Performance of an OO Compute Kernel on the JVM Revisiting Java as a Language for Scientific Computing Applications (Extended Version)[*]

Malin Källén Tobias Wrigstad

November 5, 2019

Abstract

The study of Java as a programming language for scientific computing is warranted by simpler, more extensible and more easily maintainable code. Previous work on refactoring a C++ scientific computing code base to follow best practises of object-oriented software development revealed a coupling of such practises and considerable slowdowns due to indirections introduced by abstractions. In this paper, we explore how Java's JIT compiler handle such abstraction-induced indirection using a typical scientific computing compute kernel extracted from a linear solver written in C++. We find that the computation times for large workloads on one machine can be on-pair for C++ and Java. However, for distributed computations, a better parallelisation strategy needs to be found for non-blocking communication. We also report on the impact on performance for common "gripes": garbage collection, array bounds checking, and dynamic binding.

1 Introduction

Historically, scientific computing applications are predominantly written outside of managed languages, typically in FORTRAN or C++. Despite a recent uptick in use of Python (and R, and Julia), most complicated numerical procedures are performed in C and C++ code wrapped by dynamic front-ends that perform less performance-critical "glue code" tasks. Scientific computing applications are often at the fore-front of performance requirements, and often follow atypical development patterns (wrt. most software development): oftentimes, developers are happy to optimise an application for months just to have it run once. A large fraction of scientific computing software is developed by mathematicians, physicists, etc., with no formal training in software engineering.

Managed languages deliver many computing benefits such as portability and hardware abstraction. Furthermore, managed languages enable dynamic programming languages, like JavaScript, Python and Java, which do not compile directly to machine code but to higher-level intermediate representations, which are effectively translated into machine code (by interpretation or JIT'ing). This leaves many decisions until run-time for achieving efficient execution, but as a mitigating factor may rely on more precise information about the program to guide these choices.

This paper explores the application of Java in scientific computing. The motivation stems from a desire to achieve higher productivity and lower defect density Phipps (1999). The springboard to achieve these goals is higher levels of abstraction and software patterns, building on years of accumulated best practices of mainstream software developers. But, higher levels of abstractions are often at odds with optimisation—as abstractions tend to complicate efficient code generation by replacing concrete choices with a variety of possibilities. However, in many cases, the variety is only apparent: in practise, we tend to pick the same concrete choice consistently in almost all places; untyped Python code is inherently monomorphic Åkerblom et al. (2014); Åkerblom and Wrigstad (2015), polymorphic inline caches have a $>$90% hit frequency Hölzle et al. (1991), etc.

In this paper, we study a typical scientific computing application, or more precisely its compute kernel, where almost all its execution time is spent, written in C++. Refactoring this code base using typical OOP principles like SOLID Martin (2003), lead to a quantifiable increase in code quality/decrease in complexity, but at the cost of a 3–8X performance drop Källén et al. (2014). This was predominantly due to the introduction of abstract classes and virtual calls, which facilitate future extension. Ported to Java, the refactored program keeps the improvements of the code quality, but performance improves as the JIT compiler is able to perform critical inlining of the hottest code regions. However, porting C++ programs to Java have several other performance implications, making it hard for Java programs to match the performance of hand-tuned C++.

We present a case study comparing the performance of Java to C++. Following many prior similar studies (*e.g.* Moreira et al. (1998); Mallett (2001); Nikishkov et al. (2003); Miller et al. (2004)), we focus on multiple versions of a single code base that exhibits typical behaviour of many (but surely not all) scientific computing applications.

[*]This is an extended version of Källén and Wrigstad (2019). Code available on `https://github.com/fxpl/marielund.git`. The boxes representing execution times for Java with thread pools in Fig 3 and Fig 6 differ from the corresponding figures in Källén and Wrigstad (2019). Unfortunately, the wrong data is plotted in the conference paper.

On the C++ side, we study two different versions: a “naive” version that handle the refactoring-induced indirections (*c.f.,* Källén et al. (2014)) directly using virtual methods (dynamic polymorphism), and one that uses a much more elaborate scheme involving extensive use of template parameters to avoid virtual methods (and stay squarely inside static polymorphism) for the most frequently used methods Coplien (1995). In Java, dynamic polymorphism is the default.

We study two different parallelising approaches: using a non-standard OpenMP implementation (OMP4J by Bělohlávek and Steinhauser (2015)), which allows an almost direct transfer of OpenMP pragmas from the original C++ sources, and the “canonical way” using executor services from JDK.

Ultimately, we are interested in facilitating maintainable scientific software, including applying “battle tested” best-practises to the development of scientific computing applications (as aforementioned, *e.g.* SOLID Martin (2003) etc.). Thus, we place special interest in the ability of the Java JIT compiler to reduce the additional levels of indirection brought on by the more open and flexible code base stemming from the refactoring Källén et al. (2014), and especially performing the same well-known compiler optimisations possible using static polymorphism, in the C++ code. We also investigate which other differences in the languages affect the performance of our application, including garbage collection overheads and array abounds checking—two typical gripes from C++-oriented scientific computing developers.

In addition to serial execution (*e.g.* for simplicity), we compare the performance of the different program versions under typical deployment scenarios, both internal parallelism (OpenMP and executor services) and distributed computations on a cluster (using MPI).

Outline § 2 briefly surveys scientific computing. § 3 introduces Marielund, our compute kernel extracted from a larger application (HAParaNDA by Gustafsson and Holmgren (2010)). § 4 analyses performance C++ and Java version of our compute kernel. § 5 discusses related work. § 6 discusses our results and § 7 concludes.

2 Scientific Computing

With the advent of cheaper and more powerful computers, the possibility to make computer simulations is increasing. Computer simulations can be used to study processes that would be dangerous, expensive or even impossible to reconstruct in physical experiments. Examples of such processes are car crashes, earth quakes and interactions between molecules. Computer simulations are also used by meteorologists to predict tomorrow’s weather.

In order to make the simulations, we need to compute (an approximation of) the values of a function, describing for example temperature, pressure or displacement, using a *numerical method.* The function values are computed for a finite set of elements, constituting a *domain.* In practice the elements of the domain are often stored in a regular array. The domain may for example represent a physical area with the elements being different positions.

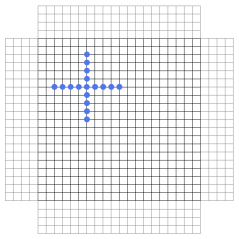


Figure 1: Two-dimensional computational block (black mesh) with ghost regions (grey mesh) and an eighth order multuncial stencil (blue dots). Here, the stencil is applied in the inner region, at the element with index $(6, 6)$.

A central, and computationally demanding, step in many numerical methods is to approximate the derivative of a function for each element in the domain. In a set of numerical methods called *finite difference methods*, this is done by applying a *stencil* on each element. Applying a stencil means computing a weighted sum of the element at which the stencil is applied and a number of neighbouring elements.

Fig 1 illustrates a two-dimensional finite difference stencil (any dimensionality > 0 is valid). The elements marked by blue dots are the elements that are included in the weighted sum computed when the stencil is applied on the middlemost element. Finite difference stencils can be of different widths. A broader stencil gives a better approximation of the derivative, but also takes longer time to apply. The stencil illustrated in Fig 1 includes eight neighbour values from along each dimension and is therefore said to be of eighth order.

Having explained the relation of stencil applications to scientific computing, we introduce the application that is at the center of our study in the next section.

3 Marielund—A Compute Kernel

HAParaNDA is a scientific computing application developed by Gustafsson and Holmgren (2010). It has, among other things, been used for solving the time-dependent Schrödinger equation Gustafsson et al. (2012b) and for numerical evaluation of the communication-avoiding Lanczos algorithm Gustafsson et al. (2012a). Currently, it consists of 12 000 lines of C++ code. HAParaNDA is written for solving high-dimensional linear problems, where high in this context typically is higher than 3. Examples of such problems are simulation of option market prices or interactions between molecules. The solver spends almost its entire execution time in a handful of methods, namely those that deal with stencil applications. HAParaNDA is an abbreviation for *High-dimensional Adaptive Parallel Numerical Dynamics API.* It is

Algorithm 1 Overview of Marielund.

```
Put inputValues in input block
Put resultValues in result block
for i = 1 : #applications do
    Apply stencil on input block and write result to
        result block
    tmp ← inputValues
    inputValues ← resultValues
    resultValues ← tmp
    Put inputValues in input block
    Put resultValues in result block
end for
```

also the name of a town in north-eastern Sweden.

As execution time of this type of application increases exponentially with the dimensionality, performance is an essential aspect of HAParaNDA, in particular the stencil applications. Thus, to facilitate the study, limit noise and increase portability, we have extracted a small compute kernel. It is this compute kernel that we have made several versions of (dynamic polymorphism, static polymorphism, Java). From this point on, we will refer to this kernel as Marielund, which is the name of a district of the town Haparanda. Algorithm 1 gives a high-level overview of how it works.

Marielund repeatedly applies an eighth order finite difference stencil on a domain, as described in § 2. Computations may be parallel, on a single node (machine) or distributed over many nodes. Each node operates on $\frac{1}{\#\text{nodes}}$ of the domain. As will be described later, the domain values are stored in an array. Each application uses one input array, which is only used for reading, and one result array, which is only used to store results. Thus, input values can be safely shared across threads, whereas the result array must be carefully "sliced" so that no two threads race to store the result value of the same element. Essentially, this means that no two threads' indices into the shared result array may alias.

A more realistic use case would also use the computed result values somehow. However, in HAParaNDA, the time spent on applications of stencils dwarf the time for pre- and post processing, which is therefore negligible in comparison. Consequently, we focus on the performance of the stencil application and only record the time spent on the applications of the stencil. Nevertheless, in order to mimic the behaviour of an iterative solver, we swap the pointers to the input values and the result values between each application.

3.1 Ghost Regions and Inter Node Communication

Under distributed deployment of Marielund (for performance or to allow data sets larger than the memory of a single node), each node only holds a small fraction of the elements. Here, we distinguish between data blocks and "ghost regions". The blocks contain the values on which the stencil is applied,

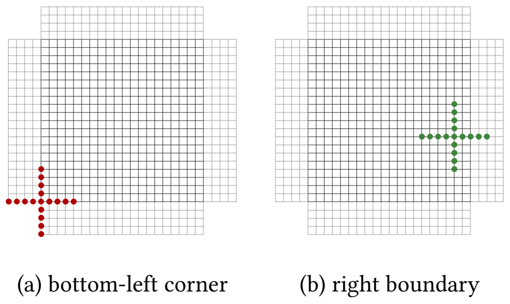


Figure 2: Two-dimensional computational blocks with eighth order multuncial stencils applied at the boundary.

while ghost regions contain a copy of the boundary values from the neighbour nodes. The latter are necessary because, when applying a stencil on an element, we need the values of W neighbouring elements in each direction, where W is the length of each stencil "arm" (in our case 4).

HAParaNDA uses MPI for inter-node communication, which is a standard technique in scientific computing. In order to make the Java and C++ codes as similar as possible, we keep this design (using the bindings of Vega-Gisbert et al. (2016)), despite the fact that it is not representative of Java applications from other domains. Algorithm 2 includes inter-node communication.

Algorithm 2 Overview of a stencil application, including communication.

```
Start sending boundary data to neighbour
Apply stencil in inner region
for i = 1 : #boundaries do
    Wait for the ghost region outside any boundary
        to be filled with values
    Apply stencil along that boundary
end for
```

3.2 Stencil Applications

Having initiated the non-blocking communication, we apply the stencil in the inner parts of the block. When explaining what this means, we refer to the element on which the stencil is currently applied as E. If the distance from E to the boundary is $\geq$W, the stencil is applied as usual. When the distance is <W in any direction, only elements that are inside E in that direction are included in the application. This procedure is summarised in Algorithm 3.

Thereafter, we apply the stencil along each boundary separately. When applying the stencil on an element E along a boundary, we only include elements that are outside E, and we add the result to the one already stored in the result block. For example, consider Fig 2. When applying the stencil in the last point of the left boundary, we apply the left arm of the stencil in Fig 2a and when we apply it in the first point of the bottom boundary, we apply the bottom arm. Likewise, we

Algorithm 3 Stencil application in the inner region of a block. w_0 denotes the centre weight of the stencil. w_{-i} and w_{+i} denote the stencil weight in the bottom and upper part of the stencil respectively, where i is the distance from the center weight. Likewise, v is the value of the element E, and $v_{+/-i}$ is the value of the element i steps above/below E. All distances are measured along the dimension D.

```
for each element E do
    result ← 0
    for each dimension D do
        if distance to bottom boundary $\geq$ extent then
            for i = 1 : extent do
                result ← result+$w_{-i} \cdot v_{-i}$
            end for
        end if
        result ← result+$w_0 * v$
        if distance to upper boundary $\geq$ extent then
            for i = 1 : extent do
                result ← result+$w_{+i} \cdot v_{+i}$
            end for
        end if
    end for
    $v_{res}$ ← result
end for
```

apply the right arm of the stencil in Fig 2b when applying the stencil along the right boundary. Before applying the stencil along a boundary, we make sure that all ghost data is received and stored in the corresponding ghost region.

3.3 Iterators

When a stencil is applied, iterators are used to step over the domain. Internally, all elements are stored in an array, and the iterator translates between linear array index and multi-dimensional indices. The numbers in a multi-dimensional index specify how many elements there are below the current point, along each dimension.

An iterator can be used to step over the domain and get and set the current value, and fetch the values of the neighbouring elements, which is needed when a stencil is applied. There are iterators that step over whole blocks, and iterators that only step along a boundary, specified by the user of the iterator. We have also implemented iterators that handle computational blocks with ghost regions, fetching neighbour values from a ghost region when needed. To facilitate reuse of code, we have created a class hierarchy for the iterators. The iterators use strategy classes for stepping over the domain and for fetching and changing data.

3.4 Intra-Node Parallelism

Marielund uses multiple threads per node. Each thread has its own iterator, which iterate over $\frac{1}{\#\text{threads}}$ of the computational blocks of the current node. Each thread applies the stencil on the elements iterated over by its own input iterator.

Inherited from HAParaNDA, the C++ versions of Marielund are parallelised using OpenMP. To deviate as little from the C++ version as possible, the Java version uses OMP4J by Bělohlávek and Steinhauser (2015), a Java library providing thread parallelisation in an OpenMP-like way.

While adoption of OMP4J is straightforward when translating C++ parallelised with OpenMP into Java, OMP4J is an independent implementation and does not contain all the functionality of OpenMP. The canonical way to parallelise Java application is using threads and concurrent elements provided by the JDK. Hence, we also carefully translated the OMP4J parallelism into using tasks for every parallel section served by a single `ThreadPoolExecutor` instance created for the entire program, in a standard task-parallel fashion. This also avoids specifying the number of threads at compile-time as is the case of OMP4J.

3.5 Block Operators

The stencil is represented by a class called `ConstFD8Stencil`. FD stands for finite difference, while 8 is the order of accuracy. `ConstFD8Stencil` inherits from the class `MultuncialStencil` which in turn inherits from `BlockOperator`.

The name `BlockOperator` is chosen because the operators operate on computational blocks. `MultuncialStencil` represents a stencil that has a mid point and one arm pointing in each direction. Each arm is parallel with one other arm, pointing in the opposite direction, and orthogonal to all other arms. In 2D, this corresponds to a cross shaped stencil, as shown in Fig 1 and Fig 2. In an eighth order finite difference stencil, each arm is of length 4. 'Multuncial' is an adjective describing an object with the shape of a *multunx*, which is our dimensionality-independent generalisation of the term *quincunx*—the pattern of five points organised as a cross in the same way as the five-spot on a six-sided dice[1].

3.6 Dynamic Binding

`MultuncialStencil` contains the two methods where the actual multiplications and additions of domain values and stencil weights are performed. One of these methods is summarised in Algorithm 3. The element values are fetched using dynamically bound methods in the iterator classes. The stencil weights are fetched from `ConstFD8Stencil` also using a dynamically bound method. This means that on average, `MultuncialStencil` calls one dynamically bound method for each arithmetic operation performed in the hot loops. (The methods of the input iterator, in turn, call dynamically bound methods in another iterator class, which call dynamically bound methods in the strategy classes.)

Even though we have only one concrete implementation of the block operator/stencil in Marielund, we keep the indi-

[1]Thanks to Kurt Otto for this suggestion!

rections/abstractions imposed by the original refactoring of HAParaNDA. This is done in order to facilitate addition of additional block operators and multuncial stencils, which are likely to be needed in HAParaNDA in the future.

3.7 From C++ to Java

We study 3 different versions of Marielund, which are denoted Marielund$_D$, Marielund$_S$ and Marielund$_J$ respectively, and are described in this section. Marielund$_D$ and Marielund$_S$ are written in C++, while Marielund$_J$ is written in Java. The key difference between Marielund$_D$ and Marielund$_S$ is that the former uses dynamic binding (virtual methods), for reasons of code quality and flexibility, whereas Marielund$_S$ uses static polymorphism through the Curiously Recurring Template Pattern (CRTP) as described by Coplien (1995), to achieve similar flexibility but without delaying resolution of method calls to run-time. The downside of using CRTP is considerably more convoluted code (as a rough indication: while the LOC only grow by 4% with CRTP, there is an 18% increase in the number of characters).

We have striven to make the Java code as similar to the dynamic C++ code as possible, but some differences are inevitable. Below, we list the most important ones.

Use of Const Modifiers in C++ The C++ code frequently uses `const` modifiers, to capture both that a method does not cause side-effects and that certain parameters will not be modified. This likely helps compiler to emit more efficient code. Java has no corresponding feature.

Use of Template Parameters in C++ The C++ code makes frequent use of template parameters to propagate dimensionality and stencil order throughout the code as a constant known at compile time. Again, this allows the C++ compiler to perform powerful optimisations like loop unrolling, since it knows *e.g.* the value of `extent` in Algorithm 3.

As of version 12, Java has no similar ability, and instead we rely on passing in constants as parameters to the JVM at startup and store these in static final fields. While this deprives the Java (source to bytecode) compiler of the possibility to perform loop unrolling (and constant folding, etc.) already at the bytecode level, the OpenJDK source to bytecode compiler emits very simple bytecode with few optimisations and leave those concerns for the JIT compiler. Inspecting the assembly output for the key methods (that vastly dominate execution time) from the JIT compiler reveals that it is still able to perform similar unrolling as C++. Another difference between the approaches is that in the case of the Java code, only one dimensionality and stencil order can be used in each execution. However, these parameters can be expected to stay constant through a single execution.

Note that on the C++-side, as a side-effect of this optimisation, the code must be recompiled for each configuration of the program. Java overcomes this naturally by delaying compilation to run-time. The Java code is arguably also cleaner as 195 propagations of template parameters can be removed completely (in about 2700 LOC depending on the version).

Multiple Inheritance to Interface Implementation Following Java practices, we resolve the issue of translating multiple inheritance straightforwardly using interfaces. As a consequence, the iterator classes are typed using an interface type rather than a class type. Hence, many frequent method calls are compiled using the `invokeinterface` bytecode rather than `invokevirtual`. In the past, the former has been known to be slower than the latter[2], although we were not able to produce meaningful slowdowns due to interface polymorphism in informal benchmarking.

Java's MPI Bindings The Java MPI bindings that we use are a thin wrapper around the standard C implementation, outside of the normal Java heap. Our main usage of MPI is sending parts of arrays of ghost data between adjacent nodes. As the JVM may move arrays during garbage collection, data communicated using MPI must be placed in a temporary buffer with a fixed address to which a pointer can be safely passed to the underlying C implementation of MPI. When, as in Marielund, non-blocking communication is used, the Java bindings enforce these buffers to be direct buffers, which are provided as standard classes in the Java SDK (package `java.nio`). However, the iterator classes use data stored in regular arrays. Consequently, we store the data of the blocks and ghost regions in regular arrays. Before sending data to another node, we copy it into the send buffer. Likewise, after receiving data, we copy it from the receive buffers to a regular array. (Storing the data in the direct buffers only does not improve performance.)

3.8 Notes on Performance

For readability and consistency, we use a two-dimensional array to store the stencil weights, in the Java code as well as in the C++ code. The C++ code only declares methods as dynamically bound (virtual) when our design required it. In order to make the Java code as similar to the C++ code as possible, every method that is not declared as `virtual` in the C++ code is declared as `final` in the Java code.

Most of the data used in the application is the arrays containing all domain elements and the result of the stencil applications. The arrays are allocated early in the program and repeatedly operated on until the end of the execution. Consequently, the amount of allocation and deallocation is kept minimal. This pattern is typical for an important class of scientific computing applications.

[2]At least in the programming folklore, see *e.g.* `https://dzone.com/articles/abstraction-slows-you-down` but see also *e.g.* Alpern et al. (2001).

Element access in the stencil application uses prefetcher-friendly, even strides. Therefore, we believe that memory blocking would not impact performance considerably.

3.9 Representativeness

Marielund is a relatively small application, but we find it representable for an important class of scientific computing applications, which typically:

1. repeatedly operate on a large data set.
2. allocate the data early during the execution and do not deallocate it until in the end.
3. keep a constant configuration through the execution, and polymorphic types can be expected not to change.
4. are inherently parallel and the load does not change during an execution.

4 Performance Analysis

Java's performance has been scrutinised by members of the scientific computing field, as witnessed by the related work in § 5. In the past, scientists have studied manual unrolling of loops, turning off array bounds checking etc. in an effort to increase Java performance. In this section, we revisit some of these questions in the context of Java 8.0.212 and 11.0.2 (henceforth referred to simply as Java 8 and 11) using the various versions of Marielund as our test vehicle. While Java 8 is "old", there are still many programs and libraries that do not yet work with Java 9 and above, thus warranting its inclusion. Naturally, one must be careful to draw grand conclusions from studies of just a single program. Nevertheless, as discussed in § 3.9, Marielund exhibits typical behaviour of an important class of scientific computing applications and we hope that our findings might dispel some prejudice.

4.1 Data Collection

We made a number of experiments to compare the performance of Marielund$_D$, Marielund$_S$ and Marielund$_J$. Each experiments was run 30 times for each version, as separate invocations of the JVM and Marielund$_D$/Marielund$_S$ respectively. The stencil was applied 8 times in each run, that is one run corresponds to 8 turns of the loop in Algorithm 1. In Marielund$_J$, also 8 warm-up applications were performed.

Execution Time A program like HAParaNDA is designed to make large scale computations that may run for several days. Consequently, the time for set up (including warm-up) and tear down of the program is negligible compared the time spent on stencil applications. Thus, we only time the 8 stencil applications: not set up, tear down or warm-up.

In the rest of this paper, *execution time* refers to the total time spent on computations and on communication (of ghost data). The time spent on communications include MPI calls to send, receive and wait for data, and (in Marielund$_J$) the time for copying data to and from direct buffers.

Warm-Up for Marielund$_J$ To determine the number of warm-up applications for Marielund$_J$, we repeatedly ran the program with an increasing number of warm-up applications until we no longer saw any performance implications, and the program had reached a steady state. Using this method, we ended up making 8 warm-up applications of the stencil before starting any measurements for Marielund$_J$.

Deployment The experiments were conducted under three different deployments, denoted Rackham, Snowy and Ada respectively. HAParaNDA-like applications are meant to be used with large data sets on large clusters, or it is at least in these cases where performance is critical (and run-time is counted in days). Rackham is a 486-node cluster with two 10-core Intel Xeon E5 2630 CPUs per node. Snowy is a 228-node cluster with two 8-cores Intel Xeon E5 2660 CPUs per node. Both clusters run CentOS 7 (3.10.0-957.21.2.el7.x86_64) and nodes are connected with Infiniband FDR. The nodes used in our experiments all have 128 GB of RAM. Ada is a dual core laptop with one Intel i7-4600U processor and 15.6 GB RAM, running Ubuntu 18.4 (4.15.0-54-generic).

Libraries We have used Open MPI version 4.0.0 (compiled from source in order to get C++ and Java bindings) in all experiments. The C++ versions of Marielund are compiled with g++ (wrapped in `mpic++`). On the two clusters, version 8.3.0 was used, while the g++ version on Ada is 7.4.0. All binaries were compiled with the following flags: `-DMPI_LIB -fPIC -MMD -MP -fopenmp -std=c++11 -O3`.

Classes using OMP4J were compiled using OMP4J version 1.2. For all JVM executions, we used the flags `-XX:+UseNUMA -XX:-UseCompressedOops -XX:+UseParallelGC`.

Experiment Configurations In our performance analysis, we experimented with parallelism on three different levels:

Serial Single-threaded runs (run on all platforms).

Threaded Parallel runs using OpenMP/OMP4J/standard JDK services (run on Rackham/Snowy) with the number of threads set to the number of cores on each node.

Distributed Distributed runs with 16 nodes each running a parallel configuration with as many threads as cores (run on Rackham and Snowy). We used `mpirun` with `--bind-to none` to start each node.

For each of these configurations, we made both two-dimensional (2D) and six-dimensional (6D) runs. These runs exhibit somewhat different characteristics and stress different parts of the code (as will be explained shortly).

For the 2D runs, we applied a two-dimensional multuncial stencil on a domain consisting of 16383^2 elements. For the

6D runs, we applied a six-dimensional stencil on a domain consisting of 25^6 elements[3]. Note that these domain sizes are the number of elements *on each node*, that is the size of the full domain in the serial and threaded runs, but only $\frac{1}{16}$ of the total domain size of the distributed runs. For Java, we set both the minimum and the maximum heap sizes to 12 GB and 24 GB for the two-dimensional and the six-dimensional runs respectively. Moreover, we set `ParallelGCThreads` to the number of threads used by Marielund.

Statistical Rigour We compared the execution times of Marielund$_D$ and Marielund$_S$ to those of Marielund$_J$ running two different Java versions (Java 8 and Java 11). Here, we used the OMP4J version of Marielund$_J$. Thereafter, we compared the two Java versions of Marielund$_J$ for both Java 8 and Java 11. These comparisons consists of pair-wise t-tests of the mean of the execution times, performed using the `R` function `t.test`. Unfortunately, the execution time distributions are not symmetric for any experiment. To compensate for the fact that the preconditions of a t-test are not fulfilled, we also made a permutation test, resampling 1 000 000 times. Below, we present box plots of the execution times for each dimensionality and level of parallelisation.

Memory Measurements To compare the amount of memory used by the different versions of Marielund, we measured memory usage of the C++ versions of the program using the valgrind tool massif. Since the JVM segfaulted when we tried to run it through massif, instead, we used VisualVM to visualise the memory usage for the Java versions. We used threaded runs of the programs for all these measures.

Assembly Analysis To understand the behaviour of the compilers (*e.g.* inlining, etc.) we studied the assembly codes. For C++, we used callgrind with `--dump-instr=yes` and `--collect-jumps=yes`. This annotates the output with the number of times each instruction is executed.

For Java, we used JMH (Java Microbenchmark Harness) with the flag `-prof perfasm` to obtain the assembly code for the hottest regions of the Java code, together with information of how much of the execution time was spent on the most expensive and/or frequently executed instructions.

4.2 Analysis

For brevity, we focus on the Rackham cluster results.

Computational Performance Fig 3 shows box plots of the execution times on Rackham. It shows that the C++ version with dynamic binding is considerably *slower* than all Java versions, while the C++ version with static binding is considerably *faster* than all Java versions.

This agrees with the results of Tab 1, which show that the slowdown of Marielund$_D$ over Marielund$_J$ is statistically significant in all cases studied, and with Tab 2, which shows the same for the slowdown of Marielund$_J$ over Marielund$_S$.

The biggest difference between the two thread parallelisation strategies for Java is found for two-dimensional serial runs, where the Java version using native Java threads is faster. As can be seen in Tab 3, the difference is highly significant both for Java 8 and Java 11. On the contrary, in the two-dimensional threaded experiments and the six-dimensional serial experiments, we cannot see any statistically significant difference at all. In the six-dimensional threaded experiments, we see a statistically significant difference on the level 0.01 and 0.05 for Java 8 and Java 11 respectively. However, the absolute difference between the execution times of the two parallelisation strategies is generally small (*c.f.*, Fig 3/Tab 3).

The results in this subsection were reproduced at Snowy and (in the serial cases) Ada, with the only exception that Marielund$_J$ with thread pools is significantly faster than the OMP4J version also for six-dimensional experiments at Ada, see Fig 4, Tab 4 – Tab 6, Fig 5 and Tab 7 – Tab 9.

Memory Usage The memory usage of Marielund$_D$ and Marielund$_S$ are basically constant through the whole execution. Its maximum value is 4.026 GB in the two-dimensional experiments and 7.115 GB in the six-dimensional ones.

For Marielund$_J$ with thread pools, the memory usage does not vary much, and stays constant around 5 GB and 12 GB the whole run. (Certainly the memory usage increases over time, but if we perform 100 applications of the stencil, the largest memory usage is not more 5.170 GB and 11.410 GB respectively. This can be compared to the first memory usage shown by VisualVM: 4.990 GB and 10.990 GB respectively.)

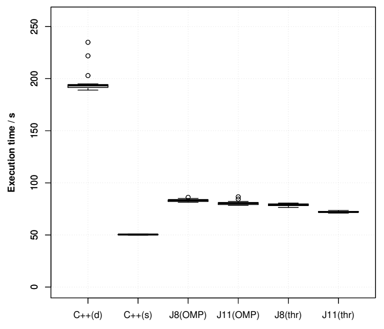
In the Marielund$_J$ version that employs OMP4J, the memory usage is considerably higher. Garbage collection is triggered at least every 12th and 6th application of the stencil in the two-dimensional and the six-dimensional case respectively. Garbage collection decreases the memory usage from just below 7 GB to 4.1 GB in the two-dimensional runs. In the six-dimensional runs, garbage collection after the warm-up phase decrease the heap size from around 13 GB to 7.1 GB.

5 Related Work

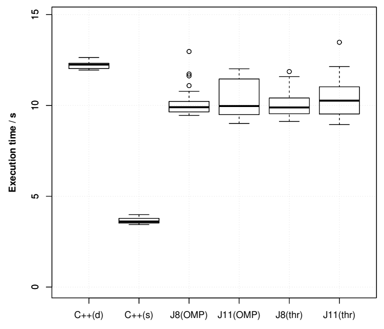
The performance of Java and C/C++ for scientific computing applications has been compared in many studies. Most of them conclude that you can obtain higher performance in C/C++ than in Java, although the relative difference varies from a couple of percent to a factor 60. Tab 10 shows an overview of the results discussed below.

Notably, most of these studies are more than 15 years old today and, naturally, cover relatively early versions of Java.

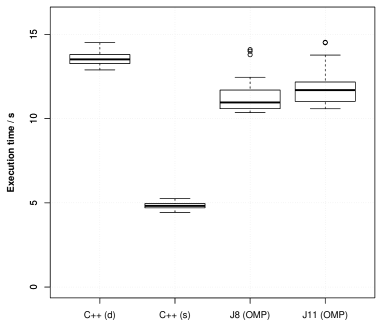
[3]We limited our domain sizes by the maximum allowed size of the MPI direct buffers used in Marielund$_J$ in order to not have to deviate further from the C++ original.



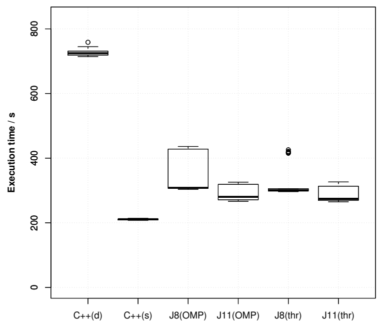
(a) Single-threaded runs, 2D.



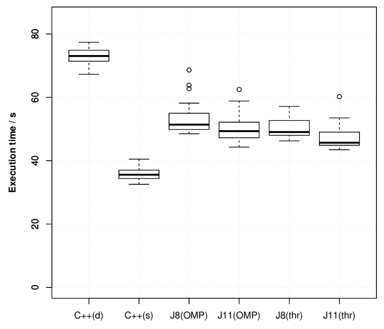
(b) Thread parallelised runs, 2D.



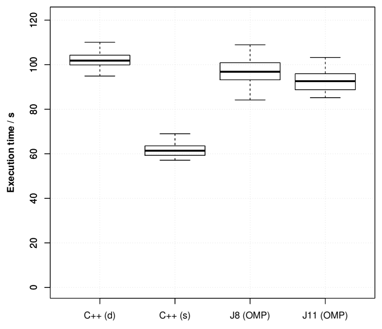
(c) Distributed runs, 2D.



(d) Single-threaded runs, 6D.



(e) Thread parallelised runs, 6D.



(f) Distributed runs, 6D.

Figure 3: Execution times on the Rackham cluster. X axes show code versions and y axes show execution times in seconds. Note variable Y axes. `s`=static, `d`=dynamic, `J`=Java, `OMP`=OMP4J, `thr`=native Java threads.

Table 1: Execution times in seconds and statistics comparing the execution times of Marielund$_D$ on the Rackham cluster with those of Marielund$_J$. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

Setup			**Execution times**		**Statistics**				
parallelisation	**dim**	**Java v**	**C++**	**Java**	**t**	p_t		p_p	
serial	2	8	195.37	83.16	64.3782	1.4965e-33	***	0.0000e+00	***
serial	6	8	726.50	344.83	35.4105	1.6489e-26	***	0.0000e+00	***
threaded	2	8	12.20	10.12	14.0344	3.4723e-15	***	0.0000e+00	***
threaded	6	8	73.03	53.34	19.9775	5.0619e-23	***	0.0000e+00	***
distributed	2	8	13.56	11.34	11.0039	6.1525e-13	***	0.0000e+00	***
distributed	6	8	102.01	96.80	4.1437	1.5005e-04	***	8.7000e-05	***
serial	2	11	195.37	80.48	65.2464	9.1230e-35	***	0.0000e+00	***
serial	6	11	726.50	294.26	91.2004	2.5774e-47	***	0.0000e+00	***
threaded	2	11	12.20	10.38	9.7546	6.2703e-11	***	0.0000e+00	***
threaded	6	11	73.03	49.88	26.0803	7.0746e-29	***	0.0000e+00	***
distributed	2	11	13.56	11.80	8.9072	1.4415e-10	***	0.0000e+00	***
distributed	6	11	102.01	92.64	9.4811	4.2104e-13	***	0.0000e+00	***

Table 2: Execution times in seconds and statistics comparing the execution times of Marielund$_S$ on the Rackham cluster with those of Marielund$_J$. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

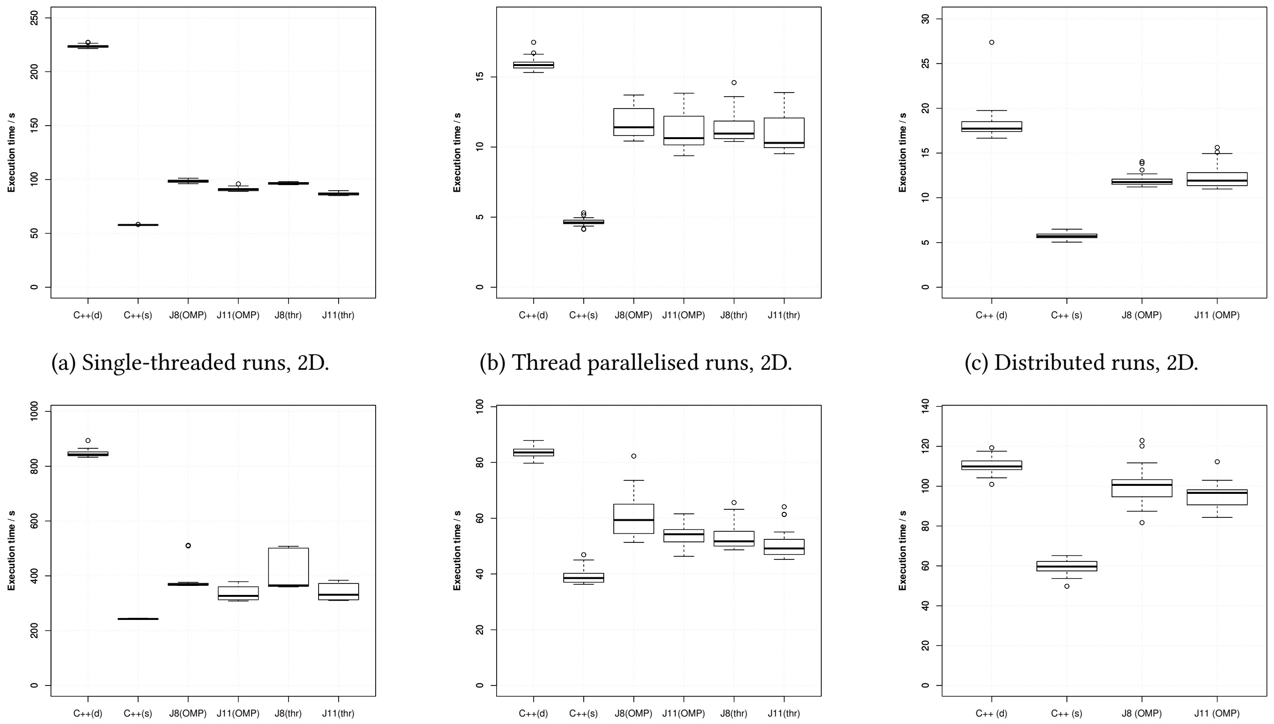
Setup			Execution times		Statistics		
parallelisation	**dim**	**Java v**	**C++**	**Java**	**t**	p_t	p_p
serial	2	8	50.38	83.16	-155.2457	9.2794e-49 ***	0.0000e+00 ***
serial	6	8	210.64	344.83	-12.6331	2.5011e-13 ***	0.0000e+00 ***
threaded	2	8	3.66	10.12	-43.9052	1.4038e-29 ***	0.0000e+00 ***
threaded	6	8	35.69	53.34	-18.5926	1.8826e-20 ***	0.0000e+00 ***
distributed	2	8	4.83	11.34	-33.2960	5.5144e-26 ***	0.0000e+00 ***
distributed	6	8	61.74	96.80	-28.4127	3.3586e-29 ***	0.0000e+00 ***
serial	2	11	50.38	80.48	-92.0714	6.0202e-39 ***	0.0000e+00 ***
serial	6	11	210.64	294.26	-19.1334	4.1494e-18 ***	0.0000e+00 ***
threaded	2	11	3.66	10.38	-36.1156	1.7448e-26 ***	0.0000e+00 ***
threaded	6	11	35.69	49.88	-16.7433	1.4110e-19 ***	0.0000e+00 ***
distributed	2	11	4.83	11.80	-36.3093	3.4517e-27 ***	0.0000e+00 ***
distributed	6	11	61.74	92.64	-32.2752	6.2988e-36 ***	0.0000e+00 ***

Table 3: Execution times in seconds and statistics comparing the execution times on the Rackham cluster when Marielund$_J$ is parallelised with OMP4J and fixed thread pools respectively. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

Setup			Execution times		Statistics		
parallelisation	**dim**	**Java v**	**OMP4J**	**threads**	**t**	p_t	p_p
serial	2	8	83.16	78.99	15.2788	7.6280e-22 ***	0.0000e+00 ***
serial	6	8	344.83	327.28	1.2388	2.2047e-01	1.9242e-01
threaded	2	8	10.12	10.09	0.1336	8.9420e-01	8.9599e-01
threaded	6	8	53.34	50.32	2.8685	6.0521e-03 **	4.3640e-03 **
serial	2	11	80.48	72.25	23.5331	1.2925e-24 ***	0.0000e+00 ***
serial	6	11	294.26	286.55	1.3081	1.9605e-01	1.9347e-01
threaded	2	11	10.38	10.35	0.0931	9.2617e-01	9.2588e-01
threaded	6	11	49.88	47.31	2.4645	1.6743e-02 *	1.5722e-02 *

Table 4: Execution times in seconds and statistics comparing the execution times of Marielund$_D$ on the Snowy cluster with those of Marielund$_J$. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

Setup			Execution times		Statistics		
parallelisation	**dim**	**Java v**	**C++**	**Java**	**t**	p_t	p_p
serial	2	8	223.60	98.35	357.1784	1.2309e-98 ***	0.0000e+00 ***
serial	6	8	846.82	396.46	41.5535	3.1632e-29 ***	0.0000e+00 ***
threaded	2	8	15.91	11.74	20.3569	2.1034e-22 ***	0.0000e+00 ***
threaded	6	8	83.72	60.99	15.5719	9.0527e-17 ***	0.0000e+00 ***
distributed	2	8	18.26	11.96	17.5238	2.3100e-19 ***	0.0000e+00 ***
distributed	6	8	110.43	100.12	6.0150	3.5437e-07 ***	0.0000e+00 ***
serial	2	11	223.60	91.09	330.2801	1.6505e-92 ***	0.0000e+00 ***
serial	6	11	846.82	334.33	101.1264	8.8907e-53 ***	0.0000e+00 ***
threaded	2	11	15.91	11.19	18.8366	4.7170e-20 ***	0.0000e+00 ***
threaded	6	11	83.72	53.89	37.4941	4.6802e-35 ***	0.0000e+00 ***
distributed	2	11	18.26	12.30	14.7008	7.1018e-20 ***	0.0000e+00 ***
distributed	6	11	110.43	95.28	11.3307	1.1440e-15 ***	0.0000e+00 ***



(a) Single-threaded runs, 2D. (b) Thread parallelised runs, 2D. (c) Distributed runs, 2D.

(d) Single-threaded runs, 6D. (e) Thread parallelised runs, 6D. (f) Distributed runs, 6D.

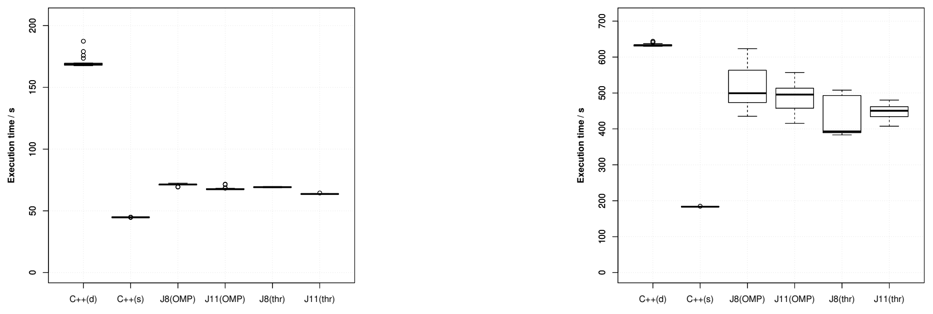
Figure 4: Execution times on the Snowy cluster. X axes show code versions and y axes show execution times in seconds. Note variable Y axes. `s`=static, `d`=dynamic, `J`=Java, `OMP`=OMP4J, `thr`=native Java threads.

Table 5: Execution times in seconds and statistics comparing the execution times of Marielund$_S$ on the Snowy cluster with those of Marielund$_J$. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

Setup			**Execution times**		**Statistics**				
parallelisation	**dim**	**Java v**	**C++**	**Java**	**t**	p_t		p_p	
serial	2	8	57.85	98.35	-162.5840	9.0793e-46	***	0.0000e+00	***
serial	6	8	242.60	396.46	-14.5209	7.6019e-15	***	0.0000e+00	***
threaded	2	8	4.64	11.74	-36.5754	6.8428e-28	***	0.0000e+00	***
threaded	6	8	39.09	60.99	-14.7127	1.1317e-16	***	0.0000e+00	***
distributed	2	8	5.75	11.96	-45.6920	3.8718e-37	***	0.0000e+00	***
distributed	6	8	59.54	100.12	-24.3699	2.7714e-25	***	0.0000e+00	***
serial	2	11	57.85	91.09	-105.1228	1.1252e-39	***	0.0000e+00	***
serial	6	11	242.60	334.33	-20.2699	9.9346e-19	***	0.0000e+00	***
threaded	2	11	4.64	11.19	-27.0459	3.5227e-23	***	0.0000e+00	***
threaded	6	11	39.09	53.89	-17.4375	4.1754e-23	***	0.0000e+00	***
distributed	2	11	5.75	12.30	-28.1727	1.2956e-24	***	0.0000e+00	***
distributed	6	11	59.54	95.28	-28.0639	3.8093e-31	***	0.0000e+00	***

Table 6: Execution times in seconds and statistics comparing the execution times on the Snowy cluster when Marielund$_J$ is parallelised with OMP4J and fixed thread pools respectively. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

Setup			**Execution times**		**Statistics**		
parallelisation	**dim**	**Java v**	**OMP4J**	**threads**	**t**	p_t	p_p
serial	2	8	98.35	96.51	6.2583	8.9148e-08 ***	0.0000e+00 ***
serial	6	8	396.46	418.98	-1.3645	1.7783e-01	1.5839e-01
threaded	2	8	11.74	11.39	1.2539	2.1494e-01	2.1411e-01
threaded	6	8	60.99	52.96	5.0454	8.3742e-06 ***	4.0000e-06 ***
serial	2	11	91.09	86.83	11.1387	2.3863e-15 ***	0.0000e+00 ***
serial	6	11	334.33	341.39	-1.0385	3.0341e-01	3.0195e-01
threaded	2	11	11.19	11.03	0.4776	6.3473e-01	6.3232e-01
threaded	6	11	53.89	50.41	3.0807	3.2166e-03 **	3.1750e-03 **



(a) Single-threaded runs, 2D.

(b) Single-threaded runs, 6D.

Figure 5: Execution times on the laptop Ada. X axes show code versions and y axes show execution times in seconds. Note variable Y axes. `s`=static, `d`=dynamic, `J`=Java, `OMP`=OMP4J, `thr`=native Java threads.

Table 7: Execution times in seconds and statistics comparing the execution times of Marielund$_D$ on the laptop Ada with those of Marielund$_J$. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

Setup			**Execution times**		**Statistics**		
parallelisation	**dim**	**Java v**	**C++**	**Java**	**t**	p_t	p_p
serial	2	8	169.90	71.28	128.8770	2.5685e-43 ***	0.0000e+00 ***
serial	6	8	633.61	514.18	12.1755	5.5646e-13 ***	0.0000e+00 ***
serial	2	11	169.90	68.04	129.4550	2.9551e-47 ***	0.0000e+00 ***
serial	6	11	633.61	491.63	20.9229	2.8504e-19 ***	0.0000e+00 ***

Table 8: Execution times in seconds and statistics comparing the execution times of Marielund$_S$ on the laptop Ada with those of Marielund$_J$. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

Setup			**Execution times**		**Statistics**		
parallelisation	**dim**	**Java v**	**C++**	**Java**	**t**	p_t	p_p
serial	2	8	44.78	71.28	-216.4792	8.4052e-49 ***	0.0000e+00 ***
serial	6	8	183.52	514.18	-33.7846	7.6542e-25 ***	0.0000e+00 ***
serial	2	11	44.78	68.04	-105.5458	3.2008e-39 ***	0.0000e+00 ***
serial	6	11	183.52	491.63	-45.6190	1.4683e-28 ***	0.0000e+00 ***

Table 9: Execution times in seconds and statistics comparing the execution times on the laptop Ada when Marielund$_J$ is parallelised with OMP4J and fixed thread pools respectively. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test.

Setup			**Execution times**		**Statistics**				
parallelisation	**dim**	**Java v**	**OMP4J**	**threads**	**t**	p_t		p_p	
serial	2	8	71.28	69.21	16.1607	8.6287e-18	***	0.0000e+00	***
serial	6	8	514.18	419.88	7.1132	1.9683e-09	***	0.0000e+00	***
serial	2	11	68.04	63.70	19.4429	8.4436e-19	***	0.0000e+00	***
serial	6	11	491.63	448.66	5.5665	1.3462e-06	***	0.0000e+00	***

To our knowledge, no comparison between C/C++ and Java has been made using a Java version newer than Java 6.

Atwood et al. (1997) implemented an iterative solver in C and Java 1.1. The C version appeared to be almost 6 times faster. Boisvert et al. (1998) compared Java 1.1 and C++ implementations of a number of linear algebra operations (axpy, dot products, dense matrix-matrix multiplications and sparse matrix-vector multiplications). They tuned the Java code by manually unrolling loops, by only using one-dimensional arrays, and by blocking loops. In best case, Java execution times are a factor 2 larger than the C ones for axpy, dot product and dense matrix-matrix multiplication. For sparse matrix-vector multiplication, Java appears to be 20–50% slower than C.

The same year, Moreira et al. (1998) compared the two languages for a matrix-matrix multiplication and addition. Their unoptimised Java implementation is about 60 times slower than the corresponding C implementation. However, by turning run-time checks of array indices off (which is not legal in Java), they got a 15-fold improvement in performance. They also wrote and compiled the C program according to “Java rules”, that is with loop unrolling and reordering of instructions disabled, by not using fused multiply-add and only using one-dimensional arrays. This makes the C program 4 times slower, and the execution times are more or less the same as those for the latter Java version of the program. Two years later, they made a study were they introduced *safe regions*, where they could prove that no array access would be illegal Moreira et al. (2000). Thereby, they could turn array index checks off in these regions. They also optimised the code using blocking and unrolling of loops, scalar replacements and activation of fused multiply-add. They applied this on the same operation as in the paper from 1998, and got a Java program that was only 17–59% slower than the corresponding C++ programs.

Mallett (2001) compared a C and a Java implementation of a molecular dynamics simulation. The Java version appeared to be between 30% and a factor 2 slower than the C version, depending on platform and problem size. Bull et al. (2003) compared Java and C++ implementations of a number of compute kernels, which they claim are representative for scientific computing software. Depending on which platform they ran the programs, the Java version was from 7% to 4 times slower than the C++ version, on average. Schatzman (2001) implemented fast Fourier transform and matrix-vector multiplication in C++ and Java. For both applications, the Java program was at most 50% slower than the corresponding C++ program. He also concluded that Java performs worse than C++ when it comes to method calls (*e.g.* getters and setters). Just as Moreira et al., he also mentions array bounds checking as a factor that limits Java’s performance.

Nikishkov et al. (2003) compared Java and C for finite element computations. In the Java program, they avoided object-oriented features with considerable overhead (*e.g.* object creation) in the performance critical parts of the code. Their results suggest that Java 1.2 is faster than Java 1.3 and Java 1.4. In an application with consecutive access of array element, Java 1.2 was about 10% slower than C. With non-consecutive array element access Java, got about 50% slower than C. They also found that in some environments, explicit unrolling of loops speeded the Java program up.

Miller et al. (2004) compared C++ and Java implementations of a finite element solver. For small (2D) problems, Java was around 3 times slower than C++, but for larger (3D) problems, the difference was less than a factor 2. They also found that using object-oriented techniques for representing tensors in the solvers resulted in higher performance. Their explanation for this is that the tensor abstraction provides a built-in framework for loop unrolling and blocking. However, in the object-oriented versions, they avoided constructor calls that might otherwise reduce performance.

Vivanco and Pizzi (2005) compared C++ and Java for a number of different applications. First, they studied the performance of array accesses and found that Java is over 2 times slower than C++, in the best case. SciMark 2.0 contains 5 basic scientific computing operations, which were also implemented in C++ and Java by Vivanco and Pizzi. Here, Java 1.3 appeared to be faster than Java 1.2. For small problem sizes, Java 1.3 was 14% slower than C++ and for large problem sizes Java 1.3 was 26% slower than C++.

Five years later, in 2009, Shafi et al. (2009) compared C and Java versions of two scientific computing applications: an n-body simulation and a finite difference solver for Maxwell’s equations. They applied a number of optimisations on the Java code, such as conversion of two-dimensional arrays to one-

Table 10: Summary of Java vs. C/C++ performance comparisons in the Scientific Computing domain −2019. † = See § 5.

Researchers	Year	Type of Application	C/C++	Java	Java Slowdown
Atwood et al. (1997)	1997	iterative solver	C	1.1	6 ×
Boisvert et al. (1998)	1998	linear algebra operations	C++	1.1	2 ×
Boisvert et al. (1998)	1998	sparse matrix–vector multiplication	C++	1.1	1.2–1.5 ×
Moreira et al. (1998)	1998	matrix operations	C	1.?	4–60 ×
Mallett (2001)	2001	n-body simulation(?)	C	1.2	1.3–2 ×
Bull et al. (2003)	2001	n-body, FFT, sparse mat.–vec. mult. etc.	C++	1.2–1.4	1.07–4 ×
Schatzman (2001)	2001	FFT & matrix–vector multiplication	C++	1.3	<1.5 ×
Nikishkov et al. (2003)	2002	finite element solver 3D	C	1.2–1.4	1.1–1.5 ×
Miller et al. (2004)	2003	finite element solver 2D / 3D	C++	1.3	3 × / <2 ×
Vivanco and Pizzi (2005)	2004	array accesses	C++	1.2–1.3	2 ×
Vivanco and Pizzi (2005)	2004	SciMark 2.0	C++	1.2–1.3	1.14–1.26 ×
Shafi et al. (2009)	2009	n-body simulation & finite diff. solver	C	1.5–1.6	†
Taboada et al. (2013)	2013	n-body simulation	C	1.6	2 ×
Oancea et al. (2011)	2010	BLAS and LAPACK methods	C	1.6	†
Vega-Gisbert et al. (2016)	2016	NPB benchmarks	C	1.6	≈1–≈3 ×
This work	2019	kernel extracted from iterative solver	C++	1.8, 1.11	*c.f.*, § 4.2

dimensional ones and replacement of objects with primitive data types. The resulting Java programs were in some cases faster than the corresponding C programs. However, the authors of the paper note that replacing Java objects with primitive data types deteriorated the readability of the code. The n-body simulation was also studied by Taboada et al. (2013), who found the Java implementation to be at least twice as slow as the C implementation.

Amedro et al. (2008) found considerable improvements in computational performance in Java 1.6 compared to Java 1.4.

In 2010, Oancea et al. (2011), implemented some BLAS and LAPACK methods in Java 1.6. They optimised the code employing among other things loop unrolling, loop invariant code motion, common sub-expression elimination and inline expansion. When it comes to matrix multiplication, their implementation is slower than the corresponding C implementation for small matrices but faster than the C implementation for large matrices. The Java implementation of LU factorisation is on pair with the C implementation for small matrices but considerably slower for large matrices.

A more recent performance comparison of C and Java 1.6 was done by Vega-Gisbert et al. (2016) using the NPB benchmark suite. For NPB's conjugate gradient solver, the Java performance was close to that of C. However, for the other four benchmark problems, the Java implementations were notably slower than the C implementations.

Only three of the aforementioned papers compare the parallel performance of C/C++ and Java: Shafi et al. (2009); Taboada et al. (2013); Vega-Gisbert et al. (2016). Of these, only Vega-Gisbert et al. (2016) use Open MPI for their experiments.

6 Discussion

In this section, we discuss performance and memory, and revisit typical folklore "Java performance gripes" from the scientific computing community.

6.1 Computational Performance

The initial motivation for this work was primarily to see if Java would be able to "compile away" the apparent virtual calls due to increased abstraction, and ultimately inline the called methods in the hot loops. Indeed, the assembly code (stabilised after 8 warm-up runs in the case of Java), shows that the dynamically bound methods in the iterator classes are inlined in Marielund$_S$ and Marielund$_J$, but not in Marielund$_D$. The method for retrieving the stencil weights (denoted w in Algorithm 3) is inlined in all versions of the code. Evidently, g++ realises that there is only a single concrete subclass of `MultuncialStencil` and leverages this information to optimise the code. Thus, adding more concrete realisations of `MultuncialStencil` may lead to performance drops, increasing the performance gap between Marielund$_D$ and the other versions of Marielund.

Despite the inlining, Marielund$_J$ is slower than Marielund$_S$. One culprit, required by Java's MPI bindings, is copying of values to and from the direct buffers for each stencil application and boundary. To investigate, we measured the execution time with the time for communication excluded. Fig 6 and Tab 11 show that the difference is considerably smaller when communication time is excluded. With Java 11, for the six-dimensional threaded runs, the difference is as small as 4%[4]. Since the performance difference of the MPI function

[4]The computation times are 19.99 and 20.78 seconds respectively. $20.78 - 19.99 = 0.88$, $\frac{0.88}{20.78} \approx 0.04$, $\frac{0.88}{19.99} \approx 0.04$.

Table 11: Computation times and statistics comparing the computation times of Marielund$_S$ on the Rackham cluster with those of Marielund$_J$. p_t denotes the p value for the t-test, while p_p denotes the p value for the permutation test. Note that in this table, the time needed for communication is excluded.

Setup			**Computation times**		**Statistics**				
parallelisation	**dim**	**Java v**	**C++**	**Java**	**t**	p_t		p_p	
serial	2	8	50.36	74.78	-128.7539	3.2299e-47	***	0.0000e+00	***
serial	6	8	199.55	307.13	-10.2106	4.0252e-11	***	0.0000e+00	***
threaded	2	8	3.60	5.50	-57.5504	4.7826e-47	***	0.0000e+00	***
threaded	6	8	19.99	23.26	-16.0978	5.3065e-18	***	0.0000e+00	***
distributed	2	8	4.38	5.73	-38.0804	8.8778e-35	***	0.0000e+00	***
distributed	6	8	21.76	32.01	-33.9480	2.2683e-28	***	0.0000e+00	***
serial	2	11	50.36	72.30	-90.9358	3.6428e-40	***	0.0000e+00	***
serial	6	11	199.55	255.25	-13.2116	7.0758e-14	***	0.0000e+00	***
threaded	2	11	3.60	5.31	-23.8759	6.8642e-25	***	0.0000e+00	***
threaded	6	11	19.99	20.78	-3.9562	3.5989e-04	***	1.4800e-04	***
distributed	2	11	4.38	5.91	-21.0316	7.3434e-24	***	0.0000e+00	***
distributed	6	11	21.76	24.50	-24.6859	1.0703e-29	***	0.0000e+00	***

calls should be minimal Vega-Gisbert et al. (2016), this suggests that the transfer of buffer values is an important reason for the performance difference seen in Fig 3 and Tab 2.

To understand the remaining difference in the execution times of Marielund$_J$ and Marielund$_S$, we studied assembly codes for the hottest loop in both versions: the loop over D in Algorithm 3. In the case of Marielund$_J$, we focused on the code generated during serial executions. For the two-dimensional stencil applications, the main differences are:

1. The C++ code for retrieving the stencil weights (w in Algorithm 3) from a 2D array is compiled down to a single load, while Java's handling of 2D arrays (as separate nested arrays) incurs considerable overhead.
2. Iteration over arrays of sizes known at compile-time still require array bounds checking in Java.
3. Java performs array bounds check on every element of arrays whose size is defined at run-time. This is done every time one of the values denoted by v (with or without subscripts) in Algorithm 3 is accessed.
4. The D loop is unrolled by g++ but not by Java's JIT compiler.
5. Instructions from different lines are more frequently reordered by the JIT compiler than by g++.
6. In Marielund$_J$, values are moved between registers, and on the stack, to a larger extent than in Marielund$_S$.

Six-dimensional runs exhibit similar differences, modulo:

1. The stencil weights array is loaded outside the hottest region, and only the sub arrays are fetched within it, eliminating some of the overhead of 2D array handling.
2. The D loop (which now goes from 0 to 5 instead of 0 to 1) is not unrolled in any of the code versions.

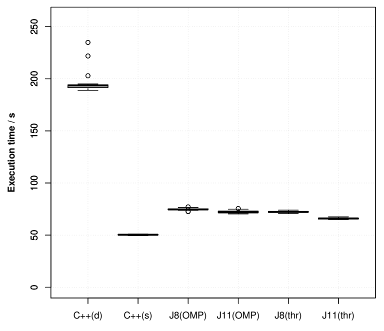
6.2 Memory Usage

When garbage collection occurs, the minimal memory foot print of Marielund$_J$ is very close to that of the C++ versions of Marielund. This indicates that the content on the heap of a Java program does not necessarily have to be much larger than that of the corresponding C++ program. In Marielund$_J$ that uses a thread pool, no garbage collection occurs. A reason that this process needs more memory is probably garbage from the warm-up phase, which is collected in the OMP4J version of Marielund$_J$. The memory of the thread pool version does increase over time (although slowly), and if we let the program run for days, garbage collection would eventually occur. Then it is reasonable to think that the memory usage would go down to the lowest memory usage level of the OMP4J version. The reason for the increase in memory usage of the thread pool version of Marielund$_J$ is most likely iterator and boundary objects that are created in each application of the stencil. These objects are explicitly deleted in the C++ versions of Marielund.

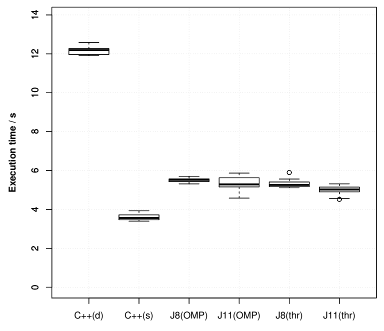
The OMP4J version allocates considerably more memory in each application than the thread pool version does. When visualising the execution in VisualVM, we saw that OMP4J seems to spawn new threads each time it enters a parallel section, as opposed to the thread pool version that creates the threads once and then reuses them. This is probably one reason for the more rapidly increasing memory usage. It is also likely that the frequent thread spawning is a reason for the lower computational performance of the OMP4J version.

6.3 Common Performance Gripes

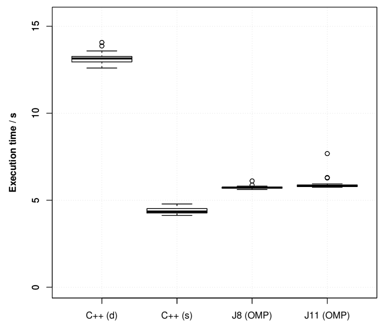
The folklore truth with respect to the use of Java for scientific computing is that C++ gives more abilities to optimise and that Java is plagued with considerable overheads: garbage collection, array bounds checking, virtual calls, JIT compilation overhead and lack of support for $>$1-dimensional arrays.



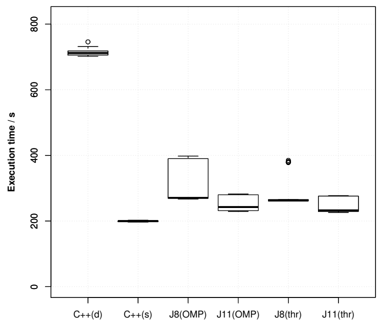
(a) Single-threaded runs, 2D.



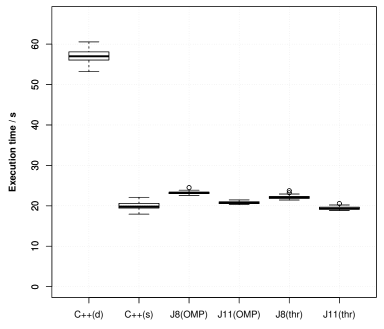
(b) Thread parallelised runs, 2D.



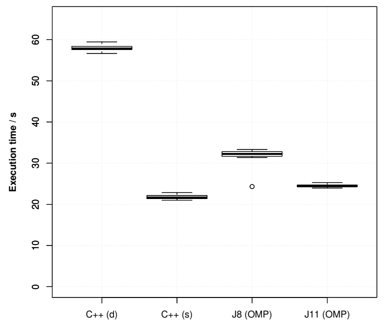
(c) Distributed runs, 2D.



(d) Single-threaded runs, 6D.



(e) Thread parallelised runs, 6D.



(f) Distributed runs, 6D.

Figure 6: Computation times on the Rackham cluster. X axes show code versions and y axes show execution times in seconds. Note variable Y axes. `s`=static, `d`=dynamic, `J`=Java, `OMP`=OMP4J, `thr`=native Java threads.

In the case of Marielund, *garbage collection* overheads was not a major pain point. The application more or less executes in constant space, and what few garbage objects are created per application are dwarfed by the size of the arrays.

With respect to *array bounds checking*, close inspection of the annotated JMH output reveals that ≈4% of the total run-time is spent on array bounds checking—in an application where arrays are very frequently accessed. This is a considerable improvement in modern Java versions, compared to the version used by Moreira et al. (1998).

On the notion of $>$*1D arrays*, when refactoring the 2D array into a 1D array we did not see any performance boost in the Marielund$_J$ version that uses a thread pool.

With respect to *virtual calls*, Java does better than C++ by allowing static indirections in the code to be removed dynamically. Java final calls are byte-compiled into `invokevirtual` and optimisation happens in the JIT compiler. Regarding *JIT compilation overhead*, we found that for a regular application, compiled code stabilised quickly; $\leq$ 8 stencil applications were enough to fully compile the hot regions. For applications like HAParaNDA that do large number of applications, the cost of compiling at run-time is thus negligible.

7 Conclusions

Our study shows that Java's JIT compiler may eliminate the performance penalty of dynamic binding when the concrete type is constant through the whole program run, which can be expected to be the case in many scientific computing programs. Nevertheless, the Java version performs worse than the statically bound C++ version that uses CRTP. The main reason for this is copying overhead for the direct buffers mandated by our use of MPI and non-blocking communication. Therefore, it might be a wise decision to look for another solution for inter-node communication when communicating large data sets using non-blocking communication.

Excluding the communication overhead, the performance difference between the six-dimensional thread parallelised runs in Java 11 and the fastest C++ version is as low as 4%, which is equal to the time spent in array bounds checking, according to our approximation in § 6.3. While Java's array bound checking has a noticeable overhead, the overhead is quite small considering the extensive array accessing. Our result further indicates that in some cases, the JIT compiler can eliminate the remaining performance differences between Java and an optimised C++ version of a scientific computing program. This agrees with the discussion in § 6.3. Still Java gives us considerably less convoluted code.

References

Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing dynamic features in Python programs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 292–295.

Beatrice Åkerblom and Tobias Wrigstad. 2015. Measuring polymorphism in Python programs. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 114–128.

Bowen Alpern, Anthony Cocchi, Stephen Fink, David Grove, and Derek Lieber. 2001. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *OOPSLA*, Vol. 1. 108–124.

Brian Amedro, Vladimir Bodnartchouk, Denis Caromel, Christian Delbe, Fabrice Huet, and Guillermo Taboada. 2008. *Current state of Java for HPC*. Technical Report RT-0353. INRIA.

Christopher A. Atwood, Rajat P. Garg, and Dennis Deryke. 1997. A prototype computational fluid dynamics case study in Java. *Concurrency: Practice and Experience* 9, 11 (1997), 1311–1318. https://doi.org/10.1002/(SICI)1096-9128(199711)9:11<1311::AID-CPE358>3.0.CO;2-5

Ronald F. Boisvert, Jack J. Dongarra, Roldan Pozo, Karin Remington, and G. W. Stewart. 1998. Developing numerical libraries in Java. *Concurrency Practice and Experience* 10, 11-13 (1998), 1117–1129. https://doi.org/10.1002/(SICI)1096-9128(199809/11)10:11/13<1117::AID-CPE386>3.0.CO;2-I

Mark Bull, Lorna Smith, Carwyn Ball, Linday Pottage, and Robin Freeman. 2003. Benchmarking Java against C and Fortran for scientific applications. *Concurrency and Computation: Practice and Experience* 15, 35 (2003), 417–430. https://doi.org/10.1002/cpe.658

Petr Bělohlávek and Antonín Steinhauser. 2015. omp4j. http://www.omp4j.org/.

James O. Coplien. 1995. Curiously Recurring Template Patterns. *C++ Rep.* 7, 2 (1995), 24–27.

Magnus Gustafsson, James Demmel, and Sverker Holmgren. 2012a. *Numerical evaluation of the Communication-Avoiding Lanczos Algorithm*. Technical Report. Department of Information Technology, Uppsala University).

Magnus Gustafsson and Sverker Holmgren. 2010. An Implementation Framework for Solving High-Dimensional PDEs on Massively Parallel Computers. In *Proceedings the 8th European Conference on Numerical Mathematics and Advanced Applications (ENUMATH2009)*. Springer Berlin Heidelberg, 417–424. https://doi.org/10.1007/978-3-642-11795-4_44

Magnus Gustafsson, Katharina Kormann, and Sverker Holmgren. 2012b. Communication-Efficient Algorithms for Numerical Quantum Dynamics. In *Applied Parallel and Scientific Computing*, Kristján Jónasson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 368–378.

Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, Pierre America (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–38.

Malin Källén and Tobias Wrigstad. 2019. Performance of an OO Compute Kernel on the JVM: Revisiting Java as a Language for Scientific Computing Applications. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 144–156. https://doi.org/10.1145/3357390.3361026

Malin Källén, Sverker Holmgren, and Ebba Hvannberg. 2014. Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 125–134. https://doi.org/10.1109/SCAM.2014.21

Michael John Mallett. 2001. A Molecular Dynamics Computer Simulation Performance Comparison of Java Versus C. *Molecular Simulation* 26, 6 (2001), 417–422. https://doi.org/10.1080/08927020108024514

Robert C. Martin. 2003. *Agile software development: principles, patterns, and practices*. Prentice Hall, Upper Saddle River, NJ, USA.

Gregory R. Miller, Pedro W. Arduino, Jaewon Jang, and Changho Choi. 2004. Localized tensor-based solvers for interactive finite element applications using C++and Java. *Computers & Structures* 81, 7 (2004), 423–437. https://doi.org/10.1016/S0045-7949(03)00014-2

José E. Moreira, Samuel Midkiff, and Meeta S. Gupta. 1998. A comparison of Java, C/C++ and FORTRAN for numerical computing. *IEEE Antennas and Propagation Magazine* 40, 5 (1998), 102–105. https://doi.org/10.1109/74.736311

José E. Moreira, Samuel P. Midkiff, and Manish Gupta. 2000. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 2 (2000), 265–295. https://doi.org/10.1145/349214.349222

Gennadiy P. Nikishkov, Yuri G. Nikishkov, and Vladimir V. Savchenko. 2003. Comparison of C and Java performance in finite element computations. *Computers & Structures* 81, 24-25 (2003), 2401–2408. https://doi.org/10.1016/S0045-7949(03)00301-8

Bogdan Oancea, Ion Gh. Rosca, Tudorel Andrei, and Andreea Iluzia Iacob. 2011. Evaluating Java performance for linear algebra numerical computations. *Procedia Computer Science* 3 (2011), 474–478. https://doi.org/10.1016/j.procs.2010.12.080

Geoffrey Phipps. 1999. Comparing observed bug and productivity rates for Java and C++. *Software – Practice and Experience* 29, 4 (1999), 345–358. https://doi.org/10.1002/(SICI)1097-024X(19990410)29:4<345::AID-SPE238>3.0.CO;2-C

James C. Schatzman. 2001. Writing high-performance Java code that runs as fast as Fortran, C, or C++. In *Proceedings of SPIE*, Sudipto Ghosh (Ed.). 106–114. https://doi.org/10.1117/12.432994

Aamir Shafi, Bryan Carpenter, Mark Baker, and Aftab Hussain. 2009. A comparative study of Java and C performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience* 21, 15 (2009), 1882–1906. https://doi.org/10.1002/cpe.1416

Guillermo L. Taboada, Sabela Ramos, Roberto R. Expósito, Juan Touriño, and Ramón Doallo. 2013. Java in the High Performance Computing arena: Research, practice and experience. *Science of Computer Programming* 78, 5 (2013), 425–444. https://doi.org/10.1016/j.scico.2011.06.002

Oscar Vega-Gisbert, Jose E. Roman, and Jeffrey M. Squyres. 2016. Design and implementation of Java bindings in Open MPI. *Parallel Comput.* 59 (2016), 1–20. https://doi.org/10.1016/j.parco.2016.08.004

Rodrigo A. Vivanco and Nicolino J. Pizzi. 2005. Scientific computing with Java and C++: a case study using functional magnetic resonance neuroimages. *Software: Practice and Experience* 35, 3 (2005), 237–254. https://doi.org/10.1002/spe.633