Making SML# a General-purpose High-performance Language

Atsushi Ohori[1] Kenjiro Taura[2] Katsuhiro Ueno[1]

1 Tohoku University 2 The University of Tokyo

Abstract

We present the progress in the development of *high-performance SML#*. The key component is the combination of lightweight thread scheduling in system research and memory management in compiler research. This part has already been achieved; the programmer easily writes code that create and use millions of threads on multicore processors. In the talk, we describe the key features of the high-performance SML# and demonstrate its feasibility through examples running on multicore processors.

1. Introduction

Today, multicore chips have become commodity hardware and are widespread in various computer platforms. Cost-efficient cloud computing platforms start providing clusters of multicore processors. Even many-core chips have already been merchandised for supercomputers. Exploiting multicore and many-core processors will become essential in future software development for any serious problem domains. The key programming technology in this direction is massive parallelism, currently mainly used in high-performance computing (HPC), such as physical simulations and other scientific computation. The general motivation of our research is to make massive parallelism on multicore processors a general-purpose programming tool for general program domains involving various algorithms and data structures.

For this purpose, a typed higher-order functional programming language should serve as an ideal framework. As argued by many authors, functional programs with collection data is a rich source of parallelism, and higher-order combinators are useful for representing parallel computation. Despite these apparent benefits, general-purpose functional programming languages have not yet become languages for massively parallel high-performance computing on multicore processors. The major obstacles include conventional garbage collection (GC) methods, which are inherently single-threaded and global, and the lack of direct interface to the underlying operating system. To overcome these obstacles, we have been developing SML# [3] with a fully concurrent GC and the direct C function interface.

Our current goal is to combine these SML# features with the system programming technology for thread-scheduling to make SML# an HPC language embodying massive parallelism on multicore processors. In the presentation, we outline the technical goals to be achieved and the major results so far obtained, and demonstrate them.

2. Requirements of a Functional HPC Language

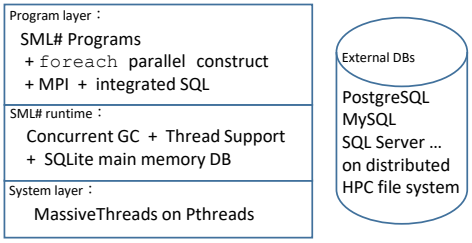
The apparent prerequisite of a high-performance functional language is the ability to run a large number of threads on multicore processors. Recent system programming research has established the technologies of lightweight thread programming. Using them, the C (or any other HPC language) programmer can easily write code that create a large number of user-level concurrent threads well over a million, which are automatically scheduled to available cores. Without this basic ability, elaborate parallel constructs in functional languages may not be useful in practical HPC applications.

Another desired feature for an HPC language is persistent bulk data access. An HPC application typically operates on a large collections of data, which must be loaded from an external storage or network by massively many computation nodes and threads. For this purpose, conventional text stream I/O is not satisfactory; a functional HPC language should provide a mechanism to access persistent bulk data in parallel.

We should also develop a high-level and declarative language constructs for writing parallel and concurrent programs. Although this aspect has been extensively studied in functional programming research, we do not see much results that show their scalability and usability with millions of threads.

3. High-performance SML# Architecture

To realize a functional HPC language that meets the above requirements, we have started a collaborative project to combine thread scheduling in system programming research and functional language compilation to form the high-performance SML#. This has been a part of an accepted plan (#86) of Japanese Master Plan of Large Research Projects 2017, Science Council of Japan. The overall architecture of the high-performance SML# is shown below.



The lowest level (system level) layer consists of the user-level lightweight thread library, MassiveThreads [1], implemented on top of POSIX threads (Pthreads) library. This layer is responsible for lightweight thread creation and scheduling. The SML# runtime lies between system and user program layers and manages the SML# heap with the non-moving (non-copying, non-compacting) concurrent GC [5]. The user program is written by using well-tuned system-provided libraries and high-level parallel programming constructs.

4. The Key Features of High-performance SML#

Direct system interface through C APIs In SML#, the data representation of basic types, such as `int` and `real`, are the same as those of C. Arrays and records also have the same memory lay-

[1] A summary of a talk to be given at ACM ML family workshop 2017. A related technical paper [6] is available on request by email.

out of their corresponding types in C. This native representation enables the programmer to call C functions from SML# without any data conversion. Huge mutable SML# matrix arrays can therefore be shared with C code. This feature allows us to import essential libraries for HPC, such as MPI and BLAS, in SML# without any runtime overhead. It is also the basis for integrating the MassiveThreads library in the SML# runtime system, as described below.

Non-moving concurrent GC The SML# runtime system incorporates a fully concurrent garbage collector [5]. This collector does not require any global barrier synchronization; it does not block any mutator threads at all due to garbage collection. Consequently, this GC method scales up to concurrently running multiple mutator threads on many-core machines. By combining it with the direct C interface, the programmer can call Pthreads functions directly from SML# code.

The MassiveThreads library integration The MassiveThreads library provides efficient creation for a massive number of user-level threads and automatic load balancing by a work stealing scheduler. Its primary characteristic distinguished from other lightweight thread runtime systems is that it is provided as a C library callable from ordinary C programs without any special compiler support. With the SML# features described so far, it should be possible to "plug-in" MassiveThreads to add the lightweight thread capability to SML#.

However, a straightforward binding would not achieve expected performance mainly due to the high memory demand. To solve this issue, we reorganize the GC algorithm in accordance with the implementation model of MassiveThreads. MassiveThreads creates a fixed number (typically the number of processor cores) of worker threads and schedules each user-level thread to a worker thread. In this user/worker thread model, a worker thread manages CPU resource for user thread scheduling. In addition to the CPU resource, SML# code also require memory resources provided by the GC module. For a SML# function code to be executed in the user/worker thread model, we must adopt that model for memory allocation and reclaimation managed by GC.

With this extension, we have successfully integrated MassiveThreads in the SML# runtime system. The following SML# program computes `fib 40` recursively in parallel:

```
open Myth.Thread
val cutOff = 10
fun fib 0 = 0
  | fib 1 = 1
  | fib n =
    if n < cutOff
    then fib (n - 2) + fib (n - 1)
    else join (create (fn () => fib (n - 2)))
         + fib (n - 1)
val _ = fib 40
```

This program uses two MassiveThreads functions: `create` f for creating a user thread computing f`()` and `join` t for waiting for the completion of thread t. It creates 3,524,577 threads in total. Figure below shows our initial benchmark result of `fib` program on a machine equipped with 64 cores of AMD Opteron 6380 1.4GHz and 252 GB main memory.

Declarative data parallel language construct To use massively parallel thread abstraction described above in data parallel programming, we developed a new language construct

> `_foreach` id `in` $dataExp$ [`where` $setUpExp$]
> `with` pat `do` $iteratorExp$ `while` $predicateExp$
> `end`

which transforms given data $dataExp$ of a recursively-defined data to another recursively-defined data. It can deal with not only arrays but also lists and other dynamically linked data structures. The underlying idea is the data parallel transformation of a system of equations [2]. This construct is suitable for representing massively parallel computation on a large cluster of multicore processors connected by MPI-style message passing.

High-performance I/O through SQL integration Data loading must also be performed in a massively parallel manner for massively many threads running on a cluster of multicore processors. The current practice is to set up a parallel file system that allows a cluster to perform collective access through MPI-IO. On top of this, there are some libraries to provide a parallel bulk data storage such as HDF5 and NetCDF. However, developing a parallel and declarative I/O interface remains a challenge. Our strategy to tackle this problem is to exploit relational databases, which are inherently parallel and declarative. SML# already has a seamless SQL integration [4]. Combining this feature with massively parallel thread abstraction, we can achieve a parallel and declarative I/O interface both in the sense of providing an SQL-based declarative alternative to HDF5 and of making SQL a parallel construct for big data processing.

For freely combining SQL programming with threads, we have refined the SQL integration of SML# in such a way that the three components of a SQL `SELECT` statement, namely a select list, `FROM` clause, and `where` clause are independently defined as polymorphic functions that are to be composed into a single query.

5. The Current Status and Future Direction

Among the features described above, the direct C interface and non-moving concurrent GC are original features of SML#. In this project, we have developed MassiveThreads integration with the necessary GC extension, `_foreach` language constructs, and SQL integration refinements. We shall release the SML# with these extensions. We are currently working on performance tuning, MPI integration, and parallel high-performance data loading through SQL integration.

References

[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] SML#. http://www.riec.tohoku.ac.jp/smlsharp/, 2006–2017.

[4] A. Ohori, K. Ueno. Making standard ML a practical database programming language. In Proc. ACM ICFP 2011, pages 307–319, 2011.

[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.

[6] A. Ohori, K. Taura, K. Ueno. Extending ML with Massive Parallelism for Multicore Processors. Unpublished manuscript.