# On the Numerical Efficiency of C++ in Scientific Computing*

Erlend Arge  Are Magnus Bruaset  Phillip B. Calvin
Joseph F. Kanney  Hans Petter Langtangen
Cass T. Miller

September 7, 1999

**Abstract**

We investigate the relative efficiency of C++ and C code versus FORTRAN 77 code through numerical experiments conducted on a range of computer platforms. The problem areas cover basic linear algebra and finite element solution of porous media fluid flow and species transport problems. The C++ codes are short and make extensive use of Diffpack, a generic library based on object-oriented programming techniques, while the FORTRAN and C programs are either based on vendor supplied numerical libraries or written and tuned particularly for the test problem. Challenges encountered in optimizing C++ codes and the efficiency of dynamic memory handling in C++ are also addressed.

Differences in computational efficiency observed for the problem areas studied were small, and tended to be problem dependent. Based on our experience with optimizing C++ code, we conclude that the use of object-oriented techniques should be confined to high-level administrative tasks, while CPU intensive numerics should be implemented using low-level C code and carefully constructed for-loops.

## 1 Introduction

As numerical simulators for increasingly large and complex problems are developed, scientific computing has a growing need for advanced software development paradigms. Recently, there has been much interest in the use of object-oriented software development techniques, and especially the programming language C++, which provides the traditional C language with powerful mechanisms for object-oriented (OO) implementation; see [4, 12]. However, the use of C++ for numerical applications has been criticized because its computational efficiency is commonly believed to be much lower than that of comparable

*This paper was originally published in *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pages 93–119. Birkhäuser, 1997.

FORTRAN codes. Two recent studies of C++ performance on low level linear algebra problems and simple vector expressions [14, 26] offer valuable insights into the problem of optimizing C++ codes, but we know of no published work which investigates the relative efficiency of C++ over a wide range of computational problems encountered in science and engineering.

We have investigated the relative efficiency question through a series of numerical experiments in which we compared C++ applications to C and FORTRAN codes for low-level linear algebra operations and to hand-optimized FORTRAN partial differential equation (PDE) solvers. These comparisons cover a range of computational procedures widely used in scientific computing, and thus provide a broad measure of performance.

For all the reported experiments, the C++ programs have been implemented as Diffpack applications. Diffpack is a large C++ library offering general tools for the numerical solution of PDEs by finite element methods; see [8, 11]. The OO design of Diffpack provides the user with access to software tools at an abstract level close to the mathematical formulation of the problem. The package also offers a high level of flexibility in that the selection of mesh generators, discretization rules, matrix storage schemes, and linear and nonlinear solvers can be made at run-time. The quest for FORTRAN-like numerical performance has been a driving force in the design of the Diffpack kernel. At the lowest levels of the Diffpack libraries, this goal has in many cases overruled the type of design considerations advocated by C++ purists. Such compromises, however, are not visible to the end-user who accesses this functionality through a clean interface at the top level.

For this investigation, the FORTRAN codes were written in FORTRAN 77, not FORTRAN 90. FORTRAN 90 is a superset of FORTRAN 77 which introduces many new features, including pointers, dynamic memory allocation, derived data types, data encapsulation, operator overloading, and a large set of powerful intrinsic functions for array manipulation [15]. FORTRAN 90 offers fewer and less powerful object-oriented features than C++, but offers direct support for data parallelism and lazy function evaluation, both of which are lacking in C++. We have not included comparisons with FORTRAN 90 since it is not as widely used as FORTRAN 77 and C at this time. However, because FORTRAN 90 (or its successor FORTRAN 95) is destined to supplant FORTRAN 77 in the coming years, we will include a few comments regarding the performance of FORTRAN 90 in our concluding remarks.

The numerical experiments performed in this investigation were run on several popular UNIX workstation configurations as summarized in Tables 1 and 2. We note that all the numerical experiments were conducted in a computing environment with relatively low user loads.

As shown in Table 2, our choice of compiler switches was relatively uniform across the languages. We experimented with aggressive optimization switches such as `+Oaggressive` and `+Oloop_unroll` for some of the low-level tests in Section 3, and while we were occasionally able to increase the performance of individual tests, the results were not consistent. A switch might increase performance for a particular problem of a certain size, but using that same switch

on another problem often reduced performance. Since fine-tuning of compiler switches was not guaranteed to increase performance across all languages, all problems, and problems of all sizes, we chose the compiler optimization levels for all tests *a priori*.

In Section 2, we discuss several C++ language features, such as dynamic memory allocation, dynamic binding, and operator overloading, that pose challenges to code optimization. In Section 3 we address the computational efficiency of commonly used vector operations from the Level 1 BLAS library, as well as matrix-vector products for dense and sparse matrices. In Section 4 we discuss full-scale comparisons of C++ programs and highly-tuned FORTRAN codes for the approximation of four different PDEs using finite element methods. Finally, in Section 5 we summarize our experiences with a few concluding remarks.

A complete collection of the test examples in this chapter, with associated source code and makefiles, is available on Internet [28].

| Identifier | Model | Operating system | CPU/MHz | RAM (Mb) |
|---|---|---|---|---|
| IBM | IBM RS6000/590 | AIX 3.2.5 | Power2/66 | 256 |
| SGI | SGI Indy | IRIX 5.3 | R4400/100 | 92 |
| HP | HP 9000/735 | HP-UX 9.05 | PA-RISC/99 | 144 |
| SUN | Sun 10/512 | SunOS 5.4 | SuperSPARC/50 | 64 |

Table 1: *The hardware platforms used for the numerical experiments.*

| Identifier | Language | Compiler version | Compilation flags |
|---|---|---|---|
| IBM | C | 2.1.3 | xlc -O3 |
| | C++ | 2.1.3 | xlC -O3 |
| | FORTRAN | 3.2.2.1 | xlf -O3 |
| SGI | C | 3.19 | cc -O2 -mips2 |
| | C++ | 3.2.1 | CC -O2 -mips2 |
| | FORTRAN | 4.0.2 | f77 -O2 -mips2 |
| HP | C | 9.69 | cc +O4 |
| | C++ | 3.50 | CC +O4 |
| | FORTRAN | 9.16 | f77 +O4 |
| SUN | C | 3.0.1 | cc -xO3 |
| | C++ | 4.0.1 | CC -O -fast |
| | FORTRAN | 3.0.1 | f77 -O -fast |

Table 2: *The compiler versions and the corresponding options used for the numerical experiments.*

# 2 Optimizing C++ for Numerical Applications

C++ provides powerful support for object-oriented programming (OOP) through language features such as virtual base classes, multiple inheritance, polymorphic functions, and operator overloading. However, since these and other features are implemented using dynamic memory allocation and deallocation and run-time binding of procedure calls, they present much greater challenges to an optimizing compiler than can those found in traditional FORTRAN codes. Additionally, due to FORTRAN's popularity and longevity, compiler writers have had much more experience in developing high-quality optimizing compilers for this language than for C++. In this section, we discuss how some of the C++ optimization issues have been addressed in the Diffpack libraries.

## 2.1 Inheritance and Class Hierarchies

The expressive power of OOP is achieved through careful design of class hierarchies using inheritance of data structures and functions. This provides the user with a flexible interface at a very high abstraction level, isolated from the low-level code. However, inheritance also presents an optimization challenge for the compiler. As an example of this complexity, we present a simplified diagram of the vector class hierarchy from Diffpack in Figure 1. For full details of the Diffpack vector implementation, we refer to [9, 7].

At the lowest level is the `VecSimplest` class, a C-style array with indexing operations implemented as *inline* functions; see [2, 7]. The next level is `VecSimple`, which inherits all the functionality of the `VecSimplest` class, and adds I/O and assignment functionality to the data structures. Class `VecSort` adds sorting functions, and the numerical vector class `Vec` adds numerical functions. It is able to perform the numerical computations that in FORTRAN are delivered by the level 1 BLAS routines, such as inner products, norms and vector additions. Side by side with this vector hierarchy are the classes `ArrayGenSimplest`, `ArrayGenSimple` and `ArrayGen`. In contrast to the one-dimensional numbering used by the traditional vector classes, these implementations of generalized arrays are able to handle multi-dimensional indexing as well. This feature is very convenient when discretizing partial differential equations by finite difference schemes.

The class `Vector` in Figure 1 is a shell class that allows the user code to employ vector objects without knowing at compile-time whether they will be of type `Vec` or `ArrayGen`. Instead, such choices are made at run-time using the mechanisms of dynamic binding; see [9, 2].

Note that there are two co-existing vector hierarchies, both built on top of the virtual base class `VecSimplest`. With regard to optimization, this means that the internal representation of a `Vec` or `ArrayGen` object may be scattered around in several memory segments in the sense that data members are grouped according to the levels in the vector hierarchy. As indicated in Figure 2, a `Vec` object occupies memory segment $S_{\mathrm{A}}$, where there is a pointer to memory

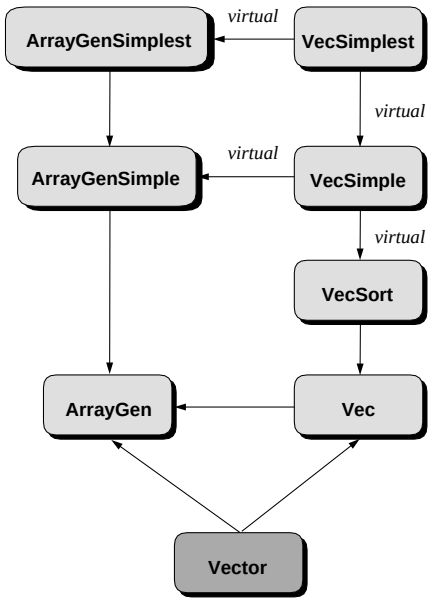


Figure 1: *The Diffpack vector hierarchy.*

segment $S_{\mathrm{B}}$ holding the data structures inherited from its virtual base class[1] `VecSimplest`. One of the inherited data members is the pointer `a` to the actual vector entries residing in memory segment $S_{\mathrm{C}}$. This situation illustrates the fact that a C++ program may have to dereference a sequence of system-defined pointers in order to access the numerical data. In most cases, this indirection can only be done at run-time, forcing the C++ optimizers to address problems that are not present for a language like FORTRAN. It is advisable to keep such matters in mind when implementing CPU intensive operations typically found as low-level member functions in numerical objects.

For example, in the vector class above, a vital member function such as the inner product could declare a local pointer and set it to point to the first entry of the C array `a`. When implementing the inner product loop using this local pointer, the problem of optimization is no worse than in a plain C code. However, we tested this technique in the inner product function defined as part of class `Vec`, and observed that it was no faster than the use of the inlined indexing operator. These observations show that the current C++ compiler technology is able to optimize a presumably difficult problem.[2] When dealing with more complicated class structures, however, compiler optimization may not be as successful.

[1] For simplicity, this discussion ignores the fact that `VecSimple` is also a virtual base class for `Vec`.

[2] With the IBM C++ compiler we have detected an efficiency increase in the dense matrix-vector product when local pointers are used instead of the indexing operator.

```
class Vec : public virtual VecSimplest
{
   public:
      Vec  (int length);
      ~Vec ();
      •••
}
```

```
class VecSimplest
{
   protected:
      double* a;
      int     n;
   public:
      VecSimplest  (int length);
      ~VecSimplest ();
      •••
}
```

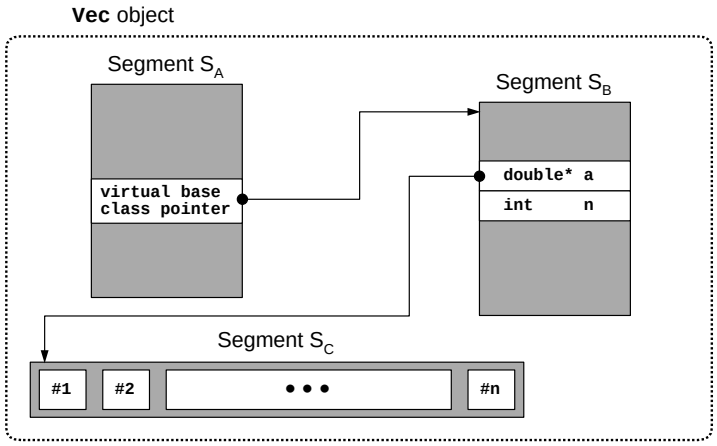


Figure 2: *An example of fragmented representation of a* `Vec` *object, having* `VecSimplest` *as a virtual base class. This object is stored in three memory segments* $S_{\mathrm{A}}$, $S_{\mathrm{B}}$ *and* $S_{\mathrm{C}}$ *according to the* `Vec`, `VecSimplest` *and the C-style array* `a` *holding the numerical values.*

## 2.2 Memory Allocation and Deallocation

In C++, as in C, we may take advantage of dynamic memory allocation, which permits us to specify at run-time the exact amount of memory needed for the computations. Depending on the application, we allocate either a few large memory segments (e.g., large matrices) or many small segments (e.g., points in 3D space). In this section we discuss how the performance of such programs is affected by the memory administration scheme, i.e., the way memory segments are allocated and freed. A significant related issue is the efficiency with which previously allocated memory is accessed, but space does not permit discussion of this topic here.

**The C++ Operators `new` and `delete`** The dynamic memory handling capabilities offered by the global operators `new` and `delete` are crucial for the design of flexible C++ programs. However, frequent calls to these operators can, in certain situations, result in unacceptable computational overhead. Consider the

| allocation (s) | deallocation (s) | | |
|---|---|---|---|
| | forward | backward | unstructured |
| 1.61 | 1.50 | 1.72 | 12.23 |

Table 3: *CPU estimates on the HP platform for allocation and deallocation of* $250,000$ *instances of class* `BasicVec`.

following simple vector class as an example:

```
class BasicVec {

public:
  int* a; int lth;
  BasicVec  ()      {a = NULL; lth = 0;}         //null constructor
  BasicVec  (int n) {a = new int[n]; lth = n;}   //constructor with length
  ~BasicVec ()      {if (a) delete a;}           //destructor
};
```

In order to experiment with allocation and deallocation of objects of type `BasicVec`, we used a simple program that allocated $\mathtt{n} = 250,000$ instances with the statements

```
BasicVec** V = new BasicVec*[n];
for (int i=0; i<n; i++)
  V[i] = new BasicVec(3);
```

The deallocation was implemented in three different ways by executing the statement `delete V[i]` in different orderings. When objects are deleted in the same order in which they were allocated, we refer to the process as *forward* deallocation and when objects are deleted in reverse order, we call it *backward* deallocation. We also deallocated the objects in an ordering unrelated to the allocation, and refer to this as *unstructured* deallocation.

Table 3 shows the time needed on the HP platform to execute our test problem with the various deallocation methods. Unstructured deallocation requires substantially more time than either of the ordered means. Since unstructured deallocation can occur frequently in complex programs, an efficient implementation should minimize the number of calls to the global operators `new` and `delete`.

Allocating a single large chunk of memory for a program mimics the standard FORTRAN approach. The disadvantage of this memory handling method is its non-dynamic nature, resulting in a large overhead in memory usage. At the other extreme, one could allocate many small chunks of memory, but this incurs a computational penalty due to the large number of calls to `new` and `delete`. The optimal choice would lie somewhere between these two extremes.

**An Improved Memory Allocation Scheme** To improve memory allocation and deallocation performance in C++, one may overload the functions `new`

and `delete` for class `BasicVec`. The overloaded versions allocate and deallocate large chunks of memory less often, storing the `BasicVec` objects in these chunks. This can be done transparently, without any changes to the main program or any modules that employ `BasicVec`. A new implementation might read as follows:

```
class BasicVec {
public:
  static MemoryPool pool;
  int* a; int lth;

  BasicVec  ()      {a = NULL; lth = 0;}
  BasicVec  (int n) {a = (int*)pool.alloc(n*sizeof(int)); lth = n;}
 ~BasicVec ()       {if (a) pool.free(a);}

  void* operator new (size_t t)  {return pool.alloc(t);}
  void operator delete (void* v) {pool.free(v);}
};
```

The memory occupied by `BasicVec` and its data members is now allocated from the storage managed by class `MemoryPool`. The operator `BasicVec::new` is called by statements of the form `BasicVec* v = new BasicVec()`. The statement `delete v` issues a call to `BasicVec::delete`.

The pool can be implemented in various ways, but in our case the implementation follows the following principles. The internal data structure of the pool is a list of memory chunks, each chunk being a class containing an array of bytes (`char* storage`), the length of the array (`int nstorage`), a pointer to the next free byte (`int nextfree`) and a counter for the number of objects allocated from the chunk (`int norefs`). Each time `MemoryPool::alloc` is called, the pool tries to allocate the requested number of bytes from the current chunk. If the current chunk contains enough free space for the request, `nextfree` is reset accordingly and `norefs` is incremented, so that no new memory is allocated. If the current chunk is not large enough, a new chunk is created with a call to the global `new`. The new chunk is allocated large enough to meet the request, though never smaller than a minimum size (the `chunk_size`) which is set by the user. When deallocating memory, `MemoryPool::free` looks up the chunk containing the address to be freed. The list of chunks is sorted on memory addresses, so the search for the right chunk is performed by binary search. The `norefs` variable in the correct chunk is decremented, and if `norefs=0` the chunk is deleted with global `delete`.

Table 4 shows the result of executing the (unaltered) main program with various sizes of `chunk_size`. It is apparent that small `chunk_size`'s give a large overhead in the workload due to extensive searching in the chunk list. The notable difference between forward and backward deallocation for `chunk_size==100` is due to the particular implementation of the binary search in the chunk list. With a slightly different implementation, the results would be reversed. However, since the optimization of unstructured deallocation is the main goal (cf. Table 3), and the difference is negligible for larger values of `chunk_size`, this behavior is not of primary importance. With large values of `chunk_size` the

| chunk_size | allocation (s) | deallocation (s) | | |
|---|---|---|---|---|
| | | forward | backward | unstructured |
| 100 | 14.53 | 267.37 | 1.80 | 143.00 |
| 1,000 | 0.65 | 3.63 | 0.91 | 4.02 |
| 10,000 | 0.67 | 0.91 | 0.82 | 1.90 |
| 100,000 | 0.60 | 0.87 | 0.82 | 1.08 |
| 1,000,000 | 0.54 | 0.85 | 0.80 | 1.01 |

Table 4: *CPU estimates on the HP platform for allocation and deallocation of* $250,000$ *instances of class* `BasicVec` *with memory allocation from the class* `MemoryPool`.

workload is notably decreased for all the deallocation schemes as compared to Table 3. The amount of work for the different deallocation schemes are approximately the same, and when compared to the use of global `new` and `delete`, this technique improves the speed of unstructured deallocation by a factor of 12. On SUN and SGI there was practically no extra cost involved in using the unstructured deallocation scheme together with the default C++ `delete` operator, thus indicating a more robust implementation of the default `new` and `delete` operators on these platforms.

## 2.3 Operator Overloading

A C++ programmer may be tempted to apply the attractive syntax and advanced language features that are often promoted in textbooks (see, for example, [4]). However, our investigations on the numerical performance of C++ programs clearly show that OOP should only be used for high-level administration, while the CPU intensive numerics should take place in functions involving plain C arrays and carefully constructed for-loops.

For example, C++ programmers are often tempted to use overloaded arithmetic operators for vectors when implementing the `DAXPY` operation. That is, one simply writes `y=y+a*x`, where `y` and `x` are vectors and `a` is a scalar. This attractive syntax involves unnecessary allocation of temporary variables since the result of `a*x` must be a vector that can be added to `y`. Testing of this syntax showed that the CPU time was increased by a factor of at least three compared to the implementation in the Diffpack library, which uses a plain member function in class `Vec`. Straight forward implementation of overloaded arithmetic operators used with large array structures will unfortunately yield a significant loss of computational efficiency using present compilers. Haney [14] addresses this problem by introducing classes and overloaded operators that can recycle temporaries using a memory management scheme similar to the one discussed in this section. Robinson [26] employs a relatively new technique called expression templates [29] to perform compile-time transformations which reduce the number of temporaries created. However, not all compilers support this technique

yet.

Our investigations show that by restricting OO techniques to high abstraction levels, current C++ compilers are able to recognize the low-level parts as FORTRAN-like and thus are able to produce better optimized executables. Such a layered design, which is in some contrast to many other C++ based efforts, has been successfully used in Diffpack. This project has also verified that the resulting modularity allows critical member functions to be re-implemented and tuned for maximum efficiency without any changes in the user code.

# 3 Low-level Linear Algebra Operations

In many numerical applications, a large fraction of the total computing time is spent on low-level operations involving vectors and matrices. In this section we will compare the computational efficiency of selected linear algebra operations, using implementations in C, C++, and FORTRAN on four different hardware platforms. The test problems are operations from the level 1 BLAS routines [17] as well as dense and sparse matrix-vector products.

## 3.1 Vector Operations

We have experimented with implementations of vector update (`DAXPY`), copy (`DCOPY`), inner product (`DDOT`) and scaling (`DSCAL`). The results from these tests were quite similar so we present only two cases in detail: (*i*) `DAXPY`: double precision vector update, i.e., $y \leftarrow \alpha x + y$; and (*ii*) `DDOT`: double precision inner product, i.e., $dot \leftarrow x^T y$. In FORTRAN, we have used the BLAS implementation,[3] while the C versions are coded from scratch. The C++ programs consist mainly of calls to the corresponding member functions of the `Vec(double)`[4] vector class in Diffpack [7].

The actual performance results observed for the BLAS operations are presented below. These experiments were conducted for a sequence of vector lengths, $n = 10^i$, $i = 1, 2, \ldots, 6$. Each vector operation was run $10^9 n^{-1}$ number of times in order to easily detect dependency of the CPU time on $n$.

The same pattern of results was observed for all languages. For the shortest vector lengths, poor performance was obtained. Medium-sized vectors ($n \leq 10^5$) were very efficient since they fit entirely within the data cache. Longer vectors incurred cache misses, forcing the machines to fetch values from the main memory; as a result, performance dropped off dramatically. We note that the relative efficiency of C++ was better for long vectors than for shorter

[3] On the IBM, SGI and HP platforms we have used BLAS libraries supplied by the vendor, whereas the SUN results are produced by the generic BLAS implementation found on netlib [23]. Experiments conducted with the ESSL library on IBM gave approximately the same results as those obtained by BLAS and are therefore not reported in this chapter.

[4] The notation `Vec(double)` refers to a vector object of type `Vec` holding values of type `double`. This template-like class parameterization is obtained by use of preprocessor macros. In the rest of this chapter we will not specify the entry type explicitly, but rather refer to `Vec` with no arguments.

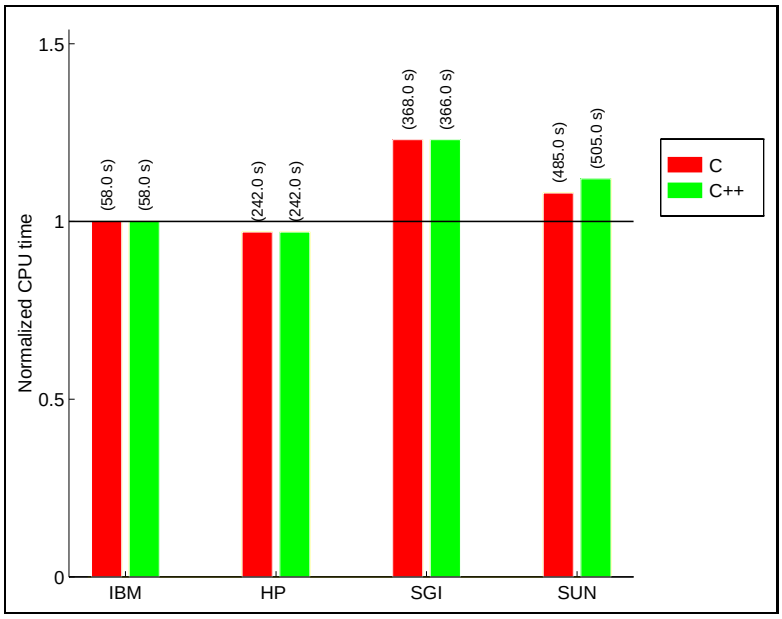


Figure 3: *The normalized CPU time of the* `DAXPY` *operation.*

ones. We interpret this as an effect of larger function call overhead in C++. A similar pattern was noted by Haney [14], who examined systems of order $10^4$ and smaller. Since large-scale simulation codes in many disciplines will typically call BLAS functions with vector lengths of $10^6$ and larger, we present results for only the $n = 10^6$ case.

**The `DAXPY` Operation** Figure 3 provides a comparison of the CPU time of a `DAXPY` operation on the chosen hardware platforms. In all plots in this chapter, the CPU times on each platform are normalized by the CPU time required by the FORTRAN code. Above each bar the accumulated CPU time for a given number of repetitions of the vector operation is given in parenthesis. On IBM and HP there are hardly any differences among FORTRAN, C++ (Diffpack) and C. We see that FORTRAN is almost 25% faster on SGI.

In initial testing, we found that a plain C implementation with standard array indexing ran significantly slower than FORTRAN and C++ (Diffpack) on the IBM platform. The optimized Diffpack functions in class `Vec` make use of direct pointer manipulation instead of plain array indexing. When we switched the C code to pointer arithmetic on the IBM, performance matched the FORTRAN and C++. This change was not necessary on the other platforms, whose compilers were able to optimize the plain C array indexing. The performance of C was also improved by using compile-time constants (as in the FORTRAN code) for the array lengths, but the results shown here correspond to dynamic arrays in C and C++.

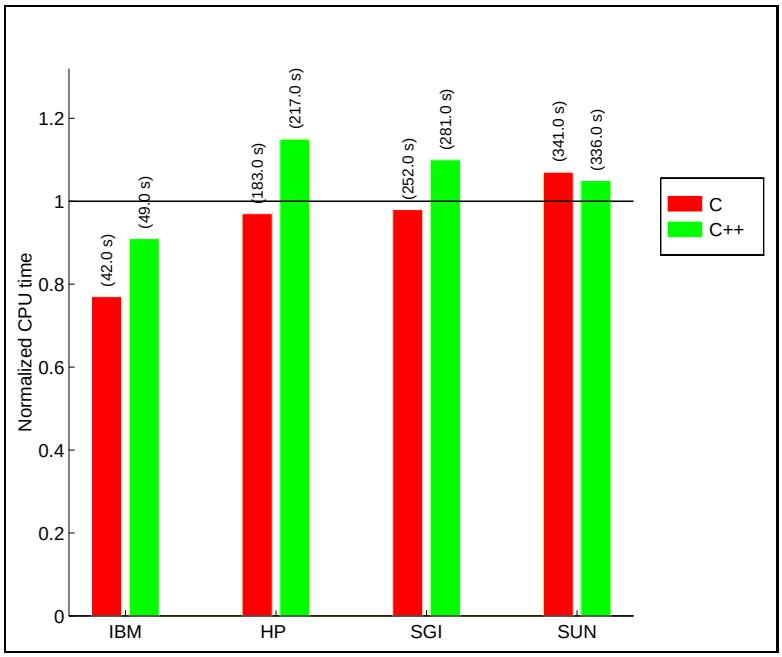


Figure 4: *The normalized CPU time of the* `DDOT` *operation.*

**The `DDOT` Operation** Figure 4 provides a comparison of the inner product computations. The DDOT test shows small differences among the relative efficiency of the implementations across various platforms. On the IBM, C++ actually ran faster than the highly-optimized BLAS and ESSL FORTRAN code in our experiments.

The main conclusion is that on all the computer environments we have used, the differences in CPU time among FORTRAN BLAS, hand-coded C implementations or generic C++ libraries are not significant. The IBM's ESSL/BLAS routines do not perform any better than plain C/C++, but IBM experts have not yet been able to find a clear reason for this result.

We also point out that we have experimented with computations of the norm of a vector, comparing naive implementations in C and Diffpack with the `DNRM2` BLAS1 routine. In this test, the `DNRM2` routine was significantly slower on all platforms. The reason is that `DNRM2` performs some scaling to avoid overflow in extreme cases. In many applications, e.g. when solving PDEs, the efficiency loss due to higher reliability in the norm function is seldom desired, so this is an example where there are clear reasons for writing one's own function instead of using the BLAS library.

## 3.2 Matrix-vector Products

In this section we investigate two types of matrix-vector products. In the first case, we study the computation for medium–sized ($n = 1000$) dense matrices. Thereafter, we focus on the matrix-vector product based on a large ($n = 63,001$) sparse matrix obtained from finite element discretization of a partial differential equation. This type of multiplication is common in numerical simulators for PDEs when the resulting algebraic systems are solved by Krylov subspace methods [5, 7].

**Dense Matrix-vector Products** Consider the matrix-vector multiplication $y = Ax$, where $x, y \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n,n}$ is a dense matrix, i.e., all $n^2$ entries of $A$ are stored and contribute to the product. We implemented this operation by hand in C and by calling the BLAS2 routine `DGEMV` in FORTRAN and compared the execution times against the results from the C++ product function in the Diffpack class `Mat`. This class, which represents dense matrices of size $m \times n$, is a part of the Diffpack matrix hierarchy. In terms of structure and complexity, the `Mat` class hierarchy in Diffpack has many similarities to the `Vector` hierarchy discussed earlier, so the comments concerning optimization of C++ code found in that discussion are valid for matrix-vector product operations as well.

We note that the BLAS routine `DGEMV` is doing more work since it actually performs an update $y \leftarrow \alpha Ax + \beta y$. However, when $\beta = 1$, the $\beta * y$ multiplication is not done, and the inner loop reduces to a fused multiply and addition $y_i \leftarrow y_i + T * A(i,j)$ where T is precomputed as $T \leftarrow \alpha * x(j)$. On a RISC processor, this should take the same amount of CPU time as the multiplication alone.

CPU time comparisons for the dense matrix-vector product are shown in Figure 5. These are the results of executing 1,000 product computations with a matrix of size $n = 1,000$. Again, the results are normalized by the CPU time required by the FORTRAN code on each platform. As in the BLAS1 tests, C and C++ were less efficient than FORTRAN for small problems ($n \leq 100$), but the results improved for larger problems.

The differences among the languages were usually small, but we noticed that C++ was clearly inferior on IBM. Our C code applies static C arrays. Switching to dynamic memory segments results in approximately the same execution time for C and C++ (40 seconds for C in Figure 5). Again we observed that pointer arithmetic in C or C++ instead of plain array indexing increased the efficiency on IBM. Replacing the `DGEMV` BLAS2 call by a call to a straightforward hand-written FORTRAN routine increased the CPU time on IBM and SGI, but not on HP.

While not directly comparable, the dense matrix-vector product test shares some basic features of the real matrix multiplication tests conducted by Haney [14] and Robinson [26]. The use of pointer arithmetic in Diffpack allows almost equivalent C and C++ performance, in contrast to Haney, who reported that a C++ code using indirect addressing was up to three times slower than C and FORTRAN. Robinson reported nearly equivalent C and C++ performance

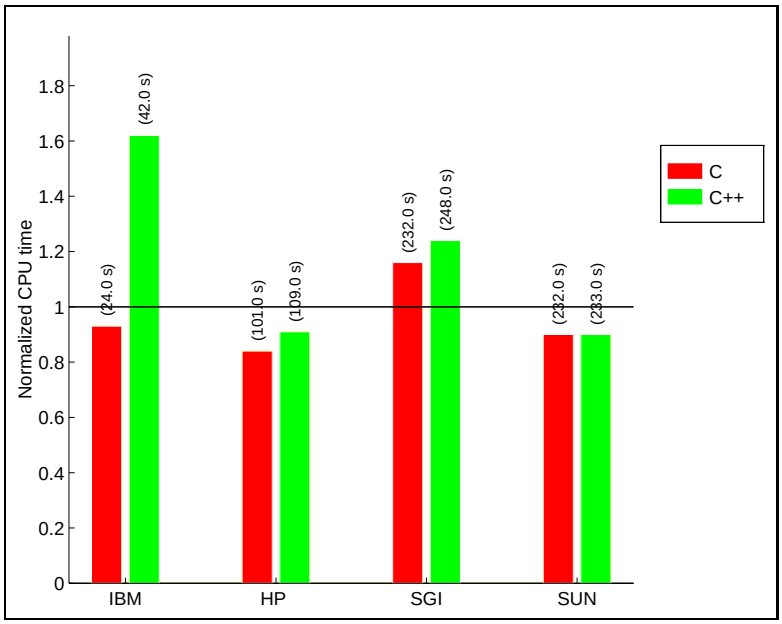


Figure 5: *The normalized CPU time of a dense matrix by vector product.*

by using language specific optimizations provided by the KCC compiler, which illustrates the positive impact that advances in compiler technology can have on C++ performance. In contrast, the CPU-intensive parts of the C++ code in Diffpack is structured so that the compiler sees almost the same code as in C or FORTRAN, and can therefore apply the same optimization technology[5].

**Sparse Matrix-vector Products** When discretizing a PDE by a finite difference or finite element method, one ultimately obtains a system of algebraic equations, $Au = b$. Typically, the coefficient matrix $A$ will be large and sparse; only a small fraction of the matrix entries are nonzero. One will usually have on the order of $kn$ nonzeroes, where $n$ is the number of matrix rows (or columns) and $k$ is some small constant integer that is independent of $n$. In order to save storage space and computation time, only the nonzeroes are stored together with some index information. Usually, such sparse systems are solved using a Krylov-subspace method [7], where the matrix $A$ enters the computation only in terms of a matrix-vector product. This sparse matrix-vector product is the most expensive part of these solvers, and is a meaningful candidate for efficiency

[5]On many platforms a significant portion of the optimization takes place on the object code level, thus at this stage invoking the same code optimization module regardless of the high-level language used. Since the layered design of Diffpack results in object code that to a large extent resembles code produced by the C and FORTRAN compilers, the backend optimizers originally developed for these languages can be successfully applied also to our C++ implementations.

comparisons.

Many different storage schemes can be used for sparse matrices with irregular patterns [3]. In this test we have chosen the Compressed Row Storage (CRS) scheme, as implemented in the sparse matrix class in Diffpack. The CRS scheme uses three vectors: `double* A`, `int* irow`, and `int* jcol`, such that `A` contains the nonzero matrix entries row by row, the first entry in row `i` is `A[irow[i]]`, and the column number of that entry is given by `jcol[irow[i]]`. It is clear that the matrix-vector product $y = Ax$ will require indirect addressing, so it is vitally important to implement the product functions so that the array `A` is accessed sequentially from the first to the last entry.

To compare the efficiency of the CRS-based matrix-vector product in different languages, the execution times for one single product involving a matrix of size $n = 63,001$ containing 564,001 nonzeroes were recorded. This matrix, which arises in a finite element discretization of a 2D advection-diffusion equation posed on the unit square, has a regular sparsity pattern where all nonzeros are located on certain diagonals; i.e., it is banded. However, this property is not exploited in the CRS implementation, which is designed for arbitrary sparsity patterns. The implementations cover standard Diffpack code for a sparse matrix-vector product (class `MatSparse`) and hand-coded FORTRAN and C routines. The CPU times ranged from 55.7 seconds on IBM to 129.7 seconds on SUN. However, there were negligible differences among the three languages. Consequently, the most time-critical part of Krylov subspace solvers, such as the conjugate gradient method, will run at the same speed whether C, C++ or FORTRAN is used for the CRS implementation.

# 4 Finite Element Applications

In this section we investigate the computational efficiency of FORTRAN versus C++ using scientific applications involving fluid flow and species transport problems solved by finite element methods. The FORTRAN applications are highly-tuned special-purpose codes, whereas the C++ codes are short and general Diffpack programs written at a high level of abstraction. This presumptively unfair comparison will show the computational penalty associated with applying general codes based upon the C++ programming language.

## 4.1 Species Transport Applications

Consider the advection-dispersion-reaction equation (ADRE) with initial and boundary conditions:

$$\frac{\partial C}{\partial t} = \nabla \cdot (\mathbf{D} \cdot \nabla C) - \mathbf{v} \cdot \nabla C - kC^2 \text{ in } \Omega \times [0,T], \quad (1)$$

$$C = C_0 \text{ on } ,_1 \times [0,T], \quad (2)$$

$$\frac{\partial C}{\partial n} = 0 \text{ on } ,_2 \times [0,T], \quad (3)$$

$$C(\mathbf{x},0) = 0 \text{ in } \Omega \quad (4)$$

Here $C(\mathbf{x},t)$ is a solute concentration in a flow field with velocity $\mathbf{v}$, $\mathbf{D}$ is the hydrodynamic dispersion tensor, $\Omega$ is the domain of interest, and $,_1$ and $,_2$ denote the boundaries: $\partial\Omega = ,_1 \cup ,_2$, with $,_1 \cap ,_2 = \emptyset$. In porous media flow, $\mathbf{v}$ is computed by solving a parabolic PDE for pressure and applying Darcy's law, but in our test problems we simply prescribe $\mathbf{v} = v\mathbf{i}$, where $\mathbf{i}$ is the unit vector in the $x_1$-direction. The $\mathbf{D}$ tensor is taken to be symmetric and constant. Moreover, $\Omega$ is a hypercube $0 \le x_1 \le L$, $0 \le x_i \le B$, $i = 2, 3$, and $,_1$ is the plane $x_1 = 0$.

We solve the problem by a standard Bubnov-Galerkin finite element method with multilinear elements of the same size. Integrals are computed by Gauss quadrature. The discretization in time is based on backward (fully implicit) finite differences. The resulting discrete equations take the form

$$[\mathbf{B} + \mathbf{A} + \mathbf{R}\,(\mathbf{c}^n)]\,\mathbf{c}^n = \mathbf{B}\mathbf{c}^{n-1} \tag{5}$$

Here $\mathbf{B}$ is the mass matrix, $\mathbf{A}$ arises from the advection and dispersion terms, $\mathbf{R}(\mathbf{c}^n)\mathbf{c}^n$ is the vector corresponding to the reaction term $kC^2$, and $\mathbf{c}$ is the vector of nodal values of $C$. The superscript $n$ denotes the time level. Note that $\mathbf{A}$ and $\mathbf{B}$ are independent of $\mathbf{c}$ and time such that these need not be recomputed at every time level.

We consider three test problems on the form (1)–(4):

- ADl: A linear advection-dispersion equation ($k = 0$) in $\mathbb{R}^3$. Then equation (5) is solved as it stands.
- ADRnl: A fully nonlinear advection-dispersion-reaction equation in $\mathbb{R}^1$. Here, Newton's method is applied to equation (5).
- ADRso: As ADRnl, except that a split-operator approach is used for the nonlinear reaction term. That is, in the first step equation (5) is solved with $\mathbf{R} = 0$, and then an ODE is solved in each node in a second step.

All test cases except ADRnl have been run on IBM, HP and SGI machines. In addition, SUN was used for ADRso, while IBM and SUN were used for the ADRnl test.

## 4.2 The Linear Advection-dispersion Test

In the ADl problem, the $\mathbf{R}$-term is zero, so the linear system (5) to be solved at each time level can be written as $\mathbf{K}\,\mathbf{c}^n = \mathbf{b}$ where $\mathbf{K} = \mathbf{A} + \mathbf{B}$ and $\mathbf{b} = \mathbf{B}\mathbf{c}^{n-1}$. For this problem, $\mathbf{K}$ and $\mathbf{b}$ are time independent and are calculated prior to the time integration. A lumped mass matrix $\mathbf{B}$ is used, and integrals are approximated by two-point Gauss quadrature in each space direction. We apply the iterative Orthomin(5) method with Jacobi (diagonal) preconditioning for solving linear systems, using $\mathbf{c}^{n-1}$ as a start vector for the iterations. The iteration is stopped when the norm of the residual of the preconditioned system, relative to the right hand side, is less than $10^{-7}$. Since the finite element assembly process is only carried out initially, the CPU time in this test problem

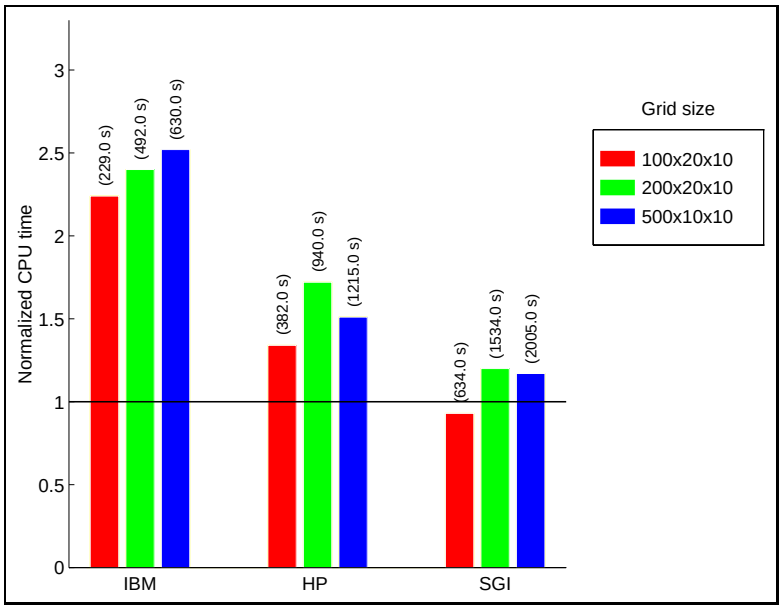


Figure 6: *The normalized CPU time of C++ versus FORTRAN in the ADl test.*

will in practice be spent in the iterative solver. Thus, the timing results will reflect the efficiency of matrix-vector products, `DAXPY` updates, dot products, and similar computations, embedded in a real simulation code. The FORTRAN code applies an efficient stencil sparse matrix storage scheme [24] based on an assumed regular sparsity pattern, while the Diffpack code uses the general CRS scheme suitable for arbitrary grids and arbitrary sparsity patterns.

Three spatial problem sizes are considered: (*i*) $100 \times 20 \times 10$ 8-node brick elements; (*ii*) $200 \times 20 \times 10$ elements; and (*iii*) $500 \times 10 \times 10$ elements, giving from 23,331 to 60,621 spatial nodes. To obtain reliable CPU time measurements, each of the finite element test problems in this chapter was executed five times to estimate the uncertainty. The standard deviation was always less than five percent. The average timing results for the ADl test are shown in Figure 6. Like in our previous studies, the CPU time on each platform is normalized by the time of the FORTRAN code on that platform. The average CPU time of a single run of the simulator is shown above each bar in parenthesis. The results in Figure 6 show significant variation with platform type. FORTRAN is clearly superior on IBM, whereas the differences are quite small on SGI. Except for the IBM, there does not seem to be a clear trend with respect to problem size. Taking into account the more efficient matrix storage scheme and the special optimizations in the FORTRAN code, the results show that, on the whole, there is only a relatively modest loss of efficiency in using the C++ language and a generic library like Diffpack.

### 4.3 The Non-linear Advection-dispersion-reaction Test

For $k = 2$, the equations (1)–(4) become nonlinear. Using a Newton-Raphson iterative method to solve the nonlinear discrete equations, a series of linear systems $\mathbf{K}\mathbf{c}^{n,m+1} = \mathbf{b}$ must be solved, where $m$ denotes the iteration level at a time level. Both the matrix $\mathbf{K} = \mathbf{B} + \mathbf{A} + \mathbf{J}$, where $\mathbf{J}$ is the Jacobian of the $\mathbf{R}(\mathbf{c}^n)\mathbf{c}^n$ term, and the right hand side

$$\mathbf{b} = (\mathbf{J} - \mathbf{R}(\mathbf{c}^{n,m}))\mathbf{c}^{n,m} + \mathbf{B}\mathbf{c}^{n-1}$$

depends on $m$ and must be recomputed in every Newton iteration. We employ a lumped (diagonal) mass matrix $\mathbf{B}$. The solution $\mathbf{c}^{n-1}$ is used as initial guess for the Newton iteration, and integrals are computed by three-point Gauss quadrature. For this test, we consider only problems in $\mathbb{R}^1$, so a direct Gaussian elimination solver is used for the linear systems. The Newton-Raphson procedure is terminated when the Euclidean norm of the residual, $||\mathbf{b} - \mathbf{K}\mathbf{c}^{n,m+1}||_2$, is less than $10^{-6}$.

The work in this problem is dominated by the formation of the linear systems since the Gaussian elimination procedure is very fast for tridiagonal matrices. Hence, the timing results will reflect the efficiency of the finite element assembly process. Normally, the assembly process is coded as an outer loop over all elements, where for each element one iterates over the integration points with loops over the elemental matrix and vector entries as the innermost loops. This assembly process can be very efficiently coded by reversing the loops, so that the innermost loop is the long loop over all elements. The FORTRAN code employs these reversed loops for efficiency, but this restricts the code to the case where all elements are of the same type and size. In contrast, the Diffpack code allows both the general element-by-element computation, referred to as EBE, and a specialized approach IEL (Innermost Element Loop) as in the FORTRAN code. In the FORTRAN code, only the parts of $\mathbf{K}$ and $\mathbf{b}$ that depend explicitly on $m$ are recomputed in each Newton iteration. The Diffpack program, being more general and flexible, recomputes the entire linear system. Thus the Diffpack code is doing more work to obtain a solution.

Three problem sizes are considered in the ADRnl test: 1,000, 10,000 and 50,000 elements. The results from the FORTRAN code and the IEL mode of the Diffpack code appear in Figure 7. Due to technical problems we were not able to produce reliable results on HP and SGI for the largest problem sizes, thus for this case we present results only for the IBM and SUN platforms. As seen, the differences between FORTRAN and C++ are modest. The superior efficiency of C++ on IBM for the largest grid is particularly striking given that the C++ implementation is more general than the one in FORTRAN. It is worth mentioning that the standard EBE approach increased the CPU time in this test by a factor between 4 and 5.

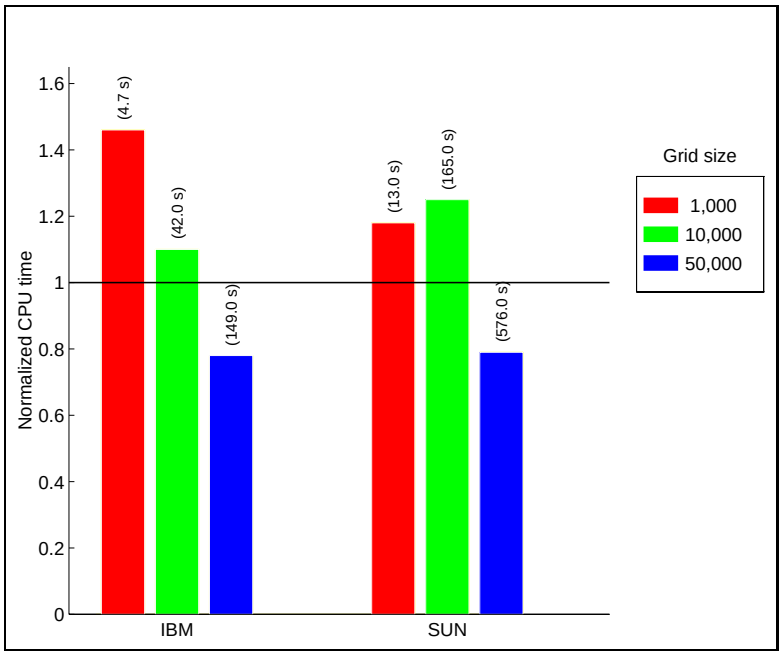


Figure 7: *The normalized CPU time of C++ versus FORTRAN in the ADRnl test.*

## 4.4 The Split-operator Test

Nonlinear reaction terms in ADREs are often resolved using split-operator approaches, which obviate the need to solve large systems of nonlinear equations. Splitting the operator may introduce additional error [22]; however, this is the only feasible method of solution for many large-scale applications such as nonlinear multiple species problems in $\mathbb{R}^3$ [16, 25]. Split-operator approaches lead to a multiple step solution process to advance a single discrete time step. The number of steps required depends on the manner in which the operator is split and whether an iterative or fixed splitting approach is employed [16]. The application considered here uses a simple two-step splitting approach without iteration [22], which contributes a splitting error of $O(\Delta t)$. This formulation first solves (1) with $k = 0$, yielding an intermediate solution $\mathbf{c}^{n-1/2}$; it then solves the ordinary differential equation $\partial C/\partial t = -kC^2$ over one time step at each nodal point, using $\mathbf{c}^{n-1/2}$ as initial condition, to yield the final solution $\mathbf{c}^n$. This two-step procedure is repeated to advance the solution in time.

Three-point Gauss quadrature rules are used to approximate integrals, and the mass matrix is lumped. We consider test problems only in $\mathbb{R}^1$, so a direct solver (Gaussian elimination) is used for the linear systems. The linear systems arising in the first step of the ADRso problem are similar to those in the ADl problem. The coefficient matrices are time independent, and can be computed initially so that updating the linear system at each time level only involves

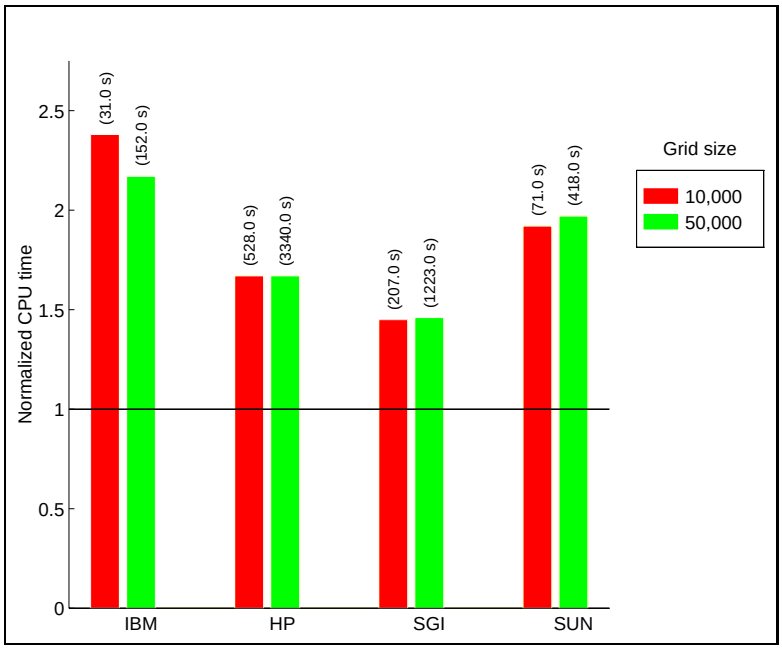


Figure 8: *The normalized CPU time of C++ versus FORTRAN in the ADRso test.*

an efficient matrix-vector product $\mathbf{B}\mathbf{c}^{n-1}$. Moreover, the second step in the operator-splitting technique can be implemented using an analytical solution of the local reaction equation. This problem will therefore test the efficiency of the linear systems solver and the node by node updating of $C$ due to the local reaction solution. Since the coefficient matrix in all the linear systems is independent of time, its LU factorization is computed only at the first time level, and almost all the CPU time is spent on the forward and back substitution at each time step. Examples were tested in $\mathbb{R}^3$; the results were consistent with the examples in $\mathbb{R}^1$, so they are omitted here.

For this test problem, the Diffpack program is essentially the same code as used for the ADl test, while the FORTRAN program is a modification of the code used for the ADRnl test. However, the effect of the reversed loops in the finite element assembly process is hardly noticed here since those operations are performed only once, and the time integration is performed for a large time interval so that the influence of the assembly process is negligible. For the ADRso test, two problem sizes corresponding to 10,000 and 50,000 elements are considered. The results appear in Figure 8 and indicate that FORTRAN was clearly superior in this problem, but only a factor less than 2.5 even in the worst case.

## 4.5 Richards' Equation

Fluid flow and species transport in porous media systems are often governed by a relatively large number of coupled, highly nonlinear PDEs [1, 13, 19]. Further, such applications often have solutions characterized by sharp fronts that propagate in space and time [21]. In this application, we consider a simple subset of the general multiphase flow and transport equations: the case of aqueous movement in a porous media system initially containing both a gas phase and an aqueous phase. Because the gas phase is much more mobile than the aqueous phase, the gas-phase pressure gradient is much smaller than the aqueous-phase pressure gradient. In the limiting case of negligible changes in gas-phase pressure, Richards' equation (RE) may be derived [18]. RE is a single equation description of aqueous flow in a two-fluid porous media. It preserves many of the features of the general multiphase equations: severe nonlinearity, relatively complex constitutive relationships, and sharp-front solutions in space and time for certain sets of auxiliary conditions.

The compressible form of RE considered here is

$$\frac{\partial \theta}{\partial t} + S_s S \frac{\partial \psi}{\partial t} = \frac{\partial}{\partial z}\left[K\left(\frac{\partial \psi}{\partial z} + 1\right)\right] \text{ in } \Omega \times [0,T], \tag{6}$$

$$\psi = 0 \text{ for } z = 0,\ 0 \le t \le T, \tag{7}$$

$$\psi = \psi_L \text{ for } z = L,\ 0 \le t \le T, \tag{8}$$

$$\psi(z,0) = -z \text{ in } \Omega, \tag{9}$$

where $\Omega = [0, L]$ is the spatial domain of interest, $\theta$ is the volume fraction of the aqueous phase, $\psi$ is the aqueous-phase pressure head, $S$ is the aqueous-phase saturation, $S_s$ is the specific storativity of the porous media resulting from compressibility of the aqueous phase, and $z$ is the vertical coordinate. The unsaturated hydraulic conductivity, $K$, is a property of the porous media and a function of $S$, which in turn depends upon $\psi$. The quantities $\theta$, $S$, and $K$ can be related to the pressure $\psi$ through constitutive relations.

The formulation described in (6)–(9) is termed a mixed-form RE (MFRE), since both $\theta$ and $\psi$ appear as dependent variables. The MFRE may be solved by one of three methods: (*i*) converting to a single dependent variable through application of the chain rule and applying any suitable nonlinear method, (*ii*) direct application of a Newton iteration procedure, or (*iii*) expansion of the $\theta$ term about the iteration level, $m$, in terms of $\psi$, followed by solution using Picard iteration [10]. We apply method (*iii*) using the truncated Taylor series approximation

$$\theta_k^{n+1,m+1} = \theta_k^{n+1,m} + (\psi_k^{n+1,m+1} - \psi_k^{n+1,m}) \left.\frac{\partial \theta_k}{\partial \psi}\right|^{n+1,m} \text{ for } k = 1, ..., N \tag{10}$$

where $N$ is the number of nodes in the system.

The constitutive relations used to close the system of equations are

$$S_e = \frac{\theta - \theta_r}{\theta_s - \theta_r} = \begin{cases} (1 + |\alpha\psi|^{n_v})^{-m_v}, & \text{for } \psi < 0 \\ 1, & \text{for } \psi \ge 0 \end{cases} \tag{11}$$

and

$$K = K_s S_e^{1/2}[1 - (1 - Se^{1/m_v})^{m_v}]^{1/2}, \tag{12}$$

where $m_v = 1 - 1/n_v$, and $\alpha$, $n_v$, and $K_s$ are measurable parameters of a porous medium.

The above initial-boundary value problem is solved using a Bubnov-Galerkin method with linear elements and a backward difference approximation of the time derivative. This yields a set of discrete nonlinear algebraic equations that are solved by Picard iteration (also called the method of successive substitutions). The mass matrix terms are lumped, and the iteration is terminated when the norm of the difference of the solution at two successive iterations, divided by the norm of the most recent solution, is less than $10^{-4}$. The coefficient matrices are reformed at each iteration using three-point Gauss quadrature. A banded LUD solver is used to solve the tridiagonal linear system at each stage of the nonlinear iteration. The known time-level solution is used as the initial guess for the nonlinear iteration at a new time level.

This test problem is more difficult than the previous problems both because it is severely nonlinear (extremely sharp fronts may propagate through the domain) and because it requires complex constitutive relations. Since most of the computational work is spent on evaluating the constitutive relations at the numerical integration points, the RE example highlights the efficiency of the finite assembly algorithms for complicated PDEs that arise in real applications. The solution of the linear systems will be very fast for this one-dimensional problem, and it is expected that the matrix assembly process will dominate the workload even in $\mathbb{R}^3$.

The FORTRAN code employs reversed loops as in the ADRnl and the ADRso examples, but this gives a negligible increase in efficiency since the function calls to constitutive relations dominate the work. The C++ program is based on the traditional and general element-by-element assembly process as given by default in Diffpack. Various constitutive models are represented by subclasses in a hierarchy. Using OOP, it is straightforward to choose the constitutive model at run-time. The computational penalty is a set of virtual function calls at every integration point. This overhead is negligible in the present problem compared to the computational burden of the constitutive relations.

The RE problem is solved for spatial mesh sizes ranging from 800 to 3,200 for 20 time steps. Timing results have been obtained from executions on HP, SGI and IBM. The CPU times are presented in Figure 9, and show that the FORTRAN code was generally faster on all platforms, but the differences on HP are small. A detailed profiling of the C++ codes showed that a large portion of the execution time is spent in the exponential `pow` function. Hence, a careful implementation of the constitutive relations, where the number of `pow` calls is reduced to a minimum, is the most important aspect of tuning these types of codes. Representing functions in constitutive relations in terms of classes in C++ makes it easy to avoid recomputation of certain expressions and at the same time achieve a user-friendly programming interface. In this way, it was easier to optimize the C++ code than the FORTRAN program.

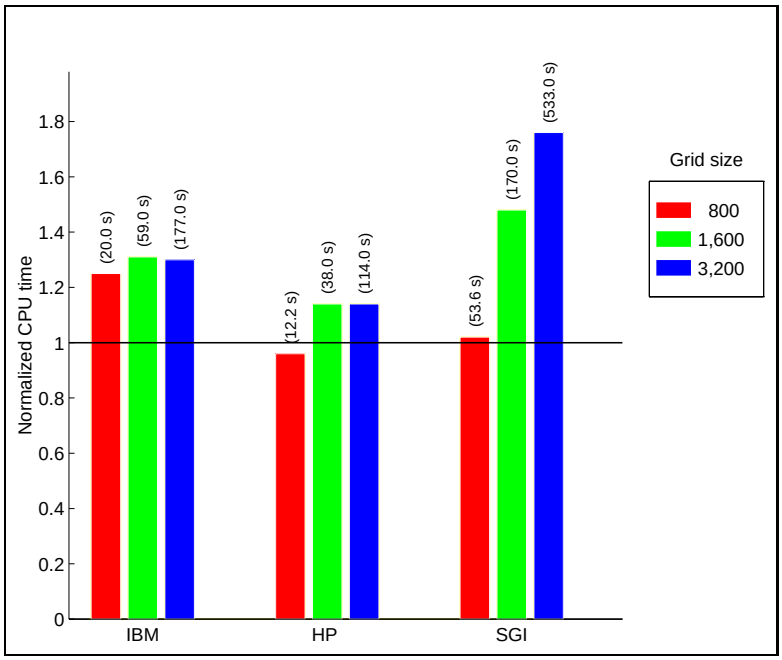


Figure 9: *The normalized CPU time of C++ versus FORTRAN in the RE test.*

# 5 Concluding Remarks

The present chapter investigated the nature and magnitude of performance penalties incurred when one programs numerical applications at high abstraction levels using general purpose C++ libraries rather than the traditional low-level FORTRAN statements. Comparisons of vector operations, matrix-vector products, dynamic memory allocation and finite element algorithms provide clear conclusions.

C++ libraries must be implemented carefully so that the CPU intensive numerics take place in functions that are easily optimized by C compilers. Most compilers can easily recognize FORTRAN-like loop constructions, containing array manipulations, for optimization. When solving partial differential equations, the time consuming parts of the code often consist of long loops with array manipulations. Given the compiler technology presently available at typical UNIX systems, we have demonstrated that object-oriented C++ mechanisms applied at high-level abstractions in combination with low-level C-style code can lead to approximately the same computational performance as for finely-tuned, procedural C and FORTRAN implementations. The attractive syntax provided by overloaded arithmetic operators for C++ vector classes, however, may drastically reduce the computational efficiency. Easily optimizable member functions should be used for maximum efficiency. Similarly, dynamic memory allocation, and deallocation provide convenient implementations of many numerical algo-

rithms, but the associated overhead can be significant, especially when working with a large number of small data structures. We have shown that by redefining the basic memory handling routines in C++, one can achieve the efficiency of FORTRAN and still have the full flexibility of C++.

A series of partial differential equation solvers based on finite element methods were implemented in FORTRAN and Diffpack. While the FORTRAN implementations were particularly tailored to and optimized for the problem in question, the Diffpack codes were more general and relied heavily on OO constructions and generic C++ libraries. We observed that run times of the C++ applications were generally higher than the FORTRAN codes. In the worst case, C++ run times were approximately 2.5 times that of FORTRAN. In the best case, the C++ application was slightly faster than FORTRAN. On average, C++ run times were approximately 50 percent higher than FORTRAN. The problem with the most challenging physics, involving a complicated nonlinear equation with nontrivial constitutive relations, demonstrated that general C++ programs written at a high abstraction level with extensive use of OO techniques can achieve a computational efficiency which comes close to hand-coded FORTRAN.

This investigation did not include comparisons with FORTRAN 90, and we know of no direct comparisons between C++ and FORTRAN 90 for the problem areas considered in this paper. However, recent performance comparisons between FORTRAN 90 and FORTRAN 77 for finite element applications [27] and case studies of FORTRAN 90 array optimization [20] indicate that some of the new array features (e.g., the MATMUL intrinsic and assumed shape arrays) can have serious negative impacts on performance. Therefore, we expect that carefully constructed C++ codes will be competitive with FORTRAN 90 for scientific computing applications.

The present experiments were conducted on UNIX workstations. One might ask whether using large-scale vector and parallel computing architectures would change the conclusions. Our impression is that current FORTRAN compilers handle vectorization-like optimization of loops better than C++ and C. However, parallel computers based on nodes with UNIX workstation-like architecture are likely to be the scientific computing environment in the future, and the test problems and conditions of the present investigations are relevant for the computational labor on a single node. Although the successful extension of complex codes such as these to parallel architectures is rarely easy, the advanced constructs provided by OO languages such as C++ will ease the process [30].

**Acknowledgements** Diffpack is being developed in a collaboration between SINTEF Applied Mathematics and the University of Oslo with financial support from the Research Council of Norway. The work performed at the Center for Multiphase Research at University of North Carolina is supported by funding from Cray Research, Inc., the North Carolina Supercomputing Center, the U.S. Army Research Office, and the U.S. Army Waterways Experiment Station.

# References


[1] L. M. Abriola and G. F. Pinder. A multiphase approach to the modeling of porous media contamination by organic compounds, 1. Equation development. *Water Resources Research*, 21(1):11–18, 1985.

[2] E. Arge, A. M. Bruaset, and H. P. Langtangen. Object-oriented numerics. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 7–26, Birkhäuser, 1997.

[3] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. A. van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM, 1993.

[4] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++. An Introduction with Advanced Techniques and Examples.* Addison-Wesley, 1994.

[5] A. M. Bruaset. *A Survey of Preconditioned Iterative Methods*, volume 328 of *Pitman Research Notes in Mathematics Series.* Addison-Wesley Longman, 1995.

[6] A. M. Bruaset. Krylov subspace iterations for sparse linear systems. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 255–280, Birkhäuser, 1997.

[7] A. M. Bruaset and H. P. Langtangen. Basic tools for linear algebra. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 27–44, Birkhäuser, 1997.

[8] A. M. Bruaset and H. P. Langtangen. A comprehensive set of tools for solving partial differential equations; Diffpack. In *Numerical Methods and Software Tools in Industrial Mathematics*, M. Dæhlen and A. Tveito (eds.), pp. 61–90, Birkhäuser, 1997.

[9] A. M. Bruaset and H. P. Langtangen: Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Trans. Math. Software*, 1997.

[10] M. A. Celia, E. T. Bouloutas and R. L. Zarba. A general mass-conservative numerical solution for the unsaturated flow equation. *Water Resources Research*, 26(7):1483–1496, 1990.

[11] Diffpack home page. *http://www.oslo.sintef.no/diffpack*

[12] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[13] R. W. Falta, K. Pruess, I. Javandel, and P. A. Witherspoon. Numerical modeling of steam injection for the removal of nonaqueous phase liquids from the subsurface, 1. Numerical formulation. *Water Resources Research*, 28(2):433–449, 1992.

[14] S. W. Haney. Is C++ fast enough for scientific computing? *Computers in Physics*, Vol 8, No 6, Nov/Dec 1994.

[15] J. Kerrigan. *Migrating to Fortran 90*. O'Reilly and Associates, 1993.

[16] W. Kinzelbach, W. Schäfer, and J. Herzer. Numerical modeling of natural and enhanced denitrification processes in aquifers. *Water Resources Research*, 27(6):1123–1135, 1991.

[17] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–329, 1979.

[18] G. de Marsily. Contaminant immobilization and containments: hydraulics – A case study. In *Subsurface Restoration Conference—Third International Conference on Water Quality Research*, pages 32–33, National Center for Ground Water Research, Houston, TX, 1992.

[19] A. S. Mayer and C. T. Miller. Simulating nonaqueous phase liquid dissolution in heterogeneous porous media. In T. F. Russell and R. E. Ewing, editors, *Mathematical Modeling in Water Resources*, pages 247–254, Computational Mechanics Publications, Southampton, UK, 1992.

[20] J. D. McCalpin. A case study of some issues in the optimization of Fortran 90 array notation. *Scientific Programming*, Vol. 5, No. 3, 1995.

[21] C. T. Miller and C. T. Kelley. A comparison of strongly convergent solution schemes for sharp-front infiltration problems. In *Proceedings of X International Conference on Computational Methods in Water Resources*, pages 325–332, Kluwer Academic Publishers, Amsterdam, The Netherlands, 1994.

[22] C. T. Miller and A. J. Rabideau. Development of split-operator, Petrov-Galerkin methods to simulate transport and diffusion problems. *Water Resources Research*, 29(7):2227–2240, 1993.

[23] Netlib home page. *http://www.netlib.org*

[24] T. Oppe. STENCIL User's Guide: A package for solving large sparse linear systems by various iterative methods. Supercomputer Computations Research Institute, Florida State University.

[25] A. J. Rabideau and C. T. Miller. Two-dimensional modeling of aquifer remediation influenced by sorption nonequilibrium and hydraulic conductivity heterogeneity. *Water Resources Research*, 30(5):1457–1470, 1994.

[26] A. D. Robinson. C++ gets faster for scientific computing. *Computers in Physics*, Vol. 10, No. 5, Sep/Oct 1996.

[27] I. M. Smith and M. A. Pettipher. Experiences with Fortran 90 in finite element analysis. Talk presented at the NAGUA Conference, Sept 13-15 1995, Numerical Algorithms Group, 1995.
Overheads available from
*http://www.hpctec.mcc.ac.uk/hpctec/courses/Fortran90/mc_f90.html*. Conference description at
*http://www.nag.co.uk:80/nagua/nagua95.html*.

[28] Source code and test cases from this chapter:
*http://www.oslo.sintef.no/diffpack/efficiency* or
*http://www.cmr.sph.unc.edu/CMR/efficiency*

[29] T. Veldhuisen. Expression Templates. *C++ Report*, 26-31, 1995. See also *http://monet.uwaterloo.ca/blitz*

[30] G. V. Wilson and P. Lu (editors). *Parallel Programming Using C++*. The MIT Press, Cambridge, Massachusetts, 1996.