マルチコア向けソフトウェア開発／デバックの基礎と実際

～アルゴリズムの並列化から並列デバックまで～

枝廣　正人（NEC，東京大学）

マルチコア向けソフトウェア開発の難しさ

マルチコアの時代になり，高性能計算機から組込み機器まで複数プロセッサを有するLSIが多く用いられるようになってきた．現在のところ，マルチコアを活かすためには並列プログラミングを行うことが有効である[1)]．

しかしながら，科学技術計算のようなアプリケーションを除き，並列プログラミングにより開発されたソフトウェアをマルチコア上で実行させても性能が上がらないといったケースが多々ある．本稿では，マルチコアで性能を阻害する要因，それをふまえマルチコア向けソフトウェアに関する課題と現在の技術，課題解決のためのスケーラブルアルゴリズムと自動並列化コンパイラの組合せ，その他マルチコアを活かすためのソフトウェア技術について紹介する．

なお，「コア」にはさまざまな種類のIPコア（LSIの機能ブロック）が想定されるが，本稿では主にCPUのことをコアとよんでいる．

マルチコアについて

本章では，マルチコアのモデルと，特に組込みシステム開発において現状多く利用されているマルチコア向けソフトウェア開発方法について説明する．

■ アーキテクチャのモデル

マルチコア計算機のモデルについては，さまざまな解説，教科書があるが，ここでは文献1）に準じSMP型マルチコアを主として考えることとする（図-1）．SMP（Symmetrical Multi-Processor）型マルチコアは，複数の同じCPUが共有バスを経由してメモリを共有し，メモリ～バス～ＣＰＵの関係においてＣＰＵは対称となり，タスクはいずれのＣＰＵでも同様に実行できる．そのためSMP OSとよばれるOSを動作させ，（OSが）負荷分散を考えながら実行可能なタスクを動的にCPUに割り付け，実行させる．ただし後述するように，完全な対称性を持つSMP型マルチコアとしてだけではなく，対称性を崩し複数のグループに分割し，それぞれに別々のOSを搭載して用いるようなマルチコア活用法もある．

■ ソフトウェア開発手順

SMP型マルチコアモデル上でのソフトウェア構築は，現状では以下のような手順になることが多い．

Step1．対象となる問題に対する並列アルゴリズム構築

Step2．並列性を抽出するための並列プログラミング

まず，自分が解きたい問題の中で並列実行可能な部分を解析し並列動作可能なアルゴリズムを構築（Step1），それを文献1）において紹介されているような方法，たとえばOpenMP，Javaなどの言語，API（Application Program Interface）を用いてプログラムを行う（Step2）ことによりマルチコア上で実行可能なソフトウェア開発を行うことができる．

■ 性能阻害要因

上述したように，並列プログラミングを行ってマルチコア上で実行させただけでは性能があがらないといったケースが多々ある．何が問題なのだろうか．ここでは5つの要因を紹介する[2)]．

要因1：アムダールの法則（Amdahl's Law）

アムダールの法則は

$$性能向上 = 1 / (F + (1-F)/N)$$

で表される．ここでFは逐次実行部分（並列化できない部分）の割合，Nはコア数である[2)]．たとえばN=16のとき10倍以上の性能向上を得るためには$F \leqq 0.04$（96%以上が並列実行可能）であり，逐次実行部分が小さくないと十分な性能向上が得られないことを示す．

要因2：メモリアクセス

マルチコアではコア間（正確には別コア上のタスク間）でメモリを共有することが多い．そのため，単一コア（のタスク間）ではキャッシュを通してデータを授受しオー

バーヘッドが生じない場合でも，マルチコアではオーバーヘッドが生じ性能が低下する場合がある．

要因 3：コア間通信

単一コア上では同時刻に複数タスクが実行されることはないため，イベント待ちのような状態でなければタスク間通信によって CPU が長時間待ち状態になることはない．たとえば，あるタスク T がデータ受信待ちになったとしよう．その場合，システム全体がイベント待ちにならなければ他に実行可能なタスクが存在する．実行可能なタスクを順次実行すれば必ずタスク T への送信が実行され，タスク T は実行可能になる（そうでなければデッドロックである）．したがって他タスクの実行により，タスク T の待ち時間は隠され，CPU が長時間待ち状態になることはない．ところがマルチコアシステムでは，通信待ち状態になって実行可能タスク数がコア数よりも下回ると，タスクの待ち状態がそのまま CPU の待ち状態になり，CPU 稼働率，ひいては並列性能が上がらない場合がある．

さらに上述のメモリ遅延により，単一コア上でのタスク間通信よりもオーバーヘッドが大きくなることが多い．

図 -2 の例では，図 -2（a）はマルチコア上で実行すれば性能向上が得られるが，図 -2（b）はほとんど性能向上しないばかりか，コア間通信オーバーヘッドによっては単一コアよりも性能低下する場合もある．

要因 4：粒度

粒度は並列化のために分割された後の個々の部分プログラム（スレッド）の大きさである．上述したように，並列化を行うとさまざまな場面でオーバーヘッドがかかるが，粒度が小さい場合，オーバーヘッドが無視できなくなり，並列性能向上の阻害要因となる．

要因 5：負荷分散

スレッドの粒度にばらつきが大きいと負荷分散がうまくいかず，分割損が生じ，十分な並列性能向上が得られない場合がある．

粒度を細かくし，同時実行可能スレッド数を増やすことにより負荷を分散させやすくなるが，一方でオーバーヘッドが無視できなくなる，スレッド間の依存関係が増える，などの性能劣化要因があらわれる．

そのため粒度と負荷分散はトレードオフの関係にあり，適切な粒度が存在する．

さて，性能向上というよりはソフトウェア生産性向上に対する阻害要因であるが，マルチコアにおいてはデバッグが重要である．

要因 6：デバッガビリティ

マルチコアにおいては，単一コア上のマルチタスクシステムよりも不具合発生の要因が多くなる．

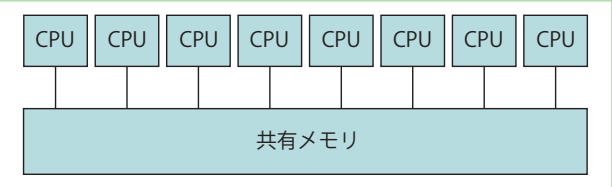


図 -1　SMP 型マルチコアアーキテクチャ

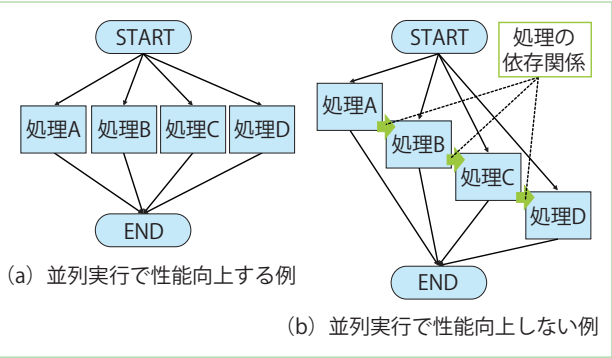


図 -2　処理の依存関係と並列性能向上

たとえば，あるデータに関し「初期状態→中間状態→終了状態」となる一連の処理に対し，処理時間が短ければ，単一コア上の実行ではその間に他のタスクが実行されることはほとんどない．ところがマルチコアでは同時に複数のタスクが実行されるため，データのロックを怠っていれば，中間状態に対して他のタスクがアクセスし，不具合が生じることもある．そのような場合は再現性が低く，かつ，いつどこで何の変数が何故想定外の値になったか，突き止めにくいことも多い．また，そこでロックを行ってデッドロックを起こすこともあり得る．

このようにマルチコアでは，同時処理（ロック忘れ），デッドロック，実行順序逆転（レーシング）などの問題が起こり，そのためのデバッグ容易化が必要である．

■ 並列化の課題

前節で示した 6 要因から，マルチコア向けソフトウェア開発のためには大きく 3 課題あることが分かる．

課題 1：分割

ソフトウェアを適切な粒度でスレッド分割すること．実行中大部分の時間において，コア数と比べて十分な数のスレッドが同時実行できるように分割すること．

課題 2：性能向上

オーバーヘッドを小さくしながら負荷分散を考え，スレッドのコア割当，データのメモリ配置を行い，最大の性能向上を達成すること．

課題 3：デバッグ容易化

ロックやデッドロックの可視化，レーシング解析（いつ何のタスクがどこで実行され，データが更新されたか，

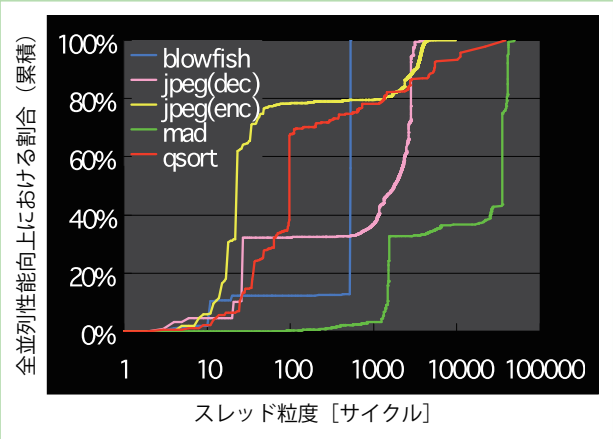


図 -3　アプリケーション並列性能向上におけるスレッド粒度の分布

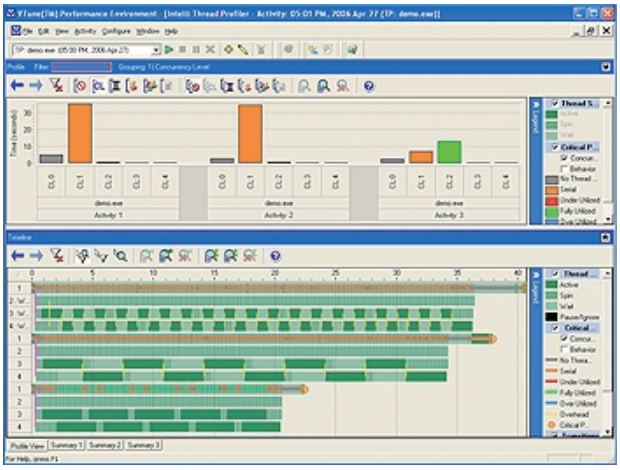


図 -4　性能解析ツール（http://www.xlsoft.com/jp/products/intel/threading/tpw/index.html）

など），性能上の不具合解析を容易にすること．

なお，課題 1 と課題 2 は関係が深い．粒度と負荷分散がトレードオフ関係にあるためである．オーバーヘッドはメモリ遅延，コア間通信遅延が関係するためアーキテクチャに依存であるが，与えられたアーキテクチャの元で最適な粒度に分割し，コア割当，メモリ配置を考えていく必要がある．これは，アーキテクチャごとに最適な粒度は異なる可能性が高いことを意味している．

図 -3 はあるアーキテクチャに向けてアプリケーションを最適化したときのスレッド粒度の分布を表している．これを見て分かるように，実際のアプリケーションでは，粒度を均一化することが難しく，さまざまな粒度を混在させ，その中で性能の最大化を考えていくことになる．

■ 並列化支援ツール

現状では上に示したソフトウェア開発手順が主に使われている．並列プログラミングを行うことにより，比較的単純なループなどは並列言語，API のライブラリが動的に負荷分散を計算し，そのときに利用可能なコア数からループを適切な回転数のサブループに分割，並列実行する．並列プログラミングに関する現状技術については文献 1）に詳しい解説があるので参照されたい．

しかし現実には並列プログラミングを行うことで最初から最高性能を達成することは多いとはいえず，また，さまざまな不具合もあり，並列化支援ツールを利用することになる．並列化支援ツールには大きく分けて性能解析ツールと，依存解析ツールがある．

- 性能解析ツール（課題 1，2 への対応）

性能解析ツールは，プロファイルのほか，並列実行情報，CPU 使用率，バス使用率などを解析する．ツールを使うことにより，プログラム開発者は，粒度と負荷分散の関係，オーバーヘッドの大きさなどを解析し，性能阻害要因を見つけ，プログラムを改善する．性能解析ツールの例を図 -4 に示す[3]．図 -4 上部は，3 バージョンのソースコードに対する並列実行時間と同時実行 CPU 数を示している．CL-0 は逐次実行コード，CL-1 ～ 4 はスレッド実行コードであり，1 ～ 4 は同時実行 CPU 数をあらわす．右端のグラフが最も並列度が高く実行時間が短縮されていることが分かる．図 -4 下部は 3 バージョンの実行において 4CPU の各 CPU における実行（濃い色），待機（薄い色）の状態を時間軸に沿って表している．図 -4 上部右端にあたる実行（下部における下の 4 本の棒グラフ）において並列実行の様子が見られる．

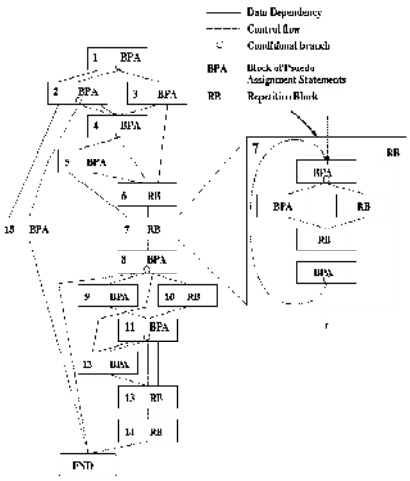
- 依存解析ツール（課題 3 への対応）

依存解析ツールは変数（メモリアドレス）の依存関係，実行順序関係，ロック状態などを解析する．

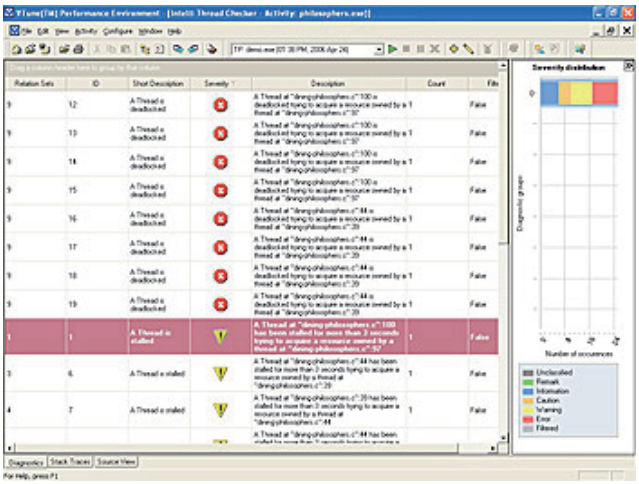
依存解析には静的解析と動的解析がある．静的解析はプログラムコードを解析することにより行う．解析の網羅性が高い反面，ポインタが使われている場合など，依存の有無が判定できない場合もある．

これに対して動的解析は，プログラムを実行させ，メモリアクセス履歴から解析することにより行う．依存の有無は正確に判定できる反面，入力データ依存であり，かつ静的解析と比べて網羅性が低いという問題がある．

静的解析の例を図 -5（a）[4]，動的解析の例を図 -5（b）に示す[3]．図 -5（a）では，プログラムをブロックに分割し，ブロック間の制御依存関係，データ依存関係をグラフ表示した結果を示している．たとえばブロック 7 と 8 に注目すると，制御としては 7 の後に 8 が実行されることになっているが，データ依存関係がないため並行実行が可能である．図 -5（b）は実際の並列実行において，同アドレスのメモリに対するアクセス順序逆転などに関する履歴が左部に，回数の統計が右部に表示されている．こ



(a) 静的解析
(早稲田大学笠原研資料より)



(b) 動的解析
(http://www.xlsoft.com/jp/products/intel/threading/tcw/index.html)

図 -5　依存解析

れにより実行時の依存関係解析などが可能となる.

なお，静的解析と動的解析を組み合わせて使ったとしても，網羅性が高く正確な依存解析は難しく，マルチコア向けソフトウェア品質面の大きな課題となっている.

マルチコア向けソフトウェア開発の将来像

さて，ここまでで現状のマルチコア向けソフトウェア開発の手順を紹介した上で，性能を阻害する要因，そして性能向上のための課題，現状技術としての並列化支援ツールについて説明してきた.

課題に関してはアーキテクチャ依存部分も多く，上述した並列ソフトウェア開発（Step1. アルゴリズム構築，Step2. 並列化プログラミング）において，実行するアーキテクチャを意識する必要がしばしばあり，汎用性を持たせて実現することは容易ではない.

現在グラフィクスなどのアプリケーション分野では，分野特化型のプロセッサおよびそれに向けた並列言語，API による並列プログラミングにより，数百コアを有するようなプロセッサに対しても実行環境が動的にプログラム分割，メモリ配置を行うため，分野内ではある程度汎用的なソフトウェア開発ができている [1].

マルチコアが広がり，コア数が増え，メニーコアとよばれる時代に向け，より一般的なアプリケーション，アーキテクチャに対しても汎用的で，品質を高めやすいソフトウェア開発方法論を確立していく必要がある.

■ ソフトウェア開発手順

今後コア数が増大，コア間接続がネットワークオンチップなどに進化し，メモリアクセス遅延，コア間通信遅延がタスクのコア割当，変数のメモリ配置によって大きく異なることを考えると，以下のような手順が望まれる.

Step1. スケーラブルアルゴリズム構築
Step2. 静的解析容易なプログラミング
Step3. 自動並列化コンパイラによるコード生成

ここで Step3 の自動並列化コンパイラの利用が前章の手順と大きく異なるところである．自動並列化コンパイラによりアーキテクチャ依存部分をコンパイラに任せることができれば，すなわちメモリ遅延，コア間通信遅延などを考慮しながら粒度を決めてプログラムを分割しデータのメモリ配置なども行うことができれば，ソフトウェア技術者はさまざまなアプリケーション，アーキテクチャに対してより汎用的な開発を進めることができる．また，コンパイラでの静的解析を最大限利用できるようなプログラミングを行うことが，ソフトウェア品質向上，ひいてはソフトウェア生産性向上の鍵と考えている.

以下ではそれぞれの Step について説明を加える.

■ スケーラブルアルゴリズム

筆者らはスケーラブルアルゴリズムを以下のように定義している [5].

要件 1. P 台の CPU を用いたときに時間に関する計算複雑度が (1/P)
要件 2. 単体 CPU での実行時間は，単体 CPU 向けに最適化された従来アルゴリズムと同等
要件 3. 複数 CPU での並列性能向上は高いことが望まれる

要件 1 の計算複雑度に関する項目に関しては，さまざまな問題に対して過去多くの理論研究がなされてきた．その際，要件 2 の視点も含めてアルゴリズムが構築されてきたわけでは必ずしもなく，現在のパソコンなど，そ

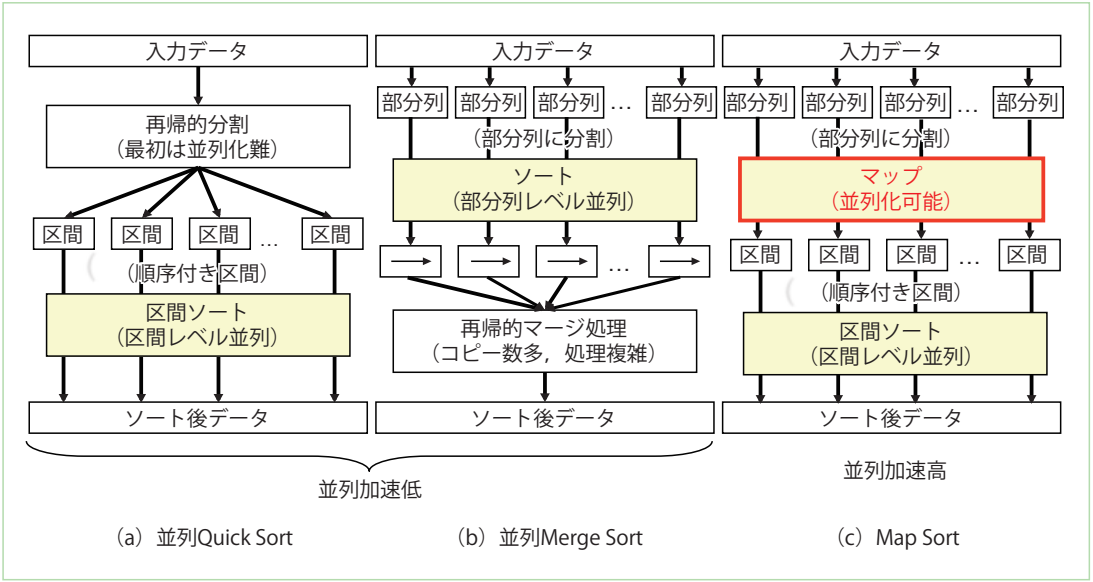


図-6　並列ソートアルゴリズムの比較

れほど多くないコア数のアーキテクチャにおいてはあまり使われていないものが多い．

要件3は前章で述べた性能阻害要因と関連があり，アーキテクチャ依存が強い．そのためアルゴリズムの汎用性を追求していく場合には考えにくい項目であるが，自動並列化コンパイラによる適切な粒度への分割，変数のメモリ配置を行いやすくするために，スケーラブルなアルゴリズムを構築していく上では，

（1）大部分が並列化可能であること
（2）静的解析により，粒度が細かく依存関係が少ない，多数の部分に分割できること
（3）メモリアクセスの静的解析が容易であること

などが留意すべき項目となる．

今後コア数増大に伴い，可能な限り多くの並列性を抽出する必要がある．これまでは科学技術計算やメディア処理などに注力されてきたが，今後は基本的なアルゴリズムも含め，スケーラブルなアルゴリズムの構築，ライブラリ化が必要である．

■ 静的解析容易なプログラミング

自動並列化コンパイラを用いるため，性能向上のためには静的な並列化解析，メモリアクセス解析などを行いやすくするようなプログラミングが必須である．前述したように，静的な解析可能性はソフトウェアの品質向上，保守容易性の向上の点からも重要である．

たとえば，C言語のポインタは静的解析を難しくするが，これに対して制約を加え，自動並列化可能にする試みも提案されている[4]．また，Javaのように元々ポインタが定義されていないような言語を用いる方法もある[1]．

今後，並列性抽出が容易で，かつ産業上使いやすい言語，APIが重要になっていくが，これまでのプログラミング言語の発展と同様，多くの提案，淘汰が行われ，成長していくものと考えられる．現在提案されている言語，APIについては文献1）を参照されたい．

■ 自動並列化コンパイラ

自動並列化コンパイラについては，スーパーコンピュータや高並列サーバ向けなどに数多くの研究がなされてきたが，現在マルチコア向けの自動並列化コンパイラとして代表的なものとしてOSCARコンパイラがあげられる．OSCARコンパイラについては文献4）を参照されたい．

■ 並列化課題に対する考察

ここで前章に列挙した並列化課題に対する効果について考察する．

まず課題1（分割）であるが，スケーラブルなアルゴリズムが構築できれば，静的解析可能な記述および並列化コンパイラの利用によって適切な分割は容易となる．

課題2（性能向上）については，並列化コンパイラにアーキテクチャモデル，遅延情報を与えることにより，適切な粒度に分割し，コアへの割当，メモリ配置を行うことが可能である．しかしながら，分岐が多いスレッドなど，スレッド実行時間の予想がつかない，入力データに依存して変動する，といった場合においては最適性を追求することは難しく，実行時における動的スケジューリングを併用するといった方策などが必要である．

課題3（デバッグ容易化）に関しては，静的解析を可能にすることにより，より多くのデバッグ情報を抽出し可視化することが可能になるため，デバッグ容易化に大きく貢献する．ただし，情報抽出，可視化はデバッグ容易性の最初の一歩に過ぎず，プログラム開発者支援方法なども含め，さらなる研究が必要である．

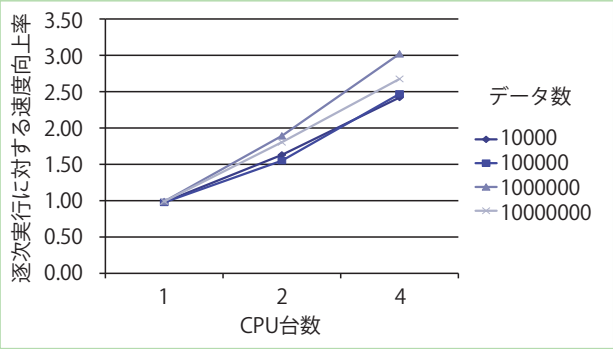


図 -7　Map Sort の自動並列化（早稲田大学笠原研資料より）
［実験環境：OSCAR コンパイラ（早大），MPCore × 4(ARM-NEC)］

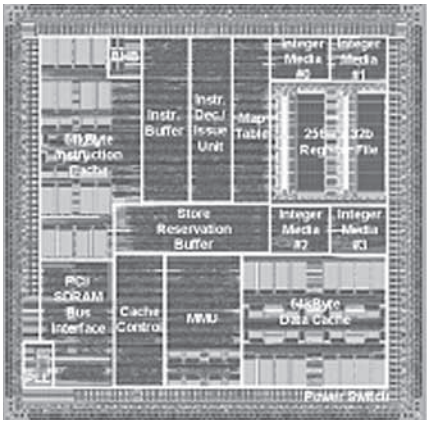


図 -8　投機マルチスレッドプロセッサ Merlot

■ 適用例：ソーティング

ここでは適用例としてソーティングをあげる．筆者らは，スケーラブルなソーティングアルゴリズムとして Map Sort を提案した[5]．従来，単一プロセッサ向けに効率の良いアルゴリズムとして Quick Sort や Merge Sort が提案されてきたが，Quick Sort はアルゴリズムの前半で再帰的に分割していく部分，Merge Sort はアルゴリズムの後半で再帰的に配列マージを行う部分で並列性能が出しにくいという問題があった（図-6）．

これを解決する方法が Map Sort である．詳細は省略するが，Quick Sort において再帰的分割を行い，複数区間に対応した複数配列に分割する処理の代わりに，Map とよぶデータ構造を用いて一括して並列分割する．

Map Sort を OSCAR コンパイラによって並列化し，NEC エレクトロニクス社 NaviEngine（ARM MPCore 4 コア）上で実行した結果を図-7 に示す．スケーラブルな性能向上が得られていることが分かる．

■ ハードウェアによる支援

これまでは主にソフトウェアに関して述べてきたが，並列ソフトウェア開発の課題解決にはハードウェア支援も有効である．ここでは 3 課題のそれぞれに対するトピックを紹介する[2]．

• 投機マルチスレッド実行

プログラムの分割を考えるにあたり，変数等の依存関係は大きな問題である．たとえばハッシュ関数 hash() を用いたプログラム，

```
for (i=1; i<n; i++) {
    j = hash(X[i]);
    A[j] の値の参照，更新；
}
```

について考えたとき，入力値配列 X[] の値の並びによっては連続する i に対してハッシュ値 j が同じになる場合が存在する．データ更新時に A[j] をロックする必要があり，オーバーヘッドが大きくなるため，プログラム分割時の制約が大きくなる．

このような場合に対するハードウェア支援として利用可能な技術が投機マルチスレッドである．2 つのスレッドに依存関係あり，並行実行の正当性が保証されていない場合にも並行実行させることを**投機実行**という．スレッドを投機的に実行させ，メモリ依存チェック，および依存があった場合の再実行をハードウェアで支援する．オーバーヘッドはゼロではないが，ソフトウェアからメモリをロックする必要がなくハードウェアで対処するため，ハッシュのように低い確率で依存がある場合には性能向上が見込まれる．図-8 は投機マルチスレッドアーキテクチャの試作 LSI Merlot である[6]．

• コア間接続アーキテクチャ

携帯型情報機器や車載情報機器においてユーザアプリケーションの実行をつかさどる組込み向け SoC（System on a Chip）のように必要最小限のメモリしか持てない場合，複数コアからのアクセスがバス混雑を引き起こし，遅延を増大させ，期待される性能向上が実現できない場合がある．

このような場合にバスを複数にする方策があるが，コスト増大を最小限にする必要がある．これに対し，実行されるアプリケーションの組合せがおおよそ分かる組込み機器の特徴を捉え，メモリアクセスを分類し，必要最小限の面積増加で性能向上を図るアーキテクチャ（カテゴライズドバスとよぶ）が提案されている．

さらに将来のコア数増大に対しては，SoC 全体をバス構造にすると電力面などで不利となるため，コア間接続をネットワーク構造とする NoC（Network on Chip）が有力視されており，現在盛んに研究されている（図 -9）[7]．

• デバッグ支援ハードウェア

マルチコア SoC システムでデバッグを行う場合，複数コアでのトレース，ブレークが基本的な課題となる．この場合，同じクロックに対するトレース情報，同じク

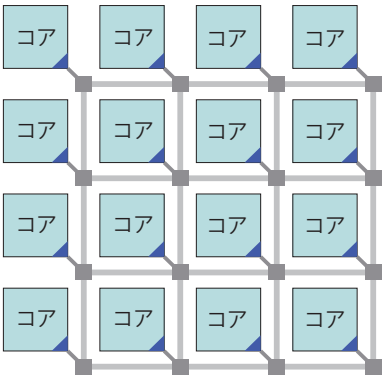


図 -9　ネットワークオンチップの例

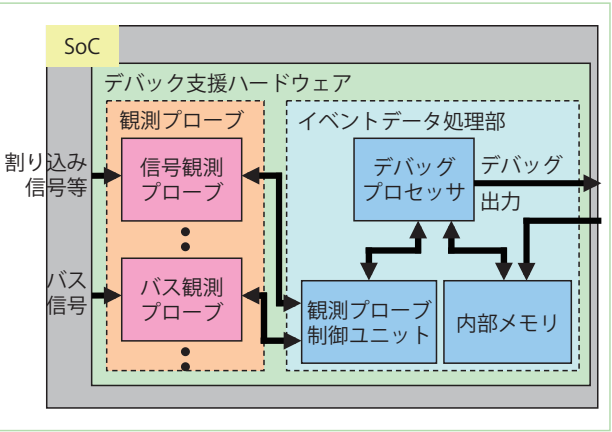


図 -10　デバッグ支援ハードウェア

ロックでのブレークなど同時性が重要となる．また，すべてのコアからのトレース結果を SoC から出力することを考えるとバンド幅，電力などが増大する．

そのため，SoC 内での同時ブレーク，トレース取得のフィルタリング（図-10）などのハードウェア支援が考えられている[8]．

マルチコア活用ソフトウェア技術

ここまでは，主に SMP 型マルチコア上において，1 つのアプリケーションを性能向上するための課題とそれを解決する方法について述べてきたが，マルチコアの活用方法はそれだけではない．物理的に分離した複数コアを活用することにより，新しい価値を生み出すことができる．本稿の最後に，マルチコアを活用するためのソフトウェア技術として，デュアル OS，動的ドメイン制御について紹介する．

■ デュアル OS

デュアル OS とは，マルチコア上で 2 つの OS を実行させることである[9]．SMP 型マルチコアシステムでは SMP OS とよばれる OS を実行させ，アプリケーションを各 CPU に動的に分散させることが多い（図-11(a)）．これに対してデュアル OS システムでは，図 -11（b）のように，CPU を分け，それぞれに別々の OS を実行させ，かつアプリケーションも分類して実行させる．元々 2 つのシステムを 1 つの SoC 上でサブシステムとして実行させているような状態になる．ここではそれぞれのサブシステムをドメインとよぶ．

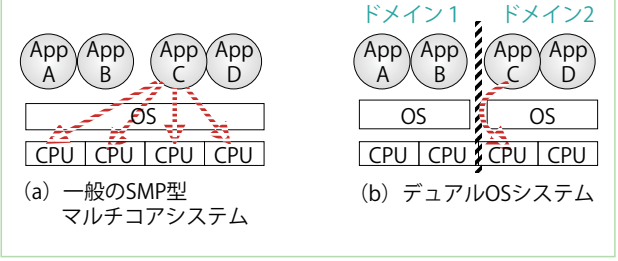


図 -11　デュアル OS システム

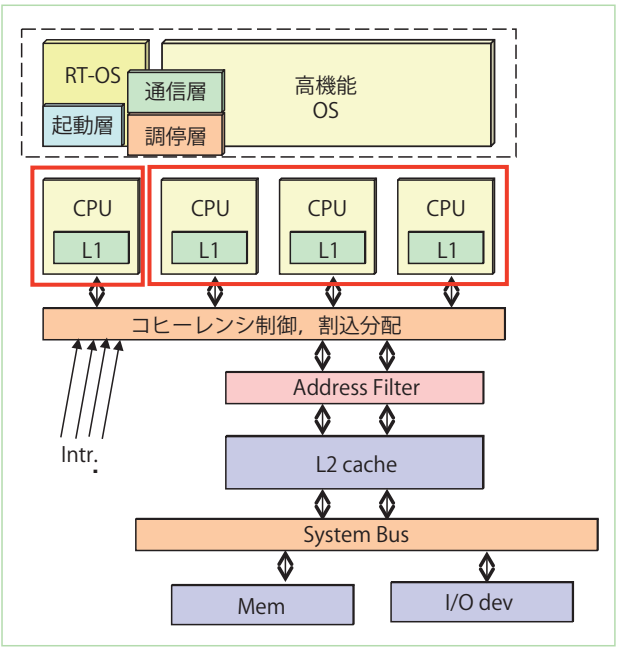


図 -12　デュアル OS の実現例

最近の組込みシステムでは，組込み制御の性質を持つアプリケーションと，ブラウザなどのアプリケーションが同時に実行されることがあり，異なる要件であるため単一プロセッサ上での実現が難しいが，マルチコア上ではリアルタイム OS と，Linux のような OS を同時実行させ，アプリケーションを分離させることにより，比較的容易に実現できる．

デュアル OS の実現例を図 -12 に示す．ハードウェアによる支援があることが望ましく，キャッシュコヒーレンシの制御，タイマや割り込み信号の分離，分配などが望まれる．ソフトウェアとしてはハードウェアリソース配分を行う起動層，OS 間の排他制御を行う調停層，OS

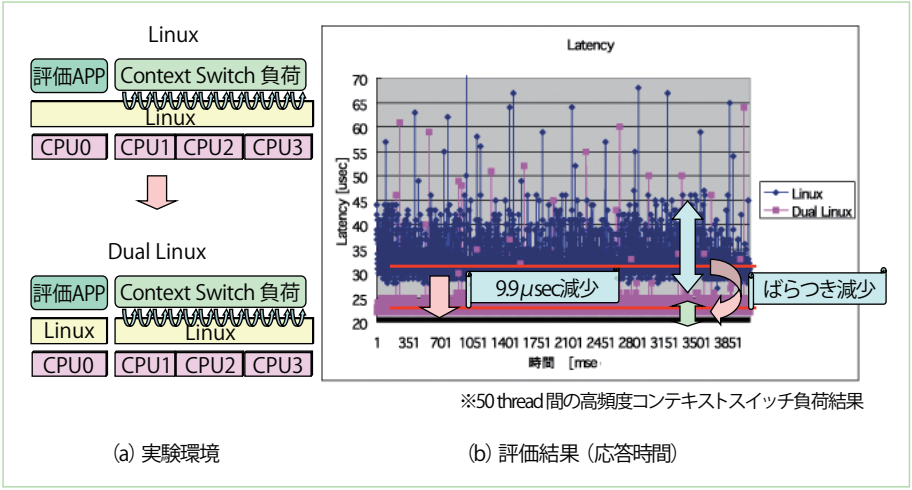


図 -13　デュアル OS の効果 (Linux 対 Dual Linux)

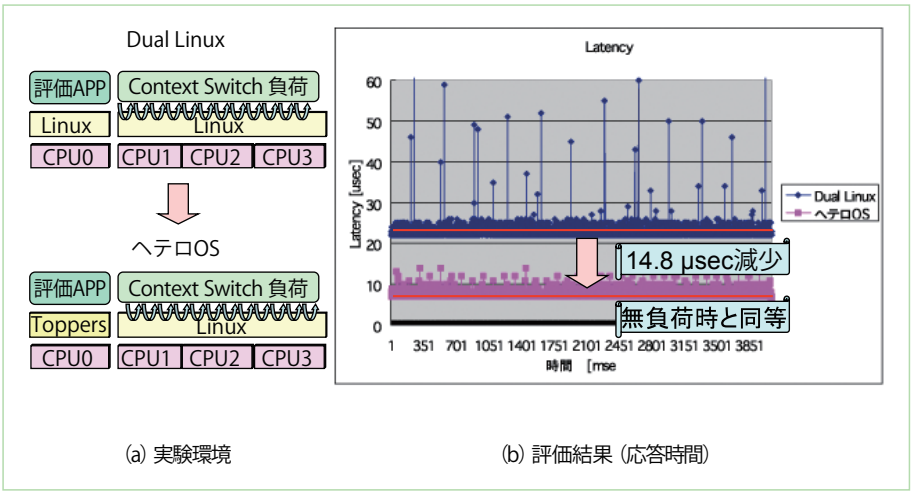


図 -14　デュアル OS の効果 (Dual Linux 対 Heterogeneous Dual OS)

間通信を行う通信層が必要である．

デュアル OS の効果を図-13，図-14 に示す．図 -13 は単一の（SMP）Linux とデュアル Linux との比較である．アプリケーションは評価対象のアプリケーションと，高頻度コンテキストスイッチ負荷をかけるアプリケーションの 2 種を同時実行させる（図 -13(a)）．図 -13（b）に示すように，Dual Linux の方が応答性に優れ，かつ応答時間のばらつきも小さいことが分かる．図 -14 は同様の実験に関するデュアル Linux とデュアル OS システム（Linux と RTOS（Toppers））との比較である．RTOS の利用により，さらに性能が改善することが分かる．

デュアル OS は次世代車載情報系プラットフォームに関するプロトタイプにも利用されている [10]．

■ 動的ドメイン制御

ドメインによりサブシステムに分離することは特に組込みシステムにおいて有効であるが，各ドメインの処理負荷に応じて動的に CPU 数を変更する処理を動的ドメイン制御とよぶ [11]．

この技術は図-15 に示すように，Linux においてホットプラグとよばれるような OS 機能を用いて実現できる．動的ドメイン制御の応用例を図-16 に示す．システムドメイン上を動く優先度の高いアプリケーション負荷に応じて，システムドメインに割り当てられる CPU 数は変化する．拡張ドメイン 1 と 2 は場合によっては同じ CPU 上で動作することになるが，アプリケーション QoS とよばれる手法を用いて優先度制御も可能である [12]．

マルチコア向けソフトウェア開発方法論の確立に向けて

本稿では，マルチコア向けソフトウェアの課題と，それに対する現状技術を紹介した．マルチコアに対して並列プログラミングは有効であるが，それによってマルチコアを有効に活かすためにはさまざまな課題がある．本稿ではマルチコアの有効利用を阻害する要因および課題を整理し，スケーラブルアルゴリズム，自動並列化コンパイラを中心とした並列化ソフトウェア開発の将来像に

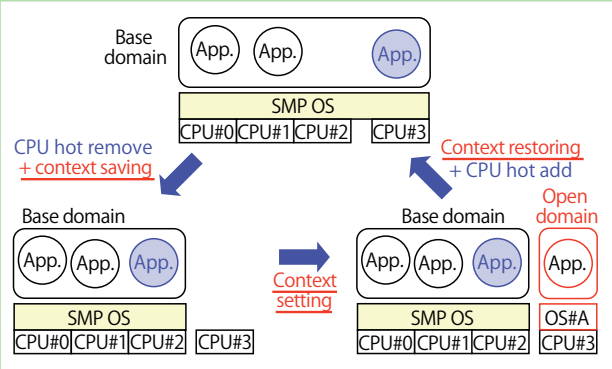


図 -15　動的ドメイン制御

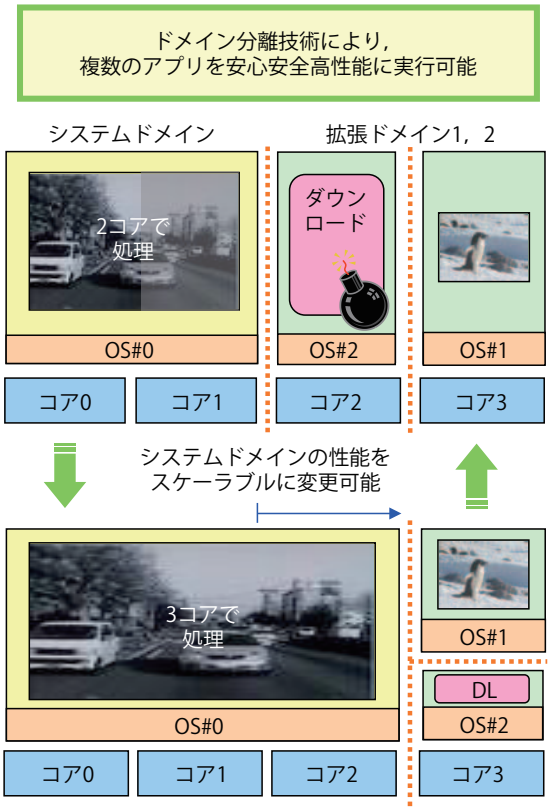


図 -16　動的ドメイン制御の応用例

ついて述べた．また，マルチコアの物理的な分離性を利用したマルチコア活用についても紹介した．今後，マルチコアの進展に伴い，ソフトウェア開発方法論を確立し，ツール等を整備していくことが必須である．

参考文献

1）松崎他：特集：マルチコアを活かすお手軽並列プログラミング，情報処理，Vol. 49, No. 12 (Dec. 2008).
2）枝廣，黒田：組込みプロセッサ技術（第 9 章マルチコア），CQ 出版社，2009.
3）http://software.intel.com/en-us/intel-sdp-home/
4）http://www.kasahara.elec.waseda.ac.jp/
5）枝廣，山下：Map Sort：マルチコアプロセッサに向けたスケーラブルなソートアルゴリズム，情報処理学会第 129 回 SLDM 研究会 (2007).
6）Nishi, N., et al.：A 1GIPS 1W *Single-Chip Tightly-Coupled Four Way Multiprocessor with Architecture Support for Multiple Control Flow Execution*, ISSCC (2000).
7）Lajolo, M., et al.：New Developments and Trends in Networks on Chip, Tutorial, (2009).
8）鈴木，酒井，鳥居：マルチコア SoC の高度な観測を可能とするプログラマブルなデバッグ支援ハードウェアの開発，情報処理学会第 134 回 SLDM 研究会 (2008).
9）阿部，酒井：組込み向けマルチコアプロセッサ MPCore を用いた応答性／機能性両立環境評価～制御処理と情報処理の融合にむけて～，情報処理学会第 134 回 SLDM 研究会 (2008).
10）NEC：次世代車載情報系プラットフォーム向けマイコン，OS を開発，プレスリリース，2007 年 12 月 11 日.
11）Inoue, H., et al.：Dynamic Security Domain Scaling on Symmetric Multiprocessors for Future High-End Embedded Systems, CODES+ISSS (2007).
12）西原，石坂，酒井，宮崎：ユーザ利用状況に応じたアプリ性能制御のためのリソース配分方法．信学技報，HCS2008-73 (2009).

（平成 21 年 6 月 29 日受付）

枝廣　正人（正会員）　eda@bp.jp.nec.com
NEC システム IP コア研究所主席研究員および東京大学情報理工学系研究科客員教授．1985 年工学修士（東京大学），1999 年 Ph.D.（プリンストン大学）．マルチコア向けソフトウェアの研究開発に従事．文部科学大臣表彰，元岡賞，本会坂井記念特別賞，山下記念研究賞などを受賞．電子情報通信学会，日本 OR 学会，IEEE 各会員．