Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 535–544

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

Replicated Synchronization for Imperative BSP Programs

Arvid Jakobsson[1,2], Frédéric Dabrowski[2], Wadoud Bousdira[2], Frédéric Loulergue[3], and Gaetan Hains[1]

[1] Huawei Technologies France Research Center
firstname.lastname@huawei.com
[2] Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France
firstname.lastname@univ-orleans.fr
[3] School of Informatics, Computing and Cyber Systems, Northern Arizona University, USA
frederic.loulergue@nau.edu

Abstract
The BSP model (Bulk Synchronous Parallel) simplifies the construction and evaluation of parallel algorithms, with its simplified synchronization structure and cost model. Nevertheless, imperative BSP programs can suffer from synchronization errors. Programs with *textually aligned barriers* are free from such errors, and this structure eases program comprehension. We propose a simplified formalization of barrier inference as data flow analysis, which verifies statically whether an imperative BSP program has *replicated synchronization*, which is a sufficient condition for textual barrier alignment.

Keywords: Parallel programming, bulk synchronous parallelism, static analysis, barrier inference

1 Introduction

Parallel architectures are ubiquitous, and the number of processing elements keeps increasing. There are existing architectures for embedded systems with hundreds of cores [4]. Therefore models and programming libraries for *scalable* parallelism are necessary. Bulk synchronous parallelism (BSP) is such a model [12]. BSP provides a high degree of abstraction like PRAM models and yet allows portable and predictable performance on any general purpose parallel architecture. BSPlib [6] is a C API proposal for BSP programming in direct mode.

Synchronization is a potential source of errors in imperative BSP programs. BSPlib programs interleave the code which handles local computation and code which handles synchronization. As a consequence, incorrect programs are easy to write but hard to debug.

In response to evolving architectures, embedded software developed at Huawei is becoming increasingly dependent on exploiting parallelism efficiently and safely. Engineers with little or no prior experience in writing parallel codes must be provided with languages and tools which alleviate the inherent complexity of parallel programming. We propose a subset of BSPlib that ensures safe synchronization and codifies Software Engineering best practices, as well as a sound static analysis that verifies whether any given BSPlib-program is in this subset.

Indeed, safe synchronization can be guaranteed by writing programs that have *textually aligned barriers*. In such programs, each barrier is the result of a synchronization request from the same source code location in all processes. It has already been noted that quality parallel code has strict synchronization patterns, and there are analyses to enforce them [2, 13], but our review of BSPlib programs in the wild suggests that the stricter convention of textually aligned barriers is sufficient to capture a large majority of correct programs. Additionally, such programs are conceptually simpler which help steer program design toward correctness, and ease other validation steps such as code review. Our static analysis verifies that a program has *replicated synchronization*: a statically inferable property which implies textually aligned barriers and safe synchronization as a result. Therefore, it reconstitutes the backbone of a BSP algorithm, which is its synchronization.

This static analysis poses higher requirements on the analyzed source code than existing analyses such as Barrier Inference [2]. For this reason, it is defined directly on the syntax of our language, assumes structured control flow and allows fewer synchronization patterns. This is intentional, since our purpose is to carve out a stricter subset of the language that complies with best practices and which nudges programmers toward correctness.

The main contribution of this paper is an adaptation of the Barrier Inference static analysis [2], formalized and proven, which verifies whether any given program has the replicated synchronization, and which thus is in the safe subset of BSPlib programs. The synchronization-backbone frames further static reasoning on the program, by giving knowledge of its degree of parallelism. Thus, this analysis is an initial building block in our envisioned set of formally justified static analyses for BSPlib programs as plug-ins for Frama-C [9], a framework for static and dynamic analysis of C programs.

The paper is organized as follows. First we present the BSP model (Section 2) and a small imperative BSP language, BSPlite (Section 3) with its operational semantics and describe the safe language subset with textually aligned barriers. Section 4 is devoted to the static analysis of this language. We discuss related work in Section 5 before concluding in Section 6.

2 The BSP Model

The BSP model offers an abstract model of parallel architectures, a model of parallel algorithms, and a cost (performance) model. A BSP computer is a distributed memory machine. It has a set of p processor-memory pairs, interconnected in such a way that point-to-point communications are possible. It has also a global synchronization unit. This model is abstract in the sense that any general purpose architecture can be seen as a BSP computer. For example, a cluster can be seen as a BSP computer with global synchronization provided by software.

A BSP program is a sequence of *super-steps*. Each super-step proceeds in three phases. In the local computation phase, each processor computes using only the data from local memory. In the communication phase, processors can request and send data from other processors. The synchronization phase ends a super-step. It is guaranteed that the data requested or sent during the communication phase has reached its destination at the end of the synchronization barrier. This constrained form of parallelism allows a realistic, yet simple, performance (or cost) model. We omit discussing it for the sake of conciseness.

BSPlib [6] provides a small set of primitives for direct mode bulk synchronous parallel programming. BSPlib follows the Single Program Multiple Data (SPMD) paradigm. It gives access to the process identifier, through a function `bsp_pid()`. It allows to write only one