SML#とMassiveThreadsの統合による超並列言語の実現

上野 雄大　大堀 淳　田浦 健次朗

マルチコア CPU のコモディティ化が進む中、プログラミング言語、特に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対応の並列処理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、マルチコア上の超並列計算を実現するシステム技術と関数型言語の言語処理系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。この連携の確立によって、種々のアーキテクチャ上で最適にスケジュールされる軽量スレッド技術の進歩が、将来に渡って、関数型言語で利用可能になると期待される。我々は、この課題の解決を目指し、システムプログラミング研究と言語コンパイラ研究を統合し、関数型言語 SML#と軽量スレッドライブラリ MassiveThreads との直接連携に成功した。本発表では、この統合の概要および SML#の MassiveThreads 拡張の概要を報告する。

1　はじめに

種々の検証や学習など，高い計算能力を必要とするアプリケーションの需要が高まるのに伴い、高性能アプリケーション構築技術が求められつつある。現状では、これらアプリケーションのための計算資源はクラウドコンピューティングやスーパーコンピュータによって集中的に提供されているが、近い将来には、認識や制御における柔軟でリアルタイムな処理の需要が拡大すると共に、個々の計算機器に高い計算能力が求められると考えられる。このようなニーズに応え得るアプローチのひとつは、マルチコア CPU を活用した超並列処理である。現在すでに、マルチコア CPU は PC から組込み機器に至る幅広いデバイスに採用されており、また 32 コア以上を搭載するメニーコアプロセッサも急速に普及しつつある。これら大量のコアをひとつの目的のためにフル活用する超並列アプリケーションを構築する技術の確立が、大規模サーバやクラウドなどに限らない、幅広いアプリケーション開発にとっての枢要な課題と言える。この課題に応える有望なアプローチのひとつは、並列処理単位を抽象し、ハードウェアを最大限に活用する、宣言的な超並列プログラミング環境の構築と考える。各スレッドの実行単位である関数をデータとして抽象し、それら第一級の関数を種々のデータ構造とともに宣言的にプログラムすることができる関数型プログラミング言語は、この超並列プログラミング技術の基盤になると期待される。

しかしながら、現状では、宣言的で高水準な記述が可能な高階関数型言語におけるマルチコア対応の並列処理のサポートは十分とは言えない。その主な要因のひとつとして、関数型言語のスレッドや並列計算プリミティブのサポートが、急速なハードウェアの進歩と、ハードウェアを活用するシステムプログラミング技術の革新に十分に追いついていないことが挙げられる。多くの既存の関数型言語コンパイラは独自の並行並列実行モデルを持ち、実行時処理系の開発では、それら並行並列実行モデルを直接実装するアプローチが取られてきた。このアプローチは、ガベージコレクション（GC）ベースのメモリ管理下で多相関数や第一級のスレッドなどを実現する上で数々の優位点を

Atsushi Ohori, 東北大学, Tohoku University.
Kenjiro Taura, 東京大学, The University of Tokyo.
Katsuhiro Ueno, 東北大学, Tohoku University.
(日本語表記は五十音順、英語表記は alphabetical order である。)

持つものの、その反面、システムライブラリとの互換性が低く、結果として，異なるコアにスケジュールされるマルチスレッドの実行さえ十分にサポートされていない（システムが提供するスレッドを利用できない、あるいは利用できたとしても期待される並列性能を発揮できない）のが現状である。

この状況を抜本的に打開する戦略の一つは、マルチコア上の超並列計算を実現す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に統合し、GC を含む実行時処理系がスレッドライブラリと直接連携する技術を確立することである。この連携によって、種々のアーキテクチャ上で最適にスケジュールされる超並列スレッド技術の進歩が、将来に渡って、そのまま関数型言語で利用可能になると期待される。この洞察の下、著者らは、関数型言語コンパイラ研究とシステムプログラミング研究を統合することによって、超並列処理を容易かつ効率よく使用できる関数型言語コンパイラの実現を目指し共同研究を推進してきた。その結果、軽量スレッドライブラリ MassiveThreads を関数型言語 SML#に直接結合することに成功した。その結果得られた言語処理系は延べ 100 万を超える軽量スレッドをメニーコアプロセッサの各コアに効率よくスケジュールできること、またその上に宣言的な超並列処理が実現できることを確認した。本発表では、この統合の技術的課題および SML#の MassiveThreads 拡張の概要を報告する。なお、本研究の初期の結果は、[3] 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。SML#の MassiveThreads 拡張は、並列スレッドスケジュール下のメモリ管理技術や超並列言語機能の型理論的基礎づけを含む、数多くの技術課題を克服した結果であり、それらの詳細は、他の機会に発表する予定である。

2 超並列関数型言語を実現する戦略

メニーコアを最大限に活用した並列プログラミングを実現する上で、最も有望と思われるアプローチのひとつは、軽量スレッドによるタスク並列計算である。軽量スレッドとは、実際のハードウェアコア数やオペレーティングシステムの制限を受けることなく大量に低コストで作成可能なスレッドを指す。このアプローチでは、プログラマには軽量スレッドの生成と同期機構を提供し、実行時処理系は軽量スレッドのハードウェアコアへの最適なスケジューリングを行う。最小限のコンテキストを持つ軽量スレッドの実現と、タスクスティーリングを基本とする軽量スレッドスケジューリング技術は、システムプログラミング分野で広く研究され、その成果として、メニーコアプロセッサ上でユーザプログラムから簡便に使用できる軽量スレッドライブラリが開発されている。この軽量スレッドアプローチは、第一級の関数と再帰的データ構造の柔軟かつ宣言的な定義を得意とする関数型言語とよく整合し、宣言的並列プログラミングの基盤となり得るものである。

本研究における我々の戦略は、前述のとおり、この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、関数型言語コンパイラに直接取り入れることである。これにより、関数型言語は、軽量スレッドの実装技術の最新成果をそのまま享受することができ、また、軽量スレッド技術は、それら技術を宣言的プログラミングに展開できる。この洞察のもと、我々は、C 言語で書かれた軽量スレッドライブラリ MassiveThreads [1] を、C 言語との直接連携機能を装備している関数型言語 SML# [4] の実行時システムに結合することを具体的な課題とした。さらに、この結合を基礎として、より高水準な並列言語機能の開発に取り組んだ。

3 超並列関数型言語を実現する上での課題

MassiveThreads と SML#は、独立して開発されたソフトウェアであり、また互いに、スレッドのスケジューリングやメモリ管理に係る低レベルの処理基盤でもある。それらの直接連携には数々の技術的課題がある。それら課題のうち主なものは以下の 2 点である。

- スレッドスケジューリングとメモリ管理の統合
  C 言語のスレッドとは異なり、関数型言語のスレッドの実行には、CPU タイムスライス以外に、関数型言語処理系が管理するヒープ（メモリ）の連続的な割当が必要である。このメモリ割当は、

本質的にグローバルな GC によって実現される。この GC ベースのメモリ管理機構を GC とは独立に並行に動作する軽量スレッドスケジューラを統合することが、軽量スレッドと関数型言語との統合の最大の課題である。CPU のタイムスライスの分配と同様、メモリの割当にはコンテキストの管理が必要である。このメモリコンテキストは、レジスタ集合やスタックといったスレッドローカルな情報からなるスレッドコンテキストとは異なり、GC を制御するためのグローバルな情報を含む。これは、共有メモリモデルにおける GC が本質的にグローバルな処理であることに由来する。これらコンテキストを、軽量スレッドスケジューラのスケジューリングオーバーヘッドを増加させずに管理する方式の確立が必要である。

- 超並列言語機能の設計

  MassiveThreads によって提供されるプログラミングモデルは、タスク並列モデルである。このモデルの下では、関数型プログラムにおける再帰的な関数呼び出しを再帰的なスレッド生成に置き換えるだけで、容易に並列処理を実現できる。その一方で、関数型言語においては、大規模集合データ（bulk data）を処理の分配や結果の集約を行うコンビネータで処理するプログラミングパターン（例えば MapReduce モデルや並列スケルトンなど）も広く普及しており、大規模データの宣言的な処理に貢献している。タスク並列に加え、これら集合データを並列処理する言語機能の設計と、その軽量スレッド基盤上での効率的な実装技術の開発が、超並列関数型言語を実現する上でのさらなる課題のひとつであると考える。さらにそれら超並列言語機能を、関数型言語が提供する高階の関数、多相型、ユーザ定義の再帰的なデータ構造などの宣言的な機能と統合することで、単なる数値配列上の MapReduce 計算を超える、幅広い並列アプリケーションの基盤となると期待される。

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

図 1　MassiveThreads–SML#プログラミングの例

4　SML#の MassiveThreads 拡張の概要

著者らは、これら課題を含む種々の課題を克服し、MassiveThreads と SML#とを統合するための枠組みと技術を開発し、超並列機能を実現する SML#コンパイラの開発に成功した。この開発成果の一部は、SML# 3.4.0 版としてすでにリリースされている。より完全なサポートは今後のリリースで順次利用可能になる予定である。

本発表では、SML#の MassiveThreads 拡張の概要を報告する。達成された枠組みと技術の詳細については、主な技術課題ごとに他の機会に発表する予定である。主要な開発項目は以下の通りである。

- MassiveThreads バインディングの作成

  SML#では軽量スレッドを生成・同期する特別な構文を用意せず、MassiveThreads の C 言語向け API を直接呼び出すバインディングライブラリのみをユーザーに提供することとした。このライブラリは、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、通常の SML#の C との直接連携機能のみを用いて構築されている。このライブラリを用いたプログラム例を図 1 に示す。

- 並行 GC の拡張

  軽量スレッドスケジューラの下での現実的でスケーラブルなメモリ管理を達成するため、マルチコア対応の並列 GC[5] のアーキテクチャを見

直し、軽量スレッドにスケールする並行並列 GC 方式を新たに構築し、実装した。この GC は、スレッドローカルストレージのみを用いて、スレッドの生成やタスクスティーリングの発生を認識し、GC とは独立に動く軽量スレッドライブラリと連携する。

- スケジューラの改良
  我々の目標は、言語からの完全な独立を保ったまま、MassiveThreads ライブラリを SML#実行時処理系に統合することである。この制約の下、軽量スレッドが言語処理系のメモリコンテキストに効率よくアクセスできることを達成するため、効率よい軽量スレッドローカルメモリの実現とチューニングを行った。
- 大規模集合データに対する並列処理構文
  リストや配列上の並列処理に加えて、近年注目されている vertex-centric 計算なども包摂する新たな宣言的並列構文と実装技術の確立を目指し、[2] によって提案された等式（方程式）の超並列処理を実現する言語構文

  `_foreach` *id* `in` *dataExp*
  [`where` *setupExp*] `with` *pat*
  `do` *iteratorExp* `while` *predicateExp*
  `end`

  とその軽量スレッドへのコンパイル技術を構築した。これは、大規模集合データ *dataExp* の各要素について、*predicateExp* が満たされている間 *iteratorExp* を繰り返し評価し、結果を大規模集合データとして返す式である。

実現された SML#は、実際に、100 万スレッドを超えるスレッドを駆動することができ、さらに、36 コアマシン上での初期的な実験により、コア数に対してスケールすることが確認されている。

5 まとめ

システムプログラミングと言語コンパイラ研究を統合し、超並列スレッドライブラリ MassiveThreads と SML#の直接統合に成功した。本発表では、その課題および拡張の概要を報告した。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。

参 考 文 献

[1] J. Nakashima, K. Taura. Massivethreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, LNCS 8665, pages 222–238, 2014.

[2] S. Nishimura, A. Ohori. Parallel functional programming on recursively defined data via data-parallel recursion. *J. Funct. Program.*, 9(4):427–462, 1999.

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

[4] SML#. http://www.riec.tohoku.ac.jp/smlsharp/, 2006–2019.

[5] K. Ueno, A. Ohori. A fully concurrent garbage collector for functional programs on multicore processors. In Proc. ACM ICFP 2016, pages 421–433, 2016.