電力制御システム向け 大規模演算処理の高速化

細川 武彦*, 相浦 利治, 高畑 泰志(三菱電機(株))

計算機の高い演算能力を積極的に利用した大規模科学技術計算や各種のシミュレーションなどは、様々な社会基盤を支える計算機システムに組み込まれ、重要な意思決定支援機能などを提供している。 電力向け計算機システムにおいても、潮流計算や安定度計算などの大規模演算を行うアプリケーションによって、系統信頼性、経済性などを実現している。 今回、これら電力制御システム向け大規模演算処理アプリケーションに対して、計算機アーキテクチャやコンパイラなど、プラットフォーム側の観点から高速化を検討した。本稿ではこれらの高速化技術、特に単体プログラムにおける最適化コンパイラ技術とコードの改善方法、並列プログラムにおける実装方法と性能解析技術について説明し、幾つかのアプリケーションに試行した結果について示す。

Performance Tuning for EMS[1] Applications

Takehiko HOSOKAWA, Toshiharu AIURA, Yasushi TAKAHATA(Mitsubishi Electric Corp.)

In the energy management systems, high reliability and efficiency have been realized by large-scale computing applications such as the optimal power flow calculation. This paper describes performance improvement for these applications and the practical results.
The two main issues addressed herein are:
(1) Coding techniques that enable well compiler optimization.
(2) Analyzer for parallel processing efficiency.
By applying these technologies, 2 to 8 times performance improvement has been achieved.

1. はじめに

計算機の高い演算能力を積極的に利用した大規模科学技術計算や各種のシミュレーションなどは、様々な社会基盤を支える計算機システムに組み込まれ、重要な意思決定支援機能などを提供している。電力向け計算機システムにおいても、潮流計算や安定度計算などの大規模演算を行うアプリケーションによって、系統信頼性、経済性などを実現している。今回、これらの電力制御システム向け大規模演算処理アプリケーションに対して、計算機アーキテクチャやコンパイラなどプラットフォーム側の観点から高速化を検討した。

本稿では特に著者らが担当した次の 2 つの高速化ポイントについて説明し、幾つかのアプリケーション(表 1)に試行した結果(表 2)について示す。

(1) 単体プログラムでの高速化

単体プログラムでの高速化では特にコンパイラによる最適化技術とそれらを有効に活用するためのコードの改善方法について示す。

(2) 並列プログラムでの高速化

並列プログラムでの高速化では特に並列処理の実装方法と並列化効率の解析手法について示す。

表 1 対象プログラム

	プログラム名	コード量	文献
①	最適潮流計算	約 5K ステップ	[1]
②	安定度解析	約 15K ステップ	[2]
③	需給運用計画	約 30K ステップ	[3]

表 2 高速化試行結果

	高速化技術	改善率[倍]		
		①	②	③
単体	最適化コンパイラ	3.6	1.2	2.4
	コードの改善	3.8	3.5	1.2
並列	処理の並列化(※)	—	—	2.2～2.7
	並列化の効率向上	—	—	1.1～1.5

(※) 4CPU の SMP 計算機による実装

1. EMS:Energy Management System

2. 単体プログラムでの高速化

単体プログラムでの高速化は古くから様々なアルゴリズムや実装方法の研究が行われてきた。近年、プロセッサ技術の高度化、複雑化によりアセンブラ言語の記述による高速化が難しくなっており、コンパイラによる最適化技術への依存度が高くなっている。また、開発するコード量も爆発的に増加してきており、これらのコードに対する最適化についてはコンパイラ任せになっているのが現状といえる。しかし、コンパイラの最適化技術が高度化するに従い、ほんの僅かなコードの差異により、性能が大きく異なる場合が増えてきている。本章では高速化のポイントとなるコンパイラの最適化技術とコードの改善方法について簡単な例を交えながら説明する。

2.1. 最適化コンパイラ技術

現在のプロセッサの大半はスーパスケーラと呼ばれる、複数の実行ユニットを用いたパイプライン処理により高速化が図られている。 プログラムの高速化を行うためにはこれらの実行ユニットやパイプラインを有効に活用することが重要であるが、(1) 構造ハザード (2)制御ハザード (3) データハザード と呼ばれる原因によりパイプラインストールが発生し、高速化を阻害する要因となっている。 コンパイラの最適化技術ではこれらのハザードを削減し、実行ユニットやパイプラインを有効に活用するようなコードの変換処理が行われる。 しかし、コンパイラの最適化では複雑なコードの場合には最適化が適用されない場合がある。 特に以下にに示すような場合、コンンパイラは最適化することができない。

データの依存関係 処理間にデータの依存関係があり、最適化の適用前後で結果が変わってしまう場合、もしくは、コンパイル対象のコードのみからではデータの依存関係が判明しない場合には最適化が適用できない。

最適化のスコープ 最適化を適用する場所を特定するアルゴリズムにより、ブロック単位、関数単位、ファイル単位などの範囲内でしか最適化は適用されない。

誤差の許容範囲 浮動小数点演算の式の変形などの最適化の場合、適用の前後で計算の丸め誤差により、計算結果が異なる場合があり、コンパイラの最適化では基本的にはこのような式の変形は適用しない。

2.2. コードの改善方法

コンパイラの最適化技術を有効に活用するためには、最適化が適用できないような記述を削減することが重要である。ここでは、幾つかの簡単な例について示す。

(1) データ依存性の影響

図 1に示すコード例はデータの依存関係がソースコードより不明な場合と明確な場合の例である。コード(A)の場合、ポインタ変数p、qの依存関係が不明であり、*pへの代入により*q の値が変更される可能性があるため、コンパイラはいくつかの最適化を利用することができない。そこで、プログラム設計段階で依存関係が無いことが明らかな場合、コード(B)のように修正を行うことにより、データの依存関係がコンパイラに解釈可能となり、最適化することができる。

(A) データ依存不明なコード

```
extern  int a[MAX][MAX], *q, *p;
foo(){
   int x, y;
   for(x = 0; x < MAX; x++){
      for(y = 0; y < MAX; y++){
         *p += a[x][y] * *q;
      }
   }
}
```

(B) データ依存が明確なコード

```
extern int a[MAX][MAX], *q, *p;
foo(){
   int x, y;
   int pt = *p;
   int qt = *q;
   for(x = 0; x < MAX; x++){
      for(y = 0; y < MAX; y++){
         pt += a[x][y] * qt;
      }
   }
   *p = pt;
}
```

図 1 データ依存性のコード例

実際にコード(A),(B)をコンパイラの最適化レベル[1]を変えてコンパイルし、実行した場合、図 2のような実行時間となる(実行時間はコード(A)の最適化レベル0を用いた場合の実行時間を 100 とした相対時間である)。

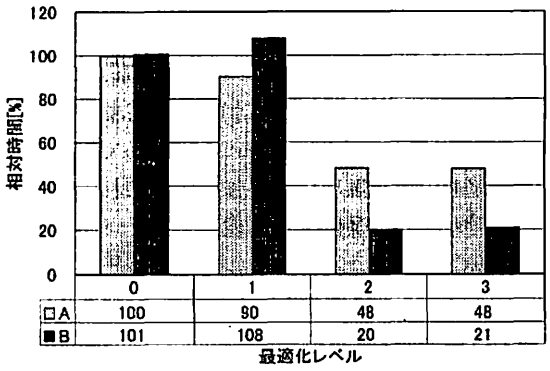


図 2 データ依存性の影響

[1] 各最適化レベルでは主に以下のような範囲内で最適化が行われる。

0 : なし, 1 : ブロック内, 2 : 関数内, 3 : ファイル内

本結果より以下のことがわかる。

・コード(A)では最適化により約 2 倍の性能向上に留まるがコード(B)のようにデータの依存関係が明確になるように修正することにより、約 5 倍の性能に向上する。

・データの依存関係が不明か、明確かにより最適化レベル 0 では大きな性能差がないが、最適化レベル 3 では約 2.3 倍の性能差があり、最適化レベルを上げた場合の性能差が大きく異なる。

(2) データアクセスパターンの影響

先のコード(A),(B)において 2 次元配列 a へのアクセス順序を以下のように変更したものをコード(C),(D)とする。

```
a[x][y]  →  a[y][x]
```

一般に C 言語での 2 次元配列へのアクセスはコード(A),(B)の方がコード(C),(D)よりも、キャッシュが有効に利用されるため効率よく行われる。

実際にコード(C),(D)をコンパイラの最適化レベルを変えてコンパイルし、実行した場合、図 3のような実行時間となる

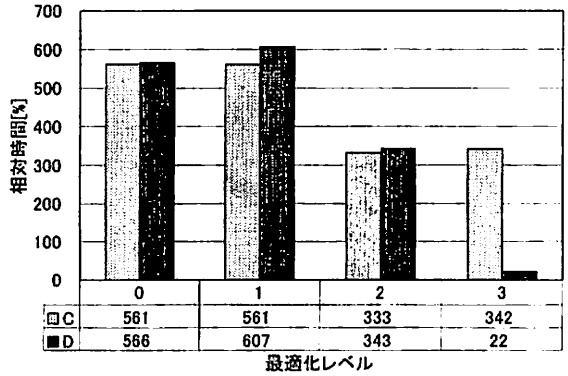


図 3 データアクセスパターンの影響

本結果より、以下のことがわかる。

・データのアクセスパターンが悪い場合に 1/5～1/6 に性能が劣化する。

・上記、劣化はデータの依存性を明確化し、最適化レベルを 3 に上げることにより、ほぼ解消できる。

・データアクセスパターンが悪い場合、データ依存性が不明か明確かにより最適化レベル 0 では大きな性能差がないが、最適化レベル 3 では約 16 倍の性能差となる。

本例からデータ依存性、データアクセスパターンについて考慮してコードを見直すことにより数倍から十数倍の性能向上が見込めることがわかる。 本例は話を単純化するためのものであり、実際のソースコードでは構造体の配列やより複雑なデータ構造が用いられる場合が多いため、ここまでの性能向上を図ることは難しいが、基本的な考え方は同様である。

(3) その他

(1),(2)ではデータの依存性、アクセスパターンの例について示したが、最適化のスコープや誤差の許容範囲などについても同様な性能を劣化させる要因や劣化を抑える修正方法がある。 以下に劣化を抑え、性能を向上させるためのポイントを簡単に示す。

・変数のスコープが明確になるように修正する。 変数のスコープをコンパイラが解釈できるようになり、より多くの最適化が利用可能となる。 また、このような修正はデータの局所性を高めることにもなるため、キャッシュミスや TLB ミスの削減にもなる。

・浮動小数点演算については適用前後の誤差、性能の検証を行い問題がなければ式の変形(乗算を加算に変更や除算を乗算に変更するなど)を行う。

・ループ内にデータ依存のある処理がある場合はループ外で行うよう修正する。 これにより、コンパイラはデータのアクセスパターンやキャッシュサイズなどから、ループの変形を行うことが可能となる。

2.3. 単体プログラムの高速化試行結果

前節に示したようなコードの改善方法を用い単体プログラムの高速化を以下のような電力制御システム向け大規模演算処理アプリケーションに対して適用した。

① **最適潮流計算**:各種の制約(発電出力・潮流・電圧など)を守り、評価関数値(発電コスト・送電損失など)を最小化する制御対象機器(発電機・変圧器・調相設備など)の状態を決定するプログラム。問題空間探索法と内点 QP 法を用いた高速化を行っている(文献[1])。

② **安定度解析**:事故発生に伴う電力動揺の計算と安定度の判別を行うプログラム。 設備情報として発電機などの一般的なモデルの他に、ユーザが任意のモデルを定義することを可能としている(文献[2])。

③ **需給運用計画**:各種の制約を守り、発電コストや電力の安定度が最適となるように運用計画を行うプログラム。 問題規模が大きいため、問題規模を分割し最適化を行っている(文献[3])。

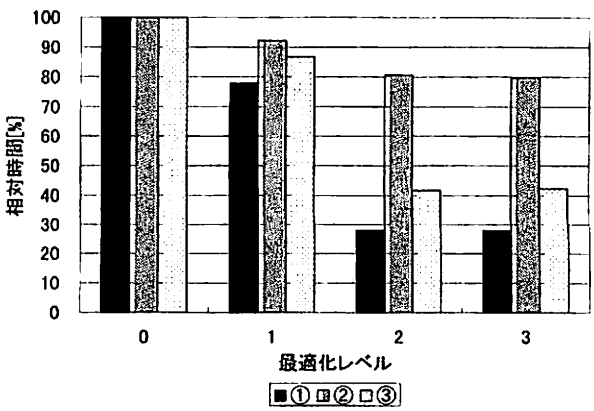


図 4 最適化の効果

各アプリケーションに最適化コンパイラを適用した場合の効果を図 4に示す(実行時間は最適化レベル0の実行時間を 100 とした場合の時間である)。 図 4からアプリケーション①は最適化によりおよそ70%実行時間が短縮されているが、アプリケーション②の場合はおよそ20%しか実行時間が短縮されていないことがわかる。

図 5に高速化の一例としてコンパイラによる最適化の効果が最も少なかったアプリケーション②に対して高速化を実施した過程を示す。

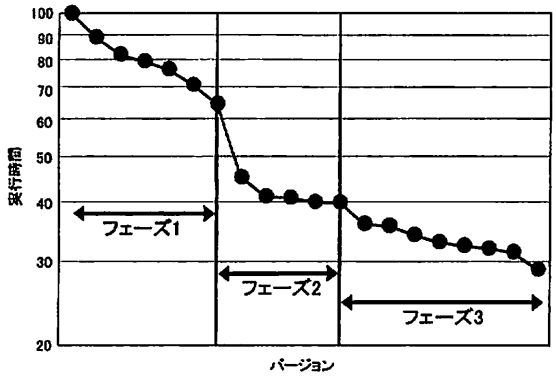


図 5 アプリケーション②の最適化の過程

フェーズ 1:プロファイラにより関数レベルの傾向を分析し、主に全体的なデータ構造や実装アルゴリズムの見直しを実施。

フェーズ 2:コンパイラの最適化オプションの調整、およびソースコードレビューによる各関数内のループや変数などの見直しを実施。

フェーズ 3:プロファイラにより、詳細なループレベルの傾向を分析し、主に詳細なループ構造、データ構造、算術式の変更を実施。

各アプリケーションの高速化の結果は以下の通りであり、約 1.2～3.8 倍程度の高速化を図ることが出来た。

表 3 : 単体プログラムの高速化結果

No.	初期値[秒]	高速化[秒]	改善[倍]
①	4.70	1.23	3.8
②	36.2	10.4	3.5
③	61.3	52.4	1.2

3. 並列プログラムでの高速化

現在、大規模演算の高速化を実現するために CPU を複数実装したSMPアーキテクチャを用いた並列プログラミングによる性能向上の試みが増えている。 実際に並列化コンパイラなども製品化されており、大きなマトリクス演算などを自動的に並列処理するようにコードの自動生成を行えるようになっている。 しかし、実際の演算では、入力データに対する処理量のばらつきにより効率的な並列処理コードの自動生成が可能な領域は限られてしまっており、特に入力データに対する処理のばらつきが大きいプログラムでは、効率的な並列処理が困難である。 以下では、問題規模が大きいため、問題規模を分割し最適化を行っているアプリケーション③について、並列化を用いて高速化した結果について示す。

3.1. 並列プログラムのモデル

今回、アプリケーション③に実際に適用した並列化のモデルは図 6に示すような単純なモデルである。

本モデルにおける高速化のポイントは如何にして子プロセスに処理を均等に分割するか、および並列処理によるオーバヘッドを削減するかである。 特にアプリケーション③は収束演算を用いた計算アルゴリズムによる高速化を行っており、入力されたデータに対して実際の処理量というのは単純に見積もることができないため、処理振り分け時に処理量を均等になるようにデータを分割することは困難である。そこで図 7のようにキューを用いることによりデータの分割を行うように設計した。

アプリケーション③に対し、単純に本モデルを用いた場合、4CPU の SMP 計算機上での 4 プロセスによる処理並列化により、単体時と比較して約 2.2～2.7 倍の高速化が実現できた。

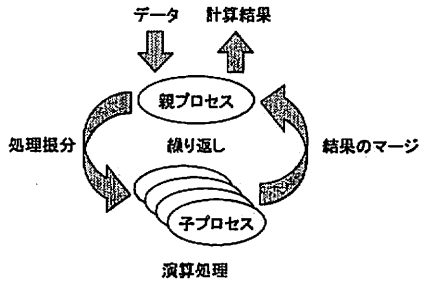


図 6 並列化のモデル

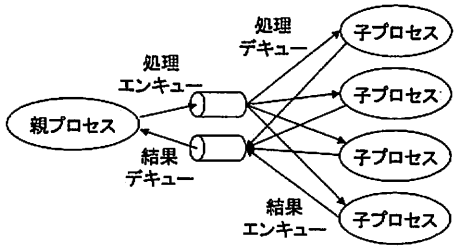


図 7 データの分割方法

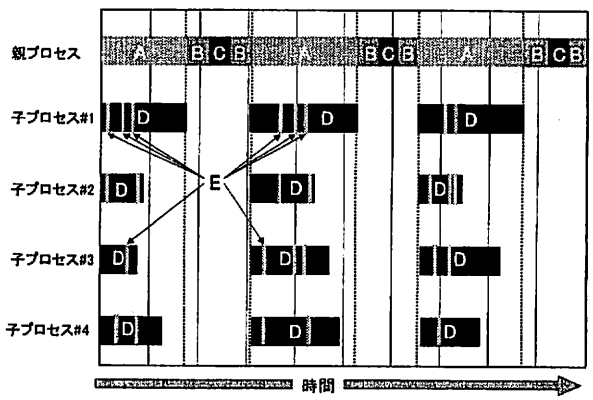


図 8 並列化の効率

3.2. 高速化のポイント

本モデルにおいて更なる高速化を実現するためのポイントは処理粒度の調整と排他制御などのオーバヘッドの削減である。処理量のばらつき、並列化に伴うオーバヘッドを最小にするような処理粒度を求めることができれば更に性能を向上することが可能となる。 図 8 を用いて並列化の効率の指標について説明する。図 8 の各処理は以下の通りである。

A: 並列処理実行部。親プロセスは子プロセス終了待ち
B: 子プロセスの起動/終了処理部。
C: 振分/マージ処理(処理エンキュー/結果デキュー)
D: 実際の計算処理部。子プロセスの実行時間
E: 処理デキュー/結果エンキュー(D 内の縞模様)

並列化の効率を以下の式で定義する。

((D の合計)÷プロセス数)÷(A の合計)×100

上記の値が 100%に近いほど効率的に計算負荷が分割されていることになり、遠いほどばらつきが大きく効率が悪くなっているといえる。 また、並列化にともなうオーバヘッドには 2 種類あり、1つめのオーバヘッドは以下の式により求めることができる。

(B の合計)+(C の合計)

もうひとつのオーバヘッドは処理 D 内から共有リソースにアクセスする場合に発生し、処理のデキュー、結果のエンキューなどで使用する排他制御によるオーバヘッドである。 このオーバヘッドは特に排他制御による待ち時間の占める割合が大きいため**処理 D 内での CPU 利用率**として観測することができる。

3.3. 性能解析ツール

上記のような並列化の効率、CPU 利用率を視覚的に確認するための性能解析ツールを開発した。

性能解析ツールでは、トレースデータを取得するライブラリ I/F を並列プログラムに提供することにより、並列プログラムの実行後にデータを集計し、解析する手法を用いた。解析データの集計結果として、各プロセスの実行をタイムチャートとして視覚化することを可能とした。このとき図 9のように、ある親プロセス(プロセス 3)から複数の子プロセス(プロセス 4#1~4)が起動される場合、各子プロセスの実行時間により並列化の効率を算出し、グラフの色を変更することにより並列化効率の悪い部分を容易に特定できるようにする。グラフにより特定できた並列化の効率の悪い部分において、処理時間の長い(短い)ものについては処理の粒度を下げる(上げる)、処理時間の長い部分について注力して高速化を行う、といった対策を立てることが可能となる。また、タイムチャートの作成時に CPU 利用率により、グラフの色を変更することも可能とし、これにより、CPU 利用率の悪いプロセス、すなわちオーバヘッドの大きい(ボトルネックを含んでいる可能性の高い)プロセスを特定することができる。

また、各機能毎の CPU 利用率のグラフを作成し、先のグラフにより特定したプロセスの詳細な CPU 利用率のグラフを取得することにより、ボトルネックとなっている機能を特定することを可能とした。特定した機能において、必要な共有リソースを親プロセスで分割して子プロセスに割り当てる、プライオリティを変更する、排他処理の粒度を変更するなど原因を取り除くための対策を立てることが可能となる。

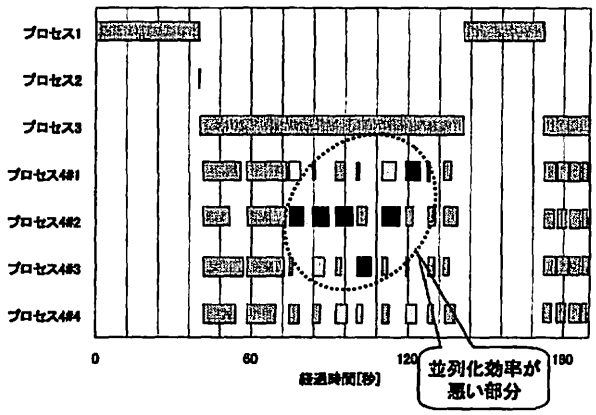


図 9 並列化効率グラフ

3.4. 並列プログラムの高速化試行結果

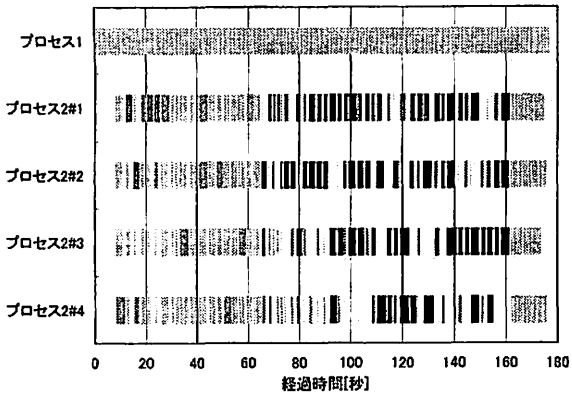
アプリケーション③に対して、開発した電力制御システム向け性能解析ツール、およびプロファイラなどを用いて検証用システム上で高速化を実施した。高速化の過程を図 9 に示す。図 9 において、色の濃いプロセスほど効率が低く、色の薄いプロセスほど効率が良いことを示している。

図 9 の(A),(B)はそれぞれ高速化実施前の並列化効率グラフ、CPU 利用率グラフであり、後半の処理に問題があることがわかる。プロセス 1 の CPU 利用率が低いのはプロセス 2 の実行待ちのためであるが、プロセス 2 の CPU 利用率が低いのはなんらかの問題があると考えられる。

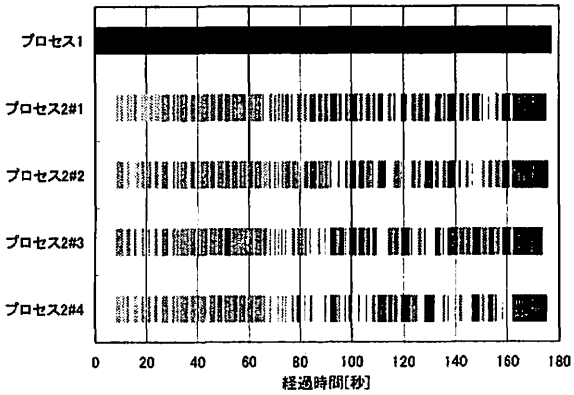
そこで、後半の処理について詳細な解析を行った結果、プロセス 2 内で共有すべきではないリソースを共有しており、競合が発生していることが判明した。

本問題を修正することにより、後半の処理の CPU 利用率が上がり、図 9 の(C)のように実行時間を短縮することができた。

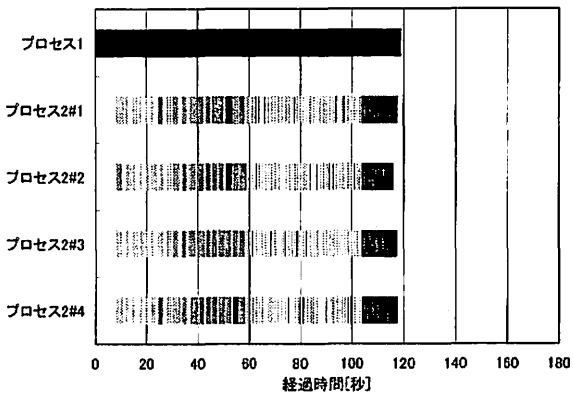
その後、計算アルゴリズムの見直しや、処理粒度の調整などを行うことにより図 9 の(D)のように実行時間を短縮することができた。 全体の実行時間がおよそ 50%程度短縮されていることがわかる。



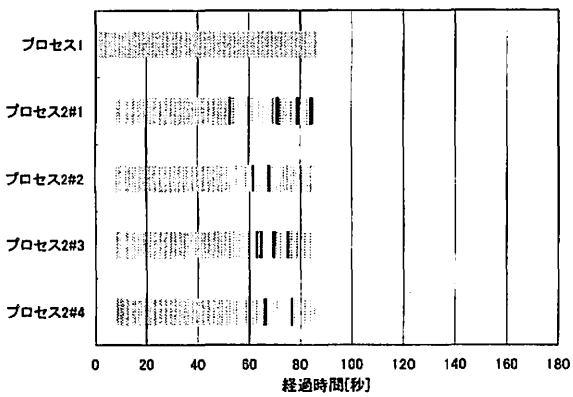
(A) 並列化効率グラフ(チューニング前)



(B) CPU 利用率グラフ(チューニング前)



(C) CPU 利用率グラフ(チューニング途中経過)



(D) 並列化効率グラフ(チューニング後)

図 10 並列化効率グラフ

4. まとめ

本稿では、電力制御システム向け大規模演算処理アプリケーションの性能確保にむけて実施した高速化として、最適化コンパイラ技術とそれらを有効に活用するためのコードの改善方法、並列処理の実装方法と並列化効率の解析手法の2つにポイントを絞って説明を行った。 これらの技術領域は電力制御システムのみならず、その他の一般的な大規模演算処理アプリケーションについても適用することが可能である。また、計算アルゴリズムによる高速化の手法は解の質と性能とのトレードオフが多くの場合に存在するが、本手法では解の質を損なうことなく、高速化を行うことが可能である。 特に最適化コンパイラ技術を有効に活用するためのコードの改善方法についてはそれらを意識したコーディングを日頃から心掛けるだけで、数倍～十数倍の性能差が出てくるものであり、広く展開していく必要があると考えている。 また、並列化効率の解析手法では今後、解析効率を上げるための機能追加(問題個所とソースコードのリンクなど)、システム上の他の性能データを取得するツール(CPU の利用状況、ネットワークトラフィックなど)と連携して解析を行う枠組みの検討などを行っていく予定である。

文 献

[1] 田村, 板屋, 他:"高速最適潮流計算アルゴリズムの開発", 電力技術・電力系統技術合同研究会, Sep. 2002

[2] 高橋, 臼井, 他:"次期中央給電指令所システムにおけるオンライン信頼度監視機能について", 電気学会電力系統技術研究会, Mar. 2002

[3] 高橋, 小野, 他:"火力・揚水・水系水力の協調を考慮した需給計画機能の開発", 電力技術・電力系統技術合同研究会, Sep. 2002"

[4] 阿部, 小島, 他:多時刻断面での潮流制約・AFC容量制約を考慮したEDCの開発", 電力技術・電力系統技術合同研究会, Sep. 2002

[5] J.Patterson, D.Hennessy:"コンピュータ・アーキテクチャ",日経 BP 社, 403～436, ISBN 4-8222-7152-8

[6] K.Wadleigh, I.Crawford:"Software Optimization for High-Performance Computing", Prentice Hall, ISBN 0-13-017008-9