Vol.2019-EMB-51 No.1
2019/6/21

小規模組込み向けFRPの実行モデルの並列化

櫻井 義孝[1,a)]　渡部 卓雄[1,b)]

概要：小規模組込みシステム向け関数リアクティブプログラミング言語 XFRP のための並列化手法を提案する．本研究の目的はマルチコア環境を活かすことで XFRP プログラムの応答時間を短縮することと，応答時間を予測可能にすることにある．OS によるサポートが期待できない小規模システムにおいてもこの目的を達成するために，静的スケジューリングにもとづく並列化手法を提案する．本発表では予備的な結果として pthread を用いた実装にもとづいた評価を行い，提案手法が応答時間の短縮に寄与することを示す．

キーワード：関数型リアクティブプログラミング，並列プログラミング，小規模組込みシステム

1. はじめに

関数型リアクティブプログラミング (Functional Reactive Programming, FRP) とは連続値である時変値 (Time-Varying Value) と離散イベントを宣言的に記述しリアクティブシステムを構築するプログラミングパラダイムである．FRP はロボティクスや GUI などのりアクティブなシステムが要求される分野に適したプログラミングパラダイムであると考えられている．

多くの FRP は Haskell のライブラリとして実装されているため，実行には豊富なリソースが要求され組込みシステムには向かない．そこでリソースの制限された小規模組込みシステムを対象とした純粋 FRP 言語として Emfrp[1] が開発された．更に，Emfrp の後続言語の 1 つに分散システム向け FRP 言語である Distributed-XFRP[2] がある．これは Erlang に翻訳され Actor モデルの上で動作し，各時変値は Actor で表現される．

Emfrp は一般的な FRP と同様に Emfrp はシステムを有向非巡回グラフ (DAG) として表す．DAG 上ではノードがそれぞれ時変値を表し，エッジは時変値の依存関係を表現する．実行の際には，各ノードの更新がノード間の依存関係であるエッジに沿って更新がデータフローのように伝搬する．Emfrp においては，システムを表現するグラフの形状はコンパイル時に決定され，実行時に変更されない．そのため，ノードの更新順序はコンパイル時に決定される．

Emfrp は全てのノードの更新を繰り返すことでリアクティブなシステムを実現する．そのため，システムの時変値の数が増えたときに全てのノードを更新するのにかかる時間が大きくなりシステムの応答性が悪くなる可能性がある．しかし，リアクティブシステムにおいて応答性の低下は重大な問題となり得る．

組込みシステム向け CPU でもマルチコアを使用可能な CPU が増加している．しかし，Emfrp の実行モデルは CPU のコア数に関わらずシングルスレッドで動作する．そのため，マルチコア使用可能な CPU では十分にリソースを活用することができない．

そこで，本研究ではマルチコア CPU において十分に計算資源を活用し，リアクティブシステムの応答性を向上するため，Emfrp の実行モデルを並列に動作可能なモデルに変更した純粋 FRP 言語 XFRP-Core を開発した．現在の XFRP-Core は Linux 上で動作するように開発され，*pthread* を使用しているが，アルゴリズム自体は *pthread* に依存するものではなく，小規模な RTOS や OS が動作しないような環境にも応用することができる．

2. 並列化アルゴリズム

XFRP-Core の並列化のための静的スケジューリングアルゴリズムを説明する．XFRP-Core ではシステムを時変値の組み合わせで構築し，システムは DAG で表現される．本研究の提案する並列化アルゴリズムは最も離れた *sink* ノードへの最長距離を基準に行われる．(以降，最長距離はノードから最も遠い *sink* ノードへの距離とする．) ここで *sink* ノードとは，DAG の最も後ろのノード，つまり出次数が 0 のノードのことである．各ノードから最も遠い *sink* ノードへの最長距離が等しい任意の 2 つのノードの間には依存関係がないため同時に更新することができる．

[1] 東京工業大学 情報理工学院 情報工学系
[a)] sakurai@psg.c.titech.ac.jp
[b)] takuo@c.titech.ac.jp

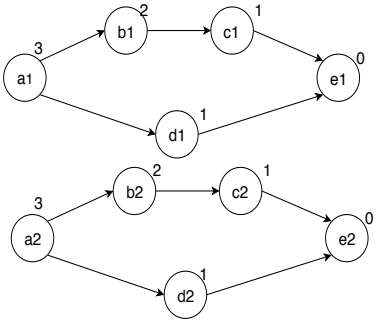

図 1 サンプルグラフ

つまり，コンパイル時に次の 2 つの手順を行う．1 つめは，各ノードから最も遠いノードへの距離を計算する．ノード i のもつこの距離を d_i とする．2 つめは，d_i の等しいノードの集合 S_{d_i} を同時に更新できるようなコードを出力する．例えば図 1 について考える．このグラフはライフゲームを XFRP-Core で実装したときにシステムを表現するグラフの一部である．図がシステムを表すグラフだとする．図において，各ノードの右上の値は最も遠い頂点への最長距離 d を表す．そして，最長距離 k が $k = d_i$ の等しいノード i の集合を同時に実行するようなコードに変換すればよい．例えば，最長距離 1 のノードの集合 S_1 は $S_1 = \{c1, d1, c2, d2\}$ となる．これを例えば 2 スレッド使用可能な環境ならばスレッド 1 は $c1, d1$ を，スレッド 2 は $c2, d2$ を実行するように割り当てればよい．このとき，使用可能なスレッド数はコンパイル時に既知であるとする．ここでの割り当ては，各スレッドが近い数のノードを処理するように行う．

全てのノードから *sink* ノードへの最長距離の時間計算量は下の Algorithm1 のように行うことで $O(V + E)$ になる。ここで，V はノード数，E は辺の数を表す．Emfrp のコンパイラはコンパイル時にノードの更新順序を決定するために DAG 上でトポロジカルソートをする必要がある．これは $O(V + E)$ の計算量を要求する．そのため，静的スケジューリングによってコンパイルの時間計算量は変更されない．

3. 評価実験

提案した並列化の性能を確かめるために，XFRP-Core 上で提案する実行モデルと Emfrp の実行モデルを実装し，一定回数のイテレーションにかかる時間を測定する評価実験を行った．実験環境には OS は ubuntu18.04，CPU は Intel(R)i5-7200U CPU 2.50GHz を使用した．また，XFRP-Core コンパイラは XFRP のソースコードから C 言語に変換し，並列実行のためのスレッドには Pthread ライブラリを使用し，スレッド間の同期にはpthread_barrierを使用した．Emfrp の実行モデルは 1 スレッドで実行し，提案する

Algorithm 1 Calculate Max-Distance

```
入力: グラフ G, 全てのノードの集合 V
出力: 各ノードの最長距離をもつ配列 dist
function dfs(node,dist):
    if G[node] is empty then
        return dist[node] = 0
    else if dist[node] ≠ inf then
        return dist[node]
    end if
    for all v in G[node] do
        d[node] = max(d[node], dfs(v))
    end for
    return dist[node]
Procedure calcMaxDistance(G, V)
    int dist[|V|] = {inf, inf, ..., inf}
    for all node in V do
        dfs(node, dist)
    end for
    return dist
```

アプリケーション	Emfrp	XFRP-Core(4 スレッド)
ライフゲーム	38.98(sec)	17.89(sec)
熱拡散シュミレータ	36.25(sec)	24.39(sec)

表 1 実験結果

XFRP-Core の実行モデルを計測するときは 4 スレッドを使用した．これは実験に使用する CPU が 2 コア 4 スレッドを使用可能だからである．実行時間の計測はそれぞれ 5 回行いその平均時間を各モデルの実行時間とした．実験対象としたアプリケーションはライフゲームと熱拡散シュミレータである．実験結果は図 3 である．

ライフゲームは 10^4 個のセルのライフゲームを 10^4 回のイテレーションにかかる時間を計測した．Emfrp の実行モデルで実行したときと比べて約 45.9%高速化されている．

熱拡散シュミレータは板を 150x150 の要素に分割し，初期値として各要素の温度を与える．XFRP 上の各イテレーションで単位時間毎の各要素の温度を計算する．このアプリケーションを 10^4 回イテレーションすることで，10^4 単位時間の間の各要素の温度を予測し，かつこのイテレーションにかかる時間を計測した．このアプリケーションでは並列化により実行時間は 67.28%になっている．

参考文献

[1] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-scale Embedded Systems, *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, New York, NY, USA, ACM, pp. 36–44 (online), DOI: 10.1145/2892664.2892670 (2016).

[2] Shibanai, K. and Watanabe, T.: Distributed Functional Reactive Programming on Actor-based Runtime, *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2018, New York, NY, USA, ACM, pp. 13–22 (online), DOI: 10.1145/3281366.3281370 (2018).