

## Parallel OO techniques for HPC applications

*Stephen Booth*  
*s.booth@epcc.ed.ac.uk*

### **1. Object orientation and High Performance Computing**

Historically HPC applications have often been relatively simple software systems (a few 10s of thousands of lines), however over recent years more complex software is becoming more common as more and more complex physical systems are being simulated. This increase in code complexity 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.

Object Orientation is a long standing technique that manages code complexity. The aim (as is always the case) is to break up a complex problem into a series of simpler problems with minimal dependencies between them. Object Orientation is based on the observation that many particularly complex dependencies occur due to shared data structures. Object Oriented techniques attempt to reduce program complexity by encapsulating data structures, and the code that operates on this data, within an opaque object with a simple well defined interface. Only the code within the object itself is allowed to access the data structures directly reducing the scope of dependencies due to these data structures.

As the internal implementation of the object and the rest of the program only interact via the interface any change to the code on one side of the interface should not introduce problems on the other side provided that the interface is maintained. This can be used to separate a large software system into a number of smaller simpler components with little dependency between them, reducing the overall complexity of the system.

The purpose of this paper is to explore if these OO programming techniques can be used to control the complexity of modern HPC codes.

### **2. Parallel programming**

Parallel programming has become the dominant paradigm for HPC. Though there are many possible ways of improving the performance of application codes few of these have the potential to reduce a programs runtime by factors of over 100 in the same way that parallelism does. There are several possible models of parallelism that can be used for HPC computing. These vary in their ease of use and their efficiency for various types of problem.

### **Parallel computing as optimisation**

For HPC applications parallelism is a means to an end rather than an end in itself. Parallel computing is necessary to achieve the necessary performance but it is essentially secondary to the primary purpose of the application. Parallel computing is therefore essentially an optimisation technique, any technique we employ to make the development of parallel programs easier must not significantly impact the scalability or performance of the resulting code.

Optimisation can be thought of as the process of refactoring a code to perform better in a particular Hardware and Software environment. Because these environments change any form of code optimisation is usually time-limited. A change that is appropriate at one point in time can easily become inappropriate later on and may need to be re-worked. Optimisations can easily damage other

important aspects of the code such as maintainability and portability. When undertaking a code optimisation exercise developers need to carefully balance the advantages and disadvantages of the proposed changes. Because of the time limited nature of many optimisations localised changes are usually more attractive than changes that require very wide ranging changes to the code.

This suggests that our aim in parallel program design should be to try and decouple the parallel nature of the application from the implementation of the algorithm. Complete decoupling is never going to be possible but we should aim to decouple these two concerns as much as possible. In addition where possible we should be aiming to de-couple the parallel implementation strategies in different parts of the application.

### **3. Types of parallel programming**

Because memory system performance is so significant for the performance of many HPC applications parallel computer architectures are often characterised by the nature of their memory systems. These different computer architectures have given rise to different models of parallel programming.

#### **Shared memory (thread level) parallelism**

Shared memory parallelism relies on having multiple processors connected to a shared memory system. The parallelism is introduced at the thread level where each thread can run on a different processor but all share access to a common data structures in a shared address space.

This is a relatively easy kind of parallelism to utilise. The parallelism can be introduced incrementally into the program. In the parallel regions of the program multiple threads are used and in the sequential regions all but one of the threads are parked, and the remaining thread runs the code. As the data structures are held in shared memory they can be accessed equally well from all of the threads. This means that the data structures can be defined independently of the decomposition of work into tasks.

Despite these advantages thread based parallelism, on its own, does not currently provide a route to cost effective parallelism across very large numbers of processors. This is probably due to economic factors driven by the rest of the computing industry though small SMP systems are very common, to a first approximation very large SMP systems seem to only be of interest to the HPC community and are therefore only available at a premium cost.

#### **Distributed memory (process level) parallelism**

Most HPC applications are currently distributed using process level parallelism. Process level parallelism distributes a single application across multiple processes each with their own separate address space. This kind of parallelism is notoriously difficult for the programmer, however it is capable of utilising very cost effective compute clusters.

Each process is usually specified as a sequential program and written in a conventional language such as C or Fortran. The processes cooperate by sending each other messages through a message passing library such as MPI. Process based parallelism can be further sub-divided depending on the capabilities of the message passing system. The most common form is that of cooperative communication as used by MPI. In cooperative communication both the sender and the receiver of a data message need to actively take part in a communication calling **send** and **receive** subroutines respectively. An alternative approach is that of one-sided communication where a single process can initiate a transfer of data between its own address space and that of a different process. This can be generalised to *active-message* communication systems where not only can a single process initiate a data transfer but the arrival of the data automatically causes a handler routine to be invoked at the destination to process the message.

Cooperative and single sided communication libraries have different advantages and disadvantages depending on the requirements of the application. Cooperative communication automatically provides

a degree of synchronisation between the sending and receiving processor which is ideal for operations like boundary swaps. When implementing the same operation using single sided communication it is necessary to add explicit synchronisation operations. On the other hand single sided communication makes random access to large distributed datasets much more straightforward.

The main disadvantage of the message passing programming model is that it puts a significant burden on the programmer. The parallel decomposition of the problem must be explicit in the code, as must all of the communications between tasks. As the problem is distributed across separate processes with independent address spaces it is not easy to incrementally add parallelism to the program in a similar fashion to thread based parallelism. The entire program has to run in parallel. As the message passing model distributes data as well as work then it is much harder to change the decomposition while a program is running. This usually requires a matching redistribution of the data, which can add significantly to the complexity of the program.

## **PGAS languages**

Recently an Partitioned Global Address Space languages such as UPC, Chapel and X10 have been proposed which combine features from both the shared and distributed memory programming models. Like distributed memory programs PGAS programs consist of cooperating processes with their own local address spaces. However in addition to local data PGAS languages also provide access to shared data structures that can be accessed from any process. PGAS languages are usually designed to permit implementations on distributed memory hardware where the shared data structures are implemented using distributed data structures and single-sided communications. In common with single-sided communication it is often necessary to include explicit synchronisation in a PGAS program.

## **4. Object Orientation**

One of the most successful techniques for dealing with software complexity is the use of Object oriented programming. It therefore seems reasonable to ask the question if it is possible to use OO techniques to simplify the task of designing and building parallel programs. In order to take advantage of developments from mainstream computing we restrict ourselves to discussing design solutions that can be implemented using existing object oriented languages rather than considering new OO languages with additional language features intended to support parallelism.

## **Object Orientation and Optimisation**

Object Orientation is intended to help control code complexity, any impact on code performance is coincidental to this primary gain. In practice the impact on performance is often detrimental because restricting direct access to the raw data structures can introduce additional overheads.

However the object oriented approach can help with the process of code optimisation and can help to control the negative side-effects associated with optimised code.

Many applications spend the majority of their run-time in a small number of “hot-spots”. If the code is designed to encapsulate these hot-spots (including their data structures) then they become easier to optimise. The hot-spot can even be totally re-written and provided the public interface is unchanged there will be no impact on the rest of the program. This includes changing the data structures used by the hot-spot. Obviously this may introduce some additional overhead whenever data is copied in and out of the optimised objects so the public interface needs to be carefully designed to reduce the number of times this is required.

Once the program hot-spots have been encapsulated in this fashion it makes the code much easier to optimise as the size of code that needs to be worked on is much smaller. Also this makes it easier to maintain multiple versions of the code in order to support multiple target platforms and regression testing because we now have a well defined interface that the different versions need to implement in order to be equivalent to each other.

## **Thread based parallelism**

Thread based parallelism is relatively easy to add to an Object Oriented system. Multiple threads can be utilised within an objects implementation to improve performance while still preserving the interface seen by external code. Similarly multiple threads of execution can occur in the external code that calls methods on an object provided that locks are added to the object implementation to ensure that each external thread sees a consistent view of the state of each object. Many modern OO languages such as Java and C# already provide explicit support in the language for multi-threaded programming.

## **Distributed memory parallelism**

Distributed memory parallel programs consist of cooperating processes running in multiple memory systems. This means that the application data also needs to be distributed across multiple memory systems. This is another potential source of dependencies between different parts of a program. Anyone who has attempted to use distributed memory numerical libraries will be aware of this problem. These libraries often require data to be distributed in a particular manner which makes it difficult to introduce these libraries into an existing application which has been written using a different data decomposition.

The OO philosophy tells us that we should try to encapsulate data layout in order to reduce dependencies between different parts of the program. By extension the same should be true of data decomposition.

In conventional OO programs each object lives in a single memory system. We need to extend this concept to cope with distributed data. The obvious extension is an Object Ensemble. Instead of representing an entity by a single Object the entity is represented by an ensemble of objects (one in each of the memory systems). Each part of the distributed data is contained in one of the objects from the ensemble. The ensemble, as a whole, needs to present a public interface that hides the data decomposition. The question is how do we define such an interface?

### ***Global interfaces***

We can define a global interface as one where any legal method call can be made on any object in the ensemble without requiring any cooperation from other members of the ensemble. This provides a high degree of encapsulation as the interface makes the ensemble appear as if it is a single Object present in each of the memory spaces. We can consider a PGAS data structure as an example of a global interface. Even though a data decomposition might be explicitly defined when the data structure is declared any process is capable of reading and writing any element of the data structure.

Global interfaces are trivial to achieve when performing read-only access to replicated data. It is much more difficult to implement a global interface that manipulates distributed data. At the very least some form of single sided communication will be required to access the remote data.

Modern OO languages like Java and C# provide some support for distributed memory programming via Object serialisation and remote method invocation (RMI).

Object serialisation provides mechanisms for the state of an Object to be written as a sequence of bytes and for Objects to be re-created from their serialised form. This provides a convenient mechanism for a message passing system to copy Objects from one processes memory space to that of another. Even when a OO language does not provide built in support for serialisation, Objects can implement methods that pack their contents into buffers for communication.

Remote method invocation is a form of active-message communication. Objects in one processes memory space are represented by stub proxy-objects in other memory spaces. When a method call is made on a stub object the method parameters are serialised and sent to the real object where the corresponding method is invoked and the method result serialised, sent to the stub object and

returned. RMI is therefore a variant of active-message communication. Java and C# both provide support for RMI in their standard class libraries and similar mechanisms can be built for other OO languages such as C++. Though originally developed for client/server programming RMI could also be used as a basis for global interface methods.

The use of global interfaces can give rise to problems with load imbalance. If we implement the interface to execute the computation where the required data is located then the calling process will be idle while waiting for the result. The pattern of computation therefore depends on both the pattern of method invocations and the data distribution. The data distribution pattern is determined by the target ensemble whereas the invocations are determined by the calling software layer. Neither part of the program has sufficient control to ensure load balance. An alternative approach would be to always perform as much of the calculation as possible in the object where the method was first invoked and to only use RMI (or some other form of single sided communication) to read and write the distributed data. In other words implementing what is effectively a PGAS data structure within the object ensemble.

### **Collective interfaces**

A different solution is to define the interface methods as collective operation. In a collective operation the same method (with consistent arguments) must be called on all objects in an ensemble at the same time. This collective style works well with cooperative communication systems as all objects in the ensemble take part in the operation and are available to send and receive data as required.

Collective methods will work best for high level operations that contain a high degree of potential parallelism. All members of the ensemble have to take part in a collective operation and if there is insufficient potential parallelism in the operation some of these processors will be under-utilised.

It is possible to define collective interfaces for lower level operations but this will often result in excessive serialisation of the overall application.

Collective method calls can provide a high degree of encapsulation of the parallel implementation. All of the communications that take place are hidden within the implementation of the class and are not exposed in the interface. It should therefore be possible to radically change the implementation and/or the data decomposition without introducing any problems in other parts of the application. This is particularly easy to achieve for method calls where all of the method parameters are replicated. In this case the only parallel data is that owned by the parent class of the method and no additional dependencies outside of this class are introduced. For methods that take other object ensembles as arguments encapsulation may be harder to achieve. In this case, data from the parameter ensembles may need to be communicated as well as data from the parent ensemble. We can avoid this communication by requiring a particular data decomposition of the parameter ensemble but this introduces a *decomposition dependency* between the two classes. While this may be acceptable for classes that are very closely related, in general we want to be able to define collective operations where the correctness of the call depends only on the interfaces exposed by the method arguments not on their underlying implementation. One solution to this problem is to require the arguments ensembles to implement a global interface. However this is not the only solution.

### **Cooperative interfaces**

A cooperative interface consists of entirely local operations, such as operations to query the decomposition and access local data. These are intended to allow the calling layer to dynamically adapt to different decompositions.

As we shall demonstrate later it is possible to implement decomposition independent collective interfaces by only requiring the argument ensembles to implement a fairly simple cooperative interface. While this meets our primary goal of allowing decomposition independent interfaces to be defined it does weaken encapsulation as the same methods could also be used to write code that is

highly dependent on the decomposition.

## Encapsulating data decomposition

One of our design aims is to encapsulate the data decomposition used by an Object ensemble. It is perfectly acceptable for closely related classes to share the same data decomposition strategy or to be explicitly aware of each others decomposition strategy. Many applications have a natural decomposition that can be used throughout the code, in which case there is little harm in making the data decomposition quite explicit in the program. However there are other applications where the correct data decomposition is less obvious or where different parts of the application may require different decompositions. In this case it is useful to be able to change the decomposition used by one section of the code without requiring a large number of corresponding changes to the rest of the program.

In these more complex cases we found it convenient to introduce Objects that represent the data decomposition. Objects from a parallel ensemble can then provide methods (from a cooperative interface) that return the decomposition they are using. Though our aim is to allow the external code to be insensitive to the data decomposition it is often necessary to expose some limited decomposition information. This exposed information is reflected in the public interface of the Decomposition objects.

The external code may use the Decomposition objects returned by Object ensembles in various ways:

- Compatibility: Some sections of code may require that two object ensembles have compatible data decompositions. The Decomposition objects may be compared in order to verify this.
- Redistribution: Decomposition objects may be used to copy data between ensembles using different decompositions.

These two operations are sufficient to reduce the decomposition dependency between classes. If a method requires a particular data decomposition from a parameter ensemble it can retrieve the decomposition object and check that its requirements are met. If the method is passed an ensemble with an incorrect decomposition then a new ensemble with the correct decomposition can be created and populated from the original.

In order to define a data decomposition it is first necessary to define some global coordinate system that addresses the items of data to be distributed. The coordinate system is usually application dependant and may represent concepts such as:

- The index tuples for a multi dimensional array.
- Particle index
- Node/vertex for a structured mesh.
- etc.

At its simplest a Decomposition encodes a mapping between such a coordinate system and a set of processors.

Let us now consider how this may be implemented in practice. The following examples follow the Java language syntax but a similar approach could be used for any OO language.

Consider the following simple case of copying distributed arrays:

```
Array a = new ArrayType1(Nx,Ny);  
Array b = new ArrayType2(Nx,Nx);  
a.copy(b); // collective copy
```

Here **a** and **b** are both members of a parallel object ensemble that represents a distributed parallel array of size  $(N_x, N_y)$ . ArrayType1 and ArrayType2 are two different implementations of the Array interface that may use different data decompositions. The **copy** method is a collective operation that copies the data from b to a performing any necessary change of data decomposition. The important question is to what extent can we implement the copy operation without writing code that is explicitly aware of the data decompositions of the two Array Types.

Many existing application codes have had to deal with this same problem and in many cases this has resulted in a very similar solution to the one we intend to present here. However in the interests of clarity we intend to explore these issues using a very simple example code.

Most HPC applications have at least one underlying coordinate system. Common operations using the coordinate system include:

- Using the coordinate to reference a storage location.
- Iterate over the coordinate space.

Often the coordinate system is only an implicit concept in the code with primitive types (such as integers) used to represent coordinates. Nested loops and rectangular arrays often provide an acceptable approximation to the concepts required by the application domain and have the advantage of being very efficiently implemented by compilers. However with distributed data things become much more complex. Different coordinates can represent data points on different memory systems so we need to be able to map a coordinate to a processor rank as well as to a memory offset. It therefore becomes attractive to use a class to represent the global coordinate system.

Normally a real application code will only have to support a small number of different coordinate systems. As this example is intended to demonstrate an approach that could be applied to any coordinate system we will introduce a **Coordinate** interface to tag all coordinate classes.

```
public interface Coordinate<T> extends Comparable<T>{  
}
```

As we will see later it is convenient to define a global ordering of all coordinates so the only requirement that we make of a **Coordinate** type is to ensure that it implements **Comparable**.

Next we define a **DataContainer** interface. This represents any object that stored data of type **D** indexed by a specified coordinate system **C** and supports a collective copy operation that can copy data from any other **DataContainer** with the same type and coordinate system. This interface also contains a number of cooperative method calls needed to implement the copy operation. This include two local put/get operations to access locally stored data and a method to return an object that describes the data decomposition. Here we are using a generic interface parametrised by the data-type and the coordinate type. In a real application it would probably be acceptable to define a separate interface for each supported coordinate system.

```

/** Data container
 * @param <D> Type of data stored
 * @param <C> Type of coordinate
 */
public interface DataContainer<D, C extends Coordinate<C>> {
    /** Collective copy
     * @param source DataContainer
     * @throws Exception
     */
    public void copy( DataContainer<D,C> source) throws Exception;
    /** Get Decomposition
     * @return Decomposition
     */
    public Decomposition<C> getDecomposition();
    /** Local get
     * @param pos Local Coordinate
     * @return Value
     */
    public D get(C pos);
    /** Local put
     * @param pos Local Coordinate
     * @param val
     */
    public void put(C pos, D val);
}

```

Decomposition is also an interface as there will be many different implementations corresponding to the different data decompositions we wish to support. The key method is the owner method that maps any coordinate to the rank of the owning processor. This method has to execute locally so this interface is only capable of supporting static decompositions (where the owning processor is a function of the coordinate and replicated data).

```

public interface Decomposition<C extends Coordinate<C>> extends
Iterable<C>{
    /** Define decomposition
     * @param c Coordinate
     * @return rank of owning processor
     */
    public abstract int owner(C c);
    /** iterator over local Coordinates
     */
    public abstract Iterator<C> iterator();
    /**
     * @return Communicator used by Decomposition
     */
    public abstract Intracomm getComm();
    /**
     * @return Iterable over global coordinates
     */
    public abstract Iterable<C> all();
    /** is a Coordinate local
     * @param c Coordinate to test
     * @return boolean true if local coordinate
     */
    public abstract boolean isLocal(C c);
    /** Are two decompositions equivalent
     * @param d Decomposition
     * @return boolean
     */
    public abstract boolean compatible(Decomposition<C> d);
}

```

**Decomposition** extends the **Iterable** interface in such a way that it will iterate over those coordinates that are owned by the local processor. It will always iterate over these in global coordinate order.

So far we have only defined interfaces. The next code fragment shows one possible implementation for the **copy** function. Note that this implementation *only* relies on the methods described in the various interfaces and is therefore capable of copying data from any class that implements the interfaces correctly. This implementation uses a utility class called **ExchangeBuffer**. This is essentially a simple wrapper around the functionality provided by `MPI_Alltoallv` and some buffers.

```

public abstract class ExchangeDataContainer<D, C extends Coordinate<C>>
implements DataContainer<D,C>{

    protected abstract ExchangeBuffer<D> getExchangeBuffer() throws
MPIException;

    public final void copy( DataContainer<D,C> source) throws
MPIException{

        ExchangeBuffer<D> buff = getExchangeBuffer();
        Decomposition<C> sd = source.getDecomposition();
        Decomposition<C> td = getDecomposition();
        for( C c : sd){ // calculate send buffer sizes
            buff.incSize(ExchangeBuffer.Type.SEND,td.owner(c));
        }
        for( C c : td){ // calculate receive buffer sizes
            buff.incSize(ExchangeBuffer.Type.RECV,sd.owner(c));
        }
        buff.allocate();
        for( C c : sd ){ // pack send
            buff.pack(td.owner(c),source.get(c));
        }
        buff.exchange(); //AlltoAllv communication
        for(C c : td){ // unpack recv
            put(c, buff.unpack(sd.owner(c)));
        }
        buff.clear();
    }
}

```

The copy operation uses the decomposition information from both the source and the target ensembles. The first two loops iterate over the local coordinates of each decomposition and use the **owner** method from the *corresponding* decomposition to work out which processor that coordinate maps to in the corresponding decomposition. These first two loops are used to calculate the buffer sizes needed for the communication.

The next loop prepares the outgoing messages. Each local element of the source is packed into a buffer selected by the **owner** method of the target decomposition. Buffers are then exchanged. Finally the receive buffers are unpacked into the target. Note that as all of the loops take place in the global coordinate order then the data in the inter-processor messages will also be in this order. This avoids any requirement to tag the data with its coordinate or to sort the data on receipt.

This implementation can perform the data exchange very efficiently. The pack/unpack steps may still be relatively inefficient, in particular due to the large number of small objects created and the large number of calls to the **owner** method, but as this is local computation any lack of efficiency in the pack/unpack should not affect the scalability of the parallel code. Note that this implementation uses two additional loops to calculate the required buffer sizes, if the owner method is relatively expensive it would be better to use automatically resizing buffers within the ExchangeBuffer class.

Various optimisations can be added to this basic algorithm, for example testing the decomposition objects for compatibility and using a local copy if they are.

This algorithm can be used to implement data re-mapping between any *static* decomposition (That is

where the owning processor of a data point is some function of its global coordinates). New **DataContainer** implementations using different data decompositions could be introduced into the code without any need to refactor the existing classes. The underlying communication step is mapped onto a **MPI\_AlltoAllv** and should be reasonably efficient. However encapsulating the decomposition in this way is not without cost. For example the copy operation requires a large amount of memory space (double that used by the source and target Objects) . Any data re-decomposition that occurs within time critical regions of the code will probably require specialised code that is aware of both the relevant decompositions, nevertheless this general mechanism for data re-decomposition is a significant asset for code development.

This implementation only uses cooperative communication. If a DataContainer supports global put/get operations as well as local ones then a much simpler algorithm is available.

```
public abstract class GlobalDataContainer<D, C extends Coordinate<C>>
implements DataContainer<D,C> {
    public abstract void global_put(C coord, D val);
    public abstract D global_get(C coord);
    public abstract void barrier();
    public void copy(DataContainer<D, C> source) throws Exception {
        Decomposition<C> sd = source.getDecomposition();
        barrier();
        for(C c : sd){
            global_put(c, source.get(c)); //local get global put
        }
        barrier();
    }
}
```

Again this works on any source DataContainer because the global operations are only used on the target object. This will result in a very large number of small communications which may result in poor performance. However if the underlying communication system can support them efficiently then this avoids the pack/unpack operations at each end.

This example shows that it is possible to write a collective ensemble interface that is insensitive to the data decomposition of its arguments. This has significant potential to reduce the complexity of parallel applications as different regions of code no longer need to care about each others data decomposition.

In the same way that Object Oriented approach attempts to divide a complex system into simple components with no knowledge of the data structures used by other components this allows a parallel program to be divided into modules with no knowledge of the data decomposition used by other modules.

We used this basic framework to implement a simple parallel image processing application that implemented the following steps:

1. Read in an image file
2. Perform a 2 dimensional Fast Fourier Transform.
3. Filter the image in Fourier space.
4. Perform the reverse transform.
5. Write out the resultant image.

This naturally breaks down into three modules

1. File access
2. FFT
3. Filter

We chose to define the interfaces to these as decomposition independent collective interfaces. The public methods are always written so as to accept any decomposition in the DataContainers passed as arguments but the decomposition of DataContainers returned by a method is unspecified and the implementation is free to return any decomposition. This decomposition may change if the implementation of the method is changed so the calling layer is not allowed to make any assumptions about the decomposition of the returned ensemble.

For simplicity we define an Array2D interface that represents a DataContainer defined over a two dimensional coordinate system.

```
public interface Array2D<D> extends DataContainer<D, Coordinate2D> {  
    public int getSizeX();  
    public int getSizeY();  
}
```

This interface includes methods to query the global size of the DataContainer. This is a commonly needed operation but really only makes sense in terms of a particular coordinate system.

Now if we consider each of the three modules in our simple example we can see how this approach simplifies the design:

For the file access module we wish to read and write standard image files but the overall performance is fairly unimportant. Therefore it is simpler to perform all IO on a single process. This is easily achieved by using DataContainers with a decomposition where all data lives on process-0. If at some later date the IO performance becomes more important a parallel IO routine implementing the same interface could be substituted.

```
public interface PgmAccess {  
    public Array2D<Double> read(File f) throws Exception;  
    public void write(File f, Array2D<Double> dat)  
        throws MPIException, IOException;  
}
```

A two dimensional FFT can be performed by first applying one dimensional FFTs to each row and then applying one dimensional FFTs to each column of the result. The simplest implementation of a parallel 2D FFT therefore uses 2 data decompositions. One where data is decomposed by rows and one where it is decomposed by column. Such an implementation can easily be prototyped using the copy operation to perform the necessary data transposes.

```

public class FFT2D {

    public Array2D<Complex> fft(boolean forward,Array2D<Complex> src)
    throws MPIException{

        FFTArray x = new FFTArray(true,src.getDecomposition().getComm(),
            src.getSizeX(),src.getSizeY());

        FFTArray y = new FFTArray(false,src.getDecomposition().getComm(),
            src.getSizeX(),src.getSizeY());

        if( forward ){
            x.copy(src);
            x.fft(forward);
            y.copy(x);
            y.fft(forward);
            return y;
        }else{
            y.copy(src);
            y.fft(forward);
            x.copy(y);
            x.fft(forward);
            return x;
        }
    }
}

```

However as the performance of the FFT routine is important this can only be considered a prototype implementation. A more realistic solution would be to use optimised custom communication routines to implement the transpose or an existing parallel FFT routine from a numerical library such as FFTW. One copy operation is still required to copy the initial data into the correct starting decomposition.

Finally the filter operation has no particular requirements as to data decomposition. Each element of the image is considered independently and masked depending on its global coordinate. The filter module can therefore operate in-place and uses whatever data decomposition is specified for the target DataContainer. In our case this has the added advantage that the input ensemble will already be in the correct starting decomposition for the reverse FFT and the copy operation will only require local communications.

Its also possible to put a wrapper around a DataContainer to provide a different view of the data. The following code is an example of this which views a DataContainer of doubles as a DataContainer of Complex.

```

public class DoubleToComplexContainer<C extends Coordinate<C>> extends
    ExchangeDataContainer<Complex, C> {
    protected DataContainer<Double, C> container;

    public DoubleToComplexContainer(DataContainer<Double, C> c) {
        container = c;
    }
    @Override
    protected ExchangeBuffer<Complex> getExchangeBuffer() throws
MPIException {
        return new ComplexExchangeBuffer(
            container.getDecomposition().getComm());
    }
    public Complex get(C pos) {
        return new Complex(container.get(pos), 0.0);
    }
    public Decomposition<C> getDecomposition() {
        return container.getDecomposition();
    }
    public void put(C pos, Complex val) {
        container.put(pos, val.re);
    }
}

```

Even in this simple example we have gained significant advantages from implementing the generic copy operation as we have been able to re-use this operation to perform several of the required communication steps and the code used to implement the copy is not significantly more complex than an specific implementation of either of these steps. In addition by defining the module interfaces in a decomposition independent way we have made them more generally useful. For example we can easily use the IO functions to write a Fourier transformed image for debugging purposes even though the real and Fourier space images happen to have different data decompositions in the code.

## 5. Conclusion

Object oriented techniques are compatible with the shared memory, distributed memory and PGAS approaches to parallel programming.

In distributed memory parallel programs data decomposition requirements are a potential source of dependencies between different parts of the code resulting in increased program complexity.

It is possible to control these dependencies using Object Oriented techniques by encapsulating the distributed data behind a interface that is independent of the data decomposition used. Though there are many ways in which this can be done it is possible to achieve very good encapsulation of data decomposition using a simple collective interface that can be implemented using only simple MPI collective communication.