OO and Distributed Memory Parallelism

Stephen Booth
EPCC, The University of Edinburgh
spb@epcc.ed.ac.uk

Introduction

- Programmer/Developers perspective.
- HPC software is a mess
  - Poor re-use
  - Poor provision of libraries
  - Overly complex and difficult
- Good experience with OO programming in other domains
  - Don't think HPC is making full use of this
- Take a step back to look at problem in abstract.
- Experimental investigation
  - Class library not language.

What is Object Orientation

- OO is a long standing technique that manages code complexity.
- I consider it a design strategy not a language feature.
  - Language features help make things easier.
- OO Strategy based on observation that many complex dependencies occur due to shared data structures.
- Solution to encapsulate program concepts into user defined types (classes).
  - Data structures are private to module/class.
  - External access only via public interface functions.
  - Dependencies are therefore explicit.

OO Trade-offs

epcc

- Reduced dependencies make it easier to write re-usable libraries of code.
- Refactoring easier
  - Free to change module internals provided backwards compatibility of module interface is preserved.
  - In particular can change the data structures.
- Some increased overheads
  - Additional memory copying to transfer information between objects
  - Additional function calls

OO and performance

epcc

- OO is intended to help control code complexity.
  - Any impact on performance is coincidental.
  - Often detrimental, additional overhead to data access.
- OO approach ***can*** help with optimisation.
  - Optimisation is the process of modifying a code to run faster in a particular hardware/software environment.
  - Requires frequent code refactoring.
    - Hardware environments change frequently.
    - Software environments change more frequently
      - Compiler versions
      - OS version etc
    - Data structures often critical to code performance.
      - Therefore need to change data structures to improve performance.
      - Especially for novel hardware like GPGPU cell etc.

OO and optimisation

- Most codes spend most run-time in a few hot-spots.
- Encapsulate these
  – Hide the data structures
  – Makes optimisation refactoring easier. Particularly those that require changes to data structures.
  – Only small faction of code needs changing.
  – May have to use non OO style within this small fraction.
- Maintain multiple versions easily.
  – Different high performance versions for different platforms.
  – Simple versions with high confidence of correctness.
  – Regression testing.

Interface design

- Need to design interface from ground up with performance optimisation in mind.
  - May change design choices from “normal” OO practice.
- Minimise data copy overhead.
  - e.g. encapsulate entire solver rather than matrix multiply
  - e.g. use factory class to generate compatible matrix/vector objects with friend access.
- Avoid unnecessary object creation and copying.
  - e.g. implement operations as methods on result object

    A.addProduct(B,C)

    – updates contents of existing objects

    A = A.add(B.mul(C))

    – generates intermediate temporary result objects

Parallelism

- Use of parallel computers is now ubiquitous in HPC.
- For HPC, Parallelism is a form of code **optimisation**.
  - Multiple processors used to generate results faster.
  - Tuning for memory and communication system.
- Most current techniques for parallelism seem to increase code complexity significantly.

Current Environments

- Most significant HPC resources use distributed memory.
  - Driven by economic factors.
  - Not driven by ease of use.
- Rise of multi-core does not seem to be changing this.
  - Shared memory only within nodes.
- Greater diversity in communication systems.
  - Basic MPI lowest common denominator.
  - RMA or active messaging operations quite common.
- Diversity in software environments.
  - Usually can only rely on lowest common denominator languages/tools.

Data decomposition

epcc

- Interacting modules may have to use the same decomposition scheme.
- This introduces inter module dependencies just as shared data structures did.
  - Many codes use a single decomposition scheme.
    - May not be appropriate for all modules.
    - Less important modules have to live with inappropriate decomposition.
  - Particular problem when using parallel numerical libraries.
    - Library routines expect a particular decomposition.
    - OK for new code development.
    - Difficult to retro-fit to an existing application that chose an incompatible decomposition.
    - Switching between libraries also complicated.

OO and distributed memory

epcc

- OO encapsulates data layout within objects to reduce dependency.
- **In parallel we should aim to do the same for data decomposition.**
- Distributed memory parallel programs consist of cooperating processes running in multiple memory systems.
  - Objects live within a single memory.
    - Cannot contain distributed data.
  - Natural extension is an Object Ensemble
    - One object per memory system.
    - The ensemble (as a set) presents a public interface that ideally hides the data decomposition.
  - How do we define such an interface?

Option 1: Global interface

- Any valid call may be made on any member of the ensemble.
- Internally object accesses the data of other members.
- Pro
  - Good encapsulation
  - Intuitive, appears as if a single object is present in every memory system.
  - Approximates shared memory parallelism.
- Con
  - Needs some kind of single sided communication to implement.
  - Needs additional synchronisation.
  - Internal decomposition is hidden but may still affect load imbalance.

Load imbalance using global interfaces

- If we move work to data.
  - Calling layer determines nature of work.
  - Implementation layer determines Data distribution (therefore work location).
  - Neither has sufficient control to ensure load balance.
- Move data to work.
  - Calling layer can distribute work evenly.
  - Large amount of data movement.
  - Some load imbalance due to data contention.

Option 2: Collective interface

- Same method (with consistent arguments) called collectively on all members of an ensemble.
  - Pro
    - Good encapsulation
    - Can be used to completely hide parallelism in upper layers of application
  - Con
    - Still lots of scope for error, e.g. Inconsistent calls.
    - Lack of flexibility.
    - Not appropriate for low-level operations
      - Introduce excessive synchronization and serialisation for operations where all processors not required to take part or not enough work to keep all processors busy.

Arguments to collective calls

- Arguments need to be consistent
- What do we mean by this?
  - Replicated data obviously ok
- But what about ensemble arguments? e.g.
  - **a.copy(b);**
  - **v2 = mat.mul(v1);**
  - If we require a particular data decomposition for the input then this breaks the encapsulation.
  - OK for closely coupled classes but not generally.
- Correctness should depend on the interface not the decomposition.
  - Collective operations built using Global interfaces allow this but are not the only solution.

Option 3: Cooperative Interface

- Interface consisting of local operations. e.g.
  - Decomposition query operations
  - Local data access
- Allows calling layer to adapt dynamically to different decompositions.
- Pro
  - Still allows decomposition to be changed without breaking code.
  - Can be used to build cleaner collective operations.
- Con
  - Encapsulation less good.

Java Investigation

- Explore these concepts using example code.
  - Simple image processing example.
  - Java 1.5
  - MPIJ
- Key aim is to encapsulate data decomposition.
  - If successful we should be able to:
    - write library code that accepts data in any decomposition.
    - copy data between 2 different ensembles that use the same coordinate space without knowing the decomposition of either.

Coordinate systems

- Most HPC applications organise data based on some concept of a coordinate system. e.g.
  - Cartesian X,Y,Z coordinates
  - Fourier wave number
  - Particle number
  - Mesh index.
  - Etc.
- Common operations include:
  - load/store based on coordinate.
  - Iterate over coordinate space.
- In many cases these are implicit concepts.
  - Primitive types are used to express coordinates.
  - Integer indexes, Nested DO loops and rectangular arrays are fine in many cases, and they are *very* fast.

Distributed data

- With distributed data things are more complex.
  - Different coordinates can live in different memory systems.
  - Need to be able to map a coordinate to a node/memory-space as well as a memory offset.
  - Need to be able to iterate over the local region of the coordinate system as well as over the global space.
  - Rectangular arrays and DO loops still work for block based decompositions but codes become more complex.
- User-defined types for coordinates and decomposition begin to look attractive.
  - Better type safety.
  - Allows run-time checking of any decomposition constraints.
  - Compiler needs to be able to implement these efficiently.
    - Small lightweight types not a Java strength.
    - For the moment ignore this.

Coordinate

- We want to support different coordinate types.
- No harm in tagging these with a Coordinate interface.
  - We will want a global ordering of Coordinates later.
  - All coordinates are therefore Comparable

```
/** Interface for objects representing a global coordinate.
 *
 * @param <T> final type of Coordinate
 *
 */
public interface Coordinate<T> extends Comparable<T>{

}
```

Data Container Interface

- An ensemble that stores data of type *D* indexed by coordinate system *C*.
  - **DataContainer<D,C extends Coordinate<C>>**
  - Would also be acceptable to define different interface for each coordinate system.
- Key requirement is to copy data from any DataContainer with the same data-type and coordinate-type independent of decomposition.
- Use a collective interface:
  - e.g. **public void copy(DataContainer<D,C> source);**
- Need to add some cooperative interface methods in order to implement this.

Data Container

```
/** 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 Interface

- Decompositions are defined over Coordinates
  - An interface as we want to support different decompositions.
  - Can represent any *static* decomposition.

```
public interface Decomposition<C extends Coordinate<C>> extends Iterable<C>{
/** Define decomposition
 * @param c Coordinate
 * @return rank of owning processor
 * @throws DecompositionException
 */
  public abstract int owner(C c) throws DecompositionException;

/** iterator over local Coordinates in coordinate rank order
 */
  public abstract Iterator<C> iterator();
/**
 * @return Communicator used by Decomposition
 */
  public abstract Intracomm getComm();
/**
 * @return Iterable over global coordinates
 */
  public abstract Iterable<C> all();
}
```

Exchange Data Container

- We can implement copy as follows:

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

Exchange copy

- This code only uses the methods defined in the various interfaces.
  - It will work for ***any*** class that implements DataContainer
- Loops over coordinates are always in Coordinate rank order
  - This defines the order of elements in the messages.
- ExchangeBuffer is a utility class that wraps MPI_Alltoallv and handles communication buffers.
- This should scale efficiently but high node-local overhead.
  - Lots of small objects.
  - Lots of method calls
  - Data copies

Possible performance improvements

epcc

- Test for compatible decompositions and perform local copy if appropriate.
- Re-implement for each coordinate type
  - Allows greater use of intrinsic types instead of temporary objects.
- Use block based decompositions
  - Loop over blocks not low level coordinates.
- Have re-usable copy-plan operations that pre-compute buffer sizes for commonly used transformations.
- Cache copy-plans in destination decomposition keyed by source decomposition.
  - Reduces overheads for subsequent copies between same decompositions.

Global copy

- If we had global put/get we could have done this.

```
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();
}
}
```

Global Copy

- Again this works on any source DataContainer
  - We are only using the global operations on the target object.
- Code is much simpler than the Exchange case.
- Data is moved as a series of small put operations.
  - Communication system has to be very good to support this efficiently.
  - However if communication system **IS** very good then it avoids the pack/unpack operations at each end.
  - Can easily switch between implementations depending on communication capabilities of target platform.

Global copy and PGAS

- Data container with global put/get is essentially a PGAS data structure implemented as a class rather than an intrinsic data-structure.
- PGAS languages decomposition managed by compiler
  - Current PGAS languages make decomposition quite explicit
    - separate local and decomposition coordinates
    - switching decomposition still relatively straightforward.
  - Compare with HPF
    - Single coordinate space
    - Could copy between arrays with same coordinate space
    - Decomposition explicit at declaration.

Data views

- Can use wrapper classes to provide a different view of data.
  - Type conversion
  - Simple coordinate transformations

```
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);
}
}
```

Impact

- We can use this to implement decomposition independent collective interfaces. e.g.

```
PgmIO pgm = new PgmIO(MPI.COMM_WORLD);
Array2D<Double> dat = pgm.read(new File("input.pgm"));

FFT2D fft = new FFT2D();
Array2D<Complex> ca = new DoubleToComplex2D(dat);
ca = fft.fft(true, ca);
```

- Methods accept ***any*** decomposition.
  - Even those programmer has not written yet.
  - Internally translate to their preferred decomposition using **copy**
- Return whatever decomposition is convenient to them.
  - e.g. File-IO could have all data on rank-0
  - Could change later if implementation changes.

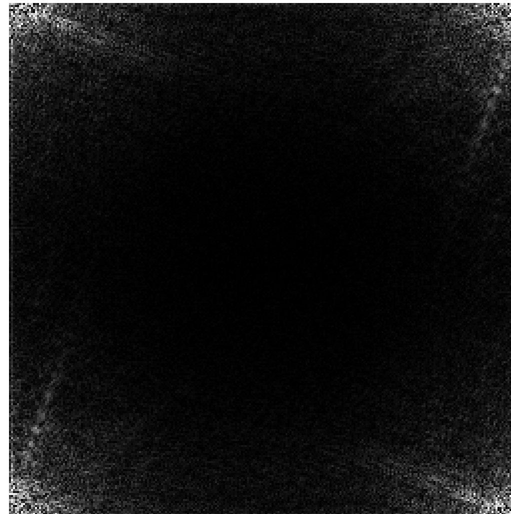
Impact-II

- Data transpose in *prototype* FFT implementation.
  - FFTArray uses row or column decomposition.
  - Real FFT would use custom communications for performance

```
public Array2D<Complex> fft(boolean forward,Array2D<Complex> src) throws MPIException,
DecompositionException{
  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;
  }
}
```

FFT result

- Decomposition independent routines seem more generally useful
  - Easy to write debug images at intermediate stages
  - Better code re-use
  - Easier to introduce new decompositions into existing code.
- Rapid prototyping



Performance

- There **is** a performance impact
  - Lots of copying
  - Lots of temporary arrays.
  - Roughly doubles communication overhead in our example.
- Fine for non performance critical sections of code.
- Easy mechanism to re-distribute data makes it easier to optimise the “hot-spot” sections.
  - No longer have to worry about data decomposition used in the rest of the program, can arrange data in best way for performance.
  - Still have to design interface so we don't redistribute too frequently.

Conclusions

- Data decomposition can be seen as a possible cause of code dependency in parallel programs analogous to data layout in sequential ones.
- We can use an OO design pattern to control these dependencies.
- Key benefits are available from existing OO languages and message passing systems.
  - Language extensions/constraint checker might help.