SML# の並列処理機能とその性能

上野 雄大　大堀 淳

マルチコア CPU がコモディティとして広く普及している一方で，宣言的で高水準な記述が可能な関数型言語におけるマルチコア CPU のサポートは十分とは言えない．この状況を打開する戦略のひとつは，関数型言語の言語処理系と並列計算を実現する軽量スレッド技術とをそれぞれ独立したモジュールとして直接連携させることである．この洞察に基づき，我々はシステムが提供する並列スレッド機能と直接連携する関数型言語処理系を実現する基盤技術を確立し，対話環境や自動的メモリ管理などの ML 系関数型言語の特徴を失うことなく高い並列計算性能を発揮する関数型言語 SML# の開発を推進してきた．最新の SML# コンパイラは，軽量スレッドの並列動作を止めない並行並列 GC を搭載し，いくつかのベンチマークプログラムにおいて C 言語に匹敵するスケーラビリティを示している．本発表では，SML# コンパイラの並列計算機能および並列処理性能の概要を報告する．

1　はじめに

マルチコア CPU はコモディティとして広く普及しており，大型計算機から組み込み機器に至るまで，計算能力を有するあらゆるデバイスに搭載されている．科学技術計算，人工知能，ビッグデータ解析などの高い計算効率が求められる分野では，これら多数のコアを持つプロセッサを最大効率で駆動するための並列プログラミング技術は必須である．メニーコア化と並列プログラミングへの流れは今後ますます加速し，近い将来には，複数のコアを高効率で駆動することがあらゆるドメインにおいて要求されるであろうことは想像に難くない．ソフトウェア開発の広い範囲において，複数のコアをひとつの目的のためにフル活用する高水準並列プログラミング技術の確立は，幅広いアプリケーション開発において枢要な課題と言える．

宣言的で高水準な記述が可能な関数型言語による並列計算は，並列プログラミングの難しさを大きく改善する可能性がある技術のひとつである．関数型言語と並列計算の相性が良いことは古くから主張されており，関数型言語をベースとした並列計算モデルや自動並列化の研究は古くから綿々と続けられている．しかしながら，関数型言語におけるマルチコア CPU 対応の並列処理のサポートは，C 言語に代表される手続型言語におけるそれと比較して，いまだ十分とは言えない．その原因のひとつは，関数型言語の実行モデルが要求する自動的なメモリ管理，およびその実装技術であるガベージコレクション（GC）にある．GC はプログラムが使用するメモリ全体を管理するため，大域的な同期を本質的に含む逐次的な処理であり，従ってプログラム全体の並列性能を落とす主要な要因となり得る．また，GC の実行にはスレッドスケジューラと GC の連携が必要不可欠であり，その連携のために，GC とマルチスレッドの両方を装備する言語の多くは独自のスレッドスケジューラを言語処理系に組み込んでいる．結果として，システムライブラリとの互換性は低くなり，最新のシステムプログラミング研究の成果を含む軽量スレッドライブラリなどの活用が難しい．

この状況を打開するひとつの戦略は，大域的な同期を含まず，かつシステムが提供するスレッドスケジューラと互いの独立性を保ったまま直接連携する GC 方式を確立することである．この GC アルゴリズムを核として関数型言語処理系の実装技術を再構築

Katsuhiro Ueno, Ohori Atsushi, 東北大学, Tohoku University.

することで，マルチコア CPU 上の並列処理を実現する最新のシステムプログラミング技術を，将来に渡って，そのまま関数型言語から利用することが可能となると期待される．この洞察の下，著者らは GC アルゴリズムおよび関数型言語処理系の研究開発を行い，関数型言語 SML#[3] から軽量スレッドライブラリ MassiveThreads[1] を直接利用する並列プログラミングを可能にした．2020 年 5 月にリリースされた SML# 3.6.0 版（以下，最新の SML# と呼ぶ）は，世界を止めない並行並列 GC を装備し，いくつかのベンチマークプログラムにおいて C 言語に匹敵する並列処理性能を発揮する．本発表では，SML# が提供する並列処理機能およびそれを実現する GC 技術を概観すると共に，最新の SML# が示す並列処理性能の概要を報告する．これらの技術的詳細は他の機会に発表する予定である．なお，本研究のこれまでの結果は，[2] や [5] で口頭発表されている．本発表ではその後の進展も踏まえた現状を報告する．

2 SML# の並列処理機能

前述の通り，高い並列性能を有する関数型言語を実現するための我々の戦略は，関数型言語とは独立に研究されているシステムプログラミング技術を，その独立性を保ったまま，関数型言語から利用可能にすることである．従って，SML# は言語独自のスレッドモデルや並列計算機能をユーザーに提供しない．代わりに，SML# は POSIX スレッド（Pthreads）API および軽量スレッドライブラリ MassiveThreads を直接サポートする．ユーザーは，C 言語との FFI を通じて，これらの API そのものや，これらの API を利用して書かれたライブラリを SML# にインポートすることができる．例えば，新たなユーザースレッドを作成する MassiveThreads の API である

```
myth_thread_t myth_create
    (void *(*)(void *), void *);
```

は，以下のようにして SML# にインポートすることができる．

```
type myth_thread_t = unit ptr
val 'a#boxed myth_create =
  _import "myth_create"
    : ('a -> unit ptr, 'a) -> myth_thread_t
```

ここで`#boxed` は，型変数がヒープにアロケートされた値のみ動く（従って C の `void*`と互換な表現を持つ）ことを表す型カインドである．`myth_create` 関数の第 1 引数はコールバック関数へのポインタである．SML# の FFI により，第 1 引数として任意の SML# 関数を `myth_create` 関数に渡すことができる．

便宜のため，Pthreads API および MassiveThreads API をほぼそのままインポートしたバインディングライブラリが SML# コンパイラに付属している．これらのライブラリは，並列性を阻害しないように注意深く書かれている以外は，前述した FFI を用いたユーザープログラムとして構築されている．これらのライブラリが提供する `Pthread` および `Myth` ストラクチャのシグネチャの一部を図 1 に示す．

SML# のスレッドはシステムライブラリが提供するスレッドそのものであるから，GC の対象ではない．従って，`create` で作成したスレッドの資源は，スレッド API を C 言語から利用する場合と同様に，ユーザープログラムから `detach` または `join` を呼ぶことで解放されなければならないことに注意が必要である．

SML# の並列処理性能を試す最も簡便な方法は，MassiveThreads API の `create` および `join` を用いてタスク並列プログラムを書き，それを SML# コンパイラの対話モードから実行することである．関数型言語の一般的な書き方で再帰関数を書き，再帰呼び出しを行う箇所を `create` と `join` で囲めばよい．ただし，計算にかかる時間が軽量スレッドの管理にかかるオーバーヘッドを上回るように，計算の規模がごく小さい場合は `create` を呼ばないようにする（カットオフを設定する）．`Myth` ストラクチャを用いて書いた `fib` 関数の例を図 2 に示す．このプログラムは `fib 40` を計算するために 3,524,577 個の軽量スレッドを生成する．MassiveThreads の軽量スレッドは，実際の CPU コア数や OS の制限を受けることなく，低コストで大量に作成することができる．MassiveThreads は大量に作成された軽量スレッドを各 CPU コアに最適にスケジューリングする．2 つ以上の CPU コアを使用するには，SML# の対話モードを起動する際に

```
structure Pthread : sig
  structure Thread : sig
    type thread
    val create : (unit -> int) -> thread
    val detach : thread -> unit
    val join : thread -> int
    val exit : int -> unit
    val self : unit -> thread
    val equal : thread * thread -> bool
  end
  structure Mutex =
  struct
    type mutex
    val create : unit -> mutex
    val lock : mutex -> unit
    val unlock : mutex -> unit
    val trylock : mutex -> bool
    val destroy : mutex -> unit
  end
  ...

structure Myth : sig
  structure Thread : sig
    type thread
    val create : (unit -> int) -> thread
    val detach : thread -> unit
    val join : thread -> int
    val exit : int -> unit
    val yield : unit -> unit
    val self : unit -> thread
    val equal : thread * thread -> bool
  end
  structure Mutex : sig
    type mutex
    val create : unit -> mutex
    val lock : mutex -> unit
    val unlock : mutex -> unit
    val trylock : mutex -> bool
    val destroy : mutex -> unit
  end
  ...
```

図 1　Pthread および Myth ライブラリのシグネチャ

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n =
    if n < 10
    then fib (n - 1) + fib (n - 2)
    else
      let
        val t = Myth.Thread.create
                  (fn _ => fib (n - 2))
      in
        fib (n - 1) + Myth.Thread.join t
      end
```

図 2　SML# による並列 fib 関数

MYTH_NUM_WORKERS 環境変数にコア数をセットする．例えば，4 コアマシンで

```
MYTH_NUM_WORKERS=4 smlsharp
```

として起動した対話モードで fib 40 を実行したときにかかる実時間は，環境変数をセットせずに起動した対話モードで実行したときに比べて約 1/4 になるはずである．

3　SML# の並行並列 GC

前述の通り，システムが提供するスレッドライブラリを直接利用可能な関数型言語を実現するにあたり中核となる技術は，スレッドスケジューラと独立性を保ったまま連携するガベージコレクタである．この性質を満たすために GC アルゴリズムに求められる要件は以下の通りである．

- ミューテータスレッドの実行を止めないこと．最大の並列性能を得るためには逐次性を徹底的に除去しなければならない．関数型言語処理系が導入する逐次性のひとつは，本質的にグローバルな GC 処理である．マルチコア CPU を駆動するスレッドスケジューラの性能を最大限に発揮させるためには，複数のコアで同時に実行されるミューテータスレッドの動作を GC の実行が妨げてはならない．この性質を満たすためには，GC は世界を止めない並行 GC である必要がある．
- ミューテータが複数のコアにスケールするのに

合わせて GC も複数のコアにスケールすること．関数型言語のスレッドの実行には，CPU タイムスライスの他に，ヒープ（メモリ）の連続的な割り当てが必要である．このメモリの割り当ては GC によって実現される．同時に実行されるミューテータが増えると，それらが要求するメモリ量も必然的に増える．複数のミューテータによるメモリ要求に応え続けるためには，メモリ要求の増加に合わせて GC の効率も高めなければならない．この性質を満たすためには，GC は並列 GC である必要がある．

- スレッドスケジューラによる自由なスケジューリングを妨げないこと．マルチコア CPU の性能を最大限に引き出す軽量スレッドスケジューラは，可能な限り多くのコアに均等に軽量スレッドを割り当てようとする．そのため，軽量スレッドはその実行中にあるコアから別のコアに移動する可能性がある．一方，メモリの割り当てには，CPU タイムスライスの分配と同様，コンテキストの管理が必要である．このコンテキストには，GC を制御するためのグローバルな情報を含む．これらコンテキストを，スレッドスケジューラの動作とは独立に，スレッドが移動する可能性も考慮して管理する方式の確立が必要である．

我々は，軽量スレッドスケジューラの下でのスケーラブルなメモリ管理を実現するため，マルチコア対応の並行 GC[4] を見直し，これらの要件を満たす並行並列 GC 方式を新たに構築し，SML# コンパイラに実装した．その概要は以下の通りである．

- SML# の並行並列 GC は，ワーカー–ユーザーモデルに基づくノンプリエンプティブな軽量スレッドに対応する．ワーカースレッドが CPU タイムスライスをユーザースレッドに供給するのと同様に，ワーカースレッドがユーザースレッドに連続的なメモリ割り当て機能を提供する．ワーカースレッドはそのために必要なコンテキストを持つ．なお，GC の正しさを保証するため，SML# コードを実行している間は，ユーザースレッドのコンテキストスイッチは起こらないものとする．
- 各ワーカースレッドはそれぞれ独立したアロケーション領域を持つ．各ワーカースレッドは，自身に割り当てられたユーザースレッドのルートセットを列挙し，生きているオブジェクトをトレースし，自分のアロケーション領域内の未使用領域を回収する．ワーカースレッドとユーザースレッドとの関連付けは，スレッドスケジューラとは独立に行う．ガベージコレクタはスレッドローカルストレージのみを用いてスレッドの生成やスレッドの移動を認識する．
- 3.5.0 版以前の SML# に実装されていた並行 GC 方式 [4] とは異なり，コレクタ専用のスレッドを用意しない．代わりに，各ワーカースレッドは時折コレクタとして振る舞う．グローバルな GC の進行はコレクタの集合によって管理される．コレクタとしての仕事は他のワーカースレッドとは並行並列に行われる．この構成により，GC の並列度はワーカースレッドが増えるに従って自動的に増加する．シングルスレッドで動作しているときは自然に逐次 GC となる．

4 SML# の並列処理性能

最新の SML# には以下のタスク並列ベンチマークプログラムが付属する．

- fib．図 2 に示された `fib` 関数を用いて `fib 40` を計算する．
- mandelbrot．2048×2048 ピクセルのマンデルブロ集合を描画する．
- nqueens．14-クイーン問題を解く．
- cilksort．4,194,304 個の浮動小数点数を並列マージソートする．

これらのベンチマークプログラムを SML# 3.5.0 版（SML#-3.5），および最新の SML# に対してバグフィックスとチューニングを行ったバージョン（SML#）で実行したときの平均実行時間（average task time）および最大 resident set size(MaxRSS) を図 3 に示す．図では，参考のため，同等のプログラムを c 言語で実行した結果（C）も示している．この結果から分かる通り，最新の SML# は従来の SML# に比べて，ほぼ同等のメモリ消費量で大幅な並列性能の向上を達成している．

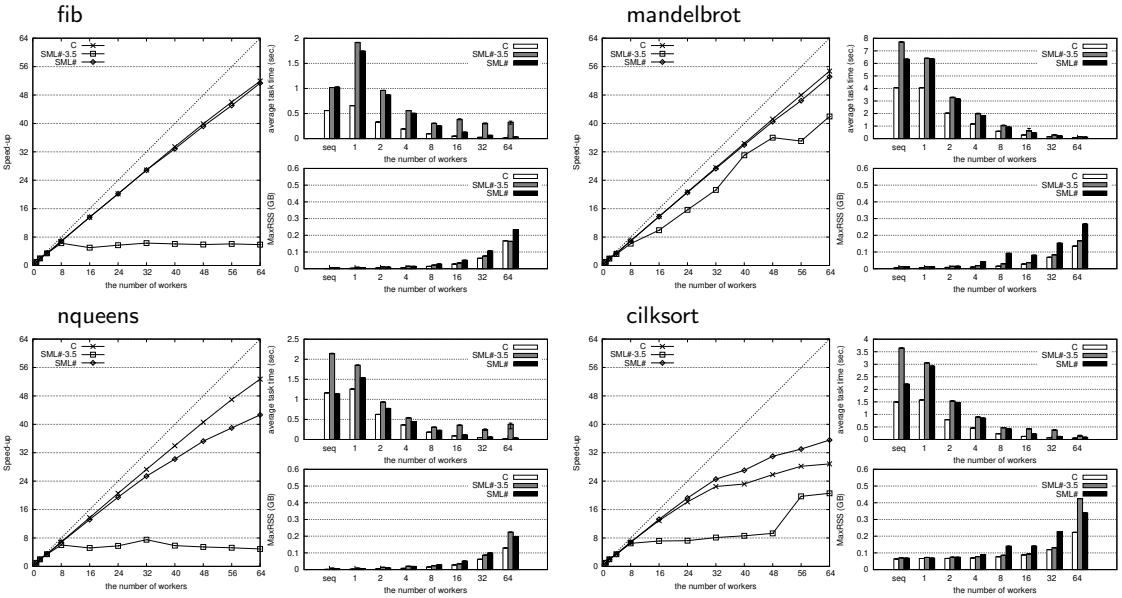

図 3　並列ベンチマーク結果

5　まとめ

大域的な同期を含まず，かつ POSIX スレッドライブラリおよび軽量スレッドライブラリ MassiveThreads と互いの独立性を保ったまま直接連携する並行並列 GC の開発に成功し，SML# に実装した．本発表では，SML# が提供する並列処理機能とその基盤となる GC 方式の概要，および最新の SML# の並列処理性能の評価結果の概略を報告した．達成された枠組みと技術の詳細は，他の機会に発表する予定である．

謝辞 本研究の一部は JSPS 科研費 19K11893 および 18K11233 の助成を受けたものです．

参 考 文 献

[1] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, *Concurrent Objects and Beyond*, Lecture Notes in Computer Science, Vol. 8665, 2014, pp. 222–238.

[2] Ohori, A., Taura, K., and Ueno, K.: Making SML# a general-purpose high-performance language, Unpublished manuscript, 2017.

[3] SML#, http://www.riec.tohoku.ac.jp/smlsharp/, 2006–2020.

[4] Ueno, K. and Ohori, A.: A Fully Concurrent Garbage Collector for Functional Programs on Multicore Processors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, New York, NY, USA, ACM, 2016, pp. 421–433.

[5] 上野雄大，大堀淳，田浦健次朗: SML#と MassiveThreads の統合による超並列言語の実現，日本ソフトウェア科学会第 *36* 回大会論文集, 2019.