



OO performance optimisations of HPC applications

Stephen Booth

*EPCC The University of Edinburgh, James Clerk Maxwell Building,
Mayfield Road, Edinburgh, EH9 3JZ, UK
s.booth@epcc.ed.ac.uk*

1. High Performance Computing

While many types of computer use are performance limited in one way or another the term High Performance Computing is usually used to refer to large numerical simulations (traditional supercomputing applications) where the performance is limited by the processing abilities of the computer rather than the network or disk. These simulations typically perform intense calculations on large volumes of floating point data. Traditionally the performance of HPC systems is usually measured in flops (floating point operations per second) However trends in computer architectures over recent decades now mean that the performance of this type of code is often more limited by the capabilities of the memory system than the ability of the processors to issue floating point instructions. Historically HPC applications have often been relatively simple software systems (a few 10s of thousands of lines), however the complexity of the codes has been increasing over recent years as more and more complex physical systems are simulated. This trend has been exacerbated by the introduction of compute clusters. Currently the only way of economically achieving the necessary performance for HPC applications is to distribute the applications across multiple processing nodes with independent memory systems. This adds significantly to the complexity of these codes.

2. Introduction to OO

The Object Oriented programming style is intended to facilitate efficient program development rather than particularly efficient programs. Any impact on program efficiency is entirely coincidental. While the increasing complexity of HPC applications make the use of OO techniques attractive to this community efficient programs are also a vital requirement. The purpose of this paper is to explore how to obtain the advantages of Object Orientation while still obtaining good program performance.

The problem that Object Orientation is attempting to solve, is that of code complexity. As programs get larger and more complex it becomes impossible for one programmer to hold all the details of the program in their head at the same time. This makes the program hard to modify without running the risk of introducing an error in some other part of the code. The only way to keep on top of this problem is to divide the program up into a number of loosely coupled modules that are small enough to work on. It is important to keep these modules as independent as possible, to reduce the chance of changes to one module causing bugs in a different module. It is also important to make

what dependencies do exist as explicit as possible in the program, so the programmer has a good idea which changes are safe and which are not.

The Object Oriented approach comes from the observation that one important source of dependencies between modules is when two modules share access to common data structures. Any change to the underlying data-structure will probably require changes to all modules that access the structure. If a program error in one of the modules causes the data structure to be corrupted or inconsistent then the programmer has to inspect all of the modules looking for the error.

Object Orientation deals with this problem by allowing programmers to define their own data types and operations on these types. These user-defined or abstract data types encapsulate complex data structures and complex operations on that data but present a relatively simple but powerful interface to the rest of the code. The rest of the program is only allowed to modify the data structure inside the type using the public interface. Because different types only interact via their interfaces this reduces the interdependencies between different parts of the program and makes what dependencies do exist explicit.

For example a **SortedIntegerSet** type may only support three operations:

1. **Add** – to add new integer values to the set.
2. **Remove** – to remove values from the set.
3. **GetContents** - to return the contents of the set, without duplicates and sorted in numerical order.

In this case the public interface for the **SortedIntegerSet** consists of these three operations plus the syntax used to create and destroy instances of the type.

It may take a take a large amount of complex code to implement this type but the rest of the program need only be concerned with this public interface. This has several advantages for the programmer:

1. When writing code using this type they only have to keep the properties of the interface in mind and need not be concerned with the implementation of the type. This allows the programmer to concentrate on higher level design decisions rather than on low level coding choices.
2. They can also safely make changes to the implementation of the type safe in the knowledge that provided the public interface is preserved that the change will not cause any problems in the rest of the program.
3. All manipulations of the type take place via the operations in the public interface, rather than duplicating explicit code fragments throughout the code. This makes any required change to the implementation easy to perform.
4. Good encapsulation not only hides complexity behind a simple interface it also should use an interface that is easy for the programmer to understand and remember. In the above example the name of the type helps to document its intended purpose and the names of the operations help to document their behaviour.

Once this new type has been defined the programmer is free to create as many instances of the type as he needs and to use them for different purposes. For example a program used for Unix system administration may use one **SortedIntegerSet** to keep track of which User ID numbers have been allocated and a second one to keep track of Group ID numbers.

These new types can also be used in the implementation of other types which in turn are used to define yet more types building more and more complex behaviour in a series of

layers. The bottom layer types in a program usually represent simple mathematical abstractions like **List**, **Set**, **Queue** etc. These types are often re-usable in different applications and versions might be available in standard libraries. Higher level types often reflect specific concepts from the problem domain of the application. For example a banking application may have types called **Branch**, **Customer**, **Account** etc.

3. Refactoring and Code optimisation

There are many different ways that a program may be written and still behave correctly. Though all of these different implementations are correct some are more desirable than others for reasons of readability, maintainability or performance. While normal code development is focused on extending or enhancing the behaviour of a program sometimes it is desirable to modify a program without changing its behaviour in order to improve the readability, maintainability or performance of the code. This is called refactoring.

Code optimisation is a special case of this where the program is modified to improve performance on a particular hardware and software platform. Code optimisation is different from other refactoring operations in a number of ways.

Code optimisation tunes a program for a particular hardware and software environment. Though there are optimisations that are useful on a wide variety of platforms, in general optimisation is not an absolute property, but is specific to a particular environment. This means that optimisations can be time limited and non portable giving rise to the need to support multiple code versions on different platforms which can in turn give rise to problems with code maintainability.

In fact optimisation is often in conflict with maintainability and readability because you often gain performance improvements by merging routines and data structures that would otherwise be independent. On the face of it this implies that the benefits of object orientation are incompatible with performance. In practice the majority of the execution time of most HPC applications is confined to a small number of key hot-spots in the program. With care, this allows the programmer to maintain a good code structure for the majority of their application while still maintaining good performance for the key hot-spots.

4. How Object Orientation impacts Performance

Memory issues

The layout of data in the memory space of a program can have a big impact on the programs performance. This is especially true for HPC applications which typically have to manipulate large volumes of floating point data, often significantly larger volumes than will fit the the processors caches.

Object Oriented program design has a big impact on the memory structure of a program. Almost all OO languages will implement an instance of a user defined type as a contiguous region of memory. This makes it very easy for the compiler to allocate space for user defined types. Of course even though the type is implemented as a contiguous region of memory it is always possible to have member variables that reference dynamically allocated data.

Either way when designing an OO code our choice of types tends to restrict our choices for the memory layout of the application data.

For example if we are writing a Molecular dynamics simulation where each particle has a number of component properties such as

- Mass, a scalar
- Velocity, a vector
- Position, a vector

The natural approach when using OO is to define a **Particle** type encapsulating these properties and to then create an array of Particles (possibly encapsulated in a **ParticleSet** type) to hold the simulation data. Whereas when using a more traditional procedural approach it would be natural to define multiple arrays, one for each property where one of the array indices is used to identify the particle.

Both approaches have some advantages and some disadvantages:

The Object approach has all of the data associated with a particle packed into a contiguous region of memory. This results in good data locality which can result in good cache use. On the other hand if a major computational loop only uses a subset of the component data (e.g. Only the particle position and mass) then the unwanted components are still likely to be loaded into the cache because they share cache lines with the required data.

This can waste a fraction of the available memory bandwidth and reduce the performance of code sections where cache misses occur. It will also reduce the size of problem that can remain resident in the cache during the computational loop.

It is possible to fine tune the memory layout of a type to some extent by permuting the order of the component data or by storing some of the data by reference rather than directly. This may allow the programmer to arrange things so that the data required/not-required by the most critical loops live on separate cache lines. This still may not solve the problem, its success depends on the nature of the loop and the target cache architecture. In addition many modern microprocessors contain prefetch engines that attempt to recognize regular memory access patterns pre-load data into the cache in a speculative fashion, these may only detect unstrided memory access patterns.

The high degree of memory locality that naturally occurs with OO may also result in the cache misses being very closely clustered together in time which on some architectures can result in much poorer performance than a traditional array based code.

Code structure issues

As well as impacting on the memory layout of the data a OO programming style also impacts the structure of the executable code. Most naïve compilation strategies for OO languages will convert each different operation on a type into a code block roughly equivalent to a subroutine or function in a procedural language. However the OO programming style encourages the programmer to keep the definition of each operation relatively simple and to build complexity out of multiple levels of types and operations on those types. The naïve compilation strategies would then result in large numbers of subroutine calls. These are not only expensive in themselves but they can inhibit many of the code optimisations that the compiler would otherwise perform. The OO programming style can therefore make much greater demands on the capabilities of the compiler than conventional procedural code. The compiler not only has to be very good at subroutine inlining but also capable of performing additional optimisation steps *after* the inlining has taken place.

Pointers

It is very common for Objects to store the addresses of other objects (either as explicit pointers or reference types). Objects given as method arguments are also often passed

by address to avoid having to make a copy of the object. Unfortunately the use of pointers and references can have a detrimental effect on compiler optimisation. In many cases the compiler cannot determine at compile time if two pointers may reference the same location or when calling a subroutine if the subroutine may have side effects on the region of memory referenced by a pointer. In these cases the compiler has to produce additional read and write instructions to ensure that values stored in CPU registers remain consistent with values in memory. This is the usual reason why C code is harder to optimise than equivalent Fortran code. It is also an important issue for many OO languages that also make heavy use of pointers and references.

High and low level types

These inherent performance issues in the OO programming style often make us take different approaches with the high and low level application types. For a high level type like the **ParticleSystem** type discussed previously there is little performance disadvantage to the use of OO. The code hotspots will be loops over particles calculating and applying inter particle forces. These occur at a lower level than the **ParticleSystem** type operations and process data that is entirely encapsulated within the **ParticleSystem** type therefore it should be possible to perform any optimisations without changing the public interface to the **ParticleSystem** type.

On the other hand very low level types like the cartesian vectors used for particle positions and velocities play an integral part in the most time critical parts of the code and have to function efficiently. In particular objects of this type may be created destroyed and copied within the code hotspots. For these performance critical low-level types only those language features that do not impact performance can be used.

However these low-level types are usually very simple and can even be replaced by explicit code using intrinsic types without too much damage to the code.

Intermediate level types like **Particle** are much trickier. **Particles** are higher level objects which can have quite complex behaviour (for example the **Particle** type may support operations to read and write particle state to and from files). This means that we really want to preserve the **Particle** type if at all possible. On the other hand the **Particle** type takes part in all of the code hotspots so it may be necessary to modify the type interface and do some damage to the encapsulation of the **Particle** type to improve the performance of the code.

5. Languages

Object Orientation is usually associated with particular programming languages. However it is really a programming style rather than a feature of any particular language. It is quite possible to use OO techniques in conventional procedural languages like C and Fortran. It is also possible to write monolithic spaghetti code in languages like C++ and Java. However for the purposes of this report I intend to concentrate on the performance implications of language features explicitly intended for the support of OO programming.

The choice of programming language is one of the most fundamental design decisions in any software project. It is of particular importance for HPC applications as the programming language dictates the choices of compiler available and much of the eventual code performance comes down to the abilities of the compiler. For many HPC systems the best compilers are those provided by the manufacturer so it is probably safest to stick to widely supported mainstream languages such as C++ and possibly Java.

6. C++

C++ is probably the most widespread of all the languages with explicit support for the OO programming style.

However it is important to realise that C++ is not just an Object Oriented language. C++ is an incredibly powerful general purpose tool. C++ aims to give the programmer the ability to tackle almost any kind of programming task from the interfacing directly with memory mapped hardware to developing high level complex software systems. C++ is essentially a super-set of the procedural C language so all the programming styles available to the C programmer are also available in C++.

The language also supports many other extensions in addition to those used for Object Orientation. For example C++ allows the programmer to define behaviour for the normal arithmetic operations for their user-defined types. Used sparingly and with care this can be a useful addition. Not only is the internal complexity of the type hidden behind a simple interface but, where appropriate, this interface may be in the form of the familiar arithmetic operations and so providing a good encapsulation. However this system can easily be abused to the point where a programmer is completely unable to determine what a piece of code is doing without tracking back through all the implementation details of the types concerned. Similar concerns can be expressed about other features of C++ such as templates.

The widespread support for C++ and its flexibility make it a good choice for the development of HPC codes. On the other hand the sheer complexity of the language means that C++ codes often hit portability problems when moved between different systems and compilers.

7. OO features of C++

So what are the features of C++ that are useful for OO programming?

Typedef and enum

These are the simplest form of type definition supported by C++ and actually date back to the C language, though C++ compilers typically perform greater type checking on enums than C compilers.

- Typedef – Allows the programmer to define a new type name in terms of other existing types.
- Enum – Allows the programmer to define a set of named integer constants.

Both of these constructs do not change the behaviour of the underlying type but they are nevertheless quite useful as they allow the programmer to name types by their intended use rather than by their implementation. For example consider the following declaration:

```
double v[3];
```

This version only indicates that v consists of an array of three doubles.

```
typedef double space_t;  
static const ndim=3;  
typedef space_t vector_t[ndim];  
enum dim { X=0, Y, Z };  
vector_t v;
```

This version is equivalent to the previous one however the variable v is now described in terms of its intended use in the program (a Cartesian vector) rather than its implementation. We also have a set of constants corresponding to the 3 dimensions that can be used to improve the readability of the code.

The typedef gives us greater flexibility to change the implementation for example we can change the floating-point precision of all vector_t variables by redefining space_t. However the possible changes are fairly limited to very closely related types as a typedef only changes the way a variable is defined not the way it is used. This makes the use of typedef much less powerful than the use of classes, however from a HPC perspective they do have the advantage that they have no detrimental performance impact (most compilers will process typedefs and enums in the initial parse stage). Typedefs are also useful because a typedef allows us to introduce explicit names for a program concept that might possibly become a different type at some future point in the development of the code, even when they currently share the same implementation as other types in the program. This makes it easier, if necessary, to refactor the code; for example by replacing a primitive type with a class, introducing a new sub-class or converting a class to a template. They can be particularly useful for performance refactoring where several different implementations of a class might be required corresponding to different hardware and software environments. There are several ways of dealing with this problem each with their own advantages and disadvantages:

1. Select different versions at compile time via the build system or using a pre-processor. In this case only a single version is compiled at any one time and it becomes hard to ensure that all versions of the class are kept consistent.

2. Use inheritance to substitute different implementation classes at run-time. This can become overly complex, wasteful of memory and can introduce additional performance overheads. However the use of inheritance does help to ensure consistency between the different versions.
3. Implement each variant as a different distinct implementation class and use typedefs (selected by pre-processor statements) to map the selected class to the class name used in the application. All the implementation classes are always compiled, which helps to detect some kinds of inconsistency, but the unused implementation classes are never referenced from the application and should not end up being included at the link stage.

Classes

The normal way to define new types in C++ is as a class. There are two parts to a class definition a declaration (normally in an include file) and an implementation. The declaration defines the members (the variables that make up the internal data structures of the new type) and function prototypes for its operations.

For example:

```
typedef double space_t;
class Cartesian{
public:
    enum dimension { X=0, Y, Z};
    static const int ndim=3;
private:
    space_t x[ndim];
public:
    // calculate the norm of the vector
    space_t norm();
    static void zero(Cartesian &val);
};
```

The implementation (normally a separate source file) provides the actual code for the defined operations.

```

#include "Cartesian.h"
space_t Cartesian::norm(){
    return sqrt((x[X]*x[X])+
                (x[Y]*x[Y])+
                (x[Z]*x[Z]));
}
void Cartesian::zero(Cartesian &a){
    a.x[X] = 0.0;
    a.x[Y] = 0.0;
    a.x[Z] = 0.0;
}

```

The declaration of a class also specifies how accessible the member variables and methods are. A private member/method is only accessible from within operations defined within the class. A public member/method is accessible anywhere. In order to implement the Object Oriented programming model the member data should usually be kept private and the public methods should provide a simple and intuitive interface to the type.

A class definition can contain two types of function. Functions declared with the static keyword are part of the type so can access the private members of the type but are otherwise the same as normal procedural functions. Functions without the static keyword are called methods. These are always associated with some instance of the defined type and cannot be called in isolation. For example:

```

space_t getLength(Cartesian &a){
    return a.norm();
}

```

This behaves exactly like a static function with an additional parameter however the method syntax makes it clearer that this is an operation defined on the new type.

The method implementation can be included directly in the class definition. Many C++ texts discourage this as it mixes the interface definition with the implementation. However most compilers take this as very strong encouragement to in-line the method so for performance reasons it can be a good idea to place the method bodies in the class definition for low-level, performance-critical types.

Constructors destructors and copying

A class defines a new type. A program may contain any number of instances of this type. We usually refer to class instances as Objects. Each Object is implemented as a contiguous region of memory. C++ provides the constructor mechanism for ensuring that each Object is initialised to a consistent state. Constructors are written as methods that have the same name as the class and return no arguments. Constructors may take arguments and multiple Constructors may be defined provided they take different types or number of arguments. Each time a new Object is created some Constructor must be

called to initialise the state of the Object. In particular a constructor taking a single constant reference to an Object of the same class specifies how the assignment operator initialises one object from another. This is usually referred to as the copy-constructor. A class may also specify a Destructor method that performs clean up operations when an Object is no longer required.

The constructor mechanism is a very important language feature for complex types, however it can add significant overheads for low-level performance critical types because they essentially add additional function calls every time an object is created, destroyed or copied.

In C++ if the programmer does not specify any constructors for the type the compiler will automatically provide a default constructors. In general low-level performance critical types should use these default constructors if at all possible, as many C++ compilers are much better at producing optimised code from the default constructors than in-lining and optimising user defined constructors.

Return value optimisations

If a C++ function returns an object typically two constructors are required. One to create a temporary result object inside the function and a second to copy the value of the result to the object being assigned to when the function returns.

```
/** function to add two vectors
 */
static Cartesian Cartesian::add(
    Cartesian &a, Cartesian &b){
    Cartesian tmp; //constructor called
    tmp.x[X]=a.x[X]+b.x[X];
    tmp.x[Y]=a.x[Y]+b.x[Y];
    tmp.x[Z]=a.x[Z]+b.x[Z];
    return tmp; // copy constructor called
}
```

Many compilers implement the return value optimisation, if the function is written to so that the constructor is called as part of the return statement then one of these constructors can be optimised away.

```

static Cartesian Cartesian::add(
    Cartesian &a, Cartesian &b){
return Cartesian(a.x[X]+b.x[X],
                a.x[Y]+b.x[Y],
                a.x[Z]+b.x[Z]);
}

```

However it is not possible to remove these constructors entirely. One strategy that can be used is to avoid functions that return objects, instead define operations as methods on the result object itself. This avoids the automatic construction of anonymous temporary objects, but may require the programmer to construct more explicitly named temporary objects at a higher level of the code. This gives rise to the usual trade-off between performance and elegant code. The programmer has complete control over where constructors are called because the temporary variables are now explicit in the code rather than being inserted by the compiler, however this is at the cost of a slightly uglier syntax.

```

/** method to set this vector to the sum
 * of two others
 */
void Cartesian::add(
    Cartesian &a, Cartesian &b){
    x[X] = a.x[X] + b.x[X];
    x[Y] = a.x[Y] + b.x[Y];
    x[Z] = a.x[Z] + b.x[Z];
}

```

```

Cartesian a,b,c;
a.add(b,c);

```

In other cases it might be more intuitive to pass a reference to the result object to the method for example:

```

Matrix m;
Vector v1,v2;
m.mult(v1,v2); // v1 = m*v2

```

Of course this requires the **Matrix** type to have access to the internal structure of the **Vector** type.

Operator overloading

C++ supports the overloading of the normal mathematical operators like (+/-/*). The set of defined operators is restricted to those that C++ defines for its intrinsic types. However the programmer can define behaviour for these operators operating on user defined classes by specifying methods with corresponding names.

This can be very useful for low level types like complex numbers and vectors where these operators have an intuitive meaning though it can be abused. Unfortunately the C++ standard defines binary operators as methods on the first argument returning a new object. If you write:

```
Cartesian a,b,c;  
a = b+c;
```

You are actually calling a copy constructor on **a** and a method on the object **b**. This means the return value optimisation should be used to suppress unnecessary constructors.

```
Cartesian operator+(const Cartesian &a){  
    return Cartesian( ... );  
}
```

C++ does use result object methods for the shorthand operators:

- +=
- -=
- *=

etc. So this syntax is often more efficient than using the more intuitive binary operations. There is no reason not to use operator overloading for the majority of a program however it can be useful to have an alternative interface available (such as result-object methods) for the performance critical regions. The downside is that this increases the complexity of the classes public interface.

Inheritance

C++ allows you to specify new classes as customised versions of existing classes. The majority of the behaviour of the sub-class is inherited from the super-class. Inheritance is a common (but not essential) feature of many OO languages as it removes the need to duplicate large fragments of code in closely related types, which is a potential source of programming errors.

The way that C++ implements inheritance is effectively to include an instance of the super-class a hidden member of the sub-class. Inheritance in C++ is therefore very closely related to “composition” where the programmer explicitly includes the old type as a member of the class. The main difference between the two techniques is that with inheritance the compiler automatically accepts any method call that is valid for a super-class as being valid for the sub-class and unless the programmer has defined a corresponding method in the sub-class it will automatically translate the method into a method call on the hidden super-class instance. With composition the programmer must explicitly write forwarding methods in the sub-class to re-implement the interface.

In both cases whenever an instance of the new type is constructed a super-class constructor must also be called. This can add to the object creation overhead when either inheritance or composition is used.

Virtual classes

Normally while it is possible to redefine methods in subclasses the behaviour of the inherited methods remain unchanged, even if internally they call some of the redefined methods. Methods that are defined in the super-class always call the version of the method defined in the super-class. In some situations this is not what is required so C++ allows you to define methods as being “virtual” which forces the sub-class method to be called. The compiler implements this by adding a pointer to a table of function pointers as a hidden member of the superclass. Whenever a virtual method is called this “vtable” is de-referenced to find the correct code to execute.

The compiler also adds code to the constructors to initialise the vtable pointer. Any class which has virtual methods (either locally or by inheritance) is called a virtual class. There are a number of important performance related issues associated with the use of virtual classes.

The vtable pointer increases the size of the object. This can be very significant for small classes which only have a few words of state.

Almost all compilers are unable to in-line virtual functions and the dereference of the vtable pointer increases the function call overhead above that of a normal function call. Constructing objects belonging to virtual classes is more expensive as the vtable pointer must be initialised for every object created.

For all of these reasons low-level performance-critical types should not be virtual.

Templates

Templates were introduced into C++ as a way of constructing types that were parameterised by other types. For example a template could define how to construct a type that implements a List of some yet to be specified type and then the template is instantiated with different type parameters to produce different types that are lists of integers or lists of floating point numbers or even lists of lists.

This template mechanism is implemented as a very powerful type-aware macro language early on in the compile process. Because templates are expanded before the normal code optimisation steps they can be abused for all sorts of performance reasons. For example even if many levels of templates are nested together to define a new type they are expanded to a single class definition as seen by the rest of the compiler and many of the method calls in the templates will already have been in-lined during the template expansion.

Even more extreme techniques exist for example template-metaprogramming where recursive template calls are used to evaluate expressions at compile time and can also be used to implement code transformations like loop unrolling.

Though these techniques are very powerful and can be used to add high performance template libraries for use in C++ applications they are not OO techniques and direct use of these techniques in application code probably increases the likelihood of programming errors rather than helping to reduce them.

8. C++ in practice -The MD code

In order to explore these issues further we can investigate the performance of a simple toy application. This is based on a small molecular dynamics code that is used as part of the coursework for the performance optimisation module in the Edinburgh University MSc in High performance computing. The advantage in using this code rather than a real application is that; it is relatively small, making it easy to work with, and the same toy problem in C and Fortran versions has been extensively by a large number of MSc students. This gives us a good understanding of what the maximum achievable performance of this problem independent of the actual implementation.

The performance of this code has been investigated using a SUN Sunblade-150 system at EPCC and the SunStudio-11 compilers in order to be roughly similar to the environment used by the Msc students. For the problem size and hardware used for this investigation most reasonable attempts at optimising this problem achieved results in the range of 15-20 seconds for 20 model timesteps.

The design of the C++ version of the code follows much of the discussion in previous sections.

The lowest level type in the C++ version this code is the **Cartesian** class which implements a cartesian vector in 3 dimensions.

```
#ifndef CARTESIAN
#define CARTESIAN
/** Cartesian
 * This is a simple efficient representation of cartesian vectors
 * It is a concrete class heavily using inline methods.
 *
 * As this is a very low level type we choose to manually unroll the
 * loops over the cartesian dimensions. There is little loss of readability
 * for this class and though the compiler should be able to manage the unroll
 * by itself it will have a better chance of unrolling loops in higher level
 * classes if these are unrolled by hand.
 */
#include <cmath>
#include <iostream>
typedef double space_t;
class Cartesian{
public:
    enum dimension { X=0, Y, Z};
    static const int ndim=3;
    space_t x[ndim];
    // zero the vector
    void zero(){
        x[X] = 0.0;
        x[Y] = 0.0;
        x[Z] = 0.0;
    }
}
```

```

// scale by a constant
Cartesian& scale(space_t a){
    x[X] *= a;
    x[Y] *= a;
    x[Z] *= a;
    return *this;
}
// calculate the norm of the vector
space_t norm(){
    return sqrt((x[X]*x[X]) + (x[Y]*x[Y]) + (x[Z]*x[Z]));
}
space_t inv_norm(){
    return 1.0/sqrt((x[X]*x[X]) + (x[Y]*x[Y]) + (x[Z]*x[Z]));
}
// result object methods
// we return a reference to the current object to allow chaining
Cartesian& add(const Cartesian &a, const Cartesian &b){
    x[X] = a.x[X] + b.x[X];
    x[Y] = a.x[Y] + b.x[Y];
    x[Z] = a.x[Z] + b.x[Z];
    return *this;
}
Cartesian& subtract(const Cartesian &a, const Cartesian &b){
    x[X] = a.x[X] - b.x[X];
    x[Y] = a.x[Y] - b.x[Y];
    x[Z] = a.x[Z] - b.x[Z];
    return *this;
}
Cartesian& scale(space_t a, const Cartesian &b){
    x[X] = a * b.x[X];
    x[Y] = a * b.x[Y];
    x[Z] = a * b.x[Z];
    return *this;
}
Cartesian& negate(Cartesian &a){
    x[X] = -a.x[X];
    x[Y] = -a.x[Y];
    x[Z] = -a.x[Z];
    return *this;
}
Cartesian& negate(){
    x[X] = -x[X];
    x[Y] = -x[Y];
    x[Z] = -x[Z];
    return *this;
}

// operator overloading

```

```

// member function overload operators are methods on the RHS argument.
// use following to convert my preferred result object syntax into overloaded
// operators hopefully the compiler can optimise away the
// empty constructor

Cartesian operator+(const Cartesian &a){
    return Cartesian().add(*this,a);
}
Cartesian operator-(const Cartesian &a){
    return Cartesian().subtract(*this,a);
}
Cartesian operator-(){
    return Cartesian().negate(*this);
}
Cartesian operator*(space_t a){
    return Cartesian().scale(a,*this);
}
// these operators are result object operators naturally
void operator+=(const Cartesian &a){
    x[X] += a.x[X];
    x[Y] += a.x[Y];
    x[Z] += a.x[Z];
}
void operator-=(const Cartesian &a){
    x[X] -= a.x[X];
    x[Y] -= a.x[Y];
    x[Z] -= a.x[Z];
}
void operator*=(const space_t a){
    x[X] *= a;
    x[Y] *= a;
    x[Z] *= a;
}
// component manipulation
void set(int i, space_t dat){ x[i] = dat; }
space_t get(int i) const { return x[i]; }
};

std::ostream& operator<<(std::ostream& s, const Cartesian &c);
std::istream& operator>>(std::istream& s, Cartesian &cart);
#endif

```

This class is very low level so all the computational methods are placed in the class definition. The short loops over the 3 cartesian dimensions are unrolled by hand. Most compilers should be able to unroll these but many have difficulty performing loop unrolling on multiple loop nests so we perform manual unrolling at the lowest level to encourage compiler unrolling at higher levels. Having a single block of code in the method may also help the compiler to perform method inlining. We also provide result object methods as an alternative to the standard operator overloading.

With the SUN compiler this class appears to perform equally well as explicit code written using intrinsic types.

The next highest type is the **Particle** class. This proved to be much more problematical.

```
#ifndef PARTICLE
#define PARTICLE
#include "Cartesian.h"
const static double radius = 1.0;
class Particle{
    Cartesian pos;
    double mass;
    Cartesian vel;
    double visc;
public:
    Particle(double m, Cartesian p, Cartesian v){
        mass=m;
        pos=p;
        vel=v;
    }
    Particle(){
        mass=0.0;
        visc=0.0;
        pos.zero();
        vel.zero();
    }

    double getMass(){ return mass; };
    double getVisc(){ return visc; };
    Cartesian getPos(){ return pos; };
    Cartesian getVel(){ return vel; };
    void setMass(double a){ mass=a; };
    void setVisc(double a){ visc=a; };
    void setPos(const Cartesian &a){ pos=a; };
    void setPos(const double a[3]){ pos.x[0]=a[0]; pos.x[1]=a[1]; pos.x[2]=a[2];};

    void setVel(const Cartesian &a){ vel=a; };
    void setVel(const double a[3]){ vel.x[0]=a[0]; vel.x[1]=a[1]; vel.x[2]=a[2];};

    void setup(int i, int size);
    Cartesian visc_force();

    friend class ParticleSet;
    friend std::ostream& operator<<(std::ostream& s, const Particle &c);
    friend std::istream& operator>>(std::istream& s, Particle &cart);
};
#endif
```

This code applies two different forces to the particles. A viscous force that is proportional to the particles velocity and a gravitational force that acts pairwise between particles and depends on the distance between them and the particle mass.

The key hotspot in the code is the gravitational force calculation as this requires a $O(N^2)$ loop over all pairs of particles. This loop takes place in the **ParticleSet** class that represents the collection of Particles in the problem. This class was deliberately introduced into the design to ensure that this performance hot-spot was well encapsulated in its own class to make optimisation easier.

As the gravitational force is symmetric between the particle pairs we also chose to implement the gravitational force method in the **ParticleSet** class and give this class **friend** access to **Particle**. This is motivated by two considerations. Firstly the force is between pairs of particles so we can argue that it naturally should be a method on **ParticleSet**. Secondly this allows us to keep the force calculation and the loop over particles in the same class which will make the code easier to optimise, as all the code corresponding to the major hot-spot is in a single place. However it is clear that we are doing some damage to the data encapsulation and introducing a strong dependency between the **Particle** and **ParticleSet** classes. The methods that are not performance critical such as the IO methods remain in **Particle** as we do not want to damage encapsulation more than is necessary.

The member data of the Particle class consists of the **pos**, **mass**, **vel** and **visc** variables. Of these only the first two are used by the gravitational force calculation. The order of the members within the class is chosen so that the members used by the gravitational force calculations are close to each other. On systems where 4 doubles occupy a whole number of cache-lines (this is true only for the level-1 cache on SUN sparc systems) this is sufficient to avoid unnecessary memory traffic.

These design choices have ensured that the most performance critical class in the application is the **ParticleSet** class. In order to explore different optimisation strategies we have several possible implementations of this class and use a **MyParticleSet typedef** to select different implementations at compile time. The simplest of these is the basic **ParticleSet** class which represents a straightforward OO implementation. In addition we have more extremely optimised versions that usually derive from the basic **ParticleSet** class.

```
#ifndef PARTICLESET
#define PARTICLESET
#include "Particle.h"
/** represents a set of Particles
 */
class ParticleSet{
protected:
    int size;
    Particle *p;
public:
    static const int max_array=4096;
    static const int array_pad=13;
    ParticleSet(int size);
    ~ParticleSet();
    void evolve(int nsteps, double dt);
};
```

```

int getSize() const { return size; };
void setup();

protected:
    Particle get(int i ) const {
        return p[i];
    };
    Particle& find(int i ) {
        return p[i];
    };
inline void force( Cartesian &f , Cartesian &f2, const Particle &a, const Particle
&b){
    const space_t G = 2.0;
    space_t tmp, s;
    space_t l;
    Cartesian diff = b.pos; diff -= a.pos;
    l = diff.norm();
    tmp = (G*a.mass*b.mass/(l*l*l));
    s = (l<=radius)?tmp:-tmp;
    diff *= s;
    f += diff;
    f2 -= diff;
}
friend std::ostream& operator<<(std::ostream& s, const ParticleSet &c);
friend std::istream& operator>>(std::istream& s, ParticleSet &cart);
};
#endif

```

The gravitational force calculation is defined as an inline method. However the main computational loops occur in the evolve method that is defined in the class implementation.

```

#include "ParticleSet.h"

ParticleSet::ParticleSet(int s){
    size = s;
    p = new Particle[size+array_pad];
}
ParticleSet::~ParticleSet(){
    delete[] p;
}

```

```

void ParticleSet::evolve(int nsteps, double dt){
    Cartesian f;
    const int unroll=8;
    Cartesian force_array[max_array+13];

    for(int s=0;s<nsteps;s++){
        std::cout << "Step " << s << '\n';
        for(int i=0;i<size;i++){
            force_array[i] = p[i].visc_force();
        }
        if( unroll > 1 ){
            for(int i=0;i<size;i+=unroll){
                /* starting loop to cover first triangular block */
                for(int ii=i; ii< i+unroll; ii++){
                    for(int j=ii+1;j<i+unroll;j++){
                        force(force_array[ii],force_array[j],p[ii],p[j]);
                    }
                }
                /* main blocked loop */
                for(int j=i+unroll;j<size;j++){
                    for(int ii=0; ii<unroll; ii++){
                        force(force_array[i+ii],force_array[j],p[i+ii],p[j]);
                    }
                }
            }
        }else{
            for(int i=0;i<size-1;i++){
                for(int j=i+1;j<size;j++){
                    force(force_array[i],force_array[j],p[i],p[j]);
                }
            }
        }
        for(int i=0;i<size;i++){
            p[i].pos += p[i].vel *dt;
            p[i].vel += (force_array[i] * (dt/p[i].mass));
        }
    }
}

```

The major optimisation in this version of the code is to introduce blocking into the loop over particle pairs in order to improve cache residency. This optimisation has also been attempted in C and Fortran versions of the code but is much less significant in these codes. Even with this blocking in place this version takes approximately 29 seconds for 20 iterations the best C and Fortran versions perform much better than this.

There are two possible causes for the poor performance. In the Ultrasparc processors the Level-2 external cache lines are large (at least 64bytes) this means that the second level cache will load the velocity and viscosity data to no useful purpose. Secondly

because all the data comes from the single Particle array the cache misses will always occur at the same point in the loop. With a conventional code there are multiple input arrays and the cache misses may occur in any of them and therefore at multiple points in the loop. This means that there is a good chance that multiple cache misses may be processed simultaneously allowing a degree of overlap. Either way the data layout that is a natural consequence of having a **Particle** class seems to be inhibiting performance.

One approach would be to do without the **Particle** class entirely and have the data held directly as arrays in the **ParticleSet** class. However in this case a much less radical refactoring is possible. As the force calculation is $O(N^2)$ in the number of particles we can consider copying the relevant data to and from local arrays at the start and end of the **evolve** method. This is a relatively local code change and allows complete freedom on the data layout to be used during the intensive part of the calculation. This approach is taken in the **ArrayParticleSet** class which overrides the evolve method in **ParticleSet**.

```
#include "ArrayParticleSet.h"

void ArrayParticleSet::evolve(int nsteps, double dt){
    Cartesian f;
    space_t visc[max_array + pad];
    Cartesian pos[max_array + pad];
    Cartesian force[max_array + pad];
    space_t mass[max_array + pad];
    Cartesian vel[max_array + pad];
    Cartesian diff;

    for(int i=0; i< size; i++){
        Particle &pp = find(i);
        mass[i] = pp.getMass();
        visc[i] = pp.getVisc();
        pos[i] = pp.getPos();
        vel[i] = pp.getVel();
    }

    for(int s=0;s<nsteps;s++){
        std::cout << "Array Step " << s << '\n';
        for(int i=0;i<size;i++){
            force[i] = vel[i]*(-visc[i]);
        }
    }
}
```

```

const space_t G = 2.0;
space_t tmp;
space_t l;
for(int i=0;i<size-1;i++){
    for(int j=i+1;j<size;j++){
        diff.subtract(pos[j],pos[i]);
        l = diff.norm();
        tmp = (G*mass[i]*mass[j]/(l*l*l));
        tmp = (radius>l)?tmp:-tmp;
        diff *= tmp;
        force[i] += diff;
        force[j] -= diff;
    }
}
for(int i=0;i<size;i++){
    pos[i] += vel[i]*dt;
    vel[i] += force[i] * (dt/mass[i]);
}
}
for(int i=0; i< size; i++){
    Particle &pp = find(i);
    pp.setMass(mass[i]);
    pp.setVisc(visc[i]);
    pp.setPos(pos[i]);
    pp.setVel(vel[i]);
}
}

```

This gives the best results of our C++ versions taking 16 seconds for 20 timesteps. To get this performance it was necessary to make the new data structures local to the evolve routine. A version with the new array structures implemented as private member fields of the **ArrayParticleSet** class was significantly slower than the original **ParticleSet** (39 seconds) . This suggests that that despite the restricted visibility of these arrays the Sun compiler was unable to determine the lack of pointer aliasing in this case.

Relative to the default **ParticleSet** this version has had to remove the encapsulation of the force term into a method. Though this is mostly due to the inability of the compiler to handle the arrays as member data. This class also shares the strong coupling with the **Particle** class. It is less dependant on the internal data structure because the **Particle** objects are manipulated via accessor methods rather than directly. However the functionality of the **visc_force** method has been duplicated.

C++ performance results

The following table summarises the performance results from various **ParticleSet** implementations.

| Class | Description | Time |
|---------------------------|--|------|
| ParticleSystem | Basic OO implementation using a Particle class. Force loop is blocked by a factor of 8 to improve cache use. | 29s |
| UnrollParticleSystem | As above but the blocked loop is extracted into its own method inlining the force calculation explicitly. | 29s |
| PackedParticleSystem | Particle class is extended to include the force variable | 29s |
| ExtremeParticleSystem | Evolve method optimised using method local arrays and only using intrinsic type. | 19s |
| ArrayParticleSystem | Evolve method optimised using method local arrays but using the Cartesian type where appropriate | 16s |
| ArrayMemberParticleSystem | As above but arrays are member variable of the ArrayMemberParticleSystem class | 39s |

Conclusions from the C++ MD code

C++ proved to provide good support for the implementation of low-level performance critical classes like the **Cartesian** class. We were able to get good performance from C++ while still using this class in the critical performance loops. Such low level classes should:

- Use the default constructors.
- Not use virtual methods.
- Define in-line methods in the class definition.
- Provide result object methods for time critical sections.

The compiler proved to be fairly good at method inlining but the OO design strategy produced a memory layout which resulted in poor performance. However it was possible to refactor the code to produce an optimised version keeping the changes fairly localised. This was made easier by ensuring the known performance hot-spots were well encapsulated in the original design.

Performance of C++ codes is very dependant on the capabilities of the compiler so we would expect more issues when porting C++ codes between different compilers than with C or Fortran codes.

9. Java

It is interesting to compare C++ and Java as languages for writing HPC codes. The Java language was heavily influenced by C++ and at first glance the syntax appears very similar. However Java is a reaction to the complexity of C++ . The design of Java is only attempting to address OO programming instead of the wide variety of uses possible with C++. Where C++ provides many alternative mechanisms for a given problem Java tries to provide a single solution for each.

The Java language

As indicated earlier the syntax of the Java language is very similar to a restricted subset of C++. However it has been simplified in a number of ways.

Java only supports classes not the procedural syntax that C++ inherited from C. It is possible to write static methods that are effectively procedural subroutines but they still have to nominally belong to a parent class.

Java does not split class definitions and implementations the way C++ does. All methods are included in the class body.

One of the biggest differences between C++ and Java is their data model. In C++ variables can be allocated from either the stack (By declaring the variable in the body of the routine) or from the heap by using the **new** operator. In Java apart from scalar variables like **ints** and **doubles** all variables are reference types. All classes and arrays must be allocated from the heap using the **new** operator. There is no equivalent of the C++ copy constructor as an assignment between object variables merely copies the address of the referenced object. If a copy of an object is required then this functionality has to be implemented as an explicit method on the class. Java uses automatic garbage collection to manage the heap so it is not necessary to explicitly delete storage that has been allocated via the new operator.

Java does not support operator overloading.

Inheritance is significantly different in Java and C++. C++ allows multiple inheritance so a class may have several superclasses. Java only allows a single superclass. This simplifies much of the implementation of inheritance. Subclasses add additional member variables to the end of the object so the superclass members are always at the same offset from the start of the object (**this** pointer). Though Java does not allow multiple inheritance it does allow a class to implement multiple interfaces. An interface is a collection of method signatures that the implementing class is required to implement. This means that objects that implement the same interface can be used interchangeably but do not share any implementation code.

Java and performance

At a first glance the biggest difference between the two languages would appear to be that Java is an interpreted language. Actually this is not that important. Most modern JVMs (Java virtual machines) include JIT (Just in time) compilers that dynamically translate java bytecode into native machine instructions while the program is running.

As HPC codes typically run for long periods of time and spend most of that time in a small fraction of the code, JIT compilation is very well suited to HPC applications. This can be verified with small simple benchmarks where Java can often equal the performance of C and Fortran for computational loops working on intrinsic data-types.

The biggest performance problems with the Java language are to do with other features of the language:

- All Java classes derive from the common virtual base-class called Object.
- Java does not support non-virtual method inheritance. Methods are either virtual or must be declared private or final.
- All class variables are reference types (addresses) so all objects are dynamically allocated using the new keyword.
- Java arrays are essentially a kind of object. Only 1-dimensional arrays are supported but arrays of arrays are possible. Most significantly array bounds checking is mandatory in the Java language.

One of the consequences of these features is that every Java Object has at least 2 words of additional data. This makes it very difficult to implement small low-level types efficiently in Java.

Virtual methods are difficult to in-line. The Java compiler **javac** also attempts to inline final or private methods when creating the bytecode if invoked with the -O flag and a good JIT should do the same with non optimised code. In principle JITs might be better at inlining virtual functions than conventional compilers as they can identify methods that are not declared final but have not been overridden in any of the currently loaded classes. JITs also have access to information about which variants are called most frequently and may be able to implement conditional in-lining.

To perform well the JIT has to be able to optimise away array bounds checks from the innermost loops. For simple loops like a loop over all elements of an array this is straightforward and the checks can be replaced by a single check on the loop bounds. However for more complex cases, like when explicit index calculations are used to emulate a multi-dimensional array then the compiler may fail to perform the optimisation and insert array bound checks into the inner loop.

As JITs add additional overhead to the running program they may not attempt some of the optimisations that static compilers routinely use at high optimisation levels.

However there is no reason why the basic approach taken with the C++ code of encapsulating and optimising the hot-spot should not work equally well with Java. Actually implementing the optimised version is more challenging as low-level classes need to be avoided and the optimised loop needs to work with Java arrays rather than the raw arrays available in C++.

In extreme cases Java has a well defined mechanism for cross calling to C (the JNI mechanism), so it is always possible to re-implement code hotspots in optimised C.

The MD code in JAVA

To explore the relative impact of these features of the Java language we attempted to re-implement the MD code in Java.

Our first step was to encapsulate the code hotspot in a ParticleSet class.

```

package uk.ac.ed.epcc.MD;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;

/** ParticleSet encapsulates a collection of particles.
 * This is an abstract class defining the interface used
 * by the rest of the program.
 * We subclass this to produce different optimized
 * implementations
 * @author spb
 *
 */
public abstract class ParticleSet implements Cartesian{
    protected final int size;
    public static final double G = 2.0;
    public static final double radius = 1.0;
    public ParticleSet(int n){
        size=n;
        System.out.println("size="+size);
    }
    /** evolve the system for a series of timesteps
     *
     * @param nstep number of steps
     * @param dt timestep
     */
    public abstract void evolve(int nstep, double dt);
    /** Get the specified Particle from the set,
     * Particle objects are always made within
     * the ParticleSet to allow complete freedom to
     * the ParticleSet subclass as to which
     * Particle subclass is used.
     * @param i
     * @return Particle
     */
    public abstract Particle getParticle(int i);
    public void setup() {
        for(int i=0;i<size;i++){
            Particle p = getParticle(i);
            p.setup(i,size);
        }
    }
}

```

```
public void dump(String string) throws IOException {

    OutputStream res = new FileOutputStream(string,false);
    PrintWriter pw = new PrintWriter(res,true);
    pw.println(size);
    for(int i=0;i<size;i++){
        pw.print(i+": ");
        Particle p = getParticle(i);
        p.dump(pw);
    }
    res.close();
}
}
```

We use an abstract class to specify the interface. This allows us to implement non performance critical code (like the dump method) that can be inherited by all different implementations. In a similar way we define an abstract **Particle** class to complete the specification of the external interface to **ParticleSet**. The **Particle** class contains generic code like the IO methods and abstract methods to get and set the particle properties. The Particle objects are always created inside the **ParticleSet** and returned via the **getParticle** method. Different implementations (subclasses) of **ParticleSet** are then free to use different subclasses of **Particle**.

Next we need a reference version. The following version represents a straightforward implementation using a **Particle** class and aiming at good encapsulation rather than performance. In accordance with these aims we also create a low level **CartesianClass** type.

```
package uk.ac.ed.epcc.MD;
public class FullOOParticleSet extends ParticleSet {
    protected final FullOOParticle dat[];
    public FullOOParticleSet(int n) {
        super(n);
        dat = new FullOOParticle[size];
        for(int i=0 ; i< size; i++){
            dat[i] = new FullOOParticle();
        }
    }
}
```

```

public void evolve(int nstep, double dt) {
    CartesianClass diff = new CartesianClass();
    for(int step=0;step<nstep;step++){
        System.out.println("Full OO step "+step);
        for( int i=0; i< size; i++){
            dat[i].viscForce();
        }
        for(int i=0;i<size-1;i++){
            for( int j=i+1;j<size;j++){
                dat[i].pair_force(diff,dat[j]);
            }
        }
        for( int i=0; i< size; i++){
            dat[i].update(dt);
        }
    }
}

public final Particle getParticle(int i) {
    return dat[i];
}

public static class FullOOParticle extends Particle{
    private double mass;
    private double visc;
    private final CartesianClass position;
    private final CartesianClass velocity;
    private final CartesianClass force;
    public FullOOParticle(){
        position = new CartesianClass();
        velocity = new CartesianClass();
        force = new CartesianClass();
    }

    public double getMass() {
        return mass;
    }

    public void getPos(double[] pos) {
        position.get(pos);
    }

    public void getVel(double[] vel) {
        velocity.get(vel);
    }

    public double getVisc() {
        return visc;
    }

    public void setMass(double mass) {
        this.mass=mass;
    }
}

```

```

        public void setPos(double[] pos) {
            position.set(pos);
        }
        public void setVel(double[] vel) {
            velocity.set(vel);
        }
        public void setVisc(double visc) {
            this.visc=visc;
        }
        public void viscForce(){
            force.scale(-visc,velocity);
        }
        public void pair_force(CartesianClass diff,FullOOParticle p){

            diff.sub(position,p.position);
            double l2 = diff.norm2();
            double l = Math.sqrt(l2);
            double tmp = G * mass * p.mass /(l2 * l);
            tmp = (radius>l)?tmp:-tmp;
            diff.scale(tmp);
            force.add(diff);
            p.force.sub(diff);
        }
        public void update(double dt){
            position.sadd(dt, velocity);
            velocity.sadd(dt/mass,force);
        }
    }
}

```

This version performed surprisingly well considering the poor support for low level types in Java. This version completed 20 timesteps in 50 seconds which is not very different from some of the C++ timings. We have made one compromise to improve performance here. The temporary **CartesianClass** object **diff** is allocated outside the main loops and passed into the pair_force method. If we allocate a temporary object inside this method then the runtime almost doubles.

If we introduce a sub-class that implements loop blocking similar to that used in the C++ ParticleSet class we get down to 38 seconds. However we are still quite a bit slower than the fastest versions so we can attempt versions that store the particle information in arrays.

In Java we can exploit the “inner class” syntax to implement the **Particle** class in such a way as to avoid the need to copy data between the **Particle** objects required by the external interface and the arrays internal to the **ParticleSet**. Java allows class declarations to be nested inside each other. If the nested class is declared static then this is just a convenient way of organising the source code to show that the inner class is closely related to the enclosing class. However if the inner class is not static it is a member class. Member classes are always associated with an instance of the enclosing class (often they are created in a method of their owning object). Member classes can directly access the member variables of their owning object so if we choose to store particle data in arrays within the **ParticleSet** object the set/get methods in the associated **Particle** class can access these arrays directly avoiding the need to copy data between arrays and the **Particle** objects.

```
package uk.ac.ed.epcc.MD;

public abstract class WrappedParticleSet extends ParticleSet {
    protected final double position[];
    protected final double velocity[];
    protected final double mass[];
    protected final double visc[];

    public WrappedParticleSet(int n){
        super(n);
        position = new double[size * Ndim];
        velocity = new double[size * Ndim];
        mass = new double[size];
        visc= new double[size];
    }

    public final Particle getParticle(int i) {
        if( i< 0 || i>= size){
            throw new ArrayIndexOutOfBoundsException();
        }
        return new RefParticle(i);
    }

    /** implementation of Particle that forwards the
     * set get methods onto the real
     * internal data structures.
     * @author spb
     *
     */
    private final class RefParticle extends Particle{
        int i, offset;
        public RefParticle(int n){
            i=n;
            offset=i*Ndim;
        }

        public double getMass() {
            return mass[i];
        }
    }
}
```

```

        public void getPos(double[] pos) {
            pos[X]= position[offset + X];
            pos[Y]= position[offset + Y];
            pos[Z]= position[offset + Z];
        }

        public void getVel(double[] vel) {
            vel[X]= velocity[offset + X];
            vel[Y]= velocity[offset + Y];
            vel[Z]= velocity[offset + Z];
        }
        public double getVisc() {
            return visc[i];
        }
        public void setMass(double m) {
            mass[i]=m;
        }
        public void setPos(double[] pos) {
            position[offset + X]=pos[X];
            position[offset + Y]=pos[Y];
            position[offset + Z]=pos[Z];
        }
        public void setVel(double[] vel) {
            velocity[offset + X]=vel[X];
            velocity[offset + Y]=vel[Y];
            velocity[offset + Z]=vel[Z];
        }
        public void setVisc(double v) {
            visc[i]=v;
        }
    }
}

```

In the example shown here we have chosen to store all 3 cartesian components of the position and velocity vectors in a single one dimensional array. This is a response to the lack of true multi-dimensional arrays in Java. This reduces the readability of the code but in this particular case the damage is not excessive. This is an abstract class which can be further sub-classed to explore different implementations of the evolve method using this data-layout for example:

```

package uk.ac.ed.epcc.MD;

public final class SimpleParticleSet extends WrappedParticleSet {

    public SimpleParticleSet(int n){
        super(n);
    }
}

```

```

public void evolve(int nstep, double dt) {
    final double force[] = new double[size*Ndim];

    for(int step=0;step<nstep;step++){
        System.out.println("Simple step "+step);
        for( int i=0; i< size; i++){
            final double s = (-visc[i]);
            force[(i*Ndim)+X] = s*velocity[(i*Ndim)+X];
            force[(i*Ndim)+Y] = s*velocity[(i*Ndim)+Y];
            force[(i*Ndim)+Z] = s*velocity[(i*Ndim)+Z];
        }
        for(int i=0;i<size-1;i++){
            double fx = force[(i*Ndim)+X];
            double fy = force[(i*Ndim)+Y];
            double fz = force[(i*Ndim)+Z];
            final double g = G*mass[i];
            final double xi = position[(i*Ndim) + X];
            final double yi = position[(i*Ndim) + Y];
            final double zi = position[(i*Ndim) + Z];
            for( int j=i+1;j<size;j++){
                double x = xi - position[(j*Ndim) + X];
                double y = yi - position[(j*Ndim) + Y];
                double z = zi - position[(j*Ndim) + Z];

                double l2 = x*x + y*y + z*z;
                double len = Math.sqrt(l2);
                double tmp = g*mass[j]/(len*l2);
                tmp = (radius>len)?tmp:-tmp;
                x *= tmp;
                y *= tmp;
                z *= tmp;
                fx += x;
                fy += y;
                fz += z;
                force[(j*Ndim) + X] -= x;
                force[(j*Ndim) + Y] -= y;
                force[(j*Ndim) + Z] -= z;
            }
            force[(i*Ndim)+X]=fx;
            force[(i*Ndim)+Y]=fy;
            force[(i*Ndim)+Z]=fz;
        }
    }
    for( int i=0; i< size; i++){

```

```

        double s = (dt/mass[i]);
        position[(i*Ndim)+X] += dt * velocity[(i*Ndim)+X];
        position[(i*Ndim)+Y] += dt * velocity[(i*Ndim)+Y];
        position[(i*Ndim)+Z] += dt * velocity[(i*Ndim)+Z];
        velocity[(i*Ndim)+X] += s * force[(i*Ndim)+X];
        velocity[(i*Ndim)+Y] += s * force[(i*Ndim)+Y];
        velocity[(i*Ndim)+Z] += s * force[(i*Ndim)+Z];
    }
}
}
}

```

However none of our pure java version was able to run in under 30 seconds so we went on to investigate the use of a JNI implementation. This proved to be remarkably easy as it only required minor changes to convert one of our Java implementations into a valid C implementation.

Java Performance results

The following table gives performance results for the Java version of the MD code. As the JIT compiles the code during execution which can give anomalously high timings early in the program run each timing is the median result of 5 timings performed sequentially in the same program run.

| Class | Description | Compiler | flags | Time/s |
|------------------------|---|----------------------|---------|--------|
| FullOOParticleSet | Reference OO version including CartesianClass | Sun Hotspot 1.5.0_08 | -server | 50 |
| FullBlockedParticleSet | As above with loop blocking | Sun Hotspot 1.5.0_08 | -server | 38 |
| OOParticleSet | OO version without CartesianClass | Sun Hotspot 1.5.0_08 | -server | 46 |
| BlockedParticleSet | As above with loop blocking | Sun Hotspot 1.5.0_08 | -server | 37 |
| SimpleParticleSet | Data stored in arrays | Sun Hotspot 1.5.0_08 | -server | 36 |
| MultiParticleSet | Data stored in arrays, cartesian components in different arrays | Sun Hotspot 1.5.0_08 | -server | 32 |
| JNIParticleSet | As SimpleParticleSet but evolve method in C | Sun C 5.8 2005/10/13 | | 77 |
| JNIParticleSet | As SimpleParticleSet but evolve method in C | Sun C 5.8 2005/10/13 | -fast | 16 |
| JNIParticleSet | As SimpleParticleSet but evolve method in C | Sun C 5.3 2001/05/15 | -fast | 33 |
| JNIParticleSet | As SimpleParticleSet but evolve method in C | Gcc 3.1 | -O5 | 42 |

These results show that the Hotspot JIT is certainly competitive with compiled code. The current C compiler can produce more optimised code however it is interesting to note that an older version of the same compiler performs the same as the JIT. In this particular case the difference seems to be due to the following line of code.

```
tmp = (radius>len)?tmp:-tmp;
```

The 5.8 compiler is able to implement this using a conditional move instruction rather than a branch which seems to allow more efficient optimisation of the inner loop. If this statement is removed then the older compiler is also able to perform the additional optimisation.

Conclusions from the Java MD code

The performance of Java is remarkably good compared to compiled code. Java JIT compilers are less likely to attempt the most extreme code optimisations that good static compilers can perform because fast compile time is usually a more important feature for most Java users. On the other hand code in-lining seems to be easier for the JIT compilers as they have a complete overview of the entire code where conventional compilers typically only see a single source file at a time.

Java shares many of the same OO overheads as C++. Most significantly problems where additional object fields are loaded into cache when not required. If anything this problem is worse with Java as Java objects have additional mandatory fields and are forced to store more of the data by reference.

Java has poorer support for low-level types but the biggest performance problem seem to be the cost of object creation so Java classes can effectively be used as low level types provided that temporary objects are not allocated within the performance critical loops. The use of Java objects as low level types may also significantly increase the memory requirements of the code.

Storing the particle data in arrays within the **ParticleSet** object was relatively straightforward. However this did not result in particularly faster code than the by adding blocking to the Full OO version. However it would be more efficient in its use of memory and it was relatively easy to optimise these versions further using JNI.

10. Portability of performance.

Object oriented codes rely heavily on the abilities of the compiler to achieve good performance. It is therefore instructive to look at the performance of the Object Oriented versions of MD code when ported to a completely different hardware and software platform, for example the IBM compilers and Power-5 processors used in HPCx. All the following benchmarks were run on a IBM eServer 575 system running AIX 5.3.

C++

The following results were generated using the IBM C++ compiler x1C V 7.0 and the following compiler flags **-O5 -qhot -qarch=auto -qalias=allptr**

| Class | Description | Time |
|---------------------------|--|------|
| ParticleSystem | Basic OO implementation using a Particle class. Force loop is blocked by a factor of 8 to improve cache use. | 19s |
| UnrollParticleSystem | As above but the blocked loop is extracted into its own method inlining the force calculation explicitly. | 18s |
| PackedParticleSystem | Particle class is extended to include the force variable | 11s |
| ExtremeParticleSystem | Evolve method optimised using method local arrays and only using intrinsic type. | 4s |
| ArrayParticleSystem | Evolve method optimised using method local arrays but using the Cartesian type where appropriate | 5s |
| ArrayMemberParticleSystem | As above but arrays are member variable of the ArrayMemberParticleSystem class | 11s |

These results are broadly in-line with those from the SPARC system. There is a slight performance advantage from using intrinsic types rather than the Cartesian class but the performance impact from having the data arrays as class members is less dramatic.

Java

These results come from the IBM Java5 JVM and associated JIT

| Class | Description | Compiler | flags | Time/s |
|-----------------------|---|-----------|---------|--------|
| FullOOParticleSet | Reference OO version including CartesianClass | IBM Java5 | -server | 26 |
| OOParticleSet | OO version without CartesianClass | IBM Java5 | -server | 21 |
| BlockedParticleSystem | As above with loop blocking | IBM Java5 | -server | 23 |
| SimpleParticleSystem | Data stored in arrays | IBM Java5 | -server | 11 |
| MultiParticleSystem | Data stored in arrays, cartesian components in different arrays | IBM Java5 | -server | 11 |

Again a similar picture to that on the SPARC system with performance comparable to many of the C++ implementations but with Java not able to match the very fastest version. The key difference between the two Java implementations was that the initial method call was always significantly slower than the later calls. In the IBM JVM this initial call to any method is always interpreted. If this call takes significant time to execute then JIT compilation is triggered and subsequent calls execute compiled code. On the other hand the SUN JIT implements on-stack-replacement which allows the JVM to switch between interpreted and compiled code during the initial method call.

Both the C++ and Java versions show a similar pattern of performance features however the best implementation is different for the two platforms. This indicates that for the best performance of a code that runs on multiple platforms multiple versions of the optimised classes will often have to be maintained in parallel.

11. Conclusions

Object orientation while important for controlling code complexity can give rise to performance problems as it constrains the memory layout of application data. However the encapsulation provided by the OO programming style makes it easier to perform all forms of refactoring including performance optimisation.

In particular where it is possible to consider local restructuring of the data layout to improve the performance of key sections then the encapsulation of the internal data structures of a type can make this process significantly easier.

In some cases the memory layout problems that occur when using the OO approach can be offset by adding loop blocking to the critical loops to improve cache re-use.

One of the most frequently used quotations in papers about performance optimisation is "*Premature optimisation is the root of all evil*". This is normally attributed to Tony Hoare and Donald Knuth. While it is certainly true that early optimisation can lead to all manner of problems if code performance is one of the key requirements of a software project then the *ability* to optimise a code at a later date should be one of key design criteria used when designing the code.

Optimisations are often specific to a particular hardware and software platform so it is not usually possible (or even desirable) to optimise the code during the initial development. However if it is possible to identify the potential hot-spots in the code then the design should ensure that these are well encapsulated to make future optimisation refactoring as easy as possible. In particular there should be a high level type which contains the data used by the code hotspot and has the hotspot itself entirely contained within a single method call on that type. Particular care needs to be expended in defining the interface for this type as we expect to produce a variety of different implementations as part of the optimisation process. It might be worth developing a number of paper designs for different implementations during the early design stage in order to evaluate the flexibility of the proposed interface.

However the first version to be fully implemented should always be a reference version written for correctness rather than performance. It is a lot easier to develop fast version of a code when there is a correct working reference version to compare against.

When developing optimised versions it is often necessary to handle a large number of different implementations. The inheritance features in many OO languages can be very useful to reduce the amount of new code that is required to produce a new version.