Impact of Code Refactoring using Object-Oriented Methodology on a Scientific Computing Application

Malin Källén[a,*], Sverker Holmgren[a], Ebba Þóra Hvannberg[b]

[a] *Department of Information Technology, Uppsala University, Sweden*
[b] *Computer Science, University of Iceland, Iceland*

Abstract

The effect of refactoring on the quality of software has been extensively evaluated in scientific studies. We see a need to also consider its effect on performance. To this end, we have refactored the central parts of a code base developed in academia for a class of computationally demanding scientific computing problems. We made design choices based on the SOLID principles and we used object-oriented techniques in the implementation. We discuss the effect on maintainability qualitatively and also analyze it quantitatively. Not surprisingly, we find that maintainability increased as an effect of the refactoring. We also find that dynamic binding in the most frequently executed parts of the code makes the execution times increase significantly. By exploiting static polymorphism, we reduce the increase in execution times, and in some cases even get better performance than for the original code. We argue that the code version implementing static polymorphism is less maintainable than the one using dynamic polymorphism, although both versions are considerably more maintainable than the original code. Accordingly, we conclude that static polymorphism could be used to increase maintainability for performance critical code bases. Last, we argue that static polymorphism could even improve performance in some procedural code bases.

Keywords: refactoring, object-orientation, static polymorphism, performance, maintainability

1. Introduction

Computational science will not become a third pillar for scientific discovery unless sound practices of software engineering are applied. The driving factors for the interdisciplinary collaboration needed are increasing demands for high performance of scientific software, which in part is met by exploiting parallelism, and high software development productivity, e.g. met with high maintainability of long-lived software. Additionally, because of the critical consequences of scientific research, there is a need for high credibility of results[1]. Kelly[2] presents three case studies, where she analyzes the nature of changes made to academic software and the motivation for these changes. She notes

*Corresponding author

that developers and users are themselves the main advocates of change, rather than the changes being imposed by an external policy describing process and procedures, prioritization, and coding style. However, changes suggested by developers who are not skilled professionals do not necessarily result in better software. Instead, they can result in increased cognitive load for developers and the modified code can even be less maintainable than the original version.

An extensive study of how software for scientific applications is developed and used is presented by Hannay et al.[3]. They note that scientists are generally more knowledgeable in constructing software than in designing, testing and maintaining it. Since the software is developed in a setting that includes doctoral students and post doctorates, the management of development and maintenance is often unclear and may change frequently. This makes the need for readability and other maintenance aspects particularly important.

With this in mind, we argue that many scientific code bases may benefit from refactoring.

1.1. Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, improving its internal structure without changing its external behavior[4]. A refactoring process can include minor transformations such as changing names of variables, but it can also involve more advanced transformations based eg on object-oriented design principles.

Opdyke[5] uses the following definition of unchanged behavior: 'if the program is called twice (before and after a refactoring) with the same set of inputs, the resulting set of output values will be the same'. The type of program that is studied in this paper builds upon floating-point operations, and the result consists of floating-point values as well. Program transformations (or compiler flags) that do not theoretically change the result of a computation may actually have a small effect on the result of such a program. However, unless the algorithm used for solving the problem is numerically unstable, the size of this effect is in the order of machine epsilon. When solving a problem using a numerical algorithm (which is the case here), the numerical error (ie the difference between the theoretical solution of a problem and the solution produced by the numerical algorithm) will be much larger than that. Therefore, as long as the difference between the result produced by the original and the refactored version of the program is smaller than the numerical error, we have considered the behavior of the program being unchanged.

1.2. Performance of Scientific Software

The aim of refactoring is generally to improve the *maintainability*[1] of a code base. However, code refactoring might also have an effect on other attributes, such as computational performance. Scientific software is often very computationally demanding, and the time it takes to perform a certain simulation or data analysis is critical. As an example, an execution of the software that is the target of this study can take several weeks. Hence, for practical reasons and for reasons of computational costs, performance is an important aspect of these. This means that the impact on computational performance needs to be considered as an integral part of a refactoring process.

[1] A definition of maintainability is provided by ISO/IEC[6].

1.3. Maintainability vs Performance

Many studies have evaluated the effect of refactoring on different quality aspects related to maintainability. In a systematic literature review[7], al Dallal and Abdin summarizes 76 such studies, all targeting object-oriented code bases. They find that refactoring in general has a positive effect on measured maintainability, and that it reduces coupling, complexity and polymorphism. When it comes to estimated attributes, they find that refactoring in general increases flexibility, extendability and efficiency, while it decreases adaptability. However, for most studied attributes, they see contradictory results in the studies. They conclude that refactoring does not always improve all aspects of software, which they explain by the fact that while improving one aspect of the code, many refactorings at the same time degrade another. Another reason, according to the authors, can be that refactoring sometimes is done in a non-optimal way or at the wrong places in the code.

Demeyer[8] shows that replacing conditionals with polymorphic method calls does not necessarily decrease performance, but actually often results in faster code. Refactoring on a higher level is performed by Contreras et al.[9], who (using their terminology) re-engineer a mesh generation tool into a more object-oriented version. When evaluating the effect of the re-engineering they find an overall positive effect on maintainability and basically no change of performance. Tahvildari et al.[10] develop a framework for quality-driven re-engineering, which they apply on two medium-size software system. During this evaluation the apply 6 different design patterns: *Abstract Factory*, *Composite*, *Facade*, *Factory Method*, *Iterator* and *Visitor*. They find Abstract Factory and Factory Method decreasing maintainability (measured using three different maintainability indices) while the other four patterns increase the same. When studying performance, they note that the performance is improved by application of Composite, Facade, Factory Method and Visitor, while Abstract Factory and Iterator increase execution times. They also mention an earlier study where one of the systems was migrated from C to C++. This migration increased execution times considerably, but improved maintainability (evaluated using the same maintainability indices).

In 2016, Pflüger et al.[11] performed a systematic literature review on the trade-off between scalability and efficiency on one hand and maintainability and portability on the other hand. They note that:

> There is still a need for further research on practices, methods and tools to help reduce the trade-off (keep scalability/efficiency while improving maintainability/portability).

In our study, we apply a number of major refactorings on a code base developed in an academic scientific computing setting, with focus on performance. We evaluate which effect refactoring may have on code quality, using a set of internal object-oriented software metrics, and also discuss the code qualitatively. Therafter, we compare the performance of the different versions of the code. Last, we discuss how a reasonable level of maintainability can be obtained without any substantial degradation of performance, and in some cases even with a performance gain.

1.4. Statistical Methods

In the quantitative analysis, of software metric values and of execution times, we will use statistical methods, as argued for by al Dallal and Abdin[7] and by Georges et al.[12]

respectively. The aim of the usage of statistical methods is to determine whether the differences we see between the versions of the code are real differences or only random fluctuations. When the conclusion from a statistical test is that the difference is real, you say that there is a significant difference at a significance level α. This means that the probability that the difference is due to random fluctuations is at most α. The significance level is usually set to 0.05, 0.01 or 0.001.

The statistic methods used in this paper are MANOVA, t-tests and permutation tests. A general explanation of these methods are made eg by Quinn and Keough[13]. Put in the context of computer science, t-tests and MANOVA are explained by Georges et al.[12].

Concluding that there is a significant difference when there in reality is not is called a *type I error*. The risk of making a type I error is the significance level of the test. On the contrary, failing to find a significant difference that actually exists is called a *type II error*. The term *power* is used to denote the probability that a statistical method will be able to find a significant difference, if such a difference exists. In other words, the power of a statistical method is $1 - \beta$, where β is the probability of making a type II error. The power is affected by the size of the difference, the sample size, the variance of the samples and the significance level.

2. HAParaNDA

In this analysis, we have refactored a code base called *HAParaNDA*[14], which is developed in an academic setting. HAParaNDA is an iterative solver of time-dependent, high-dimensional linear partial differential equations (PDEs) on structured computational grids. In an iterative solver, values of a function are repeatedly operated on until they are sufficiently close to the solution of the equation. For a description of structured computational grids, see eg Bruaset[15].

The class of equations addressed by HAParaNDA arises in a wide span of application areas, ranging from quantum chemistry over systems biology to computational finance. Solving an equation that is of interest in the application area can take days or weeks of continuous computing. Therefore, HAParaNDA is designed for use on large-scale parallel computers and much effort has been put on achieving close-to-optimal performance. However, less consideration has been given to maintainability. In this section, we provide a brief description of the parts of HAParaNDA that we have refactored. In the remainder of this paper, we will use the term *the original code* when referring to the code base as it was before refactoring.

HAParaNDA consists of 12 000 lines of C++ code, parallelized using a hybrid parallelization model. OpenMP[16] is used for parallelization over threads within a compute node, that is a number of cores organized as one or a few CPUs, with shared memory. MPI[17] is used for parallelization over such nodes.

2.1. Spatial Domains and Function Data

When a PDE is solved numerically, the solution is represented as a *grid function* with values given only at a discrete set of grid points representing the spatial domain. The spatial domain in HAParaNDA is an orthotope ("hyper-rectangle") of arbitrary dimensionality, d. The dimensionality may be higher than 3 and a typical use case of the

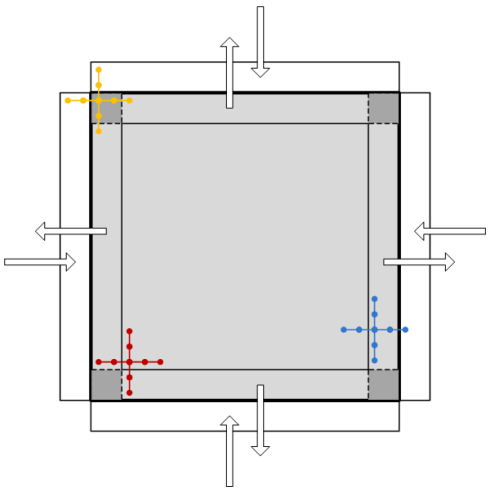

Figure 1: [Provided by M. Grandin] Communication pattern and ghost regions for the application of a finite-difference stencil in a two-dimensional grid block. The gray area represents the block, while the white rectangles represent ghost regions. The colored crosses illustrate stencils, where each dot is a weight.

code is solving six-dimensional problems. The domain is intended to be discretized using a *block-structured* grid, consisting of d-dimensional orthotopes, called *domain blocks* as described by Gustafsson et al.[18]. The idea is that the domain blocks will be stored in a tree, where the domain is the root and the successively smaller blocks form the branches. The function data is stored in the leaf blocks. Currently, the code base only provides limited support for block-structured grids with equal-sized domain blocks. In the future, support for blocks of different size, and with different ways of storing the data, will be needed.

2.2. Operators

There are two types of operators in HAParaNDA. One is *linear operators*, which can be applied on an entire grid function. The other one is *block operators*, which are applied on one block at a time. Currently, there is one type of block operator implemented, namely a *finite difference stencil*. A finite difference stencil is basically a weighted sum and is used to approximate spatial derivatives of grid functions, which is a very frequently executed operation in the PDE solver. In Figure 1, an approximative value of the derivative at the center point of a stencil is computed as a weighted sum of the function values at the points covered by the stencil. In the future, there will be a need for other block operators as well, particularly for other types of derivative approximations.

2.3. Ghost Regions

When a stencil is applied close to the boundaries of a block, data from neighboring blocks are needed. This is implemented using a *halo exchange* technique as described eg by Bruaset[15]. The data received from the neighboring blocks is stored in data buffers referred to as *ghost regions*. A block with ghost regions is illustrated in Figure 1.

3. Refactoring HAParaNDA

The parts of HAParaNDA that are refactored in this study are the ones handling the spatial domain, including ghost regions, and the block operators. The refactored code constitutes about 40% of the total code base, but contains the parts in which the main part of the execution time is spent.

3.1. Preparatory Work

3.1.1. Identification of Needs for Refactoring

As a first step in the refactoring process, we identified a number of code smells[4][19]. Three smells that has a large impact on maintainability are:

- *Divergent change* -in several classes
- *Shotgun surgery* -in several classes
- *Inappropriate intimacy* -mainly when the grid function class, which exposes several member variables publicly, is used

Thereafter, based on these smells and on the SOLID principles[20], we identified a number of refactoring actions, which we documented before starting changing the code. As an example of a refactoring action, we eliminated the occurrences of both divergent change and shotgun surgery by moving methods to classes where they belong. In some cases, this required extraction of new classes, for example the iterators described below. Our strategy for eliminating the inappropriate intimacy was mainly to hide the instance variables, in some cases after moving them to other classes. During the refactoring process, we made considerable changes in the structure of the code and we describe the resulting design in Section 3.2. We have taken performance into consideration in several ways. We describe some of them below, but we have also taken actions on a level that is too detailed for this paper.

3.1.2. Tests

For every unit of code that has been a target for the refactoring described in this paper, or that has been added or considerably changed as an effect of the refactoring, we wrote at least one test before starting changing the code. We wrote unit tests for each concrete class, with a few exceptions: When all classes used by a certain class were tested elsewhere and it was impractical to write a unit test for it, we employed an integration test instead. The results of the testing revealed several defects in the original code. Before starting the refactoring of a unit of code, we corrected all defects that we had discovered.

3.2. New Code Structure

3.2.1. Multi-Dimensional Fields

In order to apply operators (eg stencils) in HAParaNDA, one needs to iterate over the domain blocks. In the following description, we use the more general term *field* when referring to a block or to the entire domain. The dimensionality of the fields can vary from one run to another.

In the original code, two different solutions were used to implement iterations over fields. Both solutions resulted in code that was difficult to maintain, and also required that the dimensionality of the fields was known at compile-time. In the refactored code, we have specified an iterator interface, c.f. the *Iterator* pattern[21]. As we were not sure of the performance effects of defining the dimensionality at run-time, the refactored code also requires it to be known at compile time. However, we have written the code such that binding the dimensionality dynamically in the future will require a small amount of work. We have implemented iterators for both *pure fields* and *composed fields*. A pure field consists of a single field while a composed field also contains information about other fields, eg ghost regions. We wrote the pure field iterator and the composed field iterator as abstract subclasses of the iterator interface. These subclasses implement all the iterator operations. In the implementation of the composed field iterator, we used the *Decorator* pattern[21]: The composed iterator decorates the iterator for a pure field with iterators for the other fields.

The pure field iterators use strategy classes for stepping through the field and for accessing elements, c.f. the *Strategy* pattern[21]. We applied the pattern *Factory Method*[21] for creating the strategies in the pure field realizations and for creating the iterators needed in the composed field iterator realizations.

In order to get dimensionality independent code, internally, the data of each block is stored in a one dimensional data structure. From the view of the user of the iterators, the data is stored in an d-dimensional data structure, where d is the dimensionality of the data fields. Conversion between the d-dimensional and the one-dimensional indices requires one integer division and one modulo operation. This conversion is not done in the original code. In an earlier version of this paper[22], we implemented the conversion using integer divisions and modulo operations. However, an integer division (and thereby also a modulo operation) takes considerably more CPU cycles than other integer operations. We think that the use of these operations is the main reason for the difference of the performance of the original and the static code in the earlier version of the paper. Fortunately, the denominator is constant over the life span of an iterator. Therefore, we make use of *magic numbers*, as described by Warren[23] and thereby replace the modulo and the integer division with less time consuming operations.

During an application of a block operator, a large number of calls to dynamically bound methods in the iterators, and the corresponding strategy classes, are needed. This means that there is a big potential performance impact of this solution. In order to assess the performance effect, we created an alternative version of the code, where we applied *static polymorphism* on these classes using the *Curiously Recurring Template Pattern* (CRTP)[24]. This solution has also been used eg for iterators in Boost[25]. We implemented CRTP as described by Lutz et al.[26].

In the following, we will refer to the refactored code version that applies dynamic polymorphism as *the dynamic code*, while we will use the term *the static code* when referring to the alternative version. When we write *the refactored code*, we mean both the dynamic code and the static code.

We also introduced an interface `Iterable` which guarantees that iterators for the inner regions and for the boundaries can be retrieved for a data structure. All classes in which elements need to be iterated over implement this interface. However, for performance reasons, when domain block structures need to be iterated over, we integrated the iterators into the classes that hold the blocks.

3.2.2. Blocks

We implemented the tree structure described in Section 2.1 using the *Composite* pattern[21]. In order to make it easy to extend the code base with different numerical methods, requiring different ways to represent the data, we made the leaf class abstract. This class keeps track of the grid function indices stored in the block that it represents. We also introduced an additional type of block, computational blocks, to provide a temporary container for grid function values from a block that is being used in the computations. We expect these blocks to facilitate future implementation of different numerical methods. We represent this type of block by an abstract class, and realize it with pure and composed blocks respectively. The former realization can be used eg for the output from a block operator, as there is no need to use ghost regions for output data. Composed computational blocks, on the other hand, are suitable for input data of the block operators. Following the *Dependency Inversion Principle* (DIP)[20], we used an *Abstract Factory*[21] for creating the different types of blocks.

3.2.3. Grid Functions

In the original code, the grid function class keeps both function data and detailed information about the structure of the grid on which the function resides. In the refactored code, instead, we use domain blocks to store the information about the grid. The grid function class contains an iterator over the domain blocks. For performance reasons, the iterator differs from the ones described above in the sense that instead of returning a block, the `current` method takes a computational block as its argument and initializes it.

We use the Strategy pattern for initialization of the grid functions. The abstract base class employs the pattern *Template Method*[21] and delegates computation of the actual function values to its subclasses. The refactored code also provides the possibility to initialize a grid function using a composition of functions that depend on one variable each.

3.2.4. Block Operators

From the finite difference stencil in the original code, we extracted two levels of abstract base classes, one that represents block operators in general, and one that represents multuncial[2] stencils. Naturally, the block operator class is the base class of the multuncial stencil class. We implemented template methods on both levels. The concrete subclass, which represents a multuncial stencil with constant weights, uses a Strategy pattern to initialize its weights.

Like the iterators, the block operators contain polymorphic methods that are repeatedly called in the most frequently executed part of HAParaNDA. Consequently, in the static code we also applied static polymorphism in the block operator class hierarchy.

4. Threats to Validity

This paper presents a case study, involving one code base written in one language by only a few developers, and the conclusions drawn below are not necessarily generalizable

[2] "*Multuncial*" is an adjective describing an object with the shape of a *multunx*, which in turn is our dimensionality-independent generalization of the term *quincunx*. A quincunx is the pattern of five points organized as a cross in the same way as the five-spot on a six-sided dice.

to other frameworks. However, we think that the execution pattern of our program is typical for scientific computing applications, and researchers developing this type of applications could potentially benefit from the results. Notwithstanding, more studies are needed before any general conclusions can be drawn.

Another threat to validity is the set of metrics chosen, which might not always accurately reflect the maintainability of the evaluated code. To mitigate this threat, we complete the quantitative analysis with a qualitative discussion. However, we consider the metrics in this discussion, in order to reduce the impact of our subjective perception of the code.

Our study evaluates the effect of refactoring. However, we have made considerable changes to the code and there is a risk that we unintentionally have introduced changes of the behavior of the code. To mitigate this threat, we have written unit and/or integration tests (in a test-driven fashion) for all classes affected by the refactoring. This increases the probability of us noticing if the behavior was changed by the refactoring, and also forced us to familiarize more with the framework before applying any changes.

Last, our performance analysis covers only two dimensionalities and two domain sizes. We have compiled the code with one single compiler on one computer cluster. Of course, this can have an impact of the performance. However, we have made an effort to make the two program setups as different as possible, by choosing one low and one high dimensionality (the highest possible for the original HAParaNDA). We also, intentionally, chose one domain size that is divisible by the number of threads used and one that is not. However, due to the complexity of computer systems, there may also be seemingly innocuous factors that affect performance, as argued for and exemplified by Mytkowicz et al.[27].

5. Qualitative Evaluation of Maintainability

We find that the dynamic code adheres well to all five SOLID principles and that the static code adheres well to four of them. As the classes in the refactored code have limited and well-defined responsibilities, we consider the design to agree well with the *Single Responsibility Principle* (SRP)[20]. The two main differences compared to the original code are:

- Thanks to the iterators, the index handling is performed separately from the operations on the field elements. An effect of this is that the readability has increased, as the index operations are not hiding the actual computations on the field elements.

- We have separated the blocks from the grid function class instead of mixing grid structure with function data.

Our efforts to follow SRP have made all interfaces small. Consequently, with the exception of the interface `Iterable`, we have not seen any need to split the interfaces further in order to follow the *Interface Segregation Principle* (ISP)[20].

We have introduced several abstract base classes, which implement behavior that we expect to be common for all derived classes. An example is the use of Template Methods in the grid function initiators and in the block operator classes. The pattern makes it possible to add new subclasses, which alter the current behavior, with a small coding

effort and without changing the current classes. This agrees well with the *Open-Closed Principle* (OCP)[20].

Moreover, we made use of the Strategy pattern in the iterators, grid functions and block operator classes. New strategies can easily be added, and the strategies can be reused in different classes. This solution limits the efforts needed to add new iterators, grid functions or block operators. All in all, the strategies contribute to a better adherence to OCP. Before refactoring, adding functionality to the code was time-consuming and error-prone, and we are of the opinion that we have made considerable improvements on this point.

In the refactored code, we cannot find any subclass that is not substitutable for its base class, and therefore conclude that all classes follow the *Liskov Substitution Principle* (LSP)[20].

Thanks to the usages of the patterns Abstract Factory and Factory Method, we have been able to keep the number of dependencies to concrete classes low in the dynamic code, which agrees with DIP. However, at this point, CRTP comes at a price. In all places where a class that implements static polymorphism is used, the concrete type of this class must be known. In other words, we break DIP in the static code. This has a considerable effect on the changeability, and most of the smoothness brought by the dynamic binding is lost.

There are also other maintainability issues of using CRTP. The fact that the pattern does not allow virtual methods makes it harder to add a class in a hierarchy where CRTP is applied, since it is not possible to see from the interface which methods can, or must, be overridden. This is a severe limitation. It also brings the consequence that it is not possible to implement hooks, ie methods implementing default behavior, which can be overridden when needed, in the base class. In addition, the fact that virtual inheritance cannot be used at the same time as CRTP complicates our iterator class hierarchy.

In addition to better following the SOLID principles, we argue that we have improved data hiding by being able to hide several of the member variables that are public in the original code.

Finally, we note that the refactoring is likely to having improved the load balancing of the code. In the original code, a number of nested loops were used when a field was iterated over. The parallelization, ie partitioning of loop indices using OpenMP, was done only at the outermost loop level. On the contrary, as mentioned in Section 3.2.1, the iterators in the refactored code view each field as a one-dimensional data structure. This data structure can more easily be partitioned in equal chunks when the code is parallelized.

To conclude, our qualitative evaluation of the refactored code has convinced us that the maintainability has improved as an effect of the refactoring. However, we also find that there is a considerable risk of the refactoring having degraded the performance of the code. In the following two sections, we perform quantitative evaluations of the maintainability and of the computational performance.

6. Quantitative Evaluation of Maintainability

6.1. Background

A widely used object-oriented metrics suit is presented by Chidamber and Kemerer[28]. Many alternative internal metrics have been developed and used in other studies, eg fan-

in and fan-out[29], CAMC[30] and SCC[31]. Both code cohesion and code coupling are included in the suite presented by Chidamber and Kemerer[28], and these metrics are further discussed eg by Moghadam and Cinneide[32] and by Akiyama et al.[33]. Hopkins[34] collects software metrics such as cyclomatic complexity and number of knots for some numerical libraries, but to our knowledge, no earlier efforts to collect object-oriented software metrics for scientific computing software are made.

Using internal design metrics is not the only way to measure maintainability. Other metrics are related to the stability of the software, ie the number of changes done, or the number of bugs. However, we find internal metrics being most relevant to apply in this study, as the number of changes done since the refactoring is small. Moreover, external metrics such as the number of changes done may be misleading for HAParaNDA, since the current main developer has different routines for changing and checking in code than her predecessor. We have chosen to use the above-mentioned Chidamber and Kemerer suite to evaluate the refactoring, as the metrics are well motivated, and related to maintainability. Moreover, the fact that the suite is commonly used makes comparisons to other studies easier. We have also collected two additional metrics which we find relevant, namely number of *Lines Of Code* (LOC)[35] and *Average Method Size* (AMS)[35] for each class. These metrics can give an indication of the ease of understanding each class or method.

6.2. Data Collection

We collected the metrics values for the original code and the dynamic code using the tool *Understand*[35], Build 731. Unfortunately, we have not been able to make Understand identify the classes on which we have applied CRTP as classes. This means that we have not been able to collect the metrics values for the static code. Instead we resort to a qualitative discussion about the expected effect of CRTP on the metrics values at the end of this section. In cases where Understand lacks support for collecting a separate metric, we complemented the results by measurements that we performed by direct inspection of the code.

We measured at class or class template level for both the original and the dynamic code and treated each class template as one class. Likewise, we treated structs and struct templates as classes. For simplicity, we hereinafter use the term 'class' when referring to a class as well as to a struct, a struct template or a class template. In the measurements, we included all classes that have been added, removed or considerably affected by the refactoring process, but we excluded test code. We also noted the number of classes analyzed.

Parts of the original code consist of functions implemented outside classes. In order to be able to compare the different versions of the code, we treated each file that consists of such functions as a class. Our motivation for this is that these files can be seen as a manifestation of how earlier HAParaNDA developers structured the code. If these developers would have used a language that forces use of classes, they would probably have divided the code into classes in the same way as they have now divided it into files.

One class template in the original code lacked some includes and a declaration of a local variable. Here, we added the missing includes and declaration in order to make Understand able to analyze the template.

It should be noted that one of the classes (`Potential`) that we included in the measurements has not been an explicit target for refactoring, but it has nevertheless been

considerably modified: As an effect of refactoring of other parts of the code, about 70% of this particular class has been moved to other classes.

We measured LOC, AMS, *Depth of Inheritance Tree* (DIT)[28] and *Number of Children* (NOC)[28] using the Understand metrics `CountLineCode`, `AvgLineCode`, `MaxInheritanceTree` and `CountClassDerived` respectively for each class. Template parameter lists, method heads and terminating '}' are included in in `AvgLineCode`. For practical reasons, we used the metric `PercentLackOfCohesion` in Understand for collecting the *Lack of Cohesion in Methods* (LCOM)[28] values. However, the definition is different from the one proposed by Chidamber and Kemerer. The *Coupling Between Object classes* (CBO) metric[28] includes both afferent (incoming) and efferent (outgoing) coupling. We used the Understand metric `CountClassCoupled` to collect the efferent coupling for each class, and manually inspected the code in order to collect the afferent coupling values. To calculate the *Weighted Methods per Class* (WMC)[28] values, we used the metric `SumCyclomaticComplexity`. This corresponds to using the cyclomatic complexity as the weight. In order to collect the *Response For a Class* (RFC)[28] values, first, we used the metric `CountDeclMethodAll` in Understand to count the number of methods in each class, including the inherited ones. Thereafter, we inspected the code manually in order to find the number of external methods called directly from each class and its base classes, and added these values to the ones from the first step. Following Chidamber and Kemerer[28], we defined an external method to be a method or macro that is declared outside the class and is part of the framework. We included explicit constructor and destructor calls, but no implicit ones.

Thereafter, we ran a MANOVA on the collected metric values, using the `R` function `manova`. Since the metrics are measured on different scales, we cannot compare their relative importance using the raw data. Therefore, we standardized the data, that is transformed it so that the mean and standard deviation of each metric is 0 and 1 respectively. Studying the result of the MANOVA run on the standardized data, we found that the residuals were right skewed. Hence, we transformed the data once more, using the formula

$$z_t = ln(z + 2) \tag{1}$$

where z are the standardized metric values and analyzed z_t instead. The reason for us adding the constant 2 is to avoid computing the logarithm of non-positive values. To extract the result of `manova`, we called the `R` functions `summary` and `summary.aov`.

6.3. Results

The number of classes for which we collected metrics values is 25 in the original code and 48 in the dynamic one. The MANOVA shows that the change of the metric values is significant at the 0.001 level ($F = 10.4850$, $p = 2.941e - 09$). We present the F and p values of each metric separately, and mean values of each metric, in Table 1. We also provide histograms for the metric values before and after refactoring, see Figures 2-9.

Figures 2 and 3 show that a larger portion of the classes have lower LOC and AMS values after refactoring. However, as can be seen in Table 1, the change of LOC is not significant (on any level). For AMS, on the other hand, we see a decrease that is significant on the 0.01 level.

As can be seen both in Table 1 and Figures 4 and 5, the values for DIT and NOC are close to 0 in the original code. The mean of both metrics have increased considerably, and the difference is significant at the 0.001 and 0.05 level respectively.

Table 1: Metrics: Average value per class before and after refactoring, and statistic values.

Metric	Original code	Dynamic code	F	p	
LOC	105.96	61.20	2.3496	0.1298	
AMS	14.41	6.88	8.5732	0.004582	**
DIT	0.04	0.90	43.551	6.320e-09	***
NOC	0.08	0.83	6.6477	0.01200	*
LCOM	21.55	31.74	2.7470	0.1018	
CBO	4.76	4.44	0.2263	0.6358	
WMC	17.88	10.44	2.3163	0.1325	
RFC	12.80	23.81	11.1620	0.001334	**

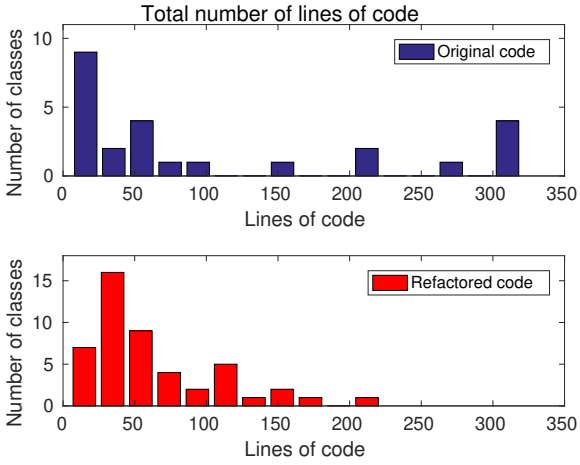

Figure 2: Number of lines of code (LOC) before and after refactoring.

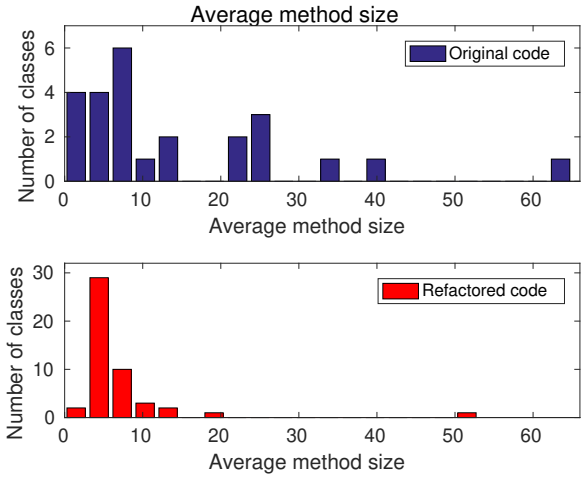

Figure 3: Average Method Size (AMS) before and after refactoring.

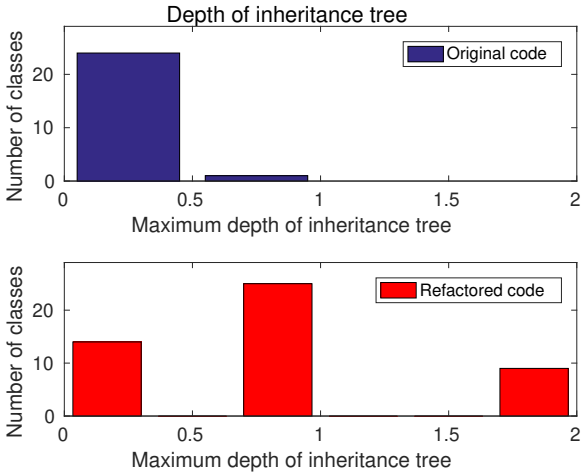

Figure 4: Depth of Inheritance Tree (DIT) before and after refactoring.

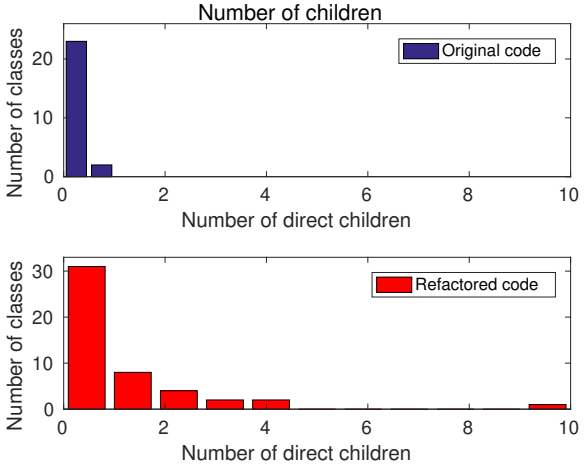

Figure 5: Number of Children (NOC) before and after refactoring.

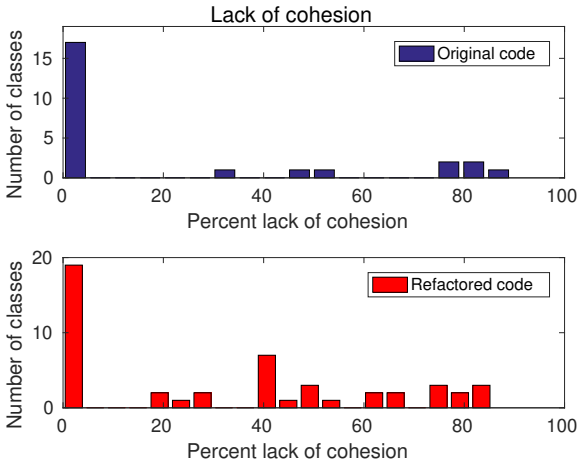

Figure 6: Lack of Cohesion in Methods (LCOM) before and after refactoring.

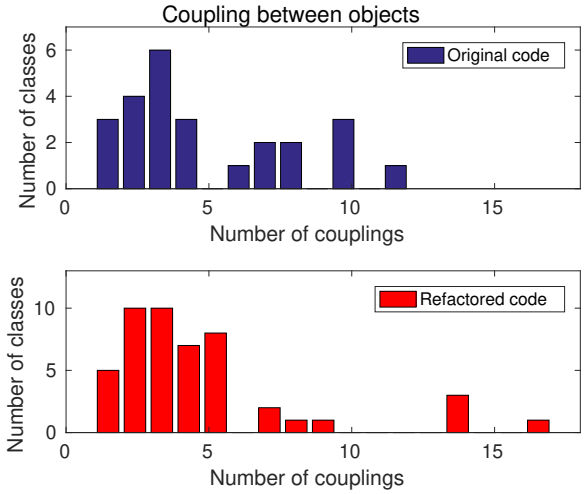

Figure 7: Coupling Between Object classes (CBO) before and after refactoring.

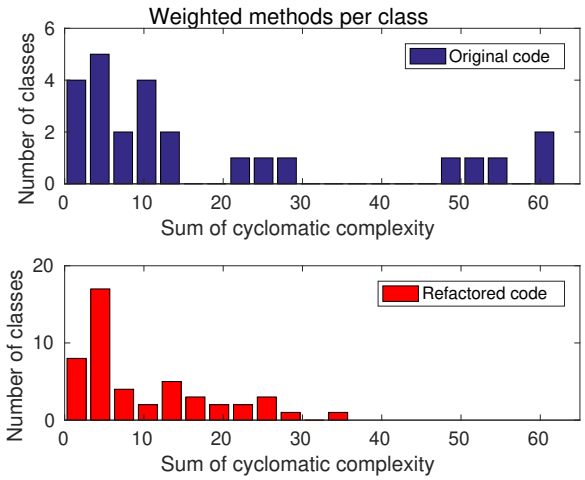

Figure 8: Weighted Methods per Class (WMC) before and after refactoring.

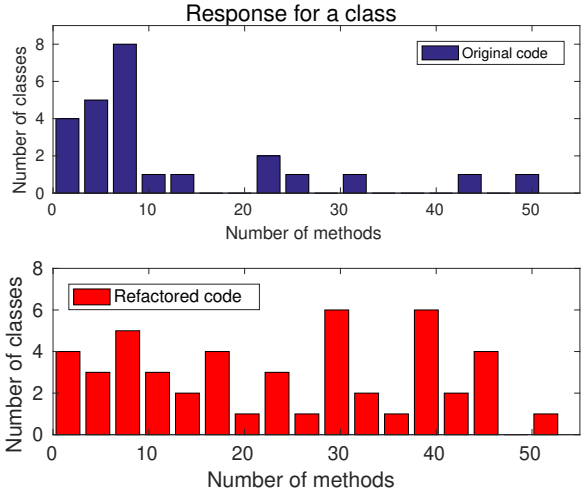

Figure 9: Response for a Class (RFC) before and after refactoring.

We find no significant changes of LCOM, CBO and WMC, see Table 1. Likewise, Figure 6 shows no obvious difference in the LCOM values before and after refactoring. However, from Figures 7 and 8, we see that after refactoring, a larger portion of the classes have small values of CBO and LCOM.

Last, we see from Table 1 that the refactoring led to an increase of RFC that is significant at the 0.01 level. This agrees with Figure 9, in which we see that most classes had a low RFC value before refactoring, while the RFC of the refactored code is higher for a larger portion of the classes.

6.4. Discussion

The fact that the MANOVA shows a highly significant difference between the two versions of the code means that the refactoring actually has changed the metric values (as a group). Below, we discuss the significance of each metric separately.

6.4.1. DIT and NOC

The low DIT and NOC values of the original code show that basically no inheritance or polymorphism was used before the start of refactoring. We argue that the significant increases of these values show that the code has gone from procedural to object-oriented. We further support this statement by noting that the portion of instance variables that are public has decreased from 44% to 6% in the refactored parts of the code, which indicates that encapsulation is used to a higher extent.

The higher value of NOC contradicts the results of al Dallal and Abdin[7], that refactoring decreases polymorphism. However, their study covers refactoring of object-oriented code bases. It seems like most of the primary studies discussed were object-oriented before the start of refactoring. They might have had too complex inheritance hierarchies, which were simplified by the refactoring. On the contrary, we have refactored a procedural code and turned into a more object-oriented version. Therefore it is natural that the amount of polymorphism increased in our case.

The increase in the values of DIT and NOC indicate that we make use of the possibilities of reuse brought by object-orientation better in the refactored versions of the code[28]. As discussed in Section 5, this facilitates extending the code. Despite the large increase, we consider DIT being low also for the dynamic code and suppose that this is an effect of us favoring object composition over inheritance in the refactoring as argued for by Gamma et al.[21]. A high value of DIT would have indicated a complex design[28].

6.4.2. AMS and RFC

Short methods increase readability of a code base (as long as the methods are defined in an adequate way)[19][4], which in turn facilitates changing the code. Therefore, we find the decrease of the AMS values being a sign of increased maintainability of HAParaNDA, which agrees with the qualitative discussion in Section 5. In Figure 3, it can be seen that one single class still has a high AMS value after refactoring. This is the class `Potential`, which has not been an explicit target of the refactoring (as mentioned in Section 6.2). Hence, we are convinced that this class will obtain a lower AMS in the future if the refactoring process is continued.

However, the decrease of the average method size comes at the price of a larger number of methods, which is a reason for the significant increase of RFC. This is in accordance

with the conclusions of al Dallal and Abdin[7]: while improving the value of one metric, many refactorings may at the same time degrade other. A higher RFC value may be a sign of the class being more complicated to test and debug[28]. However, we don't think that is the case here, since the amount of functionality hasn't changed, but only become divided between a larger number of methods. In accordance with Martin[19], we find that a large number of small methods makes the code more readable than a smaller number of large methods. Hence, we find that the changes that lead to decreased AMS values and an increased RFC values have a positive total effect on readability.

6.4.3. LOC and WMC

Although the mean values of LOC and WMC have not changed significantly, we argue that the distributions of these values have improved. Before refactoring, some classes had considerably higher LOC and WMC values than the majority of the classes. This indicates that these classes were more complex and harder to maintain[28]. A higher WMC also indicates that the classes were harder to test, and that they were more likely to be application specific, which limits the possibility of reuse[28]. The refactoring has decreased the metric values for these classes, indicating that the classes have become more maintainable.

A reason for the lack of significant decrease of these two metrics is that some classes of the original code had extremely low values. For example, four of the classes had LOC of at most 7 and two classes had a WMC value of 0. This keeps the average values of the metrics down. However, we don't think that these values indicate a high maintainability. On the contrary, these classes contribute to the complexity of the code without adding any functionality to the framework.

In summary, the fact that LOC and WMC have not decreased significantly does not necessarily mean that the complexity of an average class has not decreased. Actually, as discussed in Section 5, we believe that the refactored code adheres better to SRP, which makes the classes less complex.

6.4.4. LCOM

The better adherence to SRP should also cause the LCOM values to decrease, which it has not in this study. We argue that this may be an effect of the `PercentLackOfCohesion` being measured in a deceptive way. Firstly, as the metric considers the portion of methods using a certain variable, the usage of helper methods increases the lack of cohesion. However, this type of methods can increase the readability of the code to a large extent[4][19]. Secondly, abstract methods are included in the metric, although they naturally never use any member variable. Therefore, the introduction of abstract base classes, which we claim increase maintainability, causes an increase in the LCOM values. Finally, the metric is defined such that its value is zero for classes without member variables, and for classes without methods. As a majority of the classes in the original code lack member variables, we find that comparing the values before and after refactoring is misleading.

We see a need for more appropriate ways to measure the cohesion and indeed, considerable research has been carried out since the LCOM metrics was presented by Chidamber and Kemerer[28]. Notably, cohesion metrics are still being developed[36]. Comparing different cohesion metrics by applying them on a scientific computing application could be a direction of further research.

6.4.5. CBO

A low coupling is an indication of a modular design, which is important for maintainability[28]. During the refactoring, we have not managed to decrease the average coupling of a class. However, we see from Figure 7 that a larger portion of the classes has a relatively low coupling after the refactoring, while a few classes with high coupling keep the average value close to that in the original code. This indicates that fewer classes suffer from the rigidity caused by a high number of couplings.

Moreover, we expect the average value of the coupling of the classes covered by this refactoring to decrease even more if we continue the refactoring process. We motivate this by noting that one of the classes that has a high number of couplings is `Potential`. Another class has lots of incoming couplings from parts of the code that are not yet refactored.

6.4.6. Effects of Static Polymorphism

As mentioned in Section 5, CRTP prevents usage of virtual inheritance. This forced us to remove one class from the iterator hierarchy and merge its behavior (basically one method) into other iterator classes. As a consequence of the removal of the class, we expect small decreases in the NOC and DIT in the static code, compared to the dynamic code. We also expect a small increase in the average RFC and LCOM value as an effect of integrating the behavior from the removed iterator into other classes. This increase reflects a real, albeit very small, reduction of the adherence to SRP.

When implementing CRTP in a class hierarchy, we were forced to implement some methods, which are abstract in the dynamic code, in the base class. As an effect of this, and the fact that the pattern forces us to keep track of the concrete type of some classes, we expect the LOC and AMS metrics values to be larger than those of the dynamic code. However, we expect the increase to be small, as the number of new methods is small compared to the total number.

The basic idea of the CRTP pattern is to let the concrete type of a class be known by every class that uses it, at the same time as the "abstract" type must be known. In many cases, the concrete type is provided as a template parameter and should not be considered increasing the coupling. However, in the static code we must consider the concrete type in classes that are totally ignorant of the concrete type in the dynamic code. Therefore, we expect the application of CRTP to significantly increase the CBO value. Coupling is an important metric and the increase is a sign of decreased maintainability. This is in accordance with the qualitative evaluation in Section 5, where we concluded that CRTP decreases maintainability.

6.4.7. General notes about metrics

Our qualitative evaluation concludes that the refactored code adheres well to the SOLID principles and is more maintainable than the original one. However, these improvements are not reflected by all the metrics. This suggests that when interpreting internal metrics values, some care must be taken and the quantitative analysis must be combined with a qualitative discussion. Also note that there are maintainability aspects that none of the internal metrics would capture even in the ideal case. An example is variable and method names, which are essential for the readability of the code[4][19].

Finally, we want to raise the question whether there are internal metrics that can also give some guidance when it comes to performance. If so, internal metrics could

give designers indications about both aspects covered in this paper: maintainability and performance. As the main culprit for performance loss in our application is dynamic binding, the amount of dynamic polymorphism may indicate performance bottle necks. The metrics DIT and NOC are connected to polymorphism but a complementary addition of metrics would be needed in order to reflect the amount of dynamic binding. This could be a direction of future research.

7. Performance

7.1. Data Collection

To assess the performance impact of the refactoring, we performed a number of runs of the different versions of the code and reported the elapsed times. We used a test code that mimics the typical use of the refactored part of HAParaNDA: A finite difference stencil is repeatedly applied to a block of a grid function. We ran a test program using two-dimensional as well as six-dimensional blocks and stencils. We applied the stencil on domains whose size we chose such that the data filled up a little bit more than 50% of the memory available on each node, that is 46340^2 and 33^6 elements respectively. Note that this is the number of elements operated on *in each node*; when using more than one node, we increased the domain size accordingly. For each application, the output of the previous one was used as input, which mimics the behavior of an iterative solver (see Section 2).

We expect future users of HAParaNDA to make tens of thousands of stencil applications during an execution. The time for set-up and tear down is comparable to just a few stencil applications, which is negligible in that context. Consequently, we did not include set-up and tear down in the time measurements. For each experimental setup, we chose the number of stencil applications such that the time for computations and communication was at least 1 minute for all three versions of the code.

We executed the test code on a computing cluster that comprises 9720 computational cores in the form of 486 dual CPU (Intel Xeon E5 2630 v4) nodes with 128 GB - 1 TB RAM per node. The nodes are interconnected with Infiniband FDR. The cluster runs CentOS Linux. More information about the system is available online[37]. We ran all experiments on the 128 GB nodes, and reserved full nodes for all runs.

We compiled the code using gcc version 8.1.0 with the flags `-fPIC -O3 -DNDEBUG`. When compiling and running the parallel code, we used Open MPI version 3.1.0. We we used the flag `-bind-to none` for `mpirun`. Below, we describe the different test cases that we evaluated. A summary of the cases can be found in Table 2

First, we set up a serial experiment in order to compare the computational performance without the overhead of synchronization and communication. For this experiment, we included neither OpenMP nor MPI when compiling, and run the executables on a single core. We applied the stencil twice in each run. When choosing the number of runs, we aimed at a power of at least 99% for pairwise t-tests on a significance level of 0.001. In preliminary studies, we saw that the performance difference between the dynamic code and each of the other versions of the code was larger than that between the original and the static code. Hence, we used the preliminary execution times for the latter two when estimating the number of runs needed. The analysis indicated that we should run each version of the program 38 times, and so we did.

Table 2: Summary of the experimental setups for the performance evaluation

	Serial	**Threaded**	**Parallel**	**No inlining**
OpenMP	No	Yes	Yes	No
MPI	No	No	Yes	No
Number of nodes	1	1	16	1
Cores per node	1	20	20	1
Number of runs	38	34	31	69
Stencil applications/run	2	8	6	1
Number of elements in 2D	46340^2	46340^2	$16 \cdot 46340^2$	46340^2
Number of elements in 6D	33^6	33^6	$16 \cdot 33^6$	33^6

Next, we analyzed a threaded case. Here, we compiled HAParaNDA with OpenMP, but without MPI and run the resulting binaries on all 20 cores of one node. This is a more realistic use case scenario than using only one core, but still lets us study the performance without the impact of inter-node communication. When choosing the number of program runs, we used the same approach as for the serial experiments, and came to the conclusion that each program version needed to be run 34 times. In each program run, the stencil was applied 8 times.

Third, we evaluated the parallel performance using the programs compiled with MPI. We used 16 cluster nodes and employed hybrid (inter-node and intra-node) parallelization. In order to make use of the full nodes, again, we used all the 20 cores of each node. This is how we expect future users of HAParaNDA to use the framework. Here, since the communication times are larger than for the previous experiment, we applied the stencil 6 times only. We ran each program version 31 times. We choose this number the same way as we did for the serial and the threaded experiments.

For each program run, we measured the total time spent on computations and communication (hereinafter referred to as the execution time). Our method for measuring the time spent in a given section of the program was to record the current time stamp at the beginning and end of that section and compute the difference between the two time stamps. After having measured the execution times, we created box plots showing the execution times for each experimental setup. As suggested by Georges et al.[12], we computed the mean of the execution times for each program version (original, dynamic and static), parallelization level (serial, threaded and parallel) and dimensionality (2 and 6). For each parallelization level and dimensionality, we made pairwise t-tests of the three versions.

Unfortunately, the preconditions of t-tests are not fulfilled -the distributions of execution times are not symmetrically distributed, and we have not found any transformation that fixes this. Therefore, after each t-test, we made a permutation test, resampling 1000000 times.

Finally, in order to asses the effect of inlining, we compiled the two refactored versions of HAParaNDA with inlining disabled. Since, in preliminary experiments, the performance difference seemed to be largest for the serial code, we did not use OpenMP or MPI this time. In addition to `-fno-inline`, we used the same compiler flags as above. Since the disabling of inlining slows down the code considerably, we only applied the stencil once in each program run. We used the same domain sizes as in the other experiments,

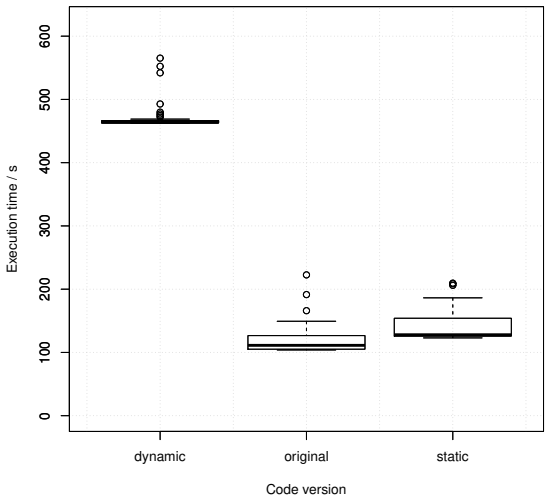

Figure 10: Execution times for the serial versions of HAParaNDA, in 2D

and aiming at a power of 99% for a t-test of the two versions, we executed each program 69 times.

Again, we computed the mean of the execution times, and made a t-test. Just as for the other experiments, we also performed a permutation test, since the execution times turned out to be highly right-skewed.

We used the `R` function `t.test` to perform all t-tests mentioned above.

7.2. Results

Figures 10-17 show box plots of the execution times of HAParaNDA for the different experimental setups. In the cases when all three versions of HAParaNDA are compared (Figures 10-15), it can be seen that the difference between the original and the static code is relatively small, while the dynamic code is considerably slower. However, when the code is compiled without inlining, the difference between the two refactored versions appears to be small.

In Table 3, we present the mean execution times for the dynamic and the original code, and the corresponding statistics. As expected, and shown in the box plots, the dynamic code is slower than the original code for all experimental setups. The difference is significant at the 0.001 level in all cases -the p values for the t-test as well as for the randomization tests are even below machine epsilon.

Table 4 shows the execution times and the corresponding statistics for the static and the original code. We see that the serial runs of the static code are slower than those of the original code. However, when it comes to the threaded and parallel runs, the refactored code is faster than the original one. This effect can also be seen in the box plots in the beginning of this section. The result is significant at the 0.01 level for the two-dimensional serial experiment, and at the 0.001 level for all other experimental setups.

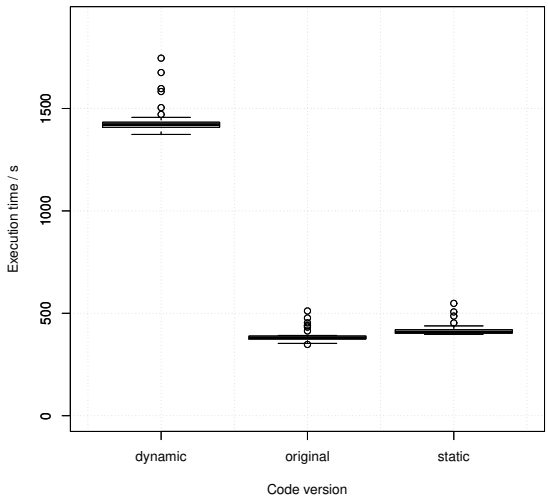

Figure 11: Execution times for the serial versions of HAParaNDA, in 6D

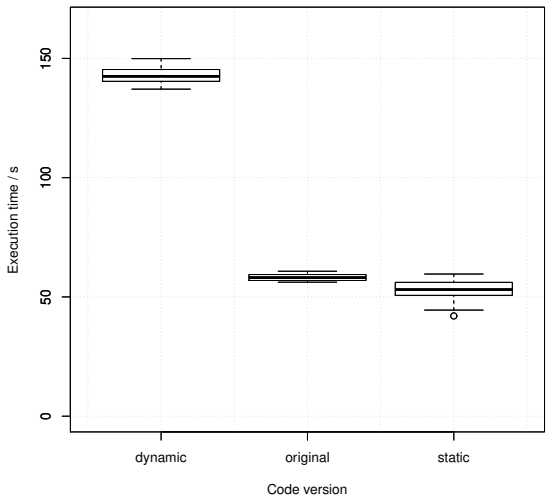

Figure 12: Execution times for the threaded versions of HAParaNDA, in 2D

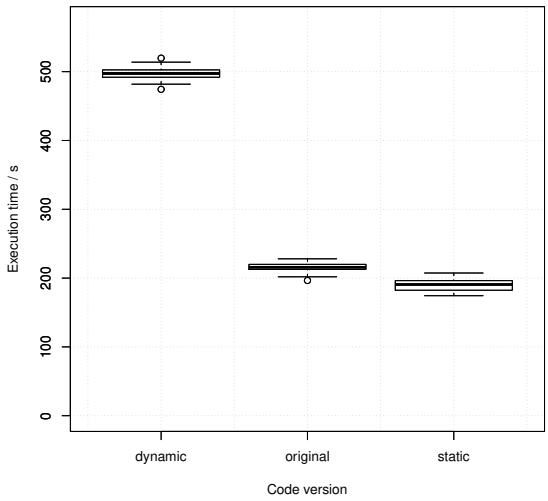

Figure 13: Execution times for the threaded versions of HAParaNDA, in 6D

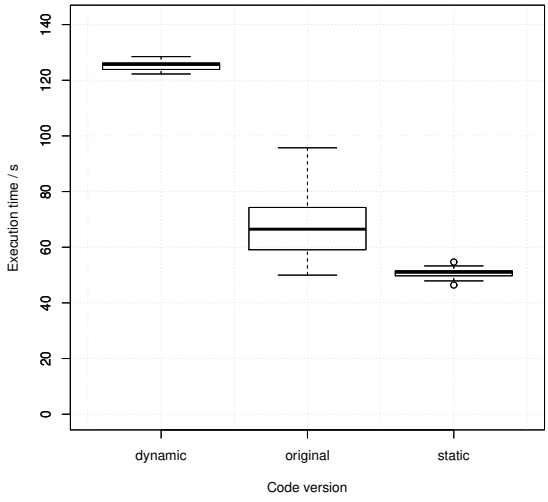

Figure 14: Execution times for the parallel versions of HAParaNDA, in 2D

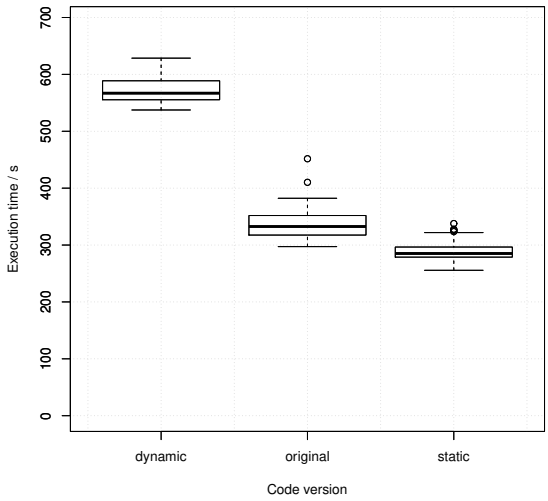

Figure 15: Execution times for the parallel versions of HAParaNDA, in 6D

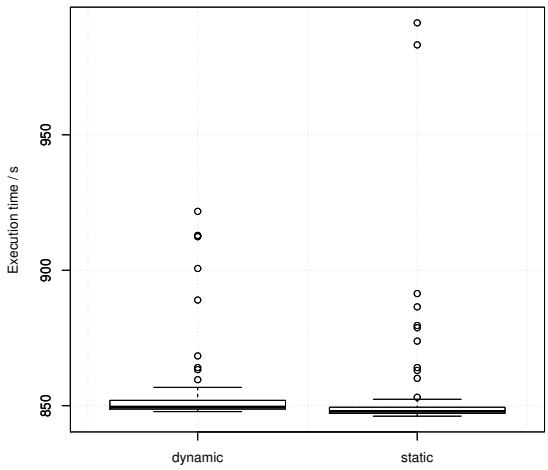

Figure 16: Execution times for the HAParaNDA versions with inlining disabled, in 2D

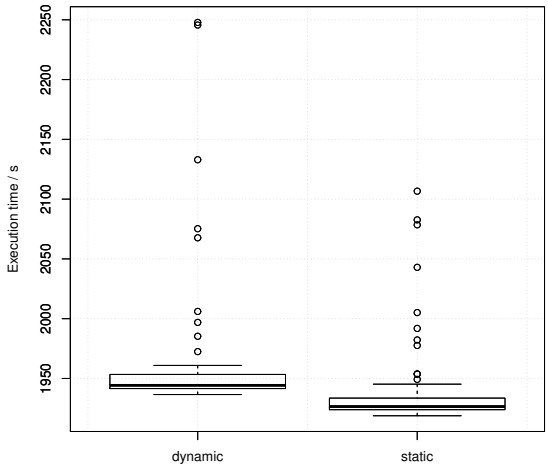

Figure 17: Execution times for the HAParaNDA versions with inlining disabled, in 6D

Table 3: Execution times and statistics comparing the dynamic and the original code. 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**				
version	**dim**	**dynamic**	**original**	**t**	p_t		p_p	
serial	2	472.91	121.51	60.8362	7.085e-65	***	0.000	***
serial	6	1443.67	390.75	75.9353	2.549e-53	***	0.000	***
threaded	2	143.23	58.19	124.5379	1.984e-55	***	0.000	***
threaded	6	497.02	215.28	143.3073	5.408e-76	***	0.000	***
parallel	2	125.18	67.15	28.5134	3.948e-24	***	0.000	***
parallel	6	571.18	338.81	31.9261	1.326e-36	***	0.000	***

Table 4: Execution times and statistics comparing the static and the original code. 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**				
version	**dim**	**static**	**original**	**t**	p_t		p_p	
serial	2	141.37	121.51	3.3791	1.163e-03	**	8.500e-04	***
serial	6	419.27	390.75	3.8181	2.779e-04	***	1.116e-04	***
threaded	2	52.72	58.19	-7.5750	2.634e-09	***	0.000	***
threaded	6	189.87	215.28	-13.4957	6.953e-20	***	0.000	***
parallel	2	50.80	67.15	-8.0580	3.931e-09	***	0.000	***
parallel	6	290.0	338.81	-6.9936	6.457e-09	***	0.000	***

Table 5: Execution times and statistics comparing the dynamic and the static code. 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**				
version	**dim**	**dynamic**	**static**	**t**	p_t		p_p	
serial	2	472.91	141.37	57.4500	4.417e-63	***	0.000	***
serial	6	1443.67	419.27	74.5193	8.541e-52	***	0.000	***
threaded	2	143.23	52.72	96.7421	1.288e-72	***	0.000	***
threaded	6	497.02	189.87	138.0745	1.047e-82	***	0.000	***
parallel	2	125.18	50.80	167.1297	4.761e-81	***	0.000	***
parallel	6	571.18	290.02	50.8743	2.438e-50	***	0.000	***
noinline	2	854.86	855.11	-0.0702	0.9441		0.9458	
noinline	6	1962.77	1940.51	2.6217	0.009916	**	0.006159	**

The p values of the permutation tests are all below 0.001.

From Table 5, we see that the dynamic code is slower than the static code in the serial as well as the threaded and parallel experiments, and that the difference is significant at the 0.001 level. This is in accordance with box plots presented in Figures 10-15. Just as in the comparison between the dynamic and the original code, all p values are below machine epsilon. However, as shown in Figures 16 and 17, the difference is considerably smaller when inlining is disabled. In the six-dimensional experiment, we see a difference that is significant on the 0.01 level, but in the two dimensional one, we see no significant difference at all; see Table 5.

7.3. Discussion

The refactored code version employing dynamic polymorphism is considerably slower than the original version of the code. When we remove the dynamic binding, using static polymorphism, this difference decreases significantly. For some experimental setups, the refactored code with static polymorphism is even faster than the original one. This indicates that the main cause for the performance loss of the first version of the refactored code is dynamic binding. This may seem contradictory to the results obtained by Demeyer[8]. However, an important difference is that there were no conditionals where we introduced polymorphism, as the original code only had support for one type of blocks, stencils etc. If new functionality was to be added to the original code, conditionals (or a considerable amount of code duplication) would be needed where we have introduced polymorphism. Hence, without the refactoring, this performance reduction could be hard to avoid when HAParaNDA, in the future, is extended with more functionality.

When we compiled the two refactored versions of the code with inlining disabled, the relative difference between the execution times for the two versions decreased considerably. In the six-dimensional case, t decreased from 74.52 to 2.62. In other words, the relative difference between the estimated values and the variation between the measurements is more than 28 times higher when the compiler is allowed to inline small and/or frequently used methods. In the two dimensional case, the performance difference became insignificant when inlining was disabled. This indicates that the main reason of the slowdown caused by dynamic binding is the hampering of inlining at compile time. This is in accordance with the results obtained by Demeyer[8] since if eg the vtable-lookups

were to blame, replacing conditionals by polymorphism would have resulted in slower code in that study.

It is interesting to note that while serial runs of the original code are faster than serial runs of the static code, the static code is faster when it comes to threaded runs. This indicates that the refactored code has better scaling properties. It is tempting to conclude that this can be explained by the improved load balancing, see Section 5. However, the difference can be seen in both the two-dimensional and the six-dimensional experiments. In the six-dimensional runs, the load is actually better distributed in the refactored versions of HAParaNDA. However, in the two-dimensional experiments, we use the domain size 46340^2 elements. 46340 is divisible by the number of threads used (20), so the load should be distributed equally well in both versions of the code. Hence, we conclude that some seemingly unrelated refactoring has improved the scaling of the code.

In the performance experiments, in total, we did 20 different t-tests. Often, advises are given against such a high number of statistical tests, because it brings a high risk of making a type I error somewhere in the analysis. However, since 17 of the t-tests showed a significant difference with $\alpha = 0.001$ and two test showed a significant difference with $\alpha = 0.01$, the risk that we have made a type I error is less than $1 - (0.999^{17} \cdot 0.99^2)$, which evaluates to 3.6%.

8. Concluding remarks

In Sections 5 and 6.4, we argued that application of static polymorphism decreased maintainability, but the static code is still more maintainable than the original one. Consequently, static polymorphism can be a compromise that gives a reasonably maintainable object-oriented code without loosing performance compared to a procedural code written for optimal performance. Actually, as argued above, future addition of functionality to the original code would probably cause a performance loss comparable to the one caused by the dynamic polymorphism. On the contrary, if that functionality was to be added to the static code, it is reasonable to believe that it would not cause any slowdown since the conditionals are replaced by static polymorphism. If so, static polymorphism could come with a considerable performance gain compared to procedural code, when applied in parts of the code that are frequently executed. However, because of the decreased maintainability, we would not recommend application of CRTP, but rather usage of dynamic polymorphism, when the performance gain is low or performance is not an issue,

Acknowledgment

We would like to thank Michael Thuné, Magnus Grandin and Kurt Otto for valuable comments regarding this study. We would also like to thank Carl Nettelblad for help on profiling the refactored code. The performance measurements were performed on resources provided by SNIC-UPPMAX.

References

References

[1] A. N. Johanson, W. Hasselbring, Software engineering for computational science: Past, present, future, Computing in Science & Engineering 20 (2) (2018) 90–109. doi:10.1109/MCSE.2018.021651343.

[2] D. Kelly, Determining factors that affect long-term evolution in scientific application software, Journal of Systems and Software 82 (5) (2009) 851–861. doi:10.1016/j.jss.2008.11.846.

[3] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, G. Wilson, How do scientists develop and use scientific software?, in: Proceedings of the ICSE Workshop on Software Engineering for Computational Science and Engineering, IEEE, 2009, pp. 1–8. doi:10.1109/SECSE.2009.5069155.

[4] M. Fowler, Refactoring: Improving the design of existing code, Addison-Wesley, Boston, MA, USA, 1999.

[5] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois (1992).

[6] ISO, Systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models, Standard 25010:2011, International Organization for Standardization (2011).

[7] J. A. Dallal, A. Abdin, Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review, IEEE Transactions on Software Engineering 44 (1) (2018) 44–69. doi:10.1109/TSE.2017.2658573.

[8] S. Demeyer, Refactor conditionals into polymorphism: what's the performance cost of introducing virtual calls?, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE, 2005, pp. 627–630. doi:10.1109/ICSM.2005.74.

[9] F. Contreras, N. Hitschfeld-Kahler, M. C. Bastarrica, C. Lillo, Balancing flexibility and performance in three dimensional meshing tools, Advances in Engineering Software 41 (3) (2010) 471–479. doi:10.1016/j.advengsoft.2009.10.005.

[10] L. Tahvildari, K. Kontogiannis, J. Mylopoulos, Quality-driven software re-engineering, Journal of Systems and Software 66 (3) (2003) 225–239. doi:10.1016/S0164-1212(02)00082-1.

[11] D. Pflüger, M. Mehl, J. Valentin, F. Lindner, D. Pfander, S. Wagner, D. Graziotin, Y. Wang, The Scalability-Efficiency/Maintainability-Portability Trade-Off in Simulation Software Engineering: Examples and a Preliminary Systematic Literature Review, in: 2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2016, pp. 26–34. doi:10.1109/SE-HPCCSE.2016.008.

[12] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, ACM SIGPLAN notices 42 (10) (2007) 57–76.

[13] G. P. Quinn, M. J. Keough, Experimental design and data analysis for biologists, Cambridge University Press, 2002.

[14] M. Gustafsson, Towards an adaptive solver for high-dimensional pde problems on clusters of multicore processors, Licentiate Thesis, Department of Information Technology, Uppsala University (2012).

[15] A. Bruaset, A. Tveito, Numerical solution of partial differential equations on parallel computers, Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2006.

[16] Openmp.
URL `http://www.openmp.org`

[17] Mpi forum.
URL `http://www.mpi-forum.org`

[18] M. Gustafsson, A. Nissen, K. Kormann, Stable difference methods for block-structured adaptive grids, Tech. Rep. 2011-022, Department of Information Technology, Uppsala University, Sweden (2011). doi:10.1007/s10915-014-9969-z.

[19] R. C. Martin, Clean code: A handbook of agile software craftsmanship, Prentice Hall, Upper Saddle River, NJ, USA, 2008.

[20] R. C. Martin, Agile software development: principles, patterns, and practices, Prentice Hall, Upper Saddle River, NJ, USA, 2003.

[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: Elements of reusable object-oriented software, Addison-Wesley, Boston, MA, USA, 1995.

[22] M. Källén, S. Holmgren, E. Hvannberg, 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, 2014, pp. 125–134. doi:10.1109/SCAM.2014.21.
[23] H. S. Warren, Hacker's delight, NJ : Pearson Education, 2013.
[24] J. O. Coplien, Curiously recurring template patterns, C++ Rep. 7 (2) (1995) 24–27.
[25] boost.
URL http://www.boost.org
[26] A. Duret-Lutz, T. Géraud, A. Demaille, Design patterns for generic programming in c++, in: 6th USENIX Conference on Object-Oriented Technologies and Systems, 2001, pp. 189–202.
[27] T. Mytkowicz, A. Diwan, M. Hauswirth, P. F. Sweeney, Producing wrong data without doing anything obviously wrong!, SIGPLAN Not. 44 (3) (2009) 265–276. doi:10.1145/1508284.1508275.
[28] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493. doi:10.1109/32.295895.
[29] T. v. Enckevort, Refactoring UML models: using openarchitectureware to measure UML model quality and perform pattern matching on UML models with OCL queries, in: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, 2009, pp. 635–646. doi:10.1145/1639950.1639959.
[30] J. Bansiya, C. Davis, L. Etzkorn, An entropy-based complexity measure for object-oriented designs, Theory and Practice of Object Systems 5 (2) (1999) 111–118.
[31] J. Al Dallal, L. C. Briand, An object-oriented high-level design-based class cohesion metric, Information and software technology 52 (12) (2010) 1346–1361. doi:10.1016/j.infsof.2010.08.006.
[32] I. H. Moghadam, M. Cinneide, Automated refactoring using design differencing, in: 16th European Conference on Software Maintenance and Reengineering, IEEE, 2012, pp. 43–52. doi:10.1109/CSMR.2012.15.
[33] M. Akiyama, S. Hayashi, T. Kobayashi, M. Saeki, Supporting design model refactoring for improving class responsibility assignment, in: Model driven engineering languages and systems, Springer Berlin Heidlelberg, 2011, pp. 455–469. doi:10.1007/978-3-642-24485-8_33.
[34] T. Hopkins, Is the quality of numerical subroutine code improving?, Tech. Rep. 1-97*, University of Kent, Computing Laboratory (January 1997). doi:10.1007/978-1-4612-1986-6_14.
[35] Understand.
URL http://www.scitools.com
[36] J. Al Dallal, Fault prediction and the discriminative powers of connectivity-based object-oriented class cohesion metrics, Information and Software Technology 54 (4) (2012) 396–416. doi:10.1016/j.infsof.2011.11.007.
[37] The rackham cluster.
URL https://www.uppmax.uu.se/resources/systems/the-rackham-cluster