Hosting OpenMP Programs on Java Virtual Machines

DOI:
10.1145/3357390.3361031

Hosting OpenMP Programs on Java Virtual Machines

Swapnil Gaikwad
University of Manchester
Manchester, UK
swapnil.gaikwad@manchester.ac.uk

Andy Nisbet
University of Manchester
Manchester, UK
andy.nisbet@manchester.ac.uk

Mikel Luján
University of Manchester
Manchester, UK
mikel.lujan@manchester.ac.uk

Abstract

To leverage existing virtual machine infrastructures is attractive for programming language implementors because competitive runtime performance may be achieved with a reduced effort. For example, the *Truffle* framework has enabled Ruby (TruffleRuby), and C (Sulong) *guest* language implementations to be hosted on a Java Virtual Machine (JVM). In this paper, we present *Sulong-OpenMP*, the first Truffle-based implementation to support parallel programs written in C/C++ and OpenMP. Our implementation adds OpenMP support to *Sulong* that executes LLVM Intermediate Representation (LLVM IR) for C/C++ programs on a JVM.

We outline the challenges faced in supporting OpenMP execution semantics, and the current limitations of Sulong-OpenMP. The geometric mean overhead of 1 thread Sulong-OpenMP compared to sequential Sulong execution was 2.6% for the NAS Parallel Benchmark suite, at peak runtime performance. Although this paper focuses on the correctness of our implementation concerning the OpenMP memory model, we also highlight the diminishing performance gap between the native execution with clang `-O2` and our *Sulong-OpenMP* as only 1.2x in the best case using 4 OpenMP threads.

CCS Concepts • **Software and its engineering → Interpreters**; **Virtual machines**; **Multithreading**; *Just-in-time compilers.*

Keywords Java Virtual Machine, OpenMP, Sulong, Truffle, GraalVM

ACM Reference Format:
Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. 2019. Hosting OpenMP Programs on Java Virtual Machines. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3357390.3361031

MPLR '19, October 21–22, 2019, Athens, Greece

1 Introduction

The Java Virtual Machine (JVM) is becoming a popular platform for hosting many programming languages beyond Java. Currently this includes, for example, Clojure, Scala, Kotlin, and using the Truffle framework C/C++, Fortran, R, Ruby, JavaScript and Python [4, 6, 11, 17]. The benefits of this approach are reduced language implementation effort with competitive runtime performance that directly leverages JVM infrastructure and its extended ecosystem of software support tools. Further, the interoperability features of Truffle [6–8] provide efficient execution support for multi-language *polyglot* applications where code in one language, can efficiently access foreign code and data in another.

A Truffle hosted language has to create an *Abstract Syntax Tree* (AST) that is initially interpreted. Profiling information is collected that helps to guide speculative optimizations during JIT compilation of *hot* frequently executed code. If any speculative assumptions for an optimization are found to be invalid, then the JVM deoptimizes the offending code back to its previous slower interpreter-based form. Truffle-hosted languages with dynamic types can benefit from speculative optimizations such as the assumed types of values. On the other hand, for statically typed languages, such as the Sulong (Truffle) implementation for C/C++ (see Section 2.1), Rigger *et al.* [16] demonstrated that additional bugs and programming errors related to undefined behaviour can be caught.

This paper explains how to extend Sulong to provid parallel execution support for C/C++ programs using OpenMP directives. Prior to this work, Sulong was limited to sequential execution of C/C++ programs. We present the design of our implementation of Sulong-OpenMP, and present results using the NAS Parallel Benchmarks (NPB) suite.

This paper describes how to reduce the execution time overheads of 1-thread Sulong-OpenMP in comparison to sequential Sulong execution, which is 2.6% geomean for NPB. The main contributions of the paper are:

- We demonstrate our approach to execute OpenMP programs on a JVM that is accomplished primarily by extending LLVM IR bitcode interpreter of Sulong.
- We describe how to map the OpenMP memory model onto the Java memory model.
- We present an evaluation of NPB suite on our implementation. We discuss the main optimizations used, the sources of overhead, and the performance impact of each optimization.

2 Background

This section provides an overview of the main components of Sulong[15], that implements an LLVM IR guest language on top of Truffle and GraalVM. In addition, we describe the main relevant aspects of OpenMP programming and functionality support necessary for our implementation.

2.1 Truffle, GraalVM & Sulong

Truffle is an implementation framework written in Java that enables a *guest* language to be implemented as an Abstract Syntax Tree (AST) interpreter [6, 21]. A Truffle-based interpreter converts a guest language program input into an AST that is executed using a Java Virtual Machine (JVM). As with normal Java, the AST is initially interpreted and its frequently executed parts are JIT compiled on demand during execution. The Truffle generated AST is specialized using observed runtime program behaviour that can then aid optimizing JIT compilers such as Graal to generate better code with speculative optimizations. For example, this can include aspects such as the observed datatype of program variables, the probability of taking a branch, and the identities of dynamically resolved function calls, and so on. Correct execution semantics are preserved by performing deoptimization at runtime back to the original Truffle AST form when speculative assumptions are found to be false.

Truffle-ASTs can be executed using any JVM, however, the *Graal* JIT compiler applies a range of optimizations including partial evaluation [20] and partial escape analysis [18], that have been found to be especially effective for improving the performance of dynamically typed languages such as Python, Ruby and JavaScript. GraalVM (a JVM using the GraalJIT compiler) is particularly useful in this context as it enables specialized custom compilation of Truffle-ASTs and specific features of Truffle hosted language features via the JVM Compiler Interface (JVMCI).

Sulong is a Truffle-based interpreter that executes the binary bitcode (`.bc`) form of the LLVM Intermediate Representation (LLVM IR) on JVMs. Therefore, by supporting the necessary LLVM IR, Sulong can host guest languages that generate LLVM IR without writing an actual parser or an interpreter for each of those languages. For example, LLVM IR can be generated from C/C++ programs using the Clang frontend, with all the benefits of the comprehensive static optimizations offered by the LLVM infrastructure.

2.2 OpenMP

OpenMP is a popular directive-based parallel programming approach for shared memory systems that is available in C, C++ and FORTRAN languages. It provides a variety of parallel programming models that can exploit various forms of work-sharing such as where iterations of a loop are collaboratively executed, and also where a task is the basic unit of parallel execution. In our initial implementation for Sulong-OpenMP, we only exploit and address the fork-join computational model of OpenMP and loop-based parallelism with a static schedule of iterations to threads.

```
1 void main() {
2  // Code before OpenMP block ...
3  #pragma omp parallel
4  {
5    printf("Thread %d\n", omp_get_thread_num());
6  }
7   // Code after OpenMP block ..
8 }
```

Listing 1. A simple OpenMP C code where each thread prints a message containing its thread id.

OpenMP parallel for-loops use a fork-join work-sharing model. In this model, threads are forked at the beginning of a program block marked with an OpenMP directive as shown in Line 2 of Listing 1, and an implicit join is automatically inserted at the end of the block. A program may contain one or more such OpenMP parallel blocks, placed sequentially or nested within each other, to achieve parallelism. When an OpenMP block contains a for-loop, it is executed by each OpenMP thread. We can place an `#pragma omp for` immediately before a for-loop to split and share the iterations of the loop amongst the available OpenMP threads. We elaborate this discussion in Section 5, while referring to the *OMP Split* optimization.

3 Implementation

This section outlines the main aspects of the Sulong-OpenMP implementation, and relevant operational aspects concerning generation of LLVM IR for OpenMP programs. Finally, we describe the implementation of key OpenMP features, and the challenges faced whilst mapping semantics of the OpenMP memory model onto the Java memory model.

```
define i32 @main() {
  ; LLVM IR for code before omp for loop
  call .. @__kmpc_fork_call(...,
  void (i32*, i32*, ...)* @.omp_outlined.),...)
 ; LLVM IR for code after omp for loop
}
define void @.omp_outlined.(..) {
 ; LLVM IR for actual OpenMP block
 call @printf(..)
}
```

Listing 2. Shortened snippet of LLVM IR generated with Clang from Listing 1.

3.1 LLVM IR for OpenMP

Clang is the LLVM compiler frontend for C, C++, Objective-C and Objective-C++ programs. It can generate LLVM IR for both sequential, and parallel OpenMP programs. The latest stable version 8.0.1 [12]. Note the latest OpenMP specification is 5.0 as of writing this paper whereas Clang has supported OpenMP 3.1 since version 3.8.0 [13].

Listing 2 shows a shortened snippet of the LLVM IR generated from Listing 1. Lines 1-7 of Listing 2 represent the equivalent LLVM IR of the `main()` function. Note that the OpenMP block in the main function is replaced by a call to the OpenMP runtime denoted using `@__kmpc_fork_call`, and the function pointer argument to the runtime call is a pointer to the function containing LLVM IR for an OpenMP block that includes a call to a `printf` function. We refer to this as an *outlining* of a function, and it is one of the ways in which the generated LLVM IR for an OpenMP program is different from a non-OpenMP program.

3.2 Sulong-OpenMP Implementation Approach

The focus of our initial implementation is the subset of OpenMP features necessary to execute the selected NPB suite benchmarks. These programs are complex and expose an important set of challenges for bringing parallel execution capabilities for the first time to Sulong. Nonetheless, we continue to extend and enhance the OpenMP support to incorporate further directives and features.

The OpenMP implementation for Clang is directly based on `Pthreads`. We could not use directly this approach, as `Pthreads` support for Sulong, although in progress, was not available at the time of implementation and writing this paper. One implementation option was to provide our own `Pthreads` support in Sulong, and to use the OpenMP runtime libraries part of LLVM. Although this approach promises to provide full OpenMP 3.1 feature completeness, the estimated effort required to run even a simple OpenMP program, such as in Listing 1, was significantly greater than the chosen alternative hybrid approach (mainly based on function morphing) that we describe below. Further, the `Pthread` approach could potentially remove or hinder opportunities to directly apply custom compilation and optimizations related to mapping OpenMP language features and memory model semantics onto Truffle and Java.

Function Morphing: Sulong maintains a registry of function definitions that map LLVM IR function names onto their corresponding Truffle-based AST representations generated by the Sulong parser. The map is searched at runtime to retrieve a corresponding Truffle-based implementation of an LLVM IR function that is then invoked. If an LLVM IR function is not found in the map, then an exception is thrown, this was initially the case for unimplemented OpenMP runtime functions. With the goal of reducing our implementation efforts to the essential OpenMP runtime subset, we performed *function morphing* where we implemented the necessary subset of OpenMP functionality with *morphed* Java implementations and directly added appropriate mappings to the registry of function definitions. This approach reduces implementation effort by avoiding extensive support for `pthreads` whilst enabling the direct exploitation of custom execution using Sulong for specific OpenMP runtime function features.

A **hybrid approach** is necessary because it becomes cumbersome to generate morphed OpenMP functions when read-/write access to local variables is required. For example, in an OpenMP for-loop the local range of iterations to be executed by a thread is typically calculated using the global iteration space and the logical id of a thread. The approach becomes cumbersome because the implementation would need to track and update any changes concerning the internal procedures of Sulong for accessing any raw memory allocated using the Unsafe API. Further, the OpenMP runtime function to perform iteration space splitting can easily exceed a few 100 lines of Java code, leading to less maintainable and potentially error-prone code.

To avoid these problems, we use a *hybrid approach* where a subset of OpenMP runtime functions can be directly implemented in C. Such C functions are converted into LLVM IR using Clang, and then added as an external library. The library is parsed by Sulong and any defined functions are added to the registry map. Therefore, hybrid functions can be invoked during execution in a Sulong compatible way. The need for this approach was minimized, and currently, we only use a single function implemented in C that returns the local range of iterations to be executed by an OpenMP thread.

This hybrid approach is very useful during prototyping, as it enables the production of simplistic and potentially incomplete implementations of OpenMP runtime functions used as placeholders for further implementation and experimentation. The approach enables the development and implementation work to provide the earliest possible successful execution of benchmarks without any exceptions arising due to unimplemented runtime functions.

3.3 Sulong-OpenMP: Challenges

The main implementation challenges for Sulong-OpenMP are discussed, they include the for-loop, master, single, critical, barrier and flush directives that are required to run the benchmark suite [9] using a fork-join model.

Fork-Join Model — Clang outlines (or wraps) the IR for a parallel execution block into a separate function, that is invoked by a runtime function (see Listing 2). Function morphing (see Section 3.2) is used to provide our own implementation for the runtime `@__kmpc_fork_call` function. Our *morphed* implementation spawns the required number of Java threads as specified by the `OMP_NUM_THREADS` environment variable. The entry point `run` method of the spawned Java threads calls the outlined function to match the behaviour of OpenMP-based execution. Our morphed function synchronizes all spawned Java threads once they finish execution of the outlined function representing the OpenMP block. After synchronization, only the master thread continues to execute outside of the OpenMP region.

OpenMP For-Loop — Each thread executes a subset of the iteration space. The local iteration space bounds of a thread are obtained from an additional call to the runtime function; `@__kmpc_for_static_init_4`. We provide an implementation of this function in C and attach its LLVM IR as an external library to Sulong using the *hybrid approach*. Executing the remaining IR including the body of the for-loop on Sulong is the same as its sequential execution. Sulong-OpenMP currently supports only the static schedule of OpenMP for-loop iterations to threads.

OpenMP Shared & Private Clauses — In the case of shared variables, only a single copy of the variable is used by all the threads and a separate copy of each private variable is created for each thread in the latter case. From the perspective of implementation, these features did not pose a high degree of difficulty because of the way Clang generates LLVM IR. Shared variables are implemented as pass-by-reference arguments to the outlined OpenMP function. Private variables are implemented by allocating thread-local copies on the private stack of each thread.

Barrier Implementation — We provide a naïve implementation of an OpenMP barrier based on Java's wait-notify mechanism. A thread can invoke a `wait()` method on an object and suspend itself until any other thread invokes `interrupt()`, `notify()` or the `notifyAll()` method on the same object. We use this feature to implement an OpenMP barrier where all threads except the last thread to enter a barrier invoke a `wait()` method on a fixed object. The last thread to reach the barrier wakes up all waiting threads.

Master Construct — A block enclosed by an OpenMP master construct is only executed by the master thread with logical id 0. Its LLVM IR is enclosed by calls to the `@__kmpc_master()` and `@__kmpc_master_end()` runtime functions. All OpenMP threads execute the `@__kmpc_master()` function that returns the value 1 only for the thread with logical id 0. Threads having a return value other than 1 are forced to skip execution of the master block. We implement this simple functionality using a hashmap to maintain the mapping between Java thread ids and their corresponding OpenMP logical thread ids.

Critical Construct — OpenMP provides a mechanism for mutual exclusion of threads where only a single thread can be executing code inside a critical section code block, and all other threads must wait until the block is exited. A critical code block is enclosed by `@__kmpc_critical()` and the `@__kmpc_critical_end()` runtime function calls. We exploit the Java `Semaphore` class to implement the two aforementioned runtime functions. Here, the thread entering a critical section acquires a semaphore and releases it while exiting the critical section to ensure an exclusive access.

Flush Construct — OpenMP uses a relaxed-consistency memory model that enables threads to have a temporary view of shared variables that may not be consistent with the view of the memory of other threads at all times [1]. The temporary view of variables of a given thread is made consistent with values written to memory after executing the flush operation associated with the OpenMP `flush` construct. This is analogous to a *memory fence* operation. Specifically, the flush operation is performed on a set of variables called the *flush-set*. If no variables are specified then the flush applies to all shared variables in the temporary view of a thread. The flush operation restricts reordering of memory operations that an implementation might undertake. Implementations must not reorder the code for a memory operation for a given variable, or the code for a flush operation for the variable, with respect to a flush operation that refers to the same variable.

Java uses a different memory consistency model based on *happens-before semantics* [14], where if a write operation by a thread is performed on a volatile variable then all previous writes to any other variables performed by that thread become visible to all other threads after those perform a read/write operation on the same volatile variable. We use this behaviour to implement the OpenMP flush operation in Java. In the generated LLVM IR, the `flush` pragma is replaced by a call to the `@__kmpc_flush` runtime function call. We implement a morphed function to increment the value of a predefined volatile variable to perform a read and a write operation on that variable. Therefore, when all the threads finish executing the OpenMP flush operation, all the writes carried out before the OpenMP flush operation are now visible to all the other Java threads.

We check the correctness of this approach using the test suite of Clang for the flush construct. Currently, this approach limits our ability to perform the flush operation on a subset of the shared variables. Thus, every flush operation, irrespective of whether it is on a subset of the shared variables or not, is executed on all the shared variables.

4 Evaluation Methodology

Firstly, we evaluate 1-thread Sulong-OpenMP execution overhead compared to sequential Sulong. Secondly, we evaluate the overhead of Sulong in comparison to native execution, and finally, we evalute the overhead of multithreaded Sulong-OpenMP against native-OpenMP along with a brief discussion on the scaling behavior for the selected benchmarks on Sulong-OpenMP and native-OpenMP.

4.1 Benchmarks

The NAS Parallel Benchmarks (NPB) suite is a widely used parallel programming benchmark suite from scientific computing that provides implementations for multiple programming languages. This paper uses the C OpenMP variant of the NPB suite 3.0 [9] and the Java NPB implementation. In this way, we can investigate the relative performance for each benchmark across pure Java, Sulong, Sulong-OpenMP, native and native-OpenMP execution mechanisms.

The NPB suite has 5 kernels and 3 pseudo-applications derived from *computational fluid dynamics* (CFD) applications. The kernels are a few hundred lines of code, while the pseudo-applications are larger; typically a few thousand lines of code. For each benchmark, there are 5 different choices of classes categorised based on the different problem sizes are available. We use the class 'W' for the kernels and class 'S' for the pseudo-applications. These are the smallest classes, and the choice is made based on the execution time for a single iteration exceeding a few minutes. Longer executions would make the generation of results for warmed-up execution state (see Section 4) unnecessarily lengthy.

4.2 Experimental Setup

A JVM-based execution of an application starts with a slower interpreted mode, and then the frequently executed parts of the application are JIT compiled whenever the JVM decides they are *hot*. This process continues until the application reaches its *peak performance* when typically all the computationally intensive parts of an application are JIT compiled. Such state is referred to as the *warmed-up* execution state where performance improvement plateaus.

We use the methodology previously presented in [15] for performance evaluation. We observe that the kernel benchmarks warmed-up in less than 10 iterations while the pseudo-applications took a maximum of 25 iterations. Consequently, we use a harness to execute a benchmark 100 times in a loop, without exiting the JVM process. Then we measured the execution time for each of the last 50 iterations and calculated a geometric mean of those values to represent the execution time of the benchmark. The variation in execution time over the last 50 iterations for each benchmark can be seen as a violin plot in Figure 1. The correct execution of each benchmark for native, Sulong, Java, native-OpenMP and Sulong-OpenMP is double checked using the test available in the NPB suite.

The experiments run on a system with 4 physical (8 hyper-threaded) cores Intel Core i7-6700 with 16GB of memory running Ubuntu 18.04 (4.15.0-48-generic). We disable the processor frequency scaling and set it to its maximum of 3.4 GHz using the *performance* governor.

To generate scalability results, we use a larger system with 16 physical cores (2 socket NUMA) Intel Xeon E5-2690 with 378GB of memory running Ubuntu 16.04 (4.15.0-36-generic). Hyper-threading is disabled, and we set the processor frequency at 2.4 GHz using the *performance* governor.

On both systems, we use Sulong commit *b0ab114* from GraalVM release candidate 1.0.0-rc6 that supports LLVM 6.0. The LLVM IR for the Sulong-based execution is generated using Clang with -O2 optimization flag.

Table 1. Comparison of execution overhead for 1-thread Sulong-OpenMP before and after optimizations (shown in Figure 3) normalized to the sequential Sulong execution time. Values less than 1 represent faster than sequential execution.

Benchmark	Before Opt.	After Opt.	1 Thread Specialization
IS	1.19	1.05	1.00
EP	1.08	1.00	0.99
MG	1.92	1.22	1.01
FT	1.05	1.03	1.02
CG	1.53	1.12	1.03
LU	**0.03**	**0.02**	**0.02**
BT	8.81	1.11	1.01
SP	32.91	1.30	1.13
GeoMean	**2.73**	**1.11**	**1.03**

5 Peak Performance Evaluation

We discuss the performance of OpenMP support in Sulong considering the following aspects:
(1) Overhead incurred by the 1-thread Sulong-OpenMP with various optimizations compared to Sulong-sequential.
(2) Relative performance with respect to the sequential native execution for 1-thread Sulong-OpenMP, 1-thread native-OpenMP and 1-thread Java.
(3) The performance gap between Sulong-OpenMP and native-OpenMP execution with 4 threads.
(4) Scaling results up to 16 processors for Sulong-OpenMP compared to native-OpenMP on a 2-socket Intel machine.

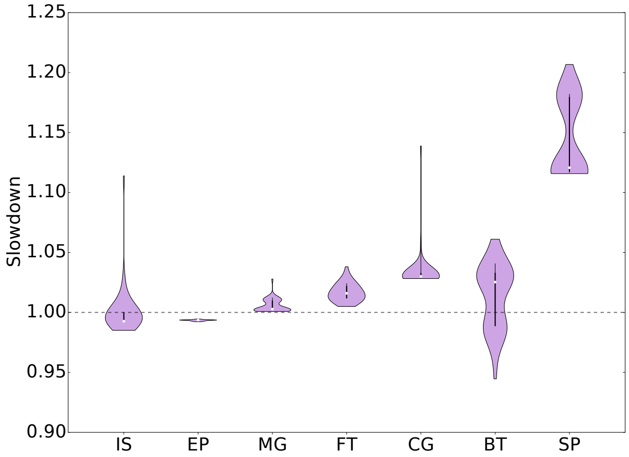

Figure 1. Violin plot of the warmed-up execution time distribution for 50 iterations of each benchmark. Overhead of Sulong-OpenMP 1-thread normalized to Sulong sequential. On the Y-axis lower is better, and values less than 1 indicate Sulong-OpenMP is faster than Sulong sequential.

5.1 Single Thread Results

The performance of 1-thread Sulong-OpenMP before and after optimizations in comparison to Sulong-sequential is shown in Table 1. To measure the performance without OpenMP support, we used the `-fopenmp` compilation flag. Although our objective is to reduce the overhead of executing OpenMP programs to zero, it is important to note here that we are comparing the sequential version (without additional OpenMP runtime function calls) to the OpenMP version with 1 thread (see Section 3.1). Figure 2 highlights the overhead of native OpenMP 1-thread against native-sequential execution using the blue bars.

Table 1 shows the aggregated impact of all the optimizations while the detailed impact of each optimization is shown in Figure 3. While comparing the overhead for executing Sulong-OpenMP to Sulong. It is important to note that the additional OpenMP runtime calls, as described in Section 3.1, effectively result in a geometric mean slowdown of 2.6% for the NPB suite benchmarks (excluding LU). This overhead is in-line with the 2.4% incurred for the executions of 1-thread native compared to native-sequential. As described in Section 4, this paper primarily focuses on correct execution semantics for Sulong-OpenMP with minimal overhead for execution with 1 OpenMP thread. In order to achieve this goal, we apply a special optimization shown in Table 1, henceforth referred to as '1 Thread Specialization'. This optimization overcomes the current limitation of our implementation for the multi-threaded OpenMP execution as described in Section 6.

In Table 1, the LU benchmark (unlike all others) shows a significant speedup compared to the sequential execution on Sulong. The main reason for this behaviour is that the `erhs` function from the benchmark cannot be JIT compiled whilst executing sequentially because its compilation unit size exceeds the maximum threshold for compilation by Graal. However, the same function has multiple OpenMP parallel blocks that are outlined into separate functions. These functions are small enough to be JIT compiled. Therefore, the performance improvement of ~48x for the OpenMP version is primarily due to JIT compilation. However, in the case of BT and SP, the sequential version was JIT compiled, but the parallel version remained as interpreted. This is a current limitation of our implementation (Section 6), that fails to compile the top-level outlined OpenMP function.

In the case of BT and SP benchmarks, this situation is exacerbated, because the functions called from the OpenMP block are inlined into the top-level OpenMP function and executed in an interpreted mode that results in a slowdown of ~9x and ~33x, respectively. This limitation impacted all the NPB suite benchmarks. The extent of the impact is proportional to the percentage of the overall computations performed in the top-level outlined function.

In the case of EP, the proportion of interpreted computation is negligible and thus, we do not see any overhead for execution with Sulong-OpenMP. The impact of such interpreted execution is not observed for LU, because it executes in interpreted mode when running on Sulong sequentially.

Figure 2 shows the 1-thread execution time overheads of Sulong-OpenMP, Java, native-OpenMP, and (sequential) Sulong normalized to the native-sequential C. The bars representing the geometric mean exclude the results for the LU benchmark because Sulong-sequential executes a large portion of its code in a slower interpreted mode. The higher overheads for executing pseudo-applications on Java should be accounted more to the implementation choices instead of the limitations of the underlying JVM platform. However, faster execution of a Java version of a benchmark can be seen as an opportunity, e.g., for MG, IS and FT benchmarks that could be exploited by Sulong-OpenMP. The performance of Sulong-OpenMP 1-thread has a better geomean slowdown in performance of 2.19 in comparison to 4.20 for Java 1-thread in comparison to native execution.

5.2 Sulong-OpenMP Optimizations

Using the performance analysis technique introduced in [5] and analysing the NPB suite, we can identify a number of performance bottlenecks. We briefly summarize the various optimizations applied to reduce the overhead of 1-thread Sulong-OpenMP compared to the Sulong-sequential execution.

(1) Calls to Intrinsics: Sulong provides implementations for various C math library functions such as `log`, `sqrt` using the Math class of Java. Previously, calls to such functions inserted a new stack frame with a synchronized method. This optimization avoids creating a new stack frame for such intrinsic functions and thus, the synchronization overhead.

(2) OpenMP Split: Targets optimization of large OpenMP parallel blocks that could not be JIT compiled. Such blocks exceed the threshold for the size of the method that Graal can compile. As a temporary workaround, the blocks are manually split into multiple blocks, typically two or three. These are then outlined into smaller functions which can then be JIT compiled by Graal.

(3) Thread Pool: This optimization highlights the benefits of implementing an OpenMP thread pool. Previously, OpenMP threads were created at the beginning of every OpenMP block and merged/destroyed at the end of the block. Implementation of the thread pool avoided the expensive operation of thread creation and destruction.

(4) Thread-private Stack Implementation: The addition of thread-private stack support is expected to help performance for benchmarks performing multiple local variable allocations, mainly the pseudo-applications. However, only the IS benchmark from NPB suite benefited from this optimization and the slowdown reduced from approximately 2.5x to 2.0x.

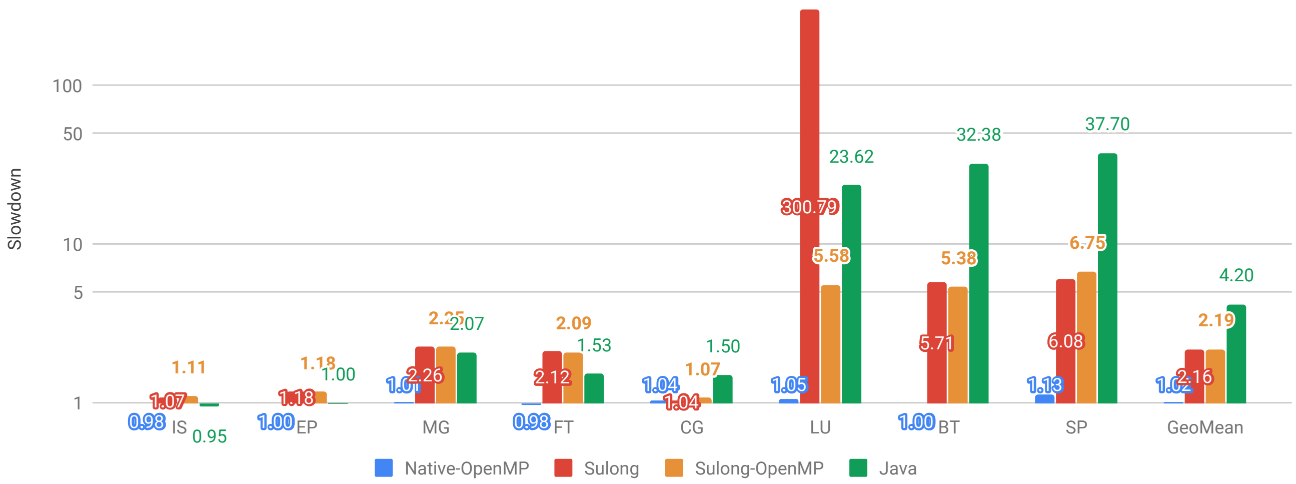

Figure 2. Comparison of overhead executing native-OpenMP, Sulong, Sulong-OpenMP (1-thread) and Java (1-thread). The execution times are normalized to execution time of the native-sequential. The Y-axis shows slowdown, on a logarithmic scale, compared to the native execution. Thus, Y-axis 1 means the execution time of Sulong with OpenMP is on par with the native execution; lower is better.

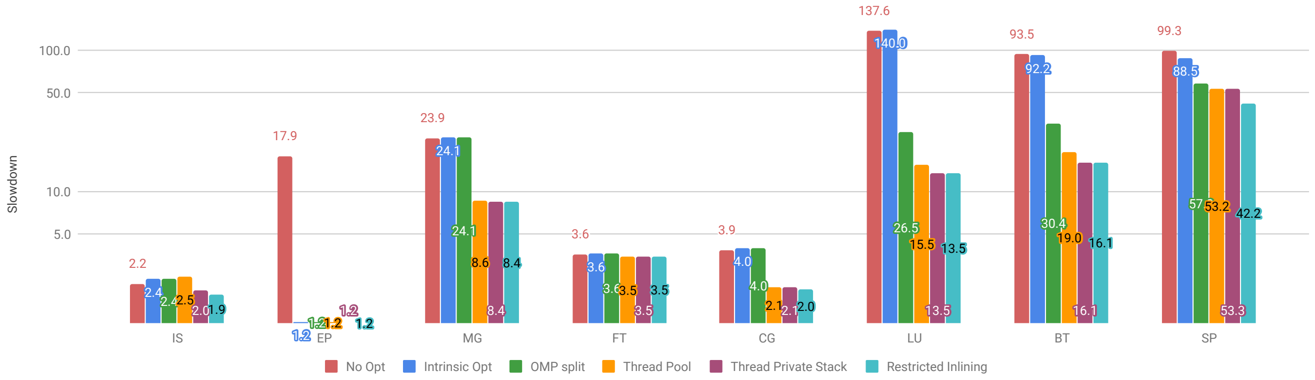

Figure 3. Effect of each optimization on the execution times of Sulong-OpenMP (4-threads) vs. execution time of native-OpenMP (4-threads). The Y-axis shows slowdown, on a logarithmic scale, compared to the native execution. Thus, Y-axis 1 means the execution time of Sulong with OpenMP is on par with the native execution; lower is better.

(5) Restricted Inlining: This optimization is only applied to the SP benchmark to minimize the impact of the existing limitation where the top-level outlined OpenMP function cannot be JIT compiled. We force Clang not to inline functions called from the outlined OpenMP functions in a parallel region using `__attribute__((noinline))` attribute. This significantly reduces the portion of computation executed in the slower interpreted mode.

5.3 Scalability Results

Figure 4 shows the scalability results for Sulong-OpenMP and native-OpenMP with EP, FT and MG benchmarks as these showed the best, intermediate and the worst scaling results for Sulong-OpenMP. We do not discuss the scaling results for the pseudo-applications because the selected problem size is too small to provide sufficient work for the number of cores available.

For the EP benchmark, Sulong-OpenMP shows scaling comparable to the native-OpenMP upto 16 cores while for the FT benchmark, Sulong-OpenMP matches the scaling of native-OpenMP until 8 cores. As seen previously in Figure 3, the MG benchmark still has a high overhead while executing on Sulong-OpenMP. This is reflected by the slowdown beyond 2 threads. Note, we have not made any attempt to optimize Sulong-OpenMP for execution on NUMA systems, nor do we attempt to pin threads to processing cores in any of the experiments.

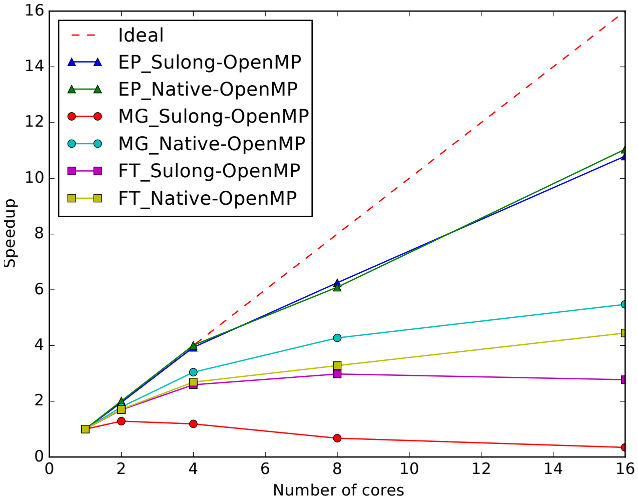

Figure 4. Speedup graph with scaling results for the EP, FT and MG kernels part of the NPB suite.

6 Limitations

The main limitations of the current Sulong-OpenMP implementation are discussed next.

A) JIT compilation of entire OpenMP Block: Section 3.1 discusses how OpenMP blocks are outlined into separate functions when the corresponding LLVM IR is generated. The threads in the OpenMP thread pool wait in a loop to execute the assigned ASTs and synchronize after executing them. There are two entities required to execute the AST of a guest language function: (i) the Truffle-based AST representation and (ii) a `VirtualFrame` object associated with a function. Passing the `VirtualFrame` as a parameter limits what outlined functions can be compiled by Graal, specifically the top-level outlined OpenMP function.

This is currently the biggest contribution to the execution time overhead for Sulong-OpenMP. Current efforts are directed towards addressing this issue. Further, we are also considering an approach using `MaterializedFrame` instead of the `VirtualFrame` for the outlined OpenMP functions.

B) Feature Completeness: As discussed in Section 3.2, we implement the subset of OpenMP features that are necessary to execute the NPB suite on Sulong-OpenMP. However, there are more OpenMP features yet to be supported. For example, various schedules to execute the OpenMP for-loop, nested parallel regions and OpenMP task directives. We will continue adding support for the most commonly used OpenMP features.

7 Related Work

There are implementations of OpenMP available from different software vendors, such as Intel, PGI, GCC, Clang LLVM and IBM. These have various degrees of completeness with respect to the latest OpenMP specification (version 5.0). All of them take an OpenMP program written in C/C++/Fortran as input and generate an executable binary by compiling Ahead-Of-Time (AOT). This is the de facto mode for executing OpenMP applications. In comparison to the JIT compiled approach presented in this paper, AOT compiled approach has a faster startup time.

For Java programs, pre-processors such as JOMP, omp4j and JaMP can convert a Java program consisting of OpenMP directives as comments in the source code to the equivalent parallel Java program using source-to-source translation [2, 3, 10]. JOMP and omp4j provide a pre-processor and a runtime library to support the execution of the generated Java source while JaMP is implemented in the research compiler Jackal [19]. JOMP and omp4j support a relatively smaller subset of the OpenMP features, while the JaMP supports all OpenMP 2.0 features and some of the OpenMP 3.0 features, such as tasks. Additionally, JaMP also supports execution of OpenMP parallel loops on CUDA-enabled graphics cards. On the other hand, our approach is for C programs converted to LLVM IR to execute on JVM. We provide the implementation for the OpenMP runtime functions emitted by Clang in LLVM IR instead of emitting code for OpenMP directives and implementing the supporting runtime functions. Sulong-OpenMP does not currently support OpenMP tasks (OpenMP 3.0), nor can it offload OpenMP parallel loops for execution by accelerators (OpenMP 4.0 & 5.0).

8 Conclusions & Future Work

We have described our implementation of OpenMP support for the Sulong framework and the main challenges faced. Sulong-OpenMP demonstrates that correct execution of OpenMP parallel programs can be achieved using the Java memory model. This paper has focused on reducing the execution time overheads of 1-thread Sulong-OpenMP in comparison to sequential Sulong execution, which is 2.6% geomean for NPB. This is very close to the 2.4% geomean overhead of native execution concerning 1-thread OpenMP to native-sequential. The observed overhead is expected in both cases as the LLVM IR generated for an OpenMP equivalent of a program (by compiling with `-fopenmp` flag) contains additional OpenMP runtime calls that are absent when the IR is generated without enabling OpenMP support.

The optimizations described in Section 5 have considerably improved the performance of multi-threaded execution using Sulong-OpenMP. Nonetheless, substantial performance and functionality gaps remain for future Sulong-OpenMP work.

Acknowledgments

We acknowledge the support of the EU Horizon 2020 ACTiCLOUD 732366 project. Swapnil Gaikwad is supported by a Ph.D. studentship funded by Oracle Labs. Mikel Luján is supported by an Arm/RAEng Research Chair award and is a Royal Society Wolfson Fellow.

References

[1] OpenMP Architecture Review Board (OpenMP ARB). 2019. OpenMP Application Programming Interface. Retrieved 30th Mar 2019 from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[2] Petr Bělohlv̀ek and Antonín Steinhauser. 2019. omp4j Project website. Retrieved 14th Jul 2019 from http://www.omp4j.org

[3] J. M. Bull and M. E. Kambites. 2000. JOMP—an OpenMP-like Interface for Java. In *Proceedings of the ACM 2000 Conference on Java Grande (JAVA '00)*. 44–53. https://doi.org/10.1145/337449.337466

[4] Oracle Corporation. 2016. TruffleRuby. Retrieved Jun 16th 2019 from https://github.com/oracle/truffleruby

[5] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. 2018. Performance Analysis for Languages Hosted on the Truffle Framework. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. Article 5, 12 pages. https://doi.org/10.1145/3237009.3237019

[6] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, Mikel Luján, and Hanspeter Mössenböck. 2018. Cross-Language Interoperability in a Multi-Language Runtime. *ACM Trans. Program. Lang. Syst.* 40, 2, Article 8 (May 2018), 43 pages. https://doi.org/10.1145/3201898

[7] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS)*. 78–90. https://doi.org/10.1145/2816707.2816714

[8] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. 1–13. https://doi.org/10.1145/2724525.2728790

[9] H. Jin, M. Frumkin, and J. Yan. 1999. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Retrieved 16th Jun 2019 from https://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf

[10] Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen. 2007. JaMP: An Implementation of OpenMP for a Java DSM. *Concurrency and Computation Practice and Experience* 19, 18 (Dec. 2007), 2333–2352. https://doi.org/10.1002/cpe.1178

[11] Joanna Kolodziej and Bruce R. Childers (Eds.). 2014. *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*. ACM. https://doi.org/10.1145/2647508

[12] LLVM. 2019. LLVM Download Page. Retrieved 29th Jun 2019 from http://releases.llvm.org

[13] LLVM. 2019. OpenMP: Support for the OpenMP language. Retrieved 29th Jun 2019 from https://openmp.llvm.org/

[14] Willam Pugh. 2019. JSR 133, Java Memory Model and Thread Specification. Retrieved 30th Mar 2019 from http://www.cs.umd.edu/~pugh/java/memoryModel/CommunityReview.pdf

[15] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL)*. 6–15. https://doi.org/10.1145/2998415.2998416

[16] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. 377–391. https://doi.org/10.1145/3173162.3173174

[17] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R Language Execution via Aggressive Speculation. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS)*. 84–95. https://doi.org/10.1145/2989225.2989236

[18] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. 165–174. https://doi.org/10.1145/2581122.2544157

[19] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. 2001. Runtime Optimizations for a Java DSM Implementation. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (JGI '01)*. 153–162. https://doi.org/10.1145/376656.376842

[20] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 662–676. https://doi.org/10.1145/3062341.3062381

[21] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. 187–204. https://doi.org/10.1145/2509578.2509581