Object Oriented Approaches to High Performance Computing

Per Eilif Træen and Carl Erik Hauge
Norwegian University of Technology and Science, Trondheim
Department of Computer and Information Science

23rd November 2001
Abstract

The use of concurrency is an important issue for achieving speed up in the execution of applications for scientific computing. Together with object oriented programming one can build efficient and general applications that work for numerous kinds of problem domains. A presentation of issues regarding the use of concurrent object oriented languages for high performance scientific computing is given, together with a discussion around efficiency matters for the object oriented languages C++ and Java.

A study of the expressiveness of C++, Java and Fortran for solving an area computation of a three dimensional mesh is conducted in the context of a prototype object oriented framework for distributed numerical computing. The framework is implemented using C++, thus a discussion on issues surrounding framework implementations for Java and Fortran is presented.

Results suggest that Fortran, C++ and Java have approximately the same expressiveness for solving the area computation problem. The authors believe this from the fact that the size and complexity of the problem is too small to give a real indication of differences in expressiveness. For the framework implementation it is shown that Java can be used just as well as C++ with regards to expressing the required functionality of the framework. Fortran, however, lack the necessary object oriented language constructs for implementing the generality and reusability offered by frameworks.

Some assumptions regarding the efficiency of the different implementation strategies are also discussed, but are seen mainly as a case for further work.
Contents

1 Introduction .................................................. 1
  1.1 Motivation .................................................. 1
  1.2 Summary and Report Structure .............................. 1

2 Programming & Object Oriented Systems ....................... 3
  2.1 Procedural Programming ................................... 3
  2.1.1 Examples of procedural languages ..................... 3
  2.2 Object Oriented Programming ............................... 6
  2.2.1 Object oriented features ............................... 6
  2.2.2 Benefits of object orientation ......................... 8
  2.2.3 Examples of Object Oriented Languages ................ 9
  2.2.4 Differences in object-oriented languages .............. 12
  2.2.5 Components ............................................ 14
  2.2.6 Frameworks ............................................. 14
  2.3 Object Oriented Systems .................................. 15

3 Concurrency and Scientific Computing with Object Oriented Languages .......................................................... 18
  3.1 Concurrent Object Oriented Programming .................... 18
  3.1.1 Object Oriented languages and parallel programming .... 18
  3.1.2 Methods for concurrent problem solving ................ 18
  3.1.3 Merging object-oriented languages and concurrency . . 19
  3.1.4 Concurrency in C++ ...................................... 21
  3.1.5 Conclusions .............................................. 24
  3.2 Object Oriented Scientific Computing ...................... 25
  3.2.1 Software for scientific computing ..................... 25
  3.2.2 Object-oriented frameworks in scientific computing .... 25
  3.2.3 Object Oriented Programming and PDE's ................ 28
  3.3 C++ & Fortran in scientific computing ..................... 28
  3.3.1 Haney's Tests .......................................... 28
  3.3.2 Preliminary Conclusions ................................ 32
  3.3.3 Robison's Tests ........................................ 33
  3.3.4 Preliminary Conclusions ................................ 36
  3.3.5 Veldhuizen & Jernigan's Tests ......................... 36
  3.3.6 Conclusions ............................................. 37
  3.3.7 Why Was C++ Slower? .................................... 38
3.3.8 How Might C++ Be Faster? ........................................ 39
3.4 Java for scientific computing ........................................ 43
  3.4.1 Critical Issues ................................................... 44
  3.4.2 Performance of Java ............................................. 47
  3.4.3 Using libraries in Java .......................................... 47

4 Case Study ............................................................... 50
  4.1 Problem description ................................................ 50
    4.1.1 Mathematical explanation ..................................... 50
    4.1.2 Algorithm ....................................................... 52
    4.1.3 Why this problem? ............................................. 54
    4.1.4 Issues with the problem ....................................... 55
    4.1.5 Focus ............................................................ 55
    4.1.6 Approaches ...................................................... 55
    4.1.7 Expectations .................................................... 55

5 Study of Object Oriented Framework for concurrent scientific computing ........................................ 56
  5.1 Code analysis ....................................................... 56
    5.1.1 Rational Rose diagram ....................................... 56
    5.1.2 Class functionality ............................................ 56
  5.2 Solving the toy problem using the existing framework ........................................ 58
  5.3 Conclusion .......................................................... 63

6 Expressiveness of Object Oriented and Procedural languages ........................................ 64
  6.1 Java implementation strategy ...................................... 64
    6.1.1 RMI - Remote Method Invocation .............................. 64
    6.1.2 Translating the framework to Java ............................ 66
    6.1.3 Toy Problem in Java .......................................... 70
    6.1.4 Implementation with the Java Array package ............... 75
    6.1.5 Conclusion ...................................................... 76
  6.2 Fortran ............................................................... 76
    6.2.1 Framework issues ............................................ 76
    6.2.2 Toy problem in Fortran ...................................... 77
  6.3 Conclusion .......................................................... 78

7 Summary ............................................................... 80
  7.1 Summary of our discussion ...................................... 80
  7.2 Future work ......................................................... 80
Preface

This report is the result of a project for fifth year students at the Norwegian University for Science and Technology in Trondheim, during the autumn term of 2001. The project was chosen as part of the study in the system development course.

The project has allowed us to look into the use of concurrent object oriented programming for scientific computing and has been very rewarding for us to work with. The subjects of object oriented programming, parallel computing and scientific computing are very extensive, and with all the material we’ve come across during our study it has been hard keeping focus on just our particular assignment.

Reidar Conradi and Roxana Diaconescu, both from our university, designed the task that we originally were presented with. The definition of our task has changed during our work, giving us the opportunity to decide exactly what kind of particular area of interest we would follow within the context of concurrent object oriented programming for scientific computing.

Our work has consisted mainly of two phases: a thorough literature study, giving us sufficient background knowledge before moving on to the next phase. This consisted of the definition of a small problem solved in the context of an object oriented framework for distributed scientific computing, focusing on the expressiveness of the programming languages C++, Java and Fortran.

We would like to thank our mentor for this project, Roxana Diaconescu, who has provided us with valuable feedback and tips during our work. Without her help this project would have been much harder to complete. Also we would like to thank Kenneth and Kjetil Pedersen with whom we have worked together with during the entire project.

A web-page with a short description of our work can be found at: http://faerun.dhs.org/project

Carl Erik Hauge

Per Eilif Træen
1 Introduction

The points below will be covered in the introduction. This will be filled in towards the end of the project.

1.1 Motivation

We have been looking into an object oriented framework for concurrent scientific computing, implemented by Roxana Diaconescu at NTNU. Our goal has been to establish whether it is possible to translate this framework from its C++ implementation into Java and/or Fortran, with focus on the expressiveness of the languages. Expressiveness is useful from a programmers point of view because it affects how much work it is for him to write the code. If it should prove feasible to translate the framework this would be useful with respect to two issues: 1) Java is today a very popular language, and most students have knowledge of it. 2) Fortran, with its history as a high-performance language when it comes to scientific computing, would most likely provide better performance than an object oriented framework. The question is whether it is possible.

To accomplish this we had to look into object oriented and procedural languages, and how they are structured. We collected information on C++, Java, and Fortran performance to have an idea of how the differences between the languages manifested into real-world applications. Subsequently we studied the previously mentioned framework, and devised a toy problem that we could use during our discussion.

1.2 Summary and Report Structure

In this report we cover the use of object-oriented languages for scientific computing. Chapter 2 starts with a presentation of procedural programming, particularly Fortran that has been the language of choice for many years in scientific computing. Then we present the concepts of the object-oriented paradigm and the features that characterize an OO language. This is followed by a brief presentation of the OO languages Smalltalk, C++ and Java, including a comparison of their features.

Chapter 3 gives an introduction to parallel programming and how this feature is incorporated into the object oriented languages. An example is provided with the C++ language. In this chapter we also move into the area of scientific computing and give an overview of existing object oriented frameworks for scientific computing. Next comes a comparison of Fortran and C++ for scientific computing based on some existing tests.
We finish the chapter with a discussion of Java for scientific computing and how Java needs to improve to be become an efficient language numerical problem solving.

In Chapter 4 we define a small problem as a basis for our discussion, the area computation of a three-dimensional mesh. We set our focus on the expressiveness of different programming languages and how well suited these are for implementing the algorithm we propose. This is also put into the context of a prototype object oriented framework for concurrent scientific computing, developed by Roxana Diaconescu at the Norwegian University of Science and Technology in Trondheim [ref1] [ref2]. We also state that we expected to find that C++ and Fortran would show better expressiveness than Java in solving our little problem.

Chapter 5 presents the basics of the framework, written in C++. We then proceed to present different approaches to implementing the algorithm in C++ using the framework. First we look at a solution with two-dimensional arrays containing information of the elements, faces and vertices comprising the mesh. Then we propose a more object oriented approach where elements, faces and vertices are represented as classes. In both these approaches we make assumptions that the framework be slightly modified in order to provide the proper data-structures for us to work with. (Of the two approaches the object oriented one proved to be the most expressive since we could create classes modelling the behaviour of the mathematical entities element, face and vertex. Operator overloading provided us with the opportunity to do arithmetic operations on the objects using mathematical operators instead of function calls)).

Our contribution comes in chapter 6 were we look at solving the problem in Java. This also includes looking at how the framework could be translated from C++ to Java, particularly how to translate features in C++ that are not present in Java like pointers, templates and global functions. Another issue is that of parallelism. In the C++ framework OOMPI is used for communication among the different process, therefore we look at how we would replace OOMPI with Java RMI. Assuming that we had the framework in Java we present code for solving the problem with the framework.

A solution is also presented in Fortran. Here we do not assume any framework, since the Fortran language does not have the features necessary to create such a framework. Still, we assume that we can use functions resembling those in the framework.

We end the report with an evaluation of our work and suggestions for further work.
2 Programming & Object Oriented systems

First we will cover the basics of object oriented and procedural languages, and give some examples. Then we will cover some information on parallel computing, before going on to scientific computing. Here we mainly focus on C++, due to its relevance to our project. Then we'll go into a fairly in-depth discussion on the efficiency of Fortran vs. C++ in scientific computing.

2.1 Procedural Programming

According to [28], the procedural programming paradigm can be stated as: “Decide which procedures you want; use the best algorithms you can find”. The focus on the processing and the algorithm that is needed for the desired computation.

In procedural programming, we think of programs as consisting of a series of functions and a set of data structures. The functions act on the data, manipulating them and extracting information. Procedural languages such as C and Pascal were built to support structured programming techniques, in which problems were decomposed into manageable sub-programs.

2.1.1 Examples of procedural languages

FORTRAN

FORTRAN is short for formula transformation and was first used mainly by people with scientific background for solving complex arithmetic problems. FORTRAN I was developed in the early 1950’s by a team from IBM. It was designed to work with IBM’s new mainframe Model 704. One of the initial goals of FORTRAN I was that it would be a language that would result in an executable machine code that was as good as one written by a specialist. FORTRAN I was indeed the first ever successful high-level language. Not only was it possible now to write large programs in a much shorter amount of time but also these programs were portable.

FORTRAN II came in 1958 and was a significant improvement; it added the capability for separate compilation of program modules and assembly language modules could also be 'linked loaded' with FORTRAN modules.

FORTRAN III came that same year but was never released to the public. It made it possible to use assembly language code right in the middle of FORTRAN code. Such
"inlined" assembly code can be more efficient, but the advantages of a high-level language are lost (e.g. portability, ease of use).

FORTRAN IV was an improvement of FORTRAN, improving details of the implementation and removing some machine-dependant language irregularities.

1966 was a milestone for FORTRAN as ASA (American Standard Association) finished the development of a new FORTRAN standard known as FORTRAN66, the first high-level language in the world.

FORTRAN has evolved continuously and in 1977 came FORTRAN 77. It had improvements such as IF-THEN-ELSE statements (had previously been IF-GOTO statements), DO loops with a decreasing control variable (index), pretest of DO loops (before FORTRAN77 DO loops were always executed at least once, so you had to add an IF GOTO before the loop) and a CHARACTER data type (before characters were always stored inside INTEGER variables).

The next standard came in 1991, FORTRAN90, and some of the improvements were modern control structures such as CASE and DO-WHILE, array notation and dynamic memory allocation.

Later on there has been the introduction of FORTRAN95 and High Performance Fortran; two different improvements of Fortran90 and Fortran95 also had some features of High Performance Fortran.

An important distinction in Fortran is that of function and subroutine. A function is used to calculate a result that basically will depend on the arguments given to the function each time it is invoked. A function has a value and is called by any mention of its name, while a subroutine is called only within a special statement, the CALL statement. A subroutine does not need to have arguments and it does not return any values. It can, however, change data specified in the arguments.

One of the main strengths of Fortran is its array structure. Fortran arrays have two major components: a block of dense, contiguous storage that hold the data elements of the array, and a descriptor that contains the address of the storage block, bounds information for each dimension, the size of individual data elements, and other information used to index the array. This indexing information is used to access elements in the array and implement array sections. Accessing an element of the storage is done in two steps: first the base address of the storage is extracted from the descriptor and the subscript, along with the indexing information in the descriptor, is used to compute the offset to the data element in the storage.

Another issue with Fortran arrays is sections. A section of a Fortran array is a view of the array that accesses some subset of elements of the array. One property of a section is that it utilizes the same storage as the array, so a change of an element in the section is also reflected in the array and vice versa. The advantage with sections is that one has the ability to pass portions of arrays to library functions.

**Cobol**

Cobol stands for Common Business Oriented Language and was introduced because FORTRAN’s scientific programming style was not adequate for describing the more complex
data in commercial programming. The idea of introducing COBOL was also to develop a language that could be used on different manufacturer's machines. COBOL made several important contributions to the future development of programming languages. It had the separation between the task to be undertaken, the description of the data involved and the working environment in which the task is carried out. Secondly it had a data description that was largely machine independent. It was very effective in handling large files, and it had the benefits of being fairly easy to read. Cobol incorporated a number of data-definition languages, making it suitable for accessing databases, a feature not existant in Fortran. Other features that were not present in Fortran at that time was the length of variable names (up to 30 characters) and also the detailed description of file records and lines for printer output.

Pascal

Niklaus Wirth developed Pascal in 1970 and the first standard was released in 1974. Pascal was developed as a teaching language with little or no intension for commercial use. The language has two characteristics that made it ideal to teach programming. First, it is a "structured" language (some say the first such language), and second it is "strongly typed".

A structured language allows the program code to be segmented into self-contained pieces that can be written independently of the total application. This allows code to be reused in separate applications and to be more easily written by teams of programmers. Pascal first used the concept of a procedure or function as a separate unit. This idea has culminated in the modern concept of a program object.

A language that is strongly typed demands that each program variable and data element be declared (i.e., announced to the program) and that its fundamental characteristics be unchanged throughout the program. For example a variable declared as an integer stays an integer, an array remains an array. This forces the programmer to avoid such traps as dividing an integer by a real and storing the result in another integer.

The weaknesses of Pascal is the language's lack of constructs for handling larger programs to be built up in a modular way and it is limited in its handling of text and the input/output facilities.

C

C came into being in the years 1969-1973, in parallel with the early development of the Unix operating system. Unix is in fact developed in C and so is most of the software running on Unix. The language was a successor of the programming language B. C operates at a very low level, something that gives the programmer freedom to control the computer in detail. This has resulted in C largely replacing assembler in the development of system programs. In addition to access to low-level machine operations, C supports user-defined data types, recursive procedures and pointer arithmetic.
2.2 Object Oriented Programming

According to [28], the object-oriented programming paradigm is: "Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance".

The first ever OOP language was SIMULA, developed in Norway in the early 1960's. SIMULA was intended for writing simulation software. The idea of encapsulation, one of the central ideas of OOP, was introduced with SIMULA, because data could not be accessed directly, only through procedure calls. SIMULA also presented the term data class, another important factor in OOP, where the clue was that data structures and the routines that affect these structures should be contained in a data class. The ideas of SIMULA never reached the majority of software developer, probably because SIMULA was aimed only at those creating simulation software.

In the 1980's the software industry was in a period of crisis. Computer hardware got faster and cheaper and software got bigger and more expensive. The rapid advance in hardware speed and cost was absorbed by the software industry that started the development of Graphical User Interfaces (GUIs) instead of the old text based interfaces. So with the ever-increasing demand for better software, programmers were being stretched to their limits. They started searching for a solution that involved taking software components from old programs and using them in new ones. The programming idea of reuse was born but one wasn't quite sure how to implement it. Then the OOP was reinvented, because one discovered that the OOP paradigm is perfect for reuse.

With the OOP paradigm the fundamental building blocks are no longer the procedures of the procedural languages, now it is the objects. Procedures are now activated only by sending messages to objects. The OO concept also makes it easier to design software since the idea is that real life entities can be mapped in to objects in an OO programming language. This works both ways in that it is easier to read OO code and understand the logic of it by mapping to objects in the real world.

2.2.1 Object oriented features

In [29] Wegner states that for a language to be object oriented, it must support classes, objects, and inheritance. Others [30] state that a language must support dynamic binding and encapsulation.

In order to fully understand and get the most of object oriented technology there are a few key terms one must learn to handle.

**Class:** A class is an abstract definition of a type that includes the data structure for the values of the type and the operations on that type. It can be seen as a prototype for an object.

**Object:** An object is an instantiation of a class. An object contains data and operations on the data. Operations are implemented as methods that manipulate the data within the object in some way. The methods of an object can be seen to represent the objects behavior, while the variables represent its state.
Method - A procedure that an object uses to perform an operation requested by a message. The method resides in the class to save storage, since all instances of a class have identical sets of methods. A method may change an object's private memory (instance variables) or send a message to other objects.

Inheritance: The basic idea behind inheritance is that one should be able to define a new class by using the definition of existing classes. The new class will then consist of all variables and operations from the existing classes in addition to values and operation defined in the new class. The new class is said to inherit the variables and operations of the existing classes. The new class is a subclass of the classes from which it was derived, also base classes. There exists two types of inheritance, simple and multiple. In simple inheritance each subclass has only one superclass. Multiple inheritance allows a subclass to have more than one superclass.

Encapsulation: Encapsulation concerns the idea of information hiding or data abstraction. Encapsulation ensures that data cannot be accessed and manipulated directly, only through function calls.

Interface: An interface is a device that unrelated objects use to interact with each other. It is used to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy.

Message: Software objects interact and communicate with each other by sending messages to each other. When object A wants object B to perform one of B's methods, object A sends a message to object B. Since an object's behavior is expressed through its methods, message passing supports all possible interactions between objects.

Dynamic-Binding Dynamic or late binding means that the code to execute a given operation is determined at run-time instead of at compile-time.

Polymorphism: A polymorphic function is one that can be utilized uniformly on a variety of objects. For instance an addition function add that can be used to add two integers or two floating-point numbers. Polymorphism is implemented with the aid of abstract classes and virtual methods. An abstract class is not meant to be used for instantiating of objects, it only defines some common behaviour that is shared by many classes, and these classes inherit from this abstract class. An abstract class can have both base methods and abstract (virtual) methods. Base methods provide behaviour that is generally useful to subclasses. Virtual methods provide default behaviour that subclasses are expected to override, that is to reimplement the method. In [38] Cardelli and Wegner developed a taxonomy of polymorphic techniques (figure 2.1).

As previously mentioned, a language must support classes, objects and inheritance to be regarded as object-oriented. However, an object-oriented language is not necessarily a pure object-oriented language. In a pure object-oriented language everything exists as
Polymorphism

Universal Ad Hoc

Generic Inclusion Overloading Coercion

Figure 2.1: Taxonomy of polymorphic techniques

an object, class or method. Bertran Meyer [32] has proposed some properties that a pure object-oriented language should contain:

- Modular structure
- Data abstraction
- Automatic Memory Management (the language deallocates unused objects without without programmer intervention)
- Classes
- Inheritance
- Polymorphism and dynamic binding
- Multiple and repeated inheritance

2.2.2 Benefits of object orientation

When using object oriented ideas and concepts it becomes possible for the user (programmer) to abstract the problem domain, and model it by using objects. This abstraction into objects fit more natural to human cognition, and it is therefore easier to understand. One example is the vehicle-class, with its subclasses cars and motorbikes. By using classes and inheritance, we may define the features of each type of vehicle, e.g. number of wheels, number of passengers, etc.

The features of encapsulation and inheritance provide the possibility of improved reuse. The reuse of software allows us not just to reuse single classes, it allows us to build frameworks and components that we can use when designing large computer systems. This results in reduced development risks for complex systems, faster development and increased quality. It is worth noting that there doesn’t exist any unambiguous empirical evidence to support the claims of increased software quality. In addition, the fact that we may group classes into “domains”, i.e. groups of classes that know little about “the world outside” is excellent for the distributed development of software. Encapsulation also leads to software which is easy to maintain. Software maintenance is partly fixing bugs but is mostly changing it to satisfy new requirements, something that will surely happen in the lifetime of a software system.
2.2.3 Examples of Object Oriented Languages

With the ever-increasing popularity of object-orientation, many OO programming languages have been created in the last couple of years. Below is a brief presentation of 3 of them.

**Smalltalk** Smalltalk is an older object-oriented language, originally developed at Xerox Palo Alto Research Center in the early 1970's. It was first intended as a research tool for implementing interactive systems, but later it evolved into a stable standard that has been implemented on a variety of hardware. The primary reference for the language is Goldberg and Robson (1983). They note, “The Smalltalk-80 system is based on the Simula language and from the visions of Alan Kay, who first encouraged us to try to create a uniformly object-oriented system.

The 5 basic terms in Smalltalk are:

- Object
- Class
- Instance
- Method
- Message

All of these terms have been explained in 2.2.1, except for *instance*. An instance is a term for an object of a class. Smalltalk is built on objects and messages. Objects interact by sending messages to each other, and in response to a message, the receiving object executes a method.

Smalltalk’s design principles were the following:

- Everything is an object. All components are objects, even classes.
- Data is private to the object
- Anb object has a notion of *self*. When an object executes a method in response to a message.

Smalltalk is more than just a language, it is an extensive program development environment with over two hundred classes and several thousand methods. Smalltalk consists of the following components:

- A language - Because everything in Smalltalk is an object, the language is lean.
- An object model from which objects are defined - The object model defines how objects behave. The object model supports inheritance, class and instance behavior, dynamic binding, messaging, and garbage collection.
• A set of reusable classes - Smalltalk has an abundance of classes that can be reused in any Smalltalk program. These classes provide the basic functions in Smalltalk plus additional support for cross-platform portability, including portable graphical user interfaces, and support for the definition and management of classes.

• A set of development tools - These tools enable users to look at and modify existing classes, rename classes, add new classes, and delete classes. They also provide source-level debugging, including the ability to add halts, look at data, modify data, and make changes to code during the execution of a program.

• A runtime environment: Smalltalk allows users to break out of the "compile-link-run" cycle. The editor, compiler, linker, debugger and the application in Smalltalk are part of the runtime image. This allows users to run the Smalltalk program while changing the source code. Changes made to the source code are reflected in the running application instantly.

It is said that “Smalltalk is not a language, but rather a system in which to define systems”. As described above, the language is a part of a big system that is made up of many components. These components include objects that provide functions usually attributed to an operating system. These are

• Automatic storage management

• File system

• Display handling

• Text and picture editing

• Keyboard and pointing device input

• Debugger

• Performance spy

• Processor scheduling.

Smalltalk, like Java, runs a virtual machine. This VM consist of the interpreter that establishes an object-oriented model for storage, a message-oriented model for processing, and a bitmap model for visual display of information. This, however, means that programs are not compiled, but interpreted, at run-time, something that invariably leads to slower performance.

One of the greatest features of Smalltalk is its high degree of reusability. Smalltalk comes with a rich set of objects that can be used directly or easily modified to fit a particular application's needs. Objects often can be used in more than one application.

One of the visions of Smalltalk was that it should be a powerful language (few keywords, high functionality), and that it should be easy to write. Due to this, Smalltalk
has proved itself well suited for prototyping of applications. However, it is less well suited for delivering applications because applications written in Smalltalk can neither run in isolation from the Smalltalk environment nor be combined with other programs written in other languages. It is possible to convert Smalltalk into C, but there are problems, for example due to the fact that Smalltalk and C has different storage management systems.

One of the drawbacks of Smalltalk is that it demands large computer resources and that programs produced usually execute slowly. The latter is due to the fact that the program is not compiled, but interpreted, at execution.

**C++**

C++ is an object oriented version of the C language and was first implemented by Bjarne Stroustrup at AT&T Bell Laboratories. It was first used in 1980 but was not made commercially available until 1985. The aims of C++ were to be a general purpose programming language that is a better C, supports data abstraction, supports object-oriented programming and supports generic programming. Because C++ retains C as a subset, it gains many of the attractive features of the C language, such as efficiency, closeness to the machine, and a variety of built-in types.

Some of C++’s features are:

- User defined types
- Strongly typed
- Class hierarchies
- Namespaces
- Access control - unlike other object-oriented languages, C++ allows objects to access parts of all of the internal data of other objects (by using friend, private, public)
- Pointers
- Generic programming via templates
- Virtual functions
- Operator Overloading

**Java**

Java was officially introduced in 1995 by Sun Microsystems engineer Arthur van Hoff. It is an object-oriented programming language very well suited for developing applications for the web. Java applications, or applets, are different from ordinary applications in that they reside on the network in centralized servers.

Java was originally designed to be used in consumer electronics, under the name Oak. The driving inspiration behind Oak was to create something that could be used on every
type of computer, from PCs to large mainframes. Oak was renamed Java in 1995, and by that time, Java had adopted a model that made it perfect for the Internet - the bytecode model. This is what makes it possible for Java to be platform independent. Every Java program is compiled into compact bytecodes that a Java Virtual Machine (JVM) then reads and interprets to run the program. The bytecodes are small, and thus well-suited for the Internet (small = fast on a slow connection). At the same time, this means that Java is not as fast as i.e. C++. The reason for this lies in the JVM. Since the JVM has to interpret the program, and then translate it into a machine-code native to the platform, performance is lost.

Java is also strong on security. This is due to the fact that the JVM can strictly monitor what goes on, when a program executes. The user can tailor Java security the way he likes it, offering a flexible solution.

**Java language features**

- **Array-bounds checking** - All access to Java arrays is bounds checked, throwing a run-time exception whenever the bounds are exceeded. The array-bounds checking can also be automatically eliminated in some code, reducing the overhead of the bound-check.

- **Null-pointer checking** - Every Java program uses pointers to refer to objects. Each time a pointer is de-referenced a null-pointer check occurs. This is an important Java feature, since de-referencing a null-pointer is a common programmer error.

- **Automatic garbage collection** - Java removes the responsibility of memory management from the programmer with its automatic garbage collection. The `new` operator allocates memory to an object and from there on the run-time system keeps track of the object’s status and automatically reclaims memory when the object is no longer used. Java’s memory management model is based on objects and references to object, there are no pointers in Java. References to objects are through `handles`. The Java memory manager keeps track of references to objects. When an object has no more references, the object is a candidate for garbage collection.

- **Multithreading** - Multithreading can be a very complicated problem for programmers, particularly in the areas of memory management and synchronisation [42]. Java has integrated support for multithreading easing much of this burden from the programmer. Locks are implicitly and automatically allocated, and there is never a need to explicitly unlock locks. Each object in Java can have a lock associated with it, and that means that one never has to allocate a lock. Exceptions will automatically release any locks and thus avoid eventual problems later on.

### 2.2.4 Differences in object-oriented languages

Below is a table (2.1) summary of some of the differences between the four languages just presented.
<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Java</th>
<th>Eiffel</th>
<th>Smalltalk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>Multiple</td>
<td>Single</td>
<td>Multiple</td>
<td>Single</td>
</tr>
<tr>
<td>Generic classes (Templates)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Static/Dynamic typing</td>
<td>Static</td>
<td>Static</td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Multithreading</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Operator overloading</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Pointer arithmetic</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Compiler/interpreter</td>
<td>Compiler</td>
<td>Interpreter</td>
<td>Compiler</td>
<td>Interpreter</td>
</tr>
</tbody>
</table>

Table 2.1

The difference between static and dynamic typing is that with a dynamic typing system variables do not need to be declared a specific type. Any variable can contain any type or object. Statically typed systems require that all variables are declared with a specific type. The compiler will then ensure that at any given time variable contains only an object compatible with that type.

Of the four languages presented, only Eiffel and Smalltalk can be said to be pure OO languages. A pure OO language is object-oriented from the ground up. That is, everything in the language exists as an object, method or a class. Voegele [39] describes a pure OO-language as one having 6 properties:

1. Encapsulation/Information Hiding
2. Inheritance
3. Polymorphism/Dynamic Binding
4. All pre-defined types are Objects
5. All operations are performed by sending messages to Objects
6. All user-defined types are Objects

Java claims to be a pure OO-language but does not satisfy property number 4 because of its inclusion of "basic" types that are not objects. It fails also to meet property 5 by implementing basic arithmetic as built-in operators, rather than messages to objects.

Below is a table (2.2) measuring the four languages against the properties for pure OO languages:

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>Java</th>
<th>Eiffel</th>
<th>Smalltalk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation/Information Hiding</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Polymorphism/Dynamic binding</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>All pre-defined types are objects</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>All operations are messages to objects</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>All user-defined types are objects</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2.2
2.2.5 Components

Components was not a new idea that came with the OO paradigm. However, with objects one has the possibility of decomposing large applications into cooperating objects and composing applications from previously created software components. Also, the features of OO programming were the same features one required from components. Information hiding and data abstraction are necessary in order to separate component implementations from component interfaces. Polymorphism and dynamic binding allow for dynamic adaptation of components to client needs. Inheritance enables the organisation of components into hierarchies.

No general definition of components exists. Bertrand Meyer in [32] defines a component as a program element that may be used by other program elements (clients), and the clients and their authors do not need to be known to the element’s authors. Saying that a program may be used by other program elements excludes programs meant only for humans or non-software systems. It is not required that the element is usable only by other program elements, it might as well be usable by humans and client software. Microsoft Word for example is meant for humans but can also be used as COM component for use by other software. The second requirement is to ensure that a component is usable in new software systems not predicted by the component’s author.

Bosch [31] defines a component as a unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes. The provided and required interfaces are for the interaction with other components. The provided interface is the access point to the component’s functionality, whereas the required interface is where the component hooks onto other components provided interfaces. The configuration interface is intended for the use of the component. There is one configuration interface for each variation point in the component. Variation points are points where the behaviour of the component can be altered.

The main advantage of components is reuse. The fact that one can create large applications by composing pre-packed software components has remarkable effects on programmer productivity and also the quality of the final application. This, however, requires a substantial amount of effort from the component developer to create a stable component that does what it is expected to. The quality of an application is only as good as the quality of it’s poorest component.

2.2.6 Frameworks

OO programming introduces features like inheritance, encapsulation and polymorphism. These features are important when one want to develop frameworks. Frameworks are an object-oriented reuse technique. As Ted Lewis defines it in [12] a framework is a class hierarchy plus a model of interaction among the objects instantiated from the framework. In [24] a framework is defined as a reusable, "semi-complete" application that can be specialized to produce custom applications. Another definition of frameworks, found in [25] is that a framework is a set of classes that embodies an abstract design for solutions to a family of related problem. A framework can be seen as the reverse of the traditional
reuse. In traditional reuse a programmer calls reusable procedures, from for example a library, in his program. When using a framework, however, the programmer instantiates objects from the framework and then provides methods for the framework to call. So a framework can be said to consist of both static and dynamic objects.

Johnson [10] describes frameworks as something in between components and patterns; they are more customisable than most components but less abstract then patterns. A framework provides a reusable context for components and also makes it easier to develop new components. When using a framework the application must adapt itself to the design and model of the framework, and in that way the framework automatically causes a pattern in the application.

Frameworks are often classified as either white-box or black-box. A white-box framework is based on inheritance. The user customizes the framework through subclassing of the framework classes and overriding predefined methods from the inherited class. A black-box framework is based on composition. The user defines components that are integrated into the framework by conforming the components to a particular interface of the framework.

Another way to categorize frameworks is into calling or called frameworks. A calling framework is an active entity that invokes other part of the application. A called framework is a passive entity that can be invoked by other parts of the application.

One example of an application framework is the Microsoft Foundation Classes (MFC). MFC is a GUI (Graphical User Interface) framework implemented in C++ for developing Windows applications. The MFC framework provides elementary user interface building blocks (e.g. buttons and menus), basic data structures (classes like CObList and CObArray) and high-level application components (such as the classes CWinApp, CDocument and CView). MFC is basically a white-box framework, where the programmer creates classes inheriting from the MFC framework, and overrides methods. As an example some classes have methods for responding to user mouse-clicks. In the MFC framework the body of these are empty, so if the programmer wants his application to respond to a mouse-click he must override the framework method and add the desired response to the mouse click event.

A framework has its advantages and disadvantages. A high quality framework can greatly increase software quality and reduce the programmer's development effort. On the other hand, to be able to use a framework will often require a lot of time to understand how the framework is built. In a white-box framework the developer needs to understand the framework's internal structure. A black-box framework is easier to use and extend but are more difficult to develop since the developer needs to define interfaces that anticipate a wide range of potential use cases. It is therefore important that a framework is well documented.

2.3 Object Oriented Systems

An object oriented system is a system built on the principles of object orientation, with for example message passing built in. However, there doesn't seem to exist any formal
definition of what an object oriented system is. We therefore present a framework for
determining what is an object oriented system.

As we state in chapter 2.2.1 the concepts of object, class and inheritance are central
to defining an object-oriented languages. As we can see from figure 2.2

- Languages that include the notion of objects are deemed to be object-based.
- Languages that includes objects and classes are referred to as class-based.
- Languages that supports objects, classes and inheritance are said to be object-oriented.

The model therefore provides a framework for discussing various languages which
feature aspects of object oriented computing. This framework is clear and concise for
most of the traditional work in object oriented languages, but limitations become apparent when it is applied to some of the recent developments in object oriented computing. Especially in the area of statically typed object oriented languages, where there is no explanation of the role of conformance, genericity and enhancement in object oriented computing.

The main limitation is that it is based on a specific set of mechanisms which object oriented languages should support. Languages which do not feature these mechanisms or which adopt alternative mechanisms are therefore not considered to be object oriented. This is unsatisfactory since new mechanisms for object oriented computing emerge all the time. [37] presents a model, based on the more general dimensions of an object-oriented environment. The proposed dimensions are encapsulation, classification, polymorphism, and interpretation.

Encapsulation - the grouping together of various properties associated with an identifiable entity in the system in a lexical and logical unit, i.e. the object. Furthermore, access to the object should be restricted to a well-defined interface.

16
Classification - the ability to group associated objects according to common properties. Various classifications can be formed representing different grouping in the system. All object within a particular grouping will share all the common properties for that grouping, but may have other differences.

Polymorphism - implies that objects can belong to more than one classification. Classifications can therefore overlap and intersect. Thus it is possible for two different classifications to share common behaviour.

Interpretation - defined as the resolution of polymorphism. In polymorphic environments, it is possible for a particular item of behaviour to have several different meanings depending on the context. It is therefore the task of interpretation to resolve this ambiguity and to determine the precise interpretation of an item of behaviour.

The model and its dimensions is not supposed to be a definition of object orientation, but it defines a design space for object oriented systems (figure 2.3).
3 Concurrency and Scientific Computing with Object Oriented Languages

3.1 Concurrent Object Oriented Programming

3.1.1 Object Oriented languages and parallel programming

The need for speed in scientific computing is often satisfied by parallel computing. Large problems and simulations require a lot of computation power. The idea is to split the problem into smaller tasks that can be solved in parallel, with some sort of coordination. Parallel computing is based on concurrency, synchronization and communication. Original OO-languages did not support concurrency but this has been added over the years through libraries or language extensions (e.g. Concurrent Smalltalk, C++ and Presto). Some new concurrent OO-languages has also been developed, like Emerald and Hybrid.

In [26] Agha defines concurrency as referring to the potentially parallel execution of parts of a computation. In a concurrent computation the components of a program may be executed sequentially or they may be executed in parallel.

Using concurrency one has the flexibility of interleaving the execution of components of a program on a single processor or to distribute it among several processors. Therefore one finds in concurrent languages constructs for creating processes (e.g. fork) and destroying them (e.g. join). Another issue for concurrency is how organization of memory is done. Physical memory may be shared by all processors or it may be distributed throughout the system so that each process only has access to a portion of the memory. The operating system will optimize the mechanisms for communication, synchronization and mutual exclusion according to the physical organization of memory. In physically shared memory communication between processes is done through sharing of variables. This forces the memory to ensure mutual exclusion of the different processes. Semaphores and spin locks are often used by these systems for data synchronization and mutual exclusion. With physically distributed memory message passing or remote procedure calls are used for process communication. Synchronization is achieved by blocking and nonblocking message calls, but when using nonblocking mechanisms the programmer is responsible for maintaining data consistency.

3.1.2 Methods for concurrent problem solving

[26] identify three common patterns of parallelism in problem solving. The first is pipeline concurrency. This involves the enumeration of potential solutions and the concurrent
testing of these solutions as they are enumerated. An example of such a strategy is the filtering of prime numbers. One generates all numbers and start to remove multiples of 2, 3, 5 etc. up to the largest prime found so far. Once a number is identified as a prime it is added to the filter. The next strategy is the divide and conquer concurrency. This involves the concurrent solving of different subproblems before joining them to obtain a solution to the overall system. Algorithms of this kind are best expressed as functions. An example of such a strategy is the problem of computing the product of numbers in a list. One could recursively subdivide the list into the multiplication of two sublists which are evaluated concurrently. In the divide and conquer strategy there is no interaction between the procedures solving the subproblems. The final strategy is therefore cooperative problem solving, which involves a dynamic complex interconnection network. As each process does its computation it may need to communicate with other processes and share intermediate results which it has computed. An example of such a system is a simulation where physical objects are modeled with logical objects that need to communicate with each other. Such a system could for example be the dynamic evolution of the paths of a number of bodies under the influence of each others gravitational fields.

3.1.3 Merging object-oriented languages and concurrency

The combination of object orientation and concurrency is made easier by the fact that the two programming styles have some similarities in their basic constructs, more accurately in their underlying concepts: classes and processes [44]. Both categories support:

- Local variables (attributes of a class, variables of a process or process type).
- Persistent data, keeping their value between successive activations.
- Encapsulated behavior (a single cycle for a process, any number of routines for a class).
- Heavy restrictions on how modules can exchange information.
- A communication mechanism usually based on some form of message passing.

[43] presents a survey on how object oriented languages incorporate parallelism. There are many directions to take in the development of concurrent object-oriented languages. Decisions made in areas like process management, communication features and inheritance form different languages.

Process management

Parallel object-oriented systems contain multiple objects, each of which can start a thread of execution. Objects and processes are however independent of each other, processes invoke methods in objects. Process creation is done in one of two approaches, explicit or implicit. In the explicit approach the languages provide mechanisms for spawning multiple processes, external to the object structure. Object integrity is ensured by mechanisms
like locks, monitors and semaphores. In the implicit approach an object invocation can spawn multiple execution threads. Processes are encapsulated within objects, and when an object receives a message it can activate one or more internal objects.

Process termination can be done either explicitly or implicitly. With implicit termination the process is terminated after replying to a message. The explicit termination lets processes continue after replying to a message and be available to respond to future messages.

Process activation can be done either upon creation or when a message is received.

Process granularity refers to the size of the schedulable unit of parallel activity and the amount of processing among messages. A fine granularity implies multiple tasks per object, while a coarse granularity is typically one task per object or even several objects.

**Communication features**

In concurrent oo-languages communication is always through message passing. Sharing of data among objects violates the encapsulation principle in object orientation.

Message types are synchronous, asynchronous and eager invocation. Synchronous communication uses remote procedure calls. Asynchronous communication eliminates the wait for synchronization. This is useful in systems where objects can keep processing without having to wait for an answer to their messages. Eager invocation is a variant of asynchronous communication. The sender of a message continues executing and a future variable holds a place for the result. Processing continues until the sender needs the results in the future variable. If the results have been returned the sender proceeds, if not it blocks and wait for the results.

Message acceptance is implicit or explicit. With implicit message acceptance the system accepts messages automatically, while in the explicit approach the objects control when they receive and process messages.

Message processing involves queues for holding incoming messages. One can choose to process the messages in the order they are received with a first-in first-out queue or one can have queues ordered according to priorities assigned by the system.

Synchronization is concerned with coordinating parallel activities to ensure they run efficiently, consistently and predictably. With inheritance involved synchronization is complicated. When a subclass inherits from a base class it sometimes needs to redefine the synchronization constraints of the inherited method. If one has a single centralized class that explicitly controls message reception, all subclasses are required to rewrite this part each time a new method is added to the class. An alternative to this centralized approach is to use critical sections for maintaining consistency. Each method becomes responsible for controlling the entry into the critical section. Access can be controlled with locks, monitors, semaphores, mutual-exclusion mechanisms or atomic variables.

[43] gives a further overview on concurrent oo-languages and how they adopt the properties just mentioned.
3.1.4 Concurrency in C++

In the foreword of [20] Bjarne Stroustrup (the creator of C++) explains why there is no initial support for concurrency in C++. The reason is that there are many forms of concurrency, for example, builders of user interfaces want one form of concurrency while people in database development want another kind and those doing high-performance numeric computations need parallelism of yet another kind. Stroustrup then concluded that no single model of concurrency would be sufficient for the variety of users.

C++ provides no primitives to support multiple threads of execution, no locking mechanism, no notion of collections of data being accessed in parallel, etc. So to be able to implement concurrency in C++ on relies on such things as library functions and/or added language primitives.

We will here present one such language extension to C++ for supporting concurrency, Compositional C++ (CC++). For a more detailed description of this and other language extensions we refer to [20].

Introduction to CC++

Compositional C++ is seen as a general-purpose programming language. It is designed to be a natural extension to C++, appropriate for parallelizing the range of applications that one would write in C++. CC++ supports integration of different parallel programming styles in a single application. It was designed to provide efficient execution on a range of parallel computing platforms, including shared and distributed-memory computers.

A parallel programming system is said to be compositional if properties that hold for a part of a program in isolation also hold when that part of the program executes in parallel with any other piece of code. By restricting the ways in which one component can access data in a another component we can achieve compositionality. Even though imposing such restrictions guarantee compositionality it also has a disadvantage. Because of the many different models for compositional programming, restricting a program to just one of these can make implementation more complex. This would be the case if for example the application does not map well into the model provided by the language. Another drawback is that in some situations performance is better with non-compositional algorithms.

CC++ has therefore been constructed with a less strict approach to compositional parallel programming. CC++ is said to have a non-deterministic behavior because there are no restrictions on how a variable can be accessed within a parallel operation.

CC++ has support for several types of compositional programming styles through language features and a design called the proper interface design. This approach sets focus on the interface between program components and the properties they must have to be able to ensure compositionality. Software components should be specified in terms of clearly defined inputs and outputs.

In order to give a simplified design of both program components and proper interfaces the design must follow these rules:

Rule 1: An action is either an input, output or internal.
• Input: The action reads shared variables as its input and reads or modifies local variables.

• Output: The action modifies shared variables as its output and reads or modifies local variables.

• Internal: The action does not reference shared variables.

Rule 2: If an input action is executable at some point in a computation, then it remains executable until it is executed. This rule prohibits the execution of a program component from changing an earlier output value.

Rule 3: An input to a program consists of the output of at most one component. If the input is output of two or more components there would be a need to deal with interference of components writing to the same input.

Overview of CC++

CC++ adds six keywords to C++; par, parfor and spawn are parallel composition operators, atomic and sync are synchronization constructs and global controls distribution and locality.

Parallel composition operators The most basic parallel construct is the parallel block or par block. A par block looks like this:

```cpp
par{
    statement1;
    statement2;
    statement3;
}
```

The statements in a par block execute in parallel, using fair interleaving. Under fair interleaving the operations of any one statement must execute in sequence. Interleaving is also such that operations from each statement will execute eventually.

The closing brace of the par block is a sequence point, meaning that statements after the brace will not execute until all the statements in the par block have terminated.

A parallel block only specifies when statements can execute, not where they execute (generally they execute in the same location).

The parfor statement provides parallel iteration. The statement is built just like a for-statement:

```cpp
parfor( initialization; test condition; update expression) {
    loop body
}
```
Each instance of the loop body is executed in parallel.

While \texttt{par} and \texttt{parfor} supplies parallelism in a structured manner (statements after \texttt{par} and \texttt{parfor} do not execute until all statements in \texttt{par} and \texttt{parfor} are terminated), \texttt{spawn} is the unstructured alternative to parallelism. \texttt{spawn} causes a function to execute in parallel with the rest of the program.

\begin{verbatim}
spawn function1();
function2();
\end{verbatim}

Statements after the spawned function are executed without regards to the status of the spawned function.

\textbf{Synchronization variables} Synchronization variables are needed in order to control variables that are treated in parallel, for example:

\begin{verbatim}
int x,y;
par {
    x=1;
    y=x+1;
}
\end{verbatim}

In this parallel block it is not possible to tell which value \texttt{y} will get, because of the interleaving.

There are two synchronization variables in CC++, \texttt{sync} and \texttt{atomic}. The \texttt{sync} variable can be seen as a \texttt{const} variable with delayed initialization. When trying to use the value of a \texttt{sync} variable before it has been initialized the reading thread will block until the variable has been initialized. In the above example, if \texttt{x} had been declared as \texttt{sync int x} then the statement \texttt{y=x+1;} would have been blocked until \texttt{x} had been initialized. Any type that can be declared \texttt{const} can also be declared \texttt{sync}.

The \texttt{atomic} keyword is used when one wants to prevent certain interleavings from occurring in a parallel execution. This can be achieved by performing these actions within the body of an \texttt{atomic} function. Within an instance of a C++ class only one \texttt{atomic} function may execute at a time. For example:

\begin{verbatim}
class A {
    atomic void function1();
    atomic void function2();
}
\end{verbatim}
A x;
par {
    x.function1();
    x.function2();
}

The two statements would not be executed at the same time. However, the execution of an atomic function may be interleaved with atomic functions of other classes, with atomic functions of another instance of the same class or with a non-atomic function in the same instance of the class.

**Specifying location** The CC++ constructs we have looked at so far only deal with when operations take place. In order to specify where operations can execute one uses the CC++ construct processor objects and global pointers.

A processor object is an instance of a program in a CC++ computation, so a CC++ computation can have multiple instances of more than one program. Definition of a processor object is done by linking one or more translations with a global class declaration. A global class declaration is just like a regular class declaration except for the keyword global:

```cpp
    global class Program A {
        public:
            function1();
    };
```

Each global class declaration is a definition of a program. Processor objects are pointers to programs and are created with the new operator and destroyed with the delete operator. Declaration of a global pointer looks like this:

```cpp
    Program A * global pA1;
```

### 3.1.5 Conclusions

As we have seen, by just adding six keywords we have been presented with a C++ well suited for concurrent programming. It gives the programmer possibilities to provide parallelism in both structured and unstructured manners, the option to control access to data and means to specifying where programs shall run. These are good options to have in scientific computing where there is need to split the tasks and solve these effectively in parallel with control over shared data.
3.2 Object Oriented Scientific Computing

Scientific computing was formerly known as numerical analysis [23]. Numerical analysis is concerned with the design and analysis of algorithms for solving mathematical problems. It deals with quantities that are continuous, as opposed to discrete, and it is concerned with functions and equations whose underlying variables (time, distance, velocity, temperature, pressure, etc.) are continuous in nature. In other words, scientific computing is the technique used to obtain solutions of mathematical models that represent some physical situation.

Most problems in continuous mathematics cannot be solved in a finite number of steps and thus are solved by an iterative process that eventually approximates a solution. One of the most important aspects of scientific computing is therefore finding fast convergent iterative algorithms and evaluating the accuracy of the resulting approximation. Consequently, it is important to have efficient algorithms that are as reliable and accurate as possible despite the variations in the approximations made along the way.

The basic general strategy to solving a computational problem is to replace a difficult problem with an easier one that has the same solution. The division of the problem domain into smaller domains that can be computed independently is one such strategy.

3.2.1 Software for scientific computing

In [22] four criteria for good programs for scientific computing are mentioned:

1. Reliability: The code does not have errors and can be trusted to compute what it is supposed to compute.

2. Robustness: The code has a wide range of applicability as well as the ability to detect bad data and other abnormal situations and deal with them in a manner that is satisfactory to the user.

3. Portability: The code can be transferred from one computer to another with a minimum of effort and without losing reliability.

4. Maintainability: Corrections or enhancements to the code should be possible with minimum effort.

3.2.2 Object-orientated frameworks in scientific computing

Having presented the basics of object-oriented programming and scientific computing, we now give a brief presentation of some frameworks used in scientific computing.

Overture Overture is an object-oriented environment for solving partial differential equations on serial and parallel architectures [36]. It contains a wide variety of C++ classes for writing PDE-solvers, for easily computing finite-difference and finite-volume discretizations on curvilinear grids, for solving matrix problems, and for plotting and saving results.
The main categories of classes in Overture are:

- Arrays: these are from the A++/P++ class library and define multidimensional arrays for serial and parallel array operations.
- Mappings: define transformations such as curves, surface, volumes and areas.
- Grids: define a discrete representation of a mapping.
- Grid functions: represent solution values (e.g. density, pressure, and velocity) at each point on a grid.
- Operators: define discrete approximations to differential operators and boundary conditions for grid functions.
- Adaptive Mesh Refinement: AMR++ consists of four separate libraries that support block-structured adaptive mesh refinement.

Load-balancing tools are provided for automatic load-balancing of computations on the adaptive overlapping grid structure on parallel computers.

**POOMA** POOMA, Parallel Object-Oriented Methods and Applications, is an object-oriented framework implemented as a set of templated C++ class libraries. It is used in scientific computing applications that require the memory and computational speed provided by high-performance parallel computing. The framework defines an interface in which the users express the fundamental scientific content and numerical methods, and optionally give hints as to how to best decompose it across processors. Classes within the POOMA framework perform the necessary data decomposition and communications.

The POOMA framework is constructed as a hierarchical layering of classes [20]. This is done in order to exploit the efficient implementations in the lower levels of the framework, while preserving an interface relevant to the application problem domain at the highest level. There are five layers:

- Application Layer
- Component Layer
- Global Layer
- Parallel Abstraction Layer
- Local Layer

The classes high in the framework represent abstractions relevant to application domains, whereas the classes lower in the framework represent the abstractions of parallelism and efficient computational kernels. The Global and Local Layers work together to define Global Data Types (GDTs) that perform matrix, field and particle operations. The
interactions between the global and local classes is mediated by objects from the Parallel Abstraction Layer, which is responsible for capturing key abstractions of parallelism, such as interprocessor communication, domain decomposition and load balancing. The Component Layer is built upon the Global Layer and contains a rich set of algorithms directly relevant to scientific simulations (e.g. interpolations, Fast Fourier Transforms and Krylov solvers). Classes in the Component Layer are generic and reusable across problem domains, whereas members of the Application Layer represent a configuration of Component and Global objects interspersed with application specific objects.

**Diffpack** Diffpack is an object-oriented problem-solving environment for the numerical solution of partial differential equations (PDE’s)[34]. Diffpack is designed to allow easy modification and combination of all numerical building blocks making up an application. It has heavy focus on the flexible construction, modelling power and computational efficiency of the numerical kernel. Diffpack is organised as a collection of C++ libraries with classes, functions and utility programs.

The Diffpack software architecture follows closely the mathematical concepts involved in the PDE solution process. For instance, when solving a PDE by a finite element method (FEM), it is from a mathematical point of view natural to rephrase the PDE in terms of its weak formulation. Such weak formulations involve the evaluation of (possibly complicated) integrals over the computational domain. It turns out that the general FEM algorithm is essentially a loop over all elements in the grid, where for each element the local contribution to the discretized problem is computed by numerical integration. The integration method itself usually takes the form of an inner loop over a collection of integration points, where the integrand needs to be evaluated for each point picked.

Observing that the PDE enters the calculation only in terms of the point samples of the integrand, this mathematical/numerical view of the solution process is mirrored in the Diffpack software. As an application programmer, the user supplies the necessary expressions to evaluate the integrand of the particular PDE for a given integration point. The rest of the solution process is automatically handled by the Diffpack library components, all the way from element definitions via numerical integration methods and matrix assembly, to the solution of the resulting algebraic problems. In addition, the user might have to supply the necessary input data concerning boundary conditions and initial conditions.

In Diffpack, object-oriented mechanisms are used only for tasks with minimal influence on computational efficiency. In particular, low-level, computationally intensive operations are always performed in a FORTRAN or C type style, while the object-oriented principles are mainly used for higher-level administrative tasks. Although object-oriented, this means that most Diffpack applications will spend almost all CPU time in functions containing very basic loops over array structures, as is the case in well written numerical FORTRAN codes. This ensures high computational performance of Diffpack applications.
3.2.3 Object Oriented Programming and PDE's

A partial differential equation (PDE) is a mathematical relation that often expresses a general law of nature. Usually it is not possible to obtain exact mathematical solutions to PDEs, and computer programs are constructed that use some kind of numerical method to solve PDEs approximately.

Traditionally, PDE solvers were written in Fortran. [21] presents a "Navier-Stokes solver" as a huge Fortran program which contains the mathematical model, boundary conditions, grids, grid functions and specifications of the numerical method to be solved. All these components of the program are tightly linked to each other, making it difficult to replace one component without having to rewrite major parts of the code. The construction of such a program is also a major task.

The introduction of parallel computers has made the drawback of the traditional approach even more apparent. In the traditional PDE solvers, the data structures are visible at the top level of the program. For portability reasons this is an unsatisfactory approach in parallel programming.

So, in conclusion, there is a need for programming languages that provide a raised level of abstraction, something one gets from using an OO-language. Both [11] and [13] present approaches to dealing with PDE’s in an object-oriented manner. An important aspect is to separate the mathematical model (the PDE to be solved) from the numerical model (solution method). When using parallel processors the idea is to divide the original solution domain of the PDE into many subdomains and assign these to the different processors. Numerical operations associated with these subdomains are typically local and sequential, and can therefore be carried out concurrently, under some global administration.

3.3 C++ & Fortran in scientific computing

For several years there has been raised questions on whether C++ is fast enough for scientific computing, and several reports have been published on the subject. The reason that the C++ vs Fortran issue is interesting, is that Fortran is arguably the performance leader on scientific applications. One of the first to conduct quantitative tests on C++ vs C vs Fortran performance was Haney [4]. In 1994 he conducted three fairly simple examples to compare C, C++ and Fortran (F77)-code on a quantitative basis. Later, similar tests were performed by Robison [5], and then by Veldhuizen. Here we try to give a summary of the tests, and the conclusions that can be drawn from them.

3.3.1 Haney’s Tests

The tests consisted of comparative performance benchmarks that were obtained by running C, C++, and Fortran 77 versions of three kernels. The kernels were matrix multiplications of real matrices, matrix multiplication of complex matrices, and a vector expression.
The tests where conducted on several platforms: Sun SPARCstation 10/41, IBM RS6000, HP 755, DEC Alpha, SGI IRIS R4000, Cray C90/16256.

Test 1 Minimal use of C++ features  The first test measured the speed of code that only used a small subset of C++: A simple class called RealMatrix provided storage management and indexing into matrices that are stored according to Fortran conventions. The test consisted of multiplying two RealMatrix objects a and b. The matrices were 100x100 in size.

The performance was largely determined by how well the compilers optimize the innermost of the three loops that were required to perform the multiplication. In C++, this loop was written:

```c
for (i = 1 ; i <= M ; i++)
    t (i, j) += temp * a (i, k);
```

In C the indexing is handled explicitly:

```c
for (i = 1 ; i <= M ; i++)
    t(i - 1 + M * (j - 1)) += temp * a[i - 1 + M * (k - 1)]
```

And the Fortran version:

```fortran
do 3000 i = 1, m
    t(i, j) = t(i, j) + temp * a(i, k)
3000 continue
```

In all three cases, `temp` was previously initialized to `b(k, j).

The tests showed that on all platforms, except Cray 90, C++ with linear addressing gave execution times that were approx 400%-500% longer than Fortrans. On the Cray C90, it was only approx. 20% slower than Fortran.

C++ with indirect addressing gave, in general, somewhat improved results compared to C++ with linear addressing. Except for the Cray C90, the execution times were approximately 300% longer than for Fortran. On the Cray however, execution times rose to about 100% of Fortran. We found this a bit surprising, since it goes against the general trend. Could this somehow be connected to the architecture of the Cray, compared to the other platforms? Unfortunately, Haney gives no explanation to this.

C code proved to execute much faster. Results showed that for the Cray and IBM, C execution times were about the same as for the compiled Fortran. On DEC Alpha, SPARC 10/41, and SGI R4000 execution times were approx 60%-70% longer, and for the HP 755 the execution time was approx 100% longer.

Haney explains the differences with optimizations in the assembly code. He points out that in the C version of the innermost loop, the two multiplications that involve M do not depend on the loop index i. The multiplications can therefore be hoisted out of the loop. This is called strength reduction, and it is an important optimization that is
required if good performance is to be obtained. When examining the Sun assembly code, Haney found that it is not done for the C++ version, but is performed for the C version.

The difference in linear addressing and indirect addressing refers to how the CPU handles the addressing of the matrix data. If we compare the C code (of the innermost loop) with an equivalent C-like pseudocode for the C++ loop:

```c
for (i = 1 ; i <= M ; i++)
  *(t->d[i - 1 + t->nRows * (j -1)]) += temp *
  *(a->d[i - 1 + a->nRows * (k -1)]);
```

The pointer references to the data members use linear addressing. The problem with it is that it inhibits strength reduction. It is possible to rewrite the code, so that it precalculates the addresses of the column starts, storing them in an array. This is referred to as indirect addressing. The fact that the addresses are precalculated will speed up the loop, but it is not a optimal solution. It complicates other matrix operations, it is wasteful of storage, and it doesn’t scale well with higher-dimension arrays.

**Test 2 Complex Arithmetic** The second test measured the cost of using a Complex class and operator overloading to simplify complex-number arithmetic. The test consists of multiplying two complex matrices. The C++ and Fortran versions of the innermost loops looked identical to the ones shown above. The C version of the innermost loops was written
for (i = 0 ; i <= M ; i++)
{
    t[i - 1 + M * (j - 1)].re += temp.re * a[i - 1 + M * (k - 1)].re -
    temp.im * a[i - 1 + M * (k - 1)].im;
    t[i - 1 + M * (j - 1)].im += temp.re * a[i - 1 + M * (k - 1)].im -
    temp.im * a[i - 1 + M * (k - 1)].re;
}

For C++ with linear addressing the execution times were between 225% - 350% longer
than for the Fortran code. On the HP 755 the execution time was about 700% longer,
and on the Cray C90 the execution time was approximately the same as for Fortran.

For C++ with indirect addressing the results again showed improved performance in
the region of 10%-25%. Except for the HP 755 and the Cray C90, the execution times
were between 150%-200% of Fortran. For the HP the execution time was over 500%
longer, and for the Cray it was approx the same as for the Fortran Code.

For C code the execution time was 50% longer on the SPARC, IBM, and SGI, 125% longer
on DEC, and 200% longer on the HP 755.

Haney states that one can gain some insight into the results by looking at output from
cfront, the C++→C translator that forms the basis for the C++ compiling systems on
many computers, including the C90. It turns out that the C++ version of the innermost
loop expands into a rather impressive array of casts and comma operators, requiring
some 30 lines of code, and these complicate optimization of the code.
Figure 3.3: From [4]. Speed comparison of C, C++, and Fortran code for vector calculations.

**Test 3 Vector Operations** The third test measured the cost of using classes and operator overloading to support arithmetic operations that involved complete arrays. Haney presents results for two versions of C++ code: one using a non-optimised RealVector class, and another using an optimized RealVector class. The optimizations consisted of eliminating the temporary creation (and subsequent destruction) of vectors. In addition, results for C code is presented. As always, the results are relative to similar Fortran code.

The non-optimised code is overall much slower than the optimised C++, C, and Fortran code. The best relative performance is on the SPARC and SGI R4000, where the compiled C++ code takes 150% longer to execute than the compiled Fortran code. On IBM and HP the code takes approx 290% longer to execute, and on the DEC almost 500% longer. The worst relative performance is on the Cray, where the C++ code takes 25 times as long as the Fortran to execute. The optimised C++ code takes between 75%-150% longer than Fortran code to execute on SPARC, IBM, HP and DEC. On SGI it is only marginally slower than Fortran, with approx 15% longer execution time, and on the Cray it takes almost 300% longer to execute than the Fortran code, although it is much faster than the non-optimised code. The C code has very similar performance to the Fortran code. On the Cray it even runs a bit faster. But in general the execution times lie between 0%-25% of the compiled Fortran code.

**3.3.2 Preliminary Conclusions**

Haney concludes his report that C++ programs that does serial calculations with arrays on workstations and vector supercomputers will probably be slower than a similar program written in C or Fortran. In addition, the difference in performance may be
uncomfortably large.

In our opinion, Haney has presented his results in a convincing manner, and we have confidence that they are sound, from a scientific perspective. However, it’s also interesting to note the difference in expressiveness between C++, and Fortran / C. The fact that what one can write on 1 line of code in C++ takes approximately 30 lines of code and 10 loops in C or Fortran, makes C++ a lot more attractive than Fortran/C with regards to code readability.

However, according to Robison [5], many of these problems do not lie with C++ itself. The problem, he states, is that the C++ style tends to differ from the C style. While C emphasizes scalar computations, C++ in contrast focuses on object computations. It is known that all modern compilers try to allocate scalar variables to registers when profitable. However, many C compilers refuse to store nonscalar variables in registers. Consequently, C optimisers fare poorly on C++ style. So, bottom line, the problem does not lie in the language in itself, but in the compilers.

In the paper Robison states that he and his colleagues at KAI (Kuck & Associates Inc.) have examined the foundations of classical optimisations for Fortran, and rebuilt them to work on C++ programs. The resulting compiler, known as KCC, was then put to work on Haney’s benchmarks. It is worth noting that Robison does not include results for the Cray C90 in his report. The reason for this is unclear. Since all the other platforms are workstations, and the Cray is a supercomputer, we assume that the reason for its absence is a matter of availability.

### 3.3.3 Robison’s Tests

**Test 1 Minimal use of C++ features** The results presented show that by using KCC, C++ performance takes a veritable leap. It is worth noting that the results for C++ with indirect addressing, C++ with linear addressing, and C, are almost the same for all platforms, so well be referring to them as one.

In Haney’s study, the C++ codes ran approx 300%-500% slower, depending on whether linear addressing or indirect addressing was used. The only exception was the Cray C90, where the execution time was somewhat better.

With the use of KCC the execution times drop to about 80% longer than Fortran code on the SPARC, approx 30% longer on the IBM, approx 30% longer on HP, about 80% longer on DEC, and almost 50% longer on the SGI platform. Overall, these are considerable improvements from the results published by Haney. It can be noted that the performance of the C++ code with linear addressing and the C++ code with indirect addressing no longer differs in a substantial way. The extra effort of adding indirect addressing therefore no longer offers significant improvement. This means that the programmer can write his entire code C++ style. He doesnt necessarily need to write the performance-critical parts of code in C style.

**Test 2 Complex Arithmetic** The second test that Haney performed measured the time to multiply single-precision 100x100 complex matrices. It then adds a complexity that C compilers are ill prepared for: temporary objects. As we stated earlier, C compilers
try to put scalar values into registers, if profitable. The temporary scalar values that arise from traditional arithmetic are put into registers instead of memory or, if they are constant, perhaps optimised away. But the fact that C++ offers operator overloading introduces new kinds of arithmetic and temporaries. Compilers used to dealing with only scalar temporaries therefore fail to put the complex temporaries into registers.

The tests run by Robison with the KCC compiler practically eliminates the penalty relative to C. In Haney’s original study, the C++ codes ran at least 200% slower than Fortran. For the SPARC platform, the execution times for C++ with linear addressing and with indirect addressing are approx 20% slower than Fortran. For C it is about 60% slower. On IBM, the results for both C++ versions are about 10%-15% slower, while C is 50% slower the Fortran. The HP platform provides the largest improvements, with a rise from about 700% slower with C++ (linear addressing), to about 50% slower than the Fortran code. With C++ (indirect addressing) the execution time rise from about 500% slower than Fortran, to 50% slower. With C code the result is about 110% slower than Fortran code, compared to 200% slower with ordinary compilers. On the DEC Alpha the two C++ variants both perform approx 60% slower than Fortran, and C approx 150% slower. On SGI R4000 the execution time for all variants are about 40% slower than Fortran code.

As we can see, the C++ code runs faster than C code on almost all platforms. This is quite interesting, when we compare to the previous tests by Haney. Apparently, KCC offers significant improvement over older compilers. It would have been interesting to see how the Cray C90 would have fared in this test. As we can see from the corresponding test by Haney, the differences between the three kernels were very small, although the C++ code was slower than the C code. It is tempting to speculate that the C++ code would have surpassed the C code on that platform also.
Test 3 Vector Operations  The third Haney test deals with vector expressions. This test required that the compiler transformed the C++ version to a C version by fusing loops. This is a common optimisation in Fortran, but in C and C++ the liberal aliasing rules put it beyond the state of the art for C and C++ in practical programs. However, Todd Veldhiuzen [14] has presented a method called template expressions. They exploit the C++ template mechanism to perform compile-time transformations of code. Haney has rewritten the C++ version to use template expressions, and the combination of KCC and template expressions closes the performance gap between C++ and C almost completely. The difference between C and C++ with template expressions is, except for the SGI, negligible.

Be aware that “optimized C++ code” = with template expressions, and “original code” = Haney’s optimized C++.

For the SPARCstation, the original C++ execution time is about 150% slower than Fortran, for the two other about 100% slower than Fortran. (Note: It seems that the performance for C is slower than in Haney’s tests!) On the IBM RS6000 the original C++ execution time is reduced from approx 300% slower, to 60% slower than Fortran code. Optimized C++ and C code is about 20% slower, compared to approx 30% and 5% slower, respectively. (Again it seems that the execution time of the C code has increased!) On the HP 755 the optimized C++ code and C code is marginally (approx 5%) faster than Fortran code. The original C++ code is almost 50% slower. Also the DEC system shows improvements. The C++ and C code execution time is about 75% slower, compared to over 100% and approx 30%, respectively. For the SGI, the difference on optimized C++ and C code is marginal. The original C++ code, however, goes from over 100% slower, to about 50% slower execution time compared to Fortran code.
3.3.4 Preliminary Conclusions

Robison concludes that, at this time, it cannot always be said that C++ is fast enough. He points to the fact that improved compilers and the new programming technique of template expressions remove the extreme performance penalty that C++ had in Haney's three tests. It is also worth mentioning that C++, through i.e. objects and overloaded operators, offer an easier programming experience compared to the soup of simple numeric variables.

3.3.5 Veldhuizen & Jernigan's Tests

By 1997 benchmarks showed that C++ was steadily closing the gap on Fortran, and Veldhuizen and Jernigan [15] posed the question: "Will C++ be faster than Fortran?". They pointed to new C++ compilers like KAI C++ (KCC) and Intel C++, and to two new programming techniques (expression templates and template metaprograms) as the reasons for this improvement.

In their paper, Velduizen and Jernigan present a benchmark\(^1\) for a 2D acoustic wave equation, using the Blitz++ library.

\(^1\)The benchmark was run on a 100 MHz IBM RS/6000 using KAI C++, XL Fortran 77, and XL Fortran 90.

Figure 3.6: From [5]. Speed comparison between C++, C, and Fortran code.
<table>
<thead>
<tr>
<th>Untuned Versions</th>
<th>Time</th>
<th>Mflops/s</th>
<th>Code Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran 90</td>
<td>93.3s</td>
<td>9.7</td>
<td>12</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>53.2s</td>
<td>17.0</td>
<td>21</td>
</tr>
<tr>
<td>Blitz++</td>
<td>57.8</td>
<td>15.7</td>
<td>9</td>
</tr>
<tr>
<td>Older C++ Math Library</td>
<td>418.2s</td>
<td>2.1</td>
<td>9</td>
</tr>
<tr>
<td>Tuned versions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortran 90</td>
<td>41.1s</td>
<td>22.1</td>
<td>19</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>27.3s</td>
<td>33.2</td>
<td>29</td>
</tr>
<tr>
<td>Blitz++</td>
<td>31.6s</td>
<td>28.7</td>
<td>8</td>
</tr>
<tr>
<td>Older C++ math library</td>
<td>361.9s</td>
<td>2.4</td>
<td>11</td>
</tr>
</tbody>
</table>

As we can see, the Blitz++ library gives a significant improvement over the older C++ math library. For the untuned version it is over 700% faster, and for the tuned version it is over 1100% faster. The difference between Fortran and C++ (using the Blitz++ library) has shrunk to just a few seconds. In addition it is interesting to note that the number of code statements is lower in C++ than in Fortran. The Blitz++ versions are highly expressive (requiring few lines of code to implement the benchmark), and compete with the performance of the Fortran versions. The tuned Blitz++ and Fortran 77 implementations do exactly the same optimizations. The tuned Fortran 90 version avoids array copying, but leaves the traversal order up to the compiler. Although the Fortran 77 version is slightly faster than Blitz++, Veldhuizen and Jernigan state that it took 30 lines of source code and several days of painstaking tuning to achieve the result.

### 3.3.6 Conclusions

In their conclusion, Veldhuizen and Jernigan make some very good remarks. They state:

C++ is now ready to compete with the performance of Fortran. The performance problems have been solved by a combination of better compilers and template techniques. It is possible that C++ will be faster than Fortran for some applications, since:

- Fortran 90 has the array abstraction built into the language. This limits the ability of the compiler to generate efficient code to expressions involving simple (e.g. dense, asymmetric) arrays. The expression templates technique in C++ has no such limitations; it can be applied to more complicated (e.g. sparse, jagged-edged) arrays.

- Template metaprograms can generate specialized algorithms well beyond the ability of current Fortran compilers.

- The generic programming features of C++ allow the creation of generic frameworks for common scientific computing tasks (such as solving finite-differenced PDE’s). Such frameworks can implement complex optimizations (such as iteration-space tilings).

- The object-oriented nature of C++ encourages favourable memory access patterns.
3.3.7 Why Was C++ Slower?

As we have seen, C++ has historically been slower than Fortran on scientific computing. Here we recite what [15] sees as the primary factors that contributed to the gap in performance between C++ and Fortran.

**Pairwise expression evaluation** The use of operator overloading in C++ permits a natural notation for array operations. One can write \( w = x + y + z \); where \( w, x, y, \) and \( z \) are vectors. However, overloaded operators in C++ are always evaluated in a pairwise manner. In a straightforward implementation, evaluation of \( w = x + y + z \) will result in two temporary vectors and three loops being generated. For large vectors, speeds of about 40% of Fortran can be achieved (depending on the expression). For tiny vectors \((N < 10)\), the cost of allocating temporaries dominates, and the speed is 5-10% of Fortran.

This problem has been solved by the expression templates technique [14]. Expression templates are used to build parse trees of expressions at compile time; the parse trees are encoded as type names. Once the parse tree has been encoded, it can be manipulated in interesting ways. One application of expression templates solves the pairwise evaluation problem: Code such as \( w = x + y + z \); is transformed into

\[
\text{for (int } i=0 ; i < N ; ++i)
\text{ w[i] = x[i] + y[i] + z[i]}
\]

Expression templates have some overhead associated with the construction of the parse tree. However, compilers like KAI C++ optimizes away this overhead completely, resulting in code that is just as fast as Fortran 77.

Although Fortran 90 supports operator overloading, there’s no comparable mechanism to expression templates. This limits the generation of efficient kernels by Fortran 90 compilers to expressions involving arrays which are dense, asymmetric, box-shaped, and linearly-addressable. The expression templates mechanism in C++ has no such limitations: it can generate efficient kernels for sparse, symmetric, jagged-edged, and/or nonlinearly-addressable arrays.

**Abstraction penalty** The promise of C++ lies in its flexible, maintainable, abstract code. In the past, abstraction in C++ has been been accompanied by a performance loss often referred to as the abstraction penalty. The abstraction penalty is what the programmer pays (in terms of performance) for writing at a more abstract level. It is measured as a ratio:

\[
\frac{\text{time(abstract version)}}{\text{time(low level version)}}
\]

E.g., an abstraction penalty of 2 means that the abstract program runs twice as slowly.

The abstraction penalty was caused by the inability of compilers to do C++ specific optimizations. The features of C++ that this concerns are small objects, virtual functions
and exceptions. More recent compilers, like KAI C++ and Intel C++, reduce this penalty substantially [16]. In addition, experience with Blitz++ has shown that performance can be tweaked to match Fortran77/90.

**Aliasing** Aliasing is the problem that two pointers could potentially point to the same data. It’s a problem primarily for code which is written at a low level of abstraction, and that uses pointers directly. If the code is written at a high level of abstraction using a class library, it’s generally possible for the library designer to avoid aliasing problems, for example by loading data into local variables which are guaranteed alias-free.

C++ compilers must assume that two pointers might refer to the same data, preventing certain optimizations from taking place. Fortran compilers don’t have this problem, since the language doesn’t permit aliasing ambiguities.

The Numerical C Extensions Group (NCEG) introduced a pointer qualifier `restrict` to advise the compiler that no aliasing is possible. A proposal to add the keyword to the ISO/ANSI C++ standard, however, was rejected, but some C++ compilers support it anyway (KAI C++, Cray C++)

### 3.3.8 How Might C++ Be Faster?

C++ is steadily approaching Fortran, and threatening it’s status as performance leader on scientific calculations. We have tried to find some factors that may contribute to C++ taking over Fortrans performance crown. In this chapter we’ll first mention some general contributing factors, then some more detailed examples like the Blitz++ library and expression templates.

**Abstraction can make optimization easier** Optimization of low-level languages like Fortran 77 requires a transformational approach. This means that the compilers first need to understand *what* the program is supposed to accomplish, and then transform it into a faster (but equivalent) program. To optimize code written at a higher level of abstraction is often easier, since a generative approach is possible.

In Fortran 90/95, the array abstraction is built into the language; The compiler generates optimized low-level code to implement the high-level program. C++ has an advantage because it provides library designers with the tools they need to build their own abstractions; optimized code can be generated by the library itself using the template techniques.

**Directed algorithm specialization** It’s often possible to speed up an algorithm by specializing it for a particular situation, and it is known that scientific computing algorithms can benefit greatly from this specialization.

There exists two approaches to generate specialized algorithms: partial evaluation and generative metaprogramming. Partial evaluators use a transformational approach to optimization. As mentioned earlier, this implies that a program is analyzed, and selected algorithms are transformed into specialized ones. Many compilers perform very limited forms of partial evaluation.
Figure 3.7: From [15]. Creating specialized algorithms for matrix multiplication with template metaprogams and Blitz++

Generative metaprogramming is a term that implies that the program generates source code for specialized algorithms. Examples include source code for fixed-size, un-rolled Fast Fourier Transforms and matrix multiplication code for matrices with known sparse structure. It turns out that C++ offers crude generative metaprogramming facilities quite by accident, as a byproduct of its template instantiation mechanism [17]. These template metaprogams are capable of doing arbitrary computations at compile time, and produce compilable C++ code as their output.

An important distinction between template metaprogams and partial evaluation is that the algorithm specializations produced by template metaprogams are unlimited in complexity and directed (one has complete control over exactly which optimizations are performed).

Encapsulation improves locality of reference. Encapsulation is a basic idea in object-oriented design. One takes the data which is closely related, and pack it as an object. A side benefit of encapsulation is that it encourages locality of reference, resulting in favorable memory access patterns [5]. This has been nicely illustrated by the Edinburgh Parallel Computing Center (EPCC) in [18]. Veldhuizen and Jernigan [15] has used the Blitz++ library; and a class that uses template metaprogams to create specialized algorithms for matrix multiplication, to run a similar benchmark (with Blitz++ and Fortran). They found that encapsulation improved the performance by about 12%. What is really interesting is that even though Veldhuizen and Jernigan hand-tuned the Fortran-version extensively, they were unable to achieve the performance of either C++ (Blitz++) version.
Expression templates

When programming it is common to pass an expression to a function. In C, expressions are passed using a pointer to a callback function that contains the function. For example, the standard C library routines qsort(), lsearch(), and bsearch() accept an argument int (*cmp)(void*, void*) which points to a user-defined function to compare two elements. Another example is passing mathematical expressions to functions. The problem with callback functions is that repeated calling generates a lot of overhead, especially if the expression that the function evaluates is simple. The technique of expression templates allows expressions to be passed to functions as an argument and inlined in the function body.

Example of expression template code  To illustrate how expression template code looks, we present two small examples, both from [14].

The first example illustrates how the return types of operators cause the compiler to infer expression types. Here we have a operator/() that produces a type representing a division between two subexpressions DExpr<A> and DExpr<B>.

```cpp
template<class A, class B>
DExpr<DBinExprOp<DExpr<A>, DExpr<B>, DApDivide> >
operator/(const DExpr<A>& a, const DExpr<A>& b)
{
    typedef DBinExprOp<DExpr<A>, DExpr<B>, DApDivide> ExprT;
    return DExpr<ExprT>(ExprT(a, b));
}
```

The second example illustrates how expressions are built.

```cpp
DVec a(10), b(10);
y = a+b;
```

Both a and b are of type DVec, and the appropriate operator+() is:

```cpp
DVecExpres<DVecBinExprOp<DVec::iterT, DVec::iterT, DApAdd> >
operator+(const DVec& a, const DVec& b)
{
    typedef DVecBinExprOp<DVec::iterT, DVec::iterT, DApAdd> ExprT;
    return DVecExpr<ExprT>(ExprT(a.begin(), b.begin()));
}
```

Blitz++  Blitz++\textsuperscript{2} is a numerical library, designed to improve the performance of C++. The core of the library has been written by Todd Veldhuizen, but several others have contributed to its development\textsuperscript{3}The primary goal for the Blitz-project is to provide a solid

\textsuperscript{2}http://ooum.cis.ksu.edu/blitz/
\textsuperscript{3}http://ooum.cis.ksu.edu/blitz/contribute/thanks.html
“base environment” of arrays, matrices and vectors for scientific computing in C++. This to develop techniques that will enable C++ to rival, and in some cases exceed, the speed of Fortran for numerical computing, while maintaining it’s object-oriented interface. Another goal of Blitz++ is to extend the conventional dense array model to incorporate new and useful features. Some examples of such extensions are flexible storage formats, tensor notation and index placeholders.

**Philosophy: The library that thinks it is a compiler** Scientific computing requires domain-specific abstractions, such as arrays, matrices, and tensors. Building such abstractions into a language (such as arrays in Fortran 90) can result in fast code, but may also be limiting: such abstractions are hard to extend or modify, and economics restrict the number of features which may be included in a commercial compiler.

An alternative is to use languages which allow library developers to create their own abstractions. Many problem domains which have been addressed in the past by custom languages (interval arithmetic, sparse arrays, adaptive mesh refinement) have now been successfully tackled by C++ libraries. Unfortunately, compilers have difficulty optimizing such libraries, because they lack semantic knowledge of the abstractions: instead of seeing sparse array operations, they see pointers and loops.

The solution may be to move high-level optimizations out of compilers and into libraries. The Blitz++ library demonstrates how this may be done in C++. The mechanisms are somewhat crude, but the results are appealing: Blitz++ arrays offer functionality and efficiency competitive with Fortran 90, but without any language extensions. The Blitz++ library is able to parse and analyze array expressions at compile time, and performs loop transformations which have until now been the responsibility of optimizing compilers.

**Features** Blitz++ can be said to be an *active library*. While a typical library can be seen as just a passive collection of routines, an active library takes an active role in the compilation process. This, among other things, mean that the responsibility for high-level optimization is shifted from the compiler to the library. Some other features of an active library are:

- Applying optimizations and generating specialized algorithms.
- Configuring and tuning themselves for a target architecture.
- Interacting with tools (profilers and debuggers) and issuing compile-time errors.
- Generating components to fulfill specified requirements.

**Example of Blitz++ Code**

```
Array <float,3> A(64,64,64)  // A 64x64x64 array
// C refers to the 2D slice A (10..63, 15, 0..63)
Array <float,2> C = A (Range(10, toEnd), 15, Range::All());
Array <float,1> D = A (Range(fromStart, 30), 15, 20);
```

42
Optimizations done by Blitz++

- Loop interchange: permute loops so that innermost loop is associated with smallest stride.
- Collapsing inner loops: If the arrays are contiguously stored, collapse the inner loops and treat as 1D arrays.
- Partial unrolling of the inner loop to expose low-level parallelism.
- Common stride optimizations: Exploit common strides and unit strides, hoist invariant stride computations.
- Detect stencils, and perform tiling for good cache use.

Conclusion  Blitz++ and expression templates give significant performance gains, and raises C++ performance to Fortran-levels and beyond. However, we believe that there is a price to be paid in terms of code readability and higher learning curve. Expression templates in particular produce complicated code in terms of readability. For somewhat inexperienced programmers this will present a stumbling block for writing efficient scientific applications.

3.4 Java for scientific computing

Java is primarily intended for developing Internet-based applications. As a language for scientific computing applications it still ranks behind C/C++ and Fortran but improvements are continuously being made. The main drawback for Java in scientific computing

43
is its speed due to the machine-independent bytecode for the JVM. Speed was, however, also a problem for both Fortran and C++ in its infancy but the rapid development of improved compilers dealt with this issue. This will most likely also be the case with Java. Because Java supports interfaces to C/C++ it is also possible for the developer to build the application in Java and then reimplement the performance-critical parts in C++.

The Java Grande Forum\(^4\) are concerned with issues regarding the use of Java in applications, scientific or industrial, that require a large amount of computing resources to solve one or more problems. They are trying to suggest improvements to Java in its language, libraries and perhaps the virtual machine that would be beneficial for such applications. The Numerics Working Group\(^5\) of the Java Grande Forum has as its goal to assess the suitability of Java for numerical computing. If Java is to become an environment for high-performance scientific applications, it must show performance comparable with what is currently achieved with other languages (Fortran or C), and it must have language features and core libraries that enable the convenient expression of mathematical algorithms. They identified five critical issues that are not currently satisfactory supported by Java’s design and implementation\([40]\).

### 3.4.1 Critical Issues

#### Complex Arithmetic

Complex numbers are not represented as primitive types in Java. Providing a \textit{complex} class using existing Java object mechanisms solves this, but not in a satisfactory manner. The object overhead of \textit{complex} methods make them inefficient, the semantics of complex objects are different from those of \textit{int} or \textit{double} (operators \texttt{-} and \texttt{==} manipulate references rather than values), and use of method calls for simple arithmetic operations lead to "ugly" code. The obvious solution would be to add \textit{complex} as a primitive in the language but this is not as easy as it first seems. The problem lies in adding support for the complex type in the JVM. Either the JVM would have to directly support the complex primitive, or complex expressions and variables in a Java program would have to be mapped into existing JVM instructions in a predetermined way. It is not recommended to make fundamental changes to the JVM as long as the given feature can be implemented with the existing functionality. The translation of a complex type into two double values also represents difficulties, both in the passing of a complex numbers to a method and the returning of complex numbers from a method. If the complex parameter is passed to a method as a pair of double parameters it would not be possible in the bytecode to distinguish between the method with the complex parameter and a method with the same name that takes two double parameters. As for the returning of a complex value, the JVM cannot return a pair of doubles unless they are contained within an array. This would cause additional memory allocation and copying overhead. In order to make Java extensible enough for complex and other numeric types (e.g. interval and decimal arithmetic) the two following capabilities are needed: lightweight classes and operator

\(^4\)http://www.javagrande.org

\(^5\)http://www.math.nist.gov/javanumerics
overloading.

**Lightweight classes**

The idea of lightweight classes is to introduce a new kind of Java object. The goal of lightweight classes is to have lightweight objects that comes with value semantics, where the `=` operator performs a deep copy (instead of changing pointers) and the `===` operator performs a deep comparison (instead of comparing pointer values). Also, a variable of a lightweight class is never null, such variables always have a value. The implementation implications for lightweight classes are discussed further in [40]. The essence is that complex number can be implemented with lightweight classes instead of ordinary Java objects. Using standard Java objects a complex class would need 32 bytes for representation, two 8-byte parts for the real and imaginary values and a 16-bit object descriptor that is present in all Java objects. Reducing this size in addition to avoiding other performance overhead of Java classes is part of the goal for lightweight classes.

**Operator overloading**

With operator overloading user defined numeric types such as complex can be used reasonably. This makes it easier to write, read and understand numeric code. Without operator overloading a simple statement such as:

\[ a = b+c* d \]

might be expressed like this:

\[ a.assign(b.plus(c.times(d))) \]

No particular solution exists to get around this problem in Java

**Use of floating point hardware**

Achieving the highest possible level of performance on numerical codes typically requires exploiting unique floating-point features in each processor. This thought is not very compatible with the Java goal of exact reproducibility of results on every platform. One of the features not supported by Java is the associativity property of mathematical operators. This could lead to the expression \((a+b)+c\) producing a different rounded result than \(a+(b+c)\). Another example of unique floating point features is the *fused multiply-add*, or *fma*, that exist on some processors\(^6\). The *fma* enables the CPU to calculate a multiplication and addition in a single CPU-cycle, effectively doubling the floating point execution rate [in certain computations].

\(^6\)POWER and POWER2-processors. It is also available on the upcoming IA-64, produced by Intel and HP.
Multi-dimensional arrays

Java does not directly support arrays of rank greater than one. A two-dimensional array is represented as an array of arrays: an array whose elements are references to one-dimensional arrays. There are some important differences between multidimensional arrays and Java’s array of arrays structure. The statement:

\[
\text{double} 
\begin{bmatrix} \text{m} \\ \text{n} \end{bmatrix} \text{arr} = \text{new} \ \text{double}[\text{m}][\text{n}];
\]

does define a two-dimensional \( m \times n \) array, but with Java there is nothing preventing the array becoming jagged at some later stage in the code with for example a statement like this:

\[
\text{for(int } \text{i}=0; \text{i}<\text{arr.length-1}; \text{i}++)
\]
\[
\{
\text{arr[i]}=\text{new} \ \text{double}[\text{n}+\text{i}]\;
\}
\]

(The array could also have been created jagged initially).

Another difference from multidimensional arrays is that two or more rows of the Java array can be aliased to the same data:

\[
\text{for(int } \text{i}=0; \text{i}<\text{arr.length-1}; \text{i}+=2)
\]
\[
\{
\text{arr[i]}=\text{arr}[\text{i}+1] \;
\}
\]

When the compiler is given a reference of type \( \text{double}[//] \) it has no information about the shape of the array or any aliasing. It is also possible with general arrays using an array of type \( \text{Object} \). In such an array an element can be of an arbitrary type.

These features come with a performance price that is way too unacceptable for numerical computing. High performance computing with arrays requires that the array has immutable rank, shape and type. Without these properties optimising for performance cannot be done. For example, not having a fixed shape of the array results in Java having to check for out-of-bounds exceptions for each row in the array, which has large effects on performance. This would be avoided if it was guaranteed that the array is rectangular (all rows of the same length).

In [45] Moreira et al point to another problem with Java and arrays. Java requires every array reference of the form \( A[\sigma] \) to be tested for validity. This means that \( A \) cannot be a null pointer (i.e. it must point to an existing array), and \( \sigma \) must be a valid index of \( A \) (i.e. greater than or equal to 0, and less than the length of array \( A \)). If any of these conditions are violated, an exception must be raised. In addition to the discontinuity that the exception causes, the cost of running explicit tests for each reference is excessive. It is also very difficult to reorder computations across array references. Code reordering is an important optimization technique used by modern compilers.
Both [40] and [41] present the Array package for Java. Its goal is to overcome the limitations inherent in Java arrays and produce performance comparable with other languages, like Fortran and C. Functionality is provided by a set of basic methods and operations that are highly optimizable by the compiler. Arrays defined in the Array package have properties like a nonjagged rectangular shape and constant bounds that are easy to detect and use by an optimising compiler.

An Array is characterized by its rank, its elemental data type and its shape, which are all immutable properties. The naming of all Array classes is like this: \(<\text{type}>\text{Array}<\text{rank}>D\). So \texttt{doubleArray2D} is a two-dimensional array of double values. Access to elements in classes of Array is through \texttt{get} and \texttt{set} methods. The full set of methods is described in [41].

### 3.4.2 Performance of Java

Boisvert et.al [35] has studied the performance of Java in scientific computing in codes taken from the SciMark benchmark\(^7\), one of the most popular benchmarks for Java. The SciMark benchmark is made up of common computational kernels found in scientific applications: Fast Fourier Transforms (FFT), successive over-relaxation (SOR) iterations, Monte-Carlo quadrature, sparse matrix multiplication, and dense matrix factorization (LU) for the solution of linear systems. Each kernel typifies a different computational style with different memory access patterns and floating point manipulations.

They collected scores for over 1,000 different Java/hardware/operating-system combinations, from laptops to high-end workstations, representing a thorough sample of Java performance across the computational landscape.

Figure 1 shows the composite score (in Mflops\(^8\) ) of this benchmark on six different architectures, and illustrates the wide range in performance over common platforms. The first observation is that Java performance is closely tied to the implementation technology of the JVM, rather than the underlying hardware performance. From this figure we see that PC platforms typically outperform high-end workstations. To demonstrate the continuous improvement in virtual machine technology, Figure 2 illustrates the performance of progressively new versions of the Sun JVM on the same hardware platform.

### 3.4.3 Using libraries in Java

Libraries are a very powerful tool for easing the development of applications. It also provides extensions to the language and presents domain-specific functionality. Many libraries exist that are designed for use in scientific computing. These are, however, developed using C/C++ or Fortran and the question is then how Java best can utilize this?

The straightforward method is using the Java Native Interface, JNI. With JNI, Java programs can access native code libraries for the platform on which they execute. This approach makes available in Java a large body of tested and optimized libraries for

---

\(^7\) (http://math.nist.gov/scimark)
\(^8\) Million floating point operations per second
Figure 3.9: Performance of Java varies greatly across computing platforms. This difference is mainly due to different implementations of the JVM, rather than underlying hardware architecture.

Figure 3.10: Evolution of Java performance on the same computational platform: a 333 MHz Sun Ultra 10.
numerical computing. It has been used, in particular, to provide Java with access to MPI and LAPACK.

This approach has disadvantages [35], in the five areas: safety, robustness, reproducibility, portability, and performance.

- Safety: native code cannot typically be executed in an environment as controlled as Java, and therefore it is not as safe to the host computer.
- Robustness: native code does not include all the run-time validity and consistency checks of Java bytecode and, therefore, is less robust.
- Reproducibility: native code, even for a standard library, is likely to have small differences from platform to platform, which may result in different outcomes in each one.
- Portability: native code is not portable across machine architectures and operating systems.
- Performance: Invoking a native method from Java incurs on run-time overhead. If the granularity of the operation is large, then the cost of the invocation can be amortised. However, for simple operations the cost of going through JNI can completely dominate the execution time.

Because of all the drawbacks of using native libraries it makes more sense to pursue the thought of developing numerical libraries in Java. One of the first such libraries was the Java Numerical Library (JNL), which has been freely distributed by Visual Numerics since 1997. It provides basic facilities in linear algebra, special functions, and elementary statistics. In 1998, another proposal for a standardized linear algebra library called JAMA was developed by MathWorks and NIST. This library includes facilities for solving linear systems and least squares problems, computing standard matrix decompositions (LU, Cholesky, QR, SVD, eigenvalue), and for computing quantities such as norms, determinants, ranks, and inverses. OpeResearch.com has developed a substantial set of Java classes to aid in the development of science and engineering applications, with an emphasis on operations research. A fairly comprehensive listing of available class libraries for numerical computing can be found on the Java Numerics Web page\(^9\).

\(^9\)(http://math.nist.gov/javanumerics)
4 Case Study

To help better illustrate our discussion later on, we found it useful to have some sort of “toy problem”. With this we mean a small problem that could possibly highlight some of the issues we have looked into, while not being to complicated to implement.

In this chapter we will present our problem of choice. We will explain it in both mathematical terms and with pseudocode. Furthermore we will try to explain why we choose this problem, it’s relevance, and some of the problems associated with it.

4.1 Problem description

Our problem is the computation of the surface area for a three-dimensional figure defined by a mesh. A mesh can be one, two, or three-dimensional, and it is made up of vertices. These vertices then make up faces which in turn make up elements. The mesh can be regular and irregular. In a regular mesh all vertices are equally spaced, unlike an irregular mesh where vertices are randomly spaced. See figure 4.1(left) for an example of an irregular two-dimensional mesh. The elements of a mesh can be a variety of polygonal figures, depending on the number of dimensions of the mesh. For two-dimensional meshes, this could be quadrilaterals, or pyramids for three-dimensional meshes. In our case, we will be dealing with three-dimensional meshes, and our elements will be tetrahedra.

4.1.1 Mathematical explanation

The 3D mesh will be divided using elements of discrete sizes. As mentioned, we will be using tetrahedra. Each tetrahedra has 4 faces, and each face has 3 vertices with coordinates [x, y, z]. To calculate the surface area of the object the following must be accomplished:

- First we must determine which faces are neighbours. That is, if the normals on two faces are directly opposite, they face each other. \( \mathbf{\pi}_1 = -\mathbf{\pi}_2 \)

- If two faces are neighbours, we do not add their area to the surface area of the figure. If not, we calculate the area, and add it to the total surface area.

To calculate the area of a given face it is necessary to know the 3 vertices that make up the face. With 3 vertices A, B and C, we get 2 vectors \( \mathbf{\overrightarrow{AB}} \) and \( \mathbf{\overrightarrow{AC}} \). By applying the crossproduct \( \mathbf{\overrightarrow{AB}} \times \mathbf{\overrightarrow{AC}} \) we can determine the area of the triangle-shaped face; ref figure 4.2
Figure 4.1: Left - Simple 2D mesh, Right - 3D mesh of a cube

Figure 4.2: ABC defines a triangle with vectors AB and AC.
\[ \text{Area}_{\text{triangle}} = \frac{AB \times AC}{2} \]

\[ \sum_{0}^{N} \text{Area}_{\text{triangle}} = \text{Area of figure} \]

\[ N = \text{number of triangles without neighbours} \]

4.1.2 Algorithm

The algorithm for our problem can be considered in two versions, a serial and a parallel version. The serial version consists of two main parts. One part consists of a function that cycles over every element and every face in the object. The other is the function that, for a given face, calculates its area.

Pseudocode

Element-cycling

for \((j = 0 ; j < \text{numberOfElements} ; j++)\) {
  for \((i = 0 ; i < 4 ; i++)\) //Number of faces = 4
    {
      if (NOT \(\text{element}[j].\text{face}[i].\text{hasNeighbour}()\))
        {
          \(\text{area} += \text{element}[j].\text{face}[i].\text{computeArea}()\);
        }
    }
}

Area computation \hspace{1em} \text{As illustrated in 4.1.1, the computations needed to find the area of a given face is to apply the crossproduct of two vectors, and divide the result by 2.}

\[ \text{areaOfFace} \text{(Vertex1[3], Vertex2[3], Vertex3[3])} \]

\{ 
  \text{Vector1[3];}
  \text{Vector2[3];}
  \text{CrossProduct[3];}

  for(\(i = 0; i<3; i++\))
    {
      \text{Vector1}[i] = \text{Vertex2}[i] - \text{Vertex1}[i];
      \text{Vector2}[i] = \text{Vertex3}[i] - \text{Vertex1}[i];
    }
  \text{CrossProduct}[0] = \text{Vector1}[1] \ast \text{Vector2}[2] - \text{Vector1}[2] \ast \text{Vector2}[1];
\]

52

To create a parallel version of the computing, we would have to decide which paradigm to use. The most common paradigms in parallel computing are:

- Task Farming, or Master / Slave
- Single Program Multiple Data (SPMD)
- Data Pipelining
- Divide and Conquer
- Speculative Parallelism

We have chosen to utilize the master/slave-paradigm. It calls for a separate “control” program termed the master that is responsible for process spawning, initialization, collection and display of results, and perhaps timing of functions. The slave programs perform the actual computation involved; they either are allocated their workloads by the master (statically or dynamically) or perform the allocations themselves.

In our toy-problem, the master would do the following:

- Partition the mesh into smaller sub-meshes.
- Distribute the sub-meshes to the slave processes.
- Receive results from the slave-processes.
- Add the results to calculate the surface area of the object.

The slave processes would do this:

- Wait to receive data from the master.
- Determine what faces are neighbours and calculate the area of the relevant faces (ref 4.1.1).
- Send the area result for a face back to the master process.
Pseudocode

Master

\[
\begin{align*}
\text{area} & = 0; \\
n & = \text{numberOfProcessors}; \\
\text{elements}[]; & \quad \text{\textit{An array of elements}} \\
\text{readElements}(); & \quad \text{\textit{reads an input file}} \\
\text{splitElements}(n); & \quad \text{\textit{splits the array of elements into n pieces}} \\
\text{for} & \quad (i = 0 \quad ; \quad i < n \quad ; \quad i++) \\
\{ & \quad \text{area} += \text{receiveResult}(i); \quad \text{\textit{receives area calculations from}} \\
\} & \quad \text{print area;}
\end{align*}
\]

Slave

\[
\begin{align*}
\text{receiveElements}(); \\
\text{for} & \quad (i = \text{first}_\text{element} \quad ; \quad i < \text{last}_\text{element} \quad ; \quad i++) \\
\{ & \quad \text{for} \quad (j = 0 \quad ; \quad j < 4 \quad ; \quad j++) \\
\{ & \quad \text{if} \quad \text{NOT} \quad \text{element}[i].\text{face}[j].\text{hasNeighbour}() \\
\{ & \quad \text{area} += \text{element}[i].\text{face}[j].\text{computeArea}(); \\
\} & \quad \}
\}
\]

\text{sendResult(area);} \quad \text{\textit{Send the results to the master-process}}
\]

The computation of the area is the same as for the serial version. We note that the major difference between the serial and the parallel version lies with the need for passing messages between the master and the slave.

4.1.3 Why this problem?

The reason for selecting this problem is first of all its simplicity. We needed a simple example for illustrative purposes in our study. If we were to define a partial differential equation and select a solving strategy based on the finite element method it would require a lot of time for us just to learn the mathematics, in addition to the effort it would take to explain the problem. Area computation does not require too much mathematical skill and it is easy both to explain and understand.

An area computation alone is not a relevant issue for scientific computing. It can, however, comprise a smaller part of a large scientific computation. The solution process of scientific computing is concerned with dividing the problem into smaller problems that can be more easily solved. This area computation problem could be such a subproblem.
4.1.4 Issues with the problem

Due to its simplicity, our toy problem has some drawbacks. To run tests we would need large meshes with many elements. Otherwise the differences in the tests would be too small to make any conclusions (statistical deviations etc). In addition, the problem does not require any communication between the slaves to keep track of internal boundaries. The only communication that is required is before and after the area calculations, not during the calculations.

4.1.5 Focus

The focus of our study will be on the expressiveness of the different languages, C++, Java and Fortran. The expressiveness of a language is concerned with the power of the expressions used in the language, i.e. number of statements needed to perform a specific task and resemblance to natural language. Expressiveness is a relative term, since a language can be more expressive in one area (arrays) than in another (text strings). What we will be interested in is each language’s expressiveness in the problem we are looking into, and also for scientific computing in general. Measuring of the expressiveness can be done by looking at the number of statements needed to express the same operations in the different languages, the structure of the functions (return values and number of arguments) and the readability of the code.

4.1.6 Approaches

We will look at strategies for solving the problem in 4.1 in C++, Java and Fortran. In the C++ approach we look at how the problem could be solved using the prototype framework implementation from [49]. For the Java approach we look at key issues for rewriting the C++ framework in Java, for example how C++ features that are not present in Java (e.g. pointers and templates) would be represented in Java. Java features not inherent in C++ (e.g. automatic garbage collection) will also be of interest, particularly the benefits one would get from these. We will look at Fortran in a slightly different perspective. A framework is an object-oriented concept; it will therefore be hard (if not impossible) to develop a framework in Fortran. We can therefore only look at implementation issues for solving the toy problem and try to look at what would be needed to achieve a behaviour resembling the one in the framework.

4.1.7 Expectations

We expect that the results of our study will show that C++ and Fortran have more expressive power than Java. The reason for this is that Fortran is very strong on mathematical operations and C++ can provide a high degree of expressiveness through operator overloading, a feature not present in Java. We are, however, not sure that our approach to Fortran will yield the most satisfying result as we were unfamiliar to this language prior to the start of the project. It will nevertheless be a learning experience for us.
5 Study of Object Oriented Framework for concurrent scientific computing

This chapter contains the study of an existing prototype object oriented framework for concurrent scientific computing. In addition we look at how this framework could be utilized to solve the toy problem from the previous chapter. This framework will also be used in the following chapter where we look into the migration of the framework to Java and Fortran.

5.1 Code analysis

Here we present the framework we will be working with. The framework is developed by Roxana Diaconescu at the Norwegian University of Science and Technology in Trondheim and is presented in [49] and [50]. It is constructed for solving of PDE's using the Finite Element Method (FEM). The key idea of FEM is to compute an approximation of the solution of the PDE by discretizing the physical domain using elements of different shape.

The computation is done at the individual element level and then assembled into a global linear system of equations.

The framework takes as input a mesh file where the user has specified the discretized physical domain information. The domain is partitioned into a number of sub-domains (using the METIS library) equal to the number of processors available, and associates each sub-domain with a single processor. The user then specifies the sequential computation for a sub-domain by composition and sub-classing of the framework. All the concurrency aspects are isolated from the user.

5.1.1 Rational Rose diagram

In figure 5.1 we present a Rational Rose-diagram of the code. Note that the diagram is somewhat abbreviated, and contains only the classes and their relations. All functions are removed from the diagram to reduce the size.

5.1.2 Class functionality

The framework uses two external libraries, METIS and OOMPI. METIS[3] is a set of programs for partitioning graphs, partitioning finite element meshes and, and for producing fill-reducing orderings for sparse matrices. OOMPI (Object Oriented Message Passing
Figure 5.1: Rational Rose diagram of the code - Overview
Interface][8] is an OO version of MPI, designed to work with OO languages like C++. The idea behind MPI is to develop a standard for writing message passing systems.

The framework consists of several classes.

- Master - The Master class reads in the mesh file from the user and constructs a partition array using the Metis library. It then creates subdomains and sends these to the processes.

- SubDomain - An object of the Subdomain class belongs to a unique process. Information local to the subdomain (vertices, elements and faces) is stored in dynamic arrays. Each subdomain has two kinds of internal boundaries, one that is part of the local domain and is sent to remote processors, and one that is received from remote processors containing duplicated data that is needed locally. The subdomain has two Ghost objects, one for sending data and one for receiving data.

- MeshStruct - The MeshStruct class is an abstract class for deriving different kinds of meshes. It contains structures for storing the faces, elements and vertices of the mesh, and also has information about the shape of the mesh (the number of faces per element, vertices per face, vertices per element and the dimension).

- UserAlg - Small class that contains a virtual main-function.

- TheAlg - Inherits from the UserAlg-class. It loops over all the elements in the mesh, and prints out the results. The results are then stored in a output file.

- Worker - The worker-class is supposed to receive a subdomain from the Master-class. In addition, it seeks to establish what remote data is needed locally, and what local data is needed remotely

- IntBndry - The IntBndry (Internal Boundary) class is a subclass of MeshStruct. It also has functionality for sending and receiving boundary data using OOMPI.

- UserData - The UserData class is a generic class that the user subclasses. The class will be used by a global update function that does all the packing, unpacking, sending and receiving of the user data.

- theData - Inherits from the UserData-class. It is basically a container class for all the different types of data that exists in the program. It contains functionality to find out the data’s size, and for reading data from a file and printing out the data.

- Ghost - The Ghost class is a container class for the data to be sent/received.

5.2 Solving the toy problem using the existing framework

The framework’s intension is to ease the programming burden of the user. The user is expected to specify the algorithm for the solution of the problem in a class inherited from the UserAlg class of the framework. The algorithm can then be called in the different
number of processes, and the user does not need to know how many processes there are or how the mesh is divided.

In the case of the problem we have described, all data about the mesh resides on a file, the number of elements, faces and vertices and how they relate. The framework reads from the file and puts all the data in the structures for the elements, faces and vertices.

Within the framework the data is contained in one-dimensional arrays, for efficiency purposes. The data is therefore ordered block-wise in the array. With the access to the structures hidden within function calls the underlying data organisation is irrelevant (the concept of encapsulation in OO), at least as long as the function does what is expected.

For the purpose of expressiveness we decide to look at how the problem can be solved assuming the data is stored in two-dimensional arrays and that the user can retrieve these arrays. For the elements array each row would hold information for each element and likewise for the faces and vertices arrays.

{
    double[3] vect1, vect2, crossproduct;
    double area;
    for (int i=0;i<3;i++)
    {
        vect1[i]=p2[i]-p1[i];
        vect2[i]=p3[i]-p1[i];
    }
    crossproduct[0] = ( vect1[1]*vect2[2] - vect1[2]*vect2[1] );
    crossproduct[1] = ( vect1[2]*vect2[0] - vect1[0]*vect2[2] );
    crossproduct[2] = ( vect1[0]*vect2[1] - vect1[1]*vect2[0] );
    return area = ( sqrt(pow(crossproduct[0],2) + pow(crossproduct[1],2) + pow(crossproduct[2],2) / 2 );
}

The code for calculating the total surface area:

double calculateSurfaceArea(Subdomain * D)
{
    int numelems = D->GetNElems();
    int numfaces = D->GetN篮板s();
    int numvertices = D->GetNVertices();
    int** elements = D->GetElems();
    int** faces = D->GetFaces();
    double** vertices = D->GetVertices();
    double area = 0;
    for(int i = 0 ; i < numelems ; i++)
    {

for(int j = 0 ; j < 4 ; j++)    //4 faces pr. element
{
    if( elements[i][j+4] == -1) //the face has no neighbour
    {
        area += triangleArea(
            vertices[ (faces[(elements[i][j])][ 0 ]],
            vertices[ (faces[(elements[i][j])][ 1 ]],
            vertices[ (faces[(elements[i][j])][ 2 ]]);
    }
}
return area;
}

This is the shortest amount of code possible to write but it is not very readable. To achieve this temporary arrays and values should be created. This would better illustrate that we work with one element at a time and that for each element we look at one face at a time. We therefore introduce:

    int* element;
    int* face;
    int neighbour;
    double[3] vertex1, vertex2, vertex3;

and the code would look like this from the for-statement:

for(int i = 0 ; i < numelems ; i++)
{
    element=elements[i];
    for(int j = 0 ; j < 4 ; j++)    //4 faces pr. element
    {
        neighbour = element[j+4];
        if( neighbour == -1)        //the face has no neighbour
        {
            face = faces[element[j]];
            vertex1 = vertices[face[0]];
            vertex2 = vertices[face[1]];
            vertex3 = vertices[face[2]];

            area += triangleArea(vertex1, vertex2, vertex3);
        }
    }
}

The code is easier to read but there is an efficiency cost involved with having temporary pointers.
One could also have selected a truly object-oriented approach. Then we would have objects of elements, faces and vertices. The user would then need to know the representation of the objects, and the methods and attributes associated with the object. For brevity we only show the class members and operations that are needed to solve our particular problem.

```java
Class Element
{
  public:
    Element getNeighbour(int i) { return neighbours[i]; }
    Face getFace(int i) { return faces[i]; }
  private:
    Face * faces
    Element* neighbours;
}
```

The structures for faces and neighbours are coordinated so that neighbour[i] is neighbour on face[i].

```java
Class Face
{
  public:
    Vertex vertex(int i) { return vertices[i]; }

  private:
    Vertex* vertices;
}
```

```java
Class Vertex
{
  public:
    Vertex operator- (Vertex v)
    {
      Vertex temp;
      temp.x = x-v.x;
      temp.y = y-v.y;
      temp.z = z-v.z;
      return temp;
    }
    Vertex operator* (Vertex v)
    {
      Vertex temp;
      temp.x = x*v.x;
      temp.y = y*v.y;
```
temp.z = z*v.z;
return temp;
}
double sum() { return (x + y + z); }
double x,y,z;
}

In addition to the standard default and value constructors and destructor, the get and set functions all need bounds-checking to ensure that the user doesn’t index outside the structures.

With the classes defined we could now express the algorithm like this:

double triangleArea (Vertex p1, Vertex p2, Vertex p3)
{
    Vertex vect1, vect2, crossproduct;
    double area;
    vect1 = p2 - p1;
    vect2 = p3 - p1;
    crossproduct.x = ( vect1.y * vect2.z - vect1.z * vect2.y );
    crossproduct.y = ( vect1.z * vect2.x - vect1.x * vect2.z );
    crossproduct.z = ( vect1.x * vect2.y - vect1.y * vect2.x );
    return area = ( sqrt ((crossproduct.crossproduct).sum()) / 2 );
}
double calculateSurfaceArea(Subdomain * D)
{
    int numelems = D->GetNElems();
    Element* elements = D->GetElems();
    Face* faces = D->GetFaces();
    Vertex* vertices = D->GetVertices();
    double area = 0;
    for(int i = 0 ; i < numelems ; i++)
    {
        for(int j = 0 ; j < 4 ; j++)  //4 faces pr. element
        {
            //face j has no neighbour
            if( Element[i].GetNeighbour(j) == null )
            {
                Face tempface = Element[i].GetFace(j);
                area += triangleArea (tempface.vertex(0), tempface.vertex(1),
                tempface.vertex(2));
            }
        }
    }
    return area;
}
With this kind of OO approach it easier to express the algorithm in a way that is more similar to natural language. Any complex operations can be hidden within well defined methods with names that say clearly what the the method does. The classes must reflect the properties of the real life concepts they model (elements, faces and vertices). The expressiveness achieved with the object-oriented approach comes at the cost of effectiveness. There is efficiency overhead involved with each object and the size of the program will also increase.

5.3 Conclusion

This chapter introduced an object oriented framework for the concurrent solution of numerical problems. The framework is still under development and therefore the presentation is based on prototype for testing framework functionality, and we only show the basic structure and core functionality.

We then showed how the framework can be used to solve our toy problem, assuming different implementations of the structures within the framework. Of the approaches we discussed the truly object oriented approach proved to be the most expressive, using objects to represent the conceptual model of the problem and utilizing the operator overloading feature.
6 Expressiveness of Object Oriented and Procedural languages

In this chapter we will present our idea of how to translate the framework presented in chapter 5 to other programming languages. First we look into Java, and then Fortran. For both languages we will discuss the particular problems associated with the conversion of the framework, and we will then show how to solve the toy problem using framework functionality. As mentioned earlier, we will be focusing on the expressiveness of the languages, but we will also mention other aspects, like performance.

6.1 Java implementation strategy

Here we will be presenting what we feel are the most significant issues relating to the migration of the framework, from C++ to Java. These are: pointers, templates, global functions, operator overloading and message passing. With respect to message passing, we have decided to use Java RMI. Due to this we will present some RMI basics in the next section. Subsequently we will show approximately how our toy problem would appear if implemented in the translated framework. We have decided to show both a serial version, and a parallel version of the toy problem.

6.1.1 RMI - Remote Method Invocation

Since we will be using RMI instead of MPI (or similar), we will now give an introduction to RMI. For more information on the subject, we refer to Sun Microsystems website\textsuperscript{1}.

An Overview of RMI Applications

RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a distributed object application. Distributed object applications need to:

\textsuperscript{1} http://java.sun.com

64
• Locate remote objects: Applications can use one of two mechanisms to obtain references to remote objects. An application can register its remote objects with RMI's simple naming facility, the rmiregistry, or the application can pass and return remote object references as part of its normal operation.

• Communicate with remote objects: Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.

• Load class bytecodes for objects that are passed around: Because RMI allows a caller to pass objects to remote objects, RMI provides the necessary mechanisms for loading an object’s code, as well as for transmitting its data.

One of the central and unique features of RMI is its ability to download the bytecodes (or simply code) of an object’s class if the class is not defined in the receiver’s virtual machine. The types and the behavior of an object, previously available only in a single virtual machine, can be transmitted to another, possibly remote, virtual machine. RMI passes objects by their true type, so the behavior of those objects is not changed when they are sent to another virtual machine. This allows new types to be introduced into a remote virtual machine, thus extending the behavior of an application dynamically. The compute engine example in this chapter uses RMI’s capability to introduce new behavior to a distributed program.

Remote Interfaces, Objects, and Methods

Like any other application, a distributed application built using Java RMI is made up of interfaces and classes. The interfaces define methods, and the classes implement the methods defined in the interfaces and, perhaps, define additional methods as well. In a distributed application some of the implementations are assumed to reside in different virtual machines. Objects that have methods that can be called across virtual machines are remote objects. An object becomes remote by implementing a remote interface, which has the following characteristics:

A remote interface extends the interface java.rmi.Remote. Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions. RMI treats a remote object differently from a non-remote object when the object is passed from one virtual machine to another. Rather than making a copy of the implementation object in the receiving virtual machine, RMI passes a remote stub for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the caller, the remote reference. The caller invokes a method on the local stub, which is responsible for carrying out the method call on the remote object. A stub for a remote object implements the same set of remote interfaces that the remote object implements. This allows a stub to be cast to any of the interfaces that the remote object implements. However, this also means that only those methods defined in a remote interface are available to be called in the receiving virtual machine.
Creating Distributed Applications Using RMI

When you use RMI to develop a distributed application, you follow these general steps.

- Design and implement the components of your distributed application.
- Compile sources and generate stubs.
- Make classes network accessible.
- Start the application.
- Design and Implement the Application Components

First, decide on your application architecture and determine which components are local objects and which ones should be remotely accessible. This step includes:

Defining the remote interfaces - A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. Part of the design of such interfaces is the determination of any local objects that will be used as parameters and return values for these methods; if any of these interfaces or classes do not yet exist, you need to define them as well.

Implementing the remote objects - Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces (either local or remote) and other methods (which are available only locally). If any local classes are to be used as parameters or return values to any of these methods, they must be implemented as well.

Implementing the clients - Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

6.1.2 Translating the framework to Java

If the C++ framework is to be translated into Java code the following issues need to be addressed:

- Pointers
- Templates
- Global functions
- Operator overloading
- How to handle message-passing
The need to change comes from the fact that Java has no pointers, templates, global functions or operator overloading, features that are present in the framework. The C++ destructors for each class can be omitted in Java since this is done automatically. In addition it is also necessary to change the parallel implementations from OOMPI to for example Java RMI.

**Global functions**

The framework has some global functions for error checking, allocating arrays and printing arrays. Since Java has the restriction that all functions belong to a class we would have to define a class Globals containing all global functions and static variables. The functions must be declared static. An example of how the class would look:

```java
class Globals {
    static int[] AllocInit(int n) {
        ...
    }
    static void printDoubleArray(double vec, int n) {
        ...
    }
}
```

**Pointers**

It is not entirely true that Java does not have pointers. In fact, in Java all objects are pointers [48] but the syntax of the language hides this. It is therefore worth noting that the assignment of object with the = operator does not assign value, but is the equivalent of pointer assignment. To get a copy of an object you have to use the `clone()` method. The same principle is for the `==` operator which is a pointer comparison. For comparison of values on has to use the `equals()` function.

Some of the implications for the framework translation is that the structures for the elements, faces and vertices must be translated accordingly:

```plaintext
int* elements -> int elements[]
int* faces -> int faces[]
double* vertices -> double vertices[]
```

The functions returning these structures need also to be changed, for example:

```plaintext
int* GetElems() -> int[] GetElems()
```
Many functions in the framework returns reference to object, e.g. in the Master class function GetDomain (a function that reads the mesh from file):

```cpp
Master& Master::GetDomain(char* filename)
{
    ...
    return *this;
}
```

In Java this would look like this:

```java
Master GetDomain(string filename)
{
    ...
    return this;
}
```

That is, simply remove the '\&' and the '*'. Also notice that instead of `char*` in the parameter type Java uses `string`.

**Templates**

This is a feature that is hard to translate from C++ to Java. In the C++ framework there are two templated classes, `UserData` and `theData` (where `theData` inherits from `UserData` and has the same template class parameter, `T`). The `theData` class contains a pointer structure of type `T` and an integer for the size of the structure, and is used in the global `Update()` function (which is also templated) which is used to update shared boundary data of type `T` during computation (packing, unpacking, sending and receiving data). The intention behind the template classes is that the data can be either char,int, double or pointer.

It is hard to find a satisfactory solution to this problem. There are two approaches one could try, one is to implement different versions of the `Update()` function, one for each kind of data. The user would need to tell the framework what kind of data it will work with by calling a function that sets a parameter within the framework. The `UserData` and `theData` classes would then be superficial, since they are used only to create arrays of the template type.

The other solution is to have the `UserData` and `theData` classes implement an array of type `Object`.

```cpp
T* data;
int size;
T GetAt (int i){
    if( (i< size) && (i>=0))
        return data[i];
}
```

```java
Object data[];
Object GetAt(int i){
    if( (i< data.length()) && (i>=0))
        return data[i];
}
```
void SetAt(int i, T val){
  if( (i< size) && (i>=0))
    data[i]=val;
}

Object SetAt(int i, Object val);
  if( (i< data.length()) && (i>=0))
    data[i]=val;
}

The size variable is not needed in Java, the length() function for arrays can be used instead. When the framework needs to know what type Object is this can be determined by calling the getClass() function.

**Operator overloading**

This is also a feature not supported in Java, but it is not used for other than the value assignment with the operator "=" between objects of classes MeshStruct and Subdomain. In Java this can be implemented with a clone() function.

**Message passing**

An important part of the framework is the passing of messages. Be it worker <-> worker or worker <-> master. As we noted in the introduction, we chose to use Java RMI. We could have used one of several existing Java MPI-versions, but we felt, however, that it would be most interesting to look at Java RMI. This due to us choosing the Task Farming, or Master / Slave paradigm for our toy problem. RMI uses a client / server approach, somewhat different in other words. It would be interesting to see whether it is applicable in a slightly different environment.

Some examples of MPI for Java are [51]:

- *mpiJava* - Modelled after the C++ binding for MPI. Implementation through JNI wrappers to native MPI software. http://www.npac.syr.edu/project/pcrc/HPJava

- *JavaMPI* - Automatic generation of wrappers to legacy MPI libraries. C-like implementation based on the JCI code generator. http://perun.hscs.wmin.ac.uk/JavaMPI

- *MPIJ* - Pure Java implementation of MPI closely based on the C++ binding. http://ccc.cs.byu.edu/DOGMA/

- *JMPl* - MPI Soft Tech Inc. have announced a commercial effort under way to develop a message passing environment for Java

**Components of the application framework** The components of the framework consist of the classes mentioned in 5.1.2. We believe that the structure would be the following (all names refering to the framework-classes):

Server-side : *Master, Ghost, MeshStruct, IntBndry, SubDomain*

Client-side : *Worker, theData, UserData, TheAlg, UserAlg*

When transforming the framework we would have to change some minor things.
• *filename.h* and *filename.c* => *filename.class*. The code in .h and .c files could be thrown together into a single class-file.

• the *Master-class* (in framework) is the equivalent of the *Server-class* (in Java RMI)

• the *Worker-class* (in framework) is the equivalent of the Client-class (in Java RMI)

**Basic functionality**  The basic functionality of the *Client* and *Server* is described in section 6.1.3. We want the *Client* to retrieve a *SubDomain*-object (a part of the total problem) from the *Server*. To accomplish this, we will implement an interface between the clients and the server. When the clients invoke the method defined in the interface, a *SubDomain* will be sent to the *Client*. After recieving the *SubDomain*-object, the *Client* will use its own methods for calculating the area, and return the answer to the *Server*.

**Problems**  It is worth noting that RMI, as a Client-Server oriented approach, would run into some problems. This is because the framework uses worker-worker communication when calculating the internal boundaries. Due to our problems simplicity, it doesn’t require this communication. To accomplish this in RMI we would have to enable the *client-class* to act as both server and client at the same time. This is possible, but tricky. We would have to divide the client into two parts. One that executes the actual computing, and one that listens for communication from other workers. An alternative would be to send the messages from client_1->server-client_2. This, however, would put unnecessary load on the server and network. We would be sending the same messages twice, effectively 100% overhead.

### 6.1.3 Toy Problem in Java

All code in header-files and c++-files would have to be combined into class-files. When implementing the serial version in Java we would have to put the relevant methods (*GetElems*, *GetVerts*, etc) in the *Master-class*. We assume we would only need a master-class, plus a class for the global methods and the algorithm.

Here is the Java-version for calculating the area of a face.

```java
public double triangleArea (double p1[3],double p2[3],double p3[3])
{
    int i;
    double v1[];
    double v2[];
    double cp[];
    v1 = new double[3];
    v2 = new double[3];
    cp = new double[3];

    for (i = 0 ; i < 3 ; i++)
```
\{
    v1[i] = p2[i] - p1[i];
    v2[i] = p3[i] - p1[i];
}\n

double area = sqrt( pow(cp[0],2), pow(cp[1],2), pow(cp[2],2));

return area;
\}

The Java-version for calculating the surface area of a model. Here we run into some issues that concerns the differences between Java and C++. The C++ version uses pointers, but since Java doesn't support pointers, we will have to solve it using other methods. As we mentioned in the beginning of this section, we would put all relevant methods into a master-class. Thus we pass "Master master" as parameter to the function, and utilize the master-class' methods GetNElems, etc.

public double calculateSurfaceArea (Master master)
{  
    int numOfElem = master.GetNElems();
    double p1[] = new double [3];
    double p2[] = new double [3];
    double p3[] = new double [3];
    int elems[][] = master.GetElems();
    int faces[][] = master.GetFaces();
    double verts[][] = master.GetVerts();
    int face[], element[];
    int neighbourid;
    double area = 0;

    for (int i = 0 ; i < numOfElem ; i++)
    {
        element = elems[i];
        for (int j = 0 ; j < 4 ; j++)
        {
            neighbourid = elems[i][j+4];  //neighbour-index
            if (neighbourid == -1)  //check for neighbour
            {
                face = element[j];
                p1 = verts[ face[0] ];
                p2 = verts[ face[1] ];
            
            
        }
p3 = verts[ face[2] ];
}
area += triangleArea (p1, p2, p3);
}
return area;
}

The differences between Java and C++ are marginal. Since Java’s two-dimensional arrays are array of arrays, we can use temporary one-dimensional arrays to work with one row at a time. In the C++ version we used pointers to point to the rows.

**Toy problem parallel version**

Our toy problem differs somewhat from the typical RMI-applications. It requires the server to delegate work to the clients, instead of clients that request the server to accomplish a given task for them. To solve our problem we will implement an interface with a method that fetches a subdomain, and sends it to a client. When the client receives its subdomain, it will invoke its method for calculating the surface-area, and then return the result to the server.

**The Server Interface**

```java
package Server;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Server extends Remote {
    SubDomain getSubDomain(int pid) throws RemoteException;
}
```

This is then the remote interface, created by importing `java.rmi.Remote` and subclassing the `Remote`-class. The interface resides between the client and the server. The `getSubDomain`-method is called from the clients with its `pid`, and a `SubDomain` is returned to it. This method is now available on all clients to run.

We also need a second interface that lie between the server and the work that it needs to do (in our example, passing subdomains to clients). This interface, named `Task`, implements one method `execute` that returns a subdomain to anyone calling the method. An object that implements this interface can be sent to a remote virtual machine as part of a remote method invocation.

**Task Interface**

```java
package server;
import java.io.Serializable;
```
public interface Task implements Serializable
{
    SubDomain execute();
}

The Server  The server’s job is to read an array of elements, split it into smaller elements, send these subdomains to the clients, and add up the results from the clients, thus computing the total surface area. Input from files is left out, but this would fit in at the end of main(). It would be natural to use some of Java’s existing functionality on I/O streams, like FileReader to to implement it. It is worth noting the installation of a security manager. This protects the access to system resources from downloaded code running on the virtual machine.

package ServerEngine;

import java.rmi.*;
import java.rmi.Server.*;
import server.*;

public class ServerEngine extends UnicastRemoteObject implements Server{
    public class ServerEngine() throws RemoteException
    {
        super();
    }

    public SubDomain getSubDomain (int pid)
    {
        /* Here comes the code that is set in the SubDomain-class for
        * sending a subdomain. In the framework this is
        * subdomain.receive();
        */
    }

    public static void main (String[] args)
    {
        //install the securitymanager
        if (System.getSecurityManager() == null)
        {
            System.setSecurityManager (new RMISecurityManager());
        }

        //
        String name = "//host/Server";
try
{
    Server server = new ServerEngine();
    Naming.rebind(name, server);
    System.out.println("ServerEngine bound to name");
}
catch (Exception e)
{
    System.err.println("ServerEngine exception: "+ e.getMessage());
    e.printStackTrace();
}

/* Read from inputfile */
...
}

The Clients

The client implements the two methods it need to calculate the area of a given subdomain (in our example, an element consisting of 4 triangular faces). The two methods, triangleArea and surfaceArea, are mostly similar to the version in 6.1.3. The only exception is the calculateSurfaceArea(). It will now recieve a SubDomain-object as parameter.

    package client;
    import java.rmi.*;
    import server.*;

    public class Area implements Task
    {
        //constants
        int num0fElem;
        double p1[] = new double[3];
        ...
        ...
        //methods
        ...
        public Object execute()
        {
            return surfaceArea(Subdomain);
        }
        /**
         * Compute area of triangular face
         */
public static double triangleArea(double p1[3], double p2[3], double p3[3])
{
    /* code repeat....Same as serial version */
    ....
}
public static double surfaceArea(SubDomain subdomain)
{
    /* Code repeat - same as serial, except as noted */
    ....
    int elems[][] = subdomain.GetElements();
    int faces[][] = subdomain.GetFaces();
    double verts[][] = subdomain.GetVertices();
    ....
}

6.1.4 Implementation with the Java Array package

The Array package was presented briefly in section 3.4.1. It was introduced to achieve higher performance in working with multidimensional arrays. These arrays are declared as objects of an array class, and access to data within the arrays is through methods get and set methods. The example shows only the triangle area computation, but this is enough to show that the code turns into an intertwinemnet of get and set calls.

private double triangleArea(doubleArray1D vert1, doubleArray1D vert2,
    doubleArray1D vert3)
{
    doubleArray1D vect1 = new doubleArray1D(3);
    doubleArray1D vect2 = new doubleArray1D(3);
    doubleArray1D cp = new doubleArray1D(3);
    for(int i=0;i<3;i++)
    {
        vect1.set(i, (vert2.get(i)-vert1.get(i)));
        vect2.set(i, (vert3.get(i)-vert1.get(i)));
    }
    cp.set(0, (vect1.get(1)*vect2.get(2) - vect1.get(2)*vect2.get(1)));
    cp.set(1, (vect1.get(2)*vect2.get(0) - vect1.get(0)*vect2.get(2)));
    cp.set(2, (vect1.get(0)*vect2.get(1) - vect1.get(1)*vect2.get(0)));
    double area = sqrt( pow(cp.get(0), 2) + pow(cp.get(1), 2) + pow(cp.get(2), 2))/2;
    return area;
}
6.1.5 Conclusion

Java and C++ have a very similar syntax, but some features in C++ must be represented differently in Java. Global functions are put within a class and declared as static and can then work just like C++ global functions. The translation from C++ pointers to Java's "pointer-free" syntax is not too hard to accomplish as long as care is taken when doing assignments ( = operator for pointer assignment and the clone() function for value assignment). Templates, however, is the main issue of concern. If the approach where the Object class is used as "template" works for the framework then we would get a translation that is very similar to the C++ framework, with roughly the same amount of code. Furthermore, we think that the use of RMI instead of OOMPI would not cause a significant change in the number of LOC\(^2\). Thus, their expressiveness is similar.

However, it turned out that RMI's client-server approach was not ideal to solve our problem. We used several days to understand how we could work our way around the differences mentioned earlier. When it comes to the expressiveness of Java it appears to us that to implement our small toy problem in Java RMI we would need approximately the same amount of code as the existing C++ version.

We have chosen to use multi-dimensional arrays over one-dimensional arrays. As discussed in section 3.4.1, this will result in a significant performance penalty, but it will result in cleaner, simpler code. As an alternative, we propose to use the Java Array package to reduce this penalty. However, this would then result in a loss of readability, and thus less expressive code.

6.2 Fortran

Fortran has for many years been the language of choice for implementing scientific computing. We want to see how a procedural language like Fortran would compare to the object oriented languages Java and C++ with regards to expressiveness.

6.2.1 Framework issues

Implementing a similar framework to the one we have presented is not achievable in Fortran. Fortran 90 introduced modules which maps to classes in the OO-languages and the same degree of encapsulation can be achieved. There are, however, too many OO features that are used in the development and use of frameworks which are not available in Fortran. Inheritance can be simulated, but the relationship between base class and "inherited" class in Fortran is a has-a relationship as opposed to the is-a relationship one gets with the OO-languages. In Fortran one is also obliged to rewrite all the methods from the base class in the inherited class. Fortran does not support templates or polymorphism, making it hard to achieve the generality and reusability provided by a framework implementation.

The C++ framework uses OOMPI for parallelisation purposes, with Fortran one can use MPI. Since OOMPI is an object oriented version of MPI there will only be minor

\[2\text{Lines Of Code}\]
differences in the implementation of the parallelism in the Fortran approach. Another approach to parallelism is to use the High Performance Fortran language which supports parallelism through built-in language features.

6.2.2 Toy problem in Fortran

Even though the Fortran approach is a little different from the one in Java and C++, we will still make some of the assumptions we did for the two OO languages. The data about the mesh is on a file with the same structure as mentioned earlier. We would then have a routine for reading the file and storing the data in arrays in addition to the number of elements, faces and vertices. These arrays are similar to the ones in Java and C++, that is a two-dimensional array containing information for each element in each row, a two-dimensional array with information about each face in each row and a two-dimensional array for each vertex.

We will also assume that we have Fortran functions that return these arrays after reading them from a file and also functions for returning the number of elements, faces and vertices. These have the same name as the ones in the framework (GetElements, GetNEElements etc.).

```fortran
FUNCTION triangleArea(vert1, vert2, vert3)
    REAL triangleArea
    REAL (3), INTENT (IN) :: vert1, vert2, vert3
    REAL (3) :: vect1, vect2, crossproduct
    vect1=vert2-vert1
    vect2=vert3-vert1
    crossproduct(0)=( vect1(1)*vect2(2) - vect1(2)*vect2(2) )
    crossproduct(1)=( vect1(2)*vect2(0) - vect1(0)*vect2(2) )
    crossproduct(2)=( vect1(0)*vect2(1) - vect1(1)*vect2(0) )
    triangleArea = SQRT( SUM ( crossproduct*crossproduct ) )/2
END FUNCTION triangleArea
FUNCTION calculateSurfaceArea(m)
    TYPE (Master) INTENT(IN) :: m
    INTEGER :: numelems
    INTEGER :: elements (: , , )
    INTEGER :: faces (: , , )
    REAL :: vertices (: , , )
    REAL :: area
    REAL(3) :: vertex1, vertex2, vertex3
    INTEGER :: elementindex, faceindex, vertexindex, i, j
    numelems=GetNEElements(m)
    elements=GetElements(m)
    faces=GetFaces(m)
    vertices=GetVertices(m)
    elementindex=faceindex=vertexindex=i=j=0
```
DO i, numelems
  DO j,4
    IF (elements(i,j+4)== -1) THEN
      faceindex=elements(i,j)
      vertex1=vertices( faces(faceindex,0) , : )
      vertex2=vertices( faces(faceindex,1) , : )
      vertex3=vertices( faces(faceindex,2) , : )
      area = area + triangleArea(vertex1,vertex2,vertex3)
    END IF
  END DO
END DO
calculateSurfaceArea = area
END calculateSurfaceArea

It is particularly in the *triangleArea* function we spot the expressive power Fortran has when working with arrays. Arithmetic operations can be done on all elements in an array without specifying loops on the array. We also use one of the intrinsic functions Fortran has for arrays, SUM, which adds all the elements of an array.

Furthermore, it is reasonable to assume that the Fortran-code is somewhat faster than the C++ and Java versions.

### 6.3 Conclusion

Based on our small example problem it has been shown that C++, Java and Fortran have roughly the same degree of expressiveness in writing code for solving the toy problem. Particularly the C++ and Java code ended up looking almost identical. This should not really come as a surprise since the two languages are known to have very similar syntax. There were some indications of differences. If one had chosen the approach of implementing elements, faces and vertices as classes, C++ would have benefited from the fact that one can use operator overloading to specify arithmetic operations in the Vertex class. The absence of operator overloading in Java was also made clear in the case where one chooses the more efficient Java Array package. For a larger example this would result in a large amount of *get* and *set* calls and the code would not only be hard to read, we would also get the awkwardness of having to specify mathematical operations without using simple operators like ‘*’
. Therefore the performance gain of using the Java Array package comes at the cost of mathematical expressiveness.

In the Fortran approach we assumed we had some functionality similar to that from the framework. We did not, however, assume having a Fortran framework since Fortran lacks the necessary features for implementing frameworks. This includes inheritance, polymorphism and templates. Therefore Fortran cannot provide the generalization and reusability one gets from object oriented frameworks. The Fortran solution to the toy problem showed why Fortran is so widely used in mathematical applications, not just for performance issues but also because of the high expressiveness it has in working with arrays.
In the case of translating the framework from C++ to Java it looks as though the C++ and Java versions would require about the same amount of code and no major rework of code is required. This is true provided that we don’t have to replace templates with multiple versions of the code, one for each datatype. Regardless of this Java would require less code since it is not necessary to do any of the memory management (e.g. freeing memory by deleting pointers) that is requisite in C++ destructors. A side issue of the transition from C++ to Java is that all Java code is contained within a .class file, while C++ has both an .h and .cpp file.

The example we used did not clearly illustrate the expectations we had that Java was not as expressive as C++ and Fortran in implementing a solution to our problem. To get a better indication of expressiveness would require a larger, more complex mathematical problem. Fortran would then benefit from its simplicity in array operations and the intrinsic functions that operate on arrays, while C++ could take advantage of the operator overloading feature.
7 Summary

7.1 Summary of our discussion

This report started with a presentation of the object oriented paradigm describing the features of object orientation and what this implies from a programming point of view. We then looked at concurrency and how this can be incorporated into object oriented languages, resulting in a variety of different languages based on how concurrent features such as process management and message passing is implemented.

Next we looked at the use of object oriented languages for scientific computing, firstly by giving a presentation of some existing object oriented frameworks for scientific computing. It was also shown that from the efficiency point-of-view C++ can achieve performance comparable with Fortran. Java ranks behind these languages mainly because of the implementation technology of the Java Virtual Machine.

We then decided to set focus on the expressiveness of the three languages C++, Java and Fortran. For this purpose we defined a small mathematical problem and put it into context of an existing OO framework for solving PDE’s in a distributed environment.

We came to the conclusion that Fortran, Java and C++ has approximately the same expressiveness for solving our small problem, and that a larger problem is required in order to get a better indication of the difference in expressiveness of these languages. As for the framework we concluded that a translation to Java is manageable without too many significant changes to the structure, while Fortran lacks features, such as inheritance and polymorphism, for developing a similar framework to that in C++.

7.2 Future work

We suggest the following for further work:

- The first issue would be to design a larger mathematical problem to solve, larger meaning more code and more mathematical operations.

- Second one could try and translate the C++ framework to Java according to the suggestions provided in the previous chapter. It would be interesting to see if the removal of pointers is as straightforward as we suggest without requiring any major restructuring of the code. The same is also the case for the suggested template approach. If this works out then the translation can be done in a rather short amount of time. Should it, however, prove to be a little more complicated than first indicated and a larger restructuring of code is required then more time is needed.
One would have to get a deeper understanding of the framework functionality in order to make sure that no functionality is lost during the code restructuring. To build the framework from scratch is something we do not believe would be necessary.

- As soon as the Java framework is up and running the next issue to look into is performance. Our guess is that the C++ version would be by far the most efficient. The work should then be centered around improving performance of the Java framework version. This would probably call for some rework of code and possibly the inclusion of the Java Array package. As a result of this one would presumably find stronger evidence of the better expressiveness of C++ versus Java in efficient mathematical applications. We also suspect that results would show that the Java version does not match the C++ version in performance either.

The conclusion of any further work would most likely be that a framework for solving partial differential equations using the finite element method is best implemented in C++.
Bibliography


[8] Indiana University Open Systems Laboratory OOMPI Home Page http://www.osl.iu.edu/research/oompi/


84


