(POPL 2025 Tutorial Proposal)

MPL: Provably Efficient Parallel Programming

Sam Westrick

Courant Institute of Mathematical Sciences
New York University
New York, NY

shw8119@nyu.edu

Abstract

Parallel programming is infamously error-prone, not only due to correctness issues but also subtle performance problems which can easily negate the benefits of parallelism. **MaPL**e (**MPL** for short) is a parallel functional language that seeks to address these issues by providing strong guarantees on both safety and efficiency. A key feature of MPL is that it is *provably efficient* relative to the work-span cost model, preserving all logical parallelism in the source program while ensuring low overhead and high scalability in practice. In this tutorial, we will present MPL, overview the key ideas underpinning its design, and give a hands-on introduction to writing code with MPL and evaluating the performance of parallel code in practice.

1 Introduction and Motivation

Recent work on provable efficiency for parallel programming [11, 1, 9, 8, 2, 10] has led to the development of **MaPL**e[1], or **MPL** for short. MPL is a parallel functional language based on Standard ML, and extended with a parallel primitive, called `par`, which evaluates two thunks in parallel and returns their results as a tuple.

```
val par: (unit -> 'a) * (unit -> 'b) -> 'a * 'b
```

The primitive `par` alone is surprisingly expressive, as it can be used (often recursively) to implement any fork-join parallel algorithm. For example, a parallel for-loop can be expressed by splitting the loop range in half and recursively processing the two halves in parallel, as shown in Figure 1.

Work and span. The function `par` only expresses logical parallelism, and we rely on the language implementation to figure out how to map computations onto processors, allowing the program to be machine-independent and portable across different architectures. To reason about performance in a portable way (independent of a particular machine), we need an abstract cost model which can be instantiated for any particular machine architecture or configuration.

A particularly popular model which has found widespread use is the *work-span* model; this model is popular due to being incredibly simple yet remarkably effective in practice, allowing for accurate prediction

[1] Available open-source on GitHub at `https://github.com/MPLLang/mpl`

```
fun parfor(i: int, j: int, f: int -> unit) =
  if i >= j then
    ()
  else if j-i = 1 then
    f(i)
  else
    let val mid = i + (j-i) div 2
    in par(fn () => parfor(i, mid, f),
           fn () => parfor(mid, j, f));
       ()
    end
```

Figure 1: Simple implementation of a parallel for-loop in terms of just `par`.

of scalability across a wide range of multicore architectures. As suggested by the name, the model is based on two quantities: *work* and *span*. Informally, *work* of a computation is the total number of instructions that need to be executed and the *span* is the number of instructions on the critical path.[2] Analyzing the work and span of a program is straightforward according to the following equations.

$$\textit{Work} \qquad \mathcal{W}(\texttt{par(f,g)}) \triangleq \mathcal{W}(\texttt{f()}) + \mathcal{W}(\texttt{g()})$$

$$\textit{Span} \qquad \mathcal{S}(\texttt{par(f,g)}) \triangleq 1 + \max(\mathcal{S}(\texttt{f()}), \mathcal{S}(\texttt{g()}))$$

Well-known and well-studied scheduling techniques (such as work-stealing) can guarantee that the execution of a program e using P processors will complete in approximately $\mathcal{W}(e)/P + \mathcal{S}(e)$ time [3, 6]; this time bound ensures nearly-optimal performance on any number of processors.

Practical concerns. The $\mathcal{W}(e)/P + \mathcal{S}(e)$ time bound, which is central to this approach, makes a number of assumptions. In practice, these assumptions can be unrealistic:

1. The bound assumes that the execution cost of `par` is free. However, typical implementations of `par` require at least a modest number of instructions. If the programmer is not careful to control the "granularity" of parallelism, the cost of `par` will dominate the computation and result in significant slowdowns.

2. The bound assumes that the only synchronization between processors occurs at either the beginning or end of a `par`. However, multi-threaded language implementations typically require other synchronizations under the hood (especially in the implementation of automatic memory management systems and garbage collectors) which limit scalability.

Provable efficiency in MPL. MPL addresses the two issues above through a combination of (and close interaction between) two features: automatic parallelism management [10], and disentangled memory management and coscheduling [11, 1, 9, 8, 2]. The resulting system ensures that the cost of the garbage collector does not interfere with parallelism, and also that the run-time cost of `par` is nearly zero. As a result, MPL is able to offer strong performance guarantees in practice with respect to the work-span model.

[2] Alternatively, one can informally think of the span of a computation as the minimum possible execution time, assuming an idealized machine with infinite processors and no communication or synchronization cost.

Motivation for this tutorial. The papers listed above have recently appeared at conferences such as POPL, ICFP, and PLDI, and therefore the topics are likely to be of interest to POPL conference-goers and tutorial attendees. As part of the tutorial, we will provide an overview of the research in this area and introduce key concepts, which will help make these lines of research more accessible to the general community. Many POPL attendees may also wish to learn more about parallel programming, and we believe that this tutorial will serve as a good hands-on introduction to the topic, especially demonstrating that state-of-the-art language implementation technologies simplify the challenge of writing efficient, scalable, and correct parallel code.

2 Overview

Objectives. Attendees will become familiar with research in the area of provably efficient implementations and will learn about key concepts and common definitions in these lines of research. Attendees will also gain hands-on experience with writing parallel code and learn how to analyze and evaluate performance, both in theory and practice.

Topics to be covered:

- Introduction to provable efficiency and abstract cost models. (This topic goes back at least to the 90s, for example with Blelloch and Greiner's work on provably efficient implementations [4, 5, 7] and languages such as NESL.)
- Overview of recent work in this area, especially work relevant to MPL, including automatic parallelism management and disentangled memory management.
- Introduction to the MPL language.
- Introduction to parallel programming, especially common patterns (e.g. divide-and-conquer), higher-order parallel primitives (`map`, `filter`, `reduce`, `scan`, etc.), and common performance issues (e.g., granularity control).
- (Hands-on) Parallel programming examples and challenges, with empirical performance analysis.

Presentation approach. Slides and interactive live coding.

Target audience and prerequisite knowledge. This tutorial is meant to be accessible generally to the POPL community. For the hands-on component, attendees should be familiar with functional programming (e.g., Standard ML or OCaml or Haskell, etc.) and recursive programming techniques, and should be comfortable with typical command-line usage. No other specific requirements. A number of programming challenges will be offered at various levels of difficulty.

Has this tutorial been held previously? No; this would be the first offering.

References

[1] Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space efficient parallel functional programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2021.

[2] Jatin Arora, Sam Westrick, and Umut A. Acar. Efficient parallel functional programming with effects. *Proc. ACM Program. Lang.*, 7(PLDI):1558–1583, 2023.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129. ACM Press, 1998.

[4] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 226–237. ACM, 1995.

[5] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM, 1996.

[6] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[7] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Transactions on Programming Languages and Systems*, 21(2):240–285, March 1999.

[8] Sam Westrick. *Efficient and Scalable Parallel Functional Programming Through Disentanglement*. PhD thesis, Carnegie Mellon University, August 2022.

[9] Sam Westrick, Jatin Arora, and Umut A. Acar. Entanglement detection with near-zero cost. In *Proceedings of the 27th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2022, 2022.

[10] Sam Westrick, Matthew Fluet, Mike Rainey, and Umut A. Acar. Automatic parallelism management. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages (POPL)*, POPL '24, 2024.

[11] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2020.