よって OpenMP の for 構文を用いて並列化を図る．

スイッチングパターンによっては計算途中で評価値の最小値を超えたり，最適化によって探索不要とされたパターンであったりした場合，計算を即座に打ち切るようにしている．これによってスイッチングパターンごとの評価関数計算にかかる時間は不均一になっている．そのため，評価関数計算の負荷分散のために OpenMP の schedule 指示節で dynamic を指定することとした．これにより，計算が打ち切られたり，計算が完了したスレッドに対して動的に次の反復を割り当てることができ，高並列性の評価値計算が実現できる．

4.2 ソートの並列化

クイックソートはピボットより小さい数のみのリストと大きい数のみのリストに 2 分割する．また，それらのリストはそれぞれがさらにソートされる．このとき，互いのリストのデータを参照することはない．よって分割されたリストにはデータ並列性がある．再帰的に実行されるリスト分割処理を OpenMP の task 構文を用いてタスク化した．こうすることで，次々と生成されるタスクは複数コアに割り当てられ，並列に実行される．このようにして並列のクイックソートを実現した．

4.3 探索の並列化

2.7 節で述べた探索手順を基に OpenMP を用いて次のように並列探索を行う．

(1) 現在の最小評価値取得
ローカル変数 Jmin_local に評価値の最小値を取得・保存する．

(2) 評価関数計算
評価関数計算途中で評価値が Jmin_local を超えた場合，リストに追加せず，枝刈りを行う．

(3) ソートおよび初期暫定解の探索
初期暫定解が求まっていない場合にのみ行う．リストをソートし，最小の評価値の子ノードを探索.

(4) 子ノード探索または最小値比較
葉ノードでない場合は探索関数の再起呼び出しを一つのタスクとし，生成する．葉ノードの場合は評価値と Jmin_local を比較し，Jmin_local よりも小さい場合は更新する．

(5) 暫定解更新全体の中で最小の評価関数が保存されている Jmin と Jmin_local を比較し，Jmin_local がより小さい場合，Jmin を更新し，その時のパターンを暫定解とする．

2.7 節で述べた手順と異なるのは次の 3 点である．

- 初期暫定解の探索
- 子ノード探索のタスク化
- ローカル変数 Jmin_local の利用

本モデルにおいて，図 7 より，初期暫定解が最適解に近い評価値となることがわかっている．そのため，この初期暫定解が先に得られていれば，並列に探索を行っても無駄な領域を探索してしまうことを最小限に抑えられる．よって，最初の暫定解が得られるまでは逐次的に探索し，暫定解が得られた以降から並列に探索を行う．

並列の探索には OpenMP のタスク化を利用する．タスクを利用する理由としては，クイックソートのように処理が再起的に実行されるためである．また，分枝限定法による枝刈りが行われるため，各ノードの負荷は不均一となる．タスクは遊んでいるスレッドに対して，動的に割り当てられるため，負荷が均一に保たれやすく，並列性を高めることができる．

分枝限定法の枝刈りは評価関数の最小値 Jmin を用いて実行される．そのため，Jmin は常に評価値の最小値であるべきである．しかし，Jmin を常に更新し，参照しようとするとスレッド間の競合が頻発するようになる．スレッド間のアクセス競合は値の不整合や並列性の低下を引き起こすため，できる限り同じ変数を同時に更新・参照することは避けるべきである．本モデルにおいて初期暫定解の評価値が十分に最適解の数値に近いため，常に Jmin を更新し続けなくても探索ノード数は大きく増えないだろうと予想し，Jmin の更新回数や参照回数を減らす．このために，スレッドローカル変数 Jmin_local を使用する．ノードの処理開始時にその時の最小評価値を取得し，以降の処理で Jmin を参照する必要がある部分は全て Jmin_local を参照する．これにより，競合の発生は大きく抑えられ，並列性を維持することができる．ただし，最後には Jmin を更新しなければならないため，OpenMP の critical 構文を用いて Jmin の参照を排他制御し，値の整合性を保つ．

5. 評価

本章では，3 章で実施した最適化と 4 章で説明した並列化を実施したことによる性能向上の評価を行う．

5.1 評価手順

まず，3 章の最適化による性能向上を評価する．最適化前のプログラムと最適化後のプログラムをそれぞれ同一の環境，シミュレーション設定でシミュレーションした場合

表 1　最適化による性能向上

プログラム	実行時間 [s]	性能向上率
最適化前	2942.85	1.00
最適化後	462.86	6.35

の実行時間を調べる．実行時間が 1/n になった場合を性能向上率 n とし，最適化による性能向上率を計算・評価する．

次に 4 章の並列化による性能向上を評価する．比較対象は最適化後のプログラムによる実行時間とし，並列処理のみによる性能向上を考慮する．コア数ごとに性能向上率を測定・計算し，コア数の変化による性能向上率の推移を評価する．また，各コアの処理時間を計測し，負荷分散を調べ，評価する．さらに全コアが探索したノード数を調べ，単純に並列探索した場合と初期暫定解を得た後に並列探索をした場合の探索ノード数の増加量を比較・評価する．

5.2　評価環境・シュミレーション設定

今回のシミュレーションは以下の環境で実行した．なお，シミュレーション設定は 3.1 節と同じ設定とした．

実行環境

- OS:CentOS 7.2.1511
- CPU：Intel Xeon E5-2695 v2 2.40GHz
- メモリ：32GB
- コンパイラ：gcc 4.8.5

なお，今回用いた CPU はハイパースレッディング技術を採用している．しかし，並列化性能を評価する上で，論理コアに割り当てられるとコア数と並列性能の関係を議論しにくい．そのため，1 物理コアに対して，1 つのスレッドが割り当てられるように GNU gcc の GOMP_CPU_AFFINITY 環境変数を設定している．

5.3　評価結果

表 1 に最適化前と最適化後の実行時間，性能向上率を示す．最適化によって性能は 6.35 倍となった．

図 11 にコア数による性能向上率の推移を示す．本研究の実行環境では 12 コアで 9.05 倍まで高速になった．また，12 コアで実行した時の負荷が最小であったコアの処理時間を 1 としたときの最大負荷のコアの処理時間を負荷差として図 12 に示す．負荷差は平均して 1.4 となった．さらに，図 13 に逐次実行時の探索ノード数を 1 とした時，暫定解を取得した後から並列探索した場合と最初から並列探索した場合の探索ノード数を示す．暫定解取得後に並列探索した場合は探索数は最大でも逐次実行時の 1.55 倍であった．一方，最初から並列探索した場合は最大で逐次実行時の 4.65 倍となっていることが確認された．なお，ノード数が増加していない場合についてはトルクの指令値とシュミレーション値の誤差が大きく，逐次実行時においても枝刈りがほとんど行われなかった場合であった．

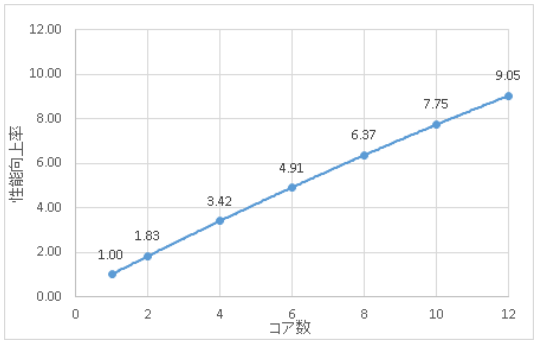

図 11　並列性能向上率

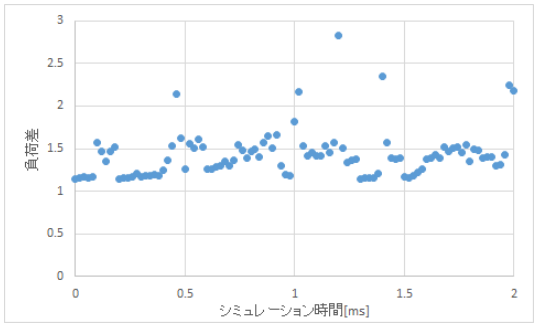

図 12　最大負荷のコアと最小負荷のコアの負荷差

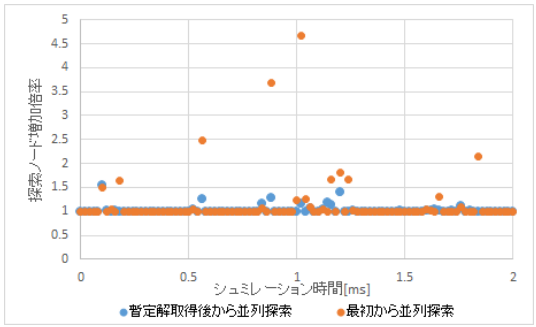

図 13　探索数増加量

5.4　考察

表 1 より，最適化による性能向上率は 6.35 だった．解空間の削減により最大 9 倍の高速化が見込まれたが，6 倍にとどまった．これはトルク誤差が大きい場合のみ解空間を削減でき，全ての場合において適用できる最適化ではなかったためと考えられる．またソートアルゴリズムの高速化を行ったが，図 5 より，ソートが全体に占める処理割合が少ないため，最適化の影響が少なかったと考えられる．

図 12 より，負荷にばらつきがあることが確認できる．負荷差の平均は 1.4 となったが，これは処理時間の約 1/3 もの時間，処理を行わないコアがあることを示している．主に OpenMP のタスクにより，並列処理を行っているため，コアに対するタスクの割り当てがうまくいっていないことが推測される．文献 [6] によると，コンパイラとして gcc を用いた場合の OpenMP によるタスク並列化のパフォーマンスは Intel コンパイラなどに比べ，非常に悪いという結果が得られており，本研究にもこの影響が出たと考えら

れる．

図 11 より，12 コアで並列性能向上率は 9.05 倍だった．一般的に並列オーバヘッド等により，コア数の数だけ倍速になるようなことはあり得ないとされているため，この性能向上率は十分な結果である考えられる．これだけの性能向上が得られた理由としては図 13 に表れているように，探索数の増加を抑えつつ並列探索が行われたからであろうと考えられる．一方で，うまく負荷分散できていないために 9 倍で留まったと考えられ，コンパイラを変更する，タスクを使わない，OpenMP 以外を使うなどといった方法をとることで，更に並列性能を伸ばすことが可能だと考えられる．

6. おわりに

本章では本研究のまとめと今後の課題について述べる．

6.1 まとめ

本研究では永久磁石同期モータモデルを題材とし，近年のハイエンド制御に対して有効とされているモデル予測制御アルゴリズムの高速化と並列化を行った．まず，最適解の分布特性を利用した解空間の削減やクイックソートのピボット選択の工夫といった最適化を行い，逐次実行の 6 倍以上の高速化を達成した．また，共有メモリ型並列化を支援する OpenMP を用いて分枝限定アルゴリズムや評価関数計算，ソートの並列化を行った．本モデルの特性を活用し，最初の暫定解が得られてから並列探索を行ったり，スレッド共有変数の参照頻度を下げるといった工夫を施すことで，12 コアで 9 倍の高速化を達成した．

6.2 今後の課題

本研究において 2 ステップ先の状態まで推定するモデル予測制御を並列化し，高速化したが，その操作の決定には 1s 程度かかることもあった．本来であれば，1 つの操作の決定に数十 μs 程度の時間しか費やすことはできないため，実用化には程遠い．モータをモデル予測制御で実用的に制御するためには操作決定の方法を変える必要があると考えられる．例えば，回転子の位置を測定，あるいは推定することで大まかな次の操作を決定できるため，その微調整として少ないパターン数から最適な操作を決定するためにモデル予測制御を用いるといった方法が考えられる．実用化はできないが，本研究による高速化を用いてシミュレーションを高速化することで，大まかな次操作決定のために必要なデータ収集を効率化できると考えられる．

参考文献

[1] 大沼巧：新しい座標系を用いた埋込磁石同期モータの位置センサレス制御に関する研究，名古屋大学 (2011).
[2] 松本純：新しい数学モデルを用いた永久磁石同期モータの位置センサレス制御系のロバスト化に関する研究，名古屋大学 (2014).
[3] 河合健司：モデル予測制御を用いた PMSM の最適制御に関する研究，三重大学 (2007).
[4] 大西克実，榎原博之，中野秀男：並列分枝限定法における分枝変数の選択に関する考察，電子情報通信学会論文誌 D-I(2001)，Vol.J84-D-I，No.9，pp.13181326.
[5] OpenMP，https://www.openmp.org/，2016
[6] Stephen L. Olivier and Jan F. Prins: *Evaluating OpenMP3.0 Run Time Systems on Unbalanced Task Graphs*, University of North Carolina at Chapel Hill.