PeerJ Computer Science

Implementing generalized deep-copy in MPI

Joss Whittle[1,*], Rita Borgo[2,*] and Mark W. Jones[1,*]

[1] Department of Computer Science, Swansea University, Swansea, United Kingdom
[2] Informatics Department, King's College London, London, United Kingdom
[*] These authors contributed equally to this work.

ABSTRACT

In this paper, we introduce a framework for implementing deep copy on top of MPI. The process is initiated by passing just the root object of the dynamic data structure. Our framework takes care of all pointer traversal, communication, copying and reconstruction on receiving nodes. The benefit of our approach is that MPI users can deep copy complex dynamic data structures without the need to write bespoke communication or serialize/deserialize methods for each object. These methods can present a challenging implementation problem that can quickly become unwieldy to maintain when working with complex structured data. This paper demonstrates our generic implementation, which encapsulates both approaches. We analyze the approach with a variety of structures (trees, graphs (including complete graphs) and rings) and demonstrate that it performs comparably to hand written implementations, using a vastly simplified programming interface. We make the source code available completely as a convenient header file.

Subjects Computer Networks and Communications, Distributed and Parallel Computing, Programming Languages
Keywords MPI extension library, Deep copy, Serialization, Marshalling, Dynamic data structures, Deserialization, Unmarshalling

Submitted 6 July 2016
Accepted 4 October 2016
Published 21 November 2016

Corresponding author
Mark W. Jones,
m.w.jones@swansea.ac.uk

Academic editor
Srikumar Venugopal

Additional Information and Declarations can be found on page 57

DOI 10.7717/peerj-cs.95

Copyright
2016 Whittle et al.

Distributed under
Creative Commons CC-BY 4.0

OPEN ACCESS

INTRODUCTION

Message passing is an established communication paradigm for both synchronous and asynchronous communication in distributed or parallel systems. Using MPI with object orientation is not always an easy task, while control on memory locality and data distribution represent extremely valuable features, dealing with the ever growing and sophisticated features of OO languages can be cumbersome.

This problem is particularly challenging for data structures employing abstractions (e.g., inheritance and polymorphism) and pointer indirection, since transferring these data structures between disjoint hosts requires deep copy semantics. For user defined objects MPI adopts shallow copy semantics, whereby default copy constructors and assignment operators perform shallow copies of the object leaving memory allocation, copy, and de-allocation to be the responsibility of the programmer, not the implementation. A similar policy is applied to MPI objects, represented as handles to opaque data that cannot be directly copied. Copy constructors and assignment operators in user defined objects that contain an MPI handle must either ensure to invoke the appropriate MPI function to copy the opaque data (deep copy) or use a reference counting scheme that will provide references to the handle (reference counted shallow

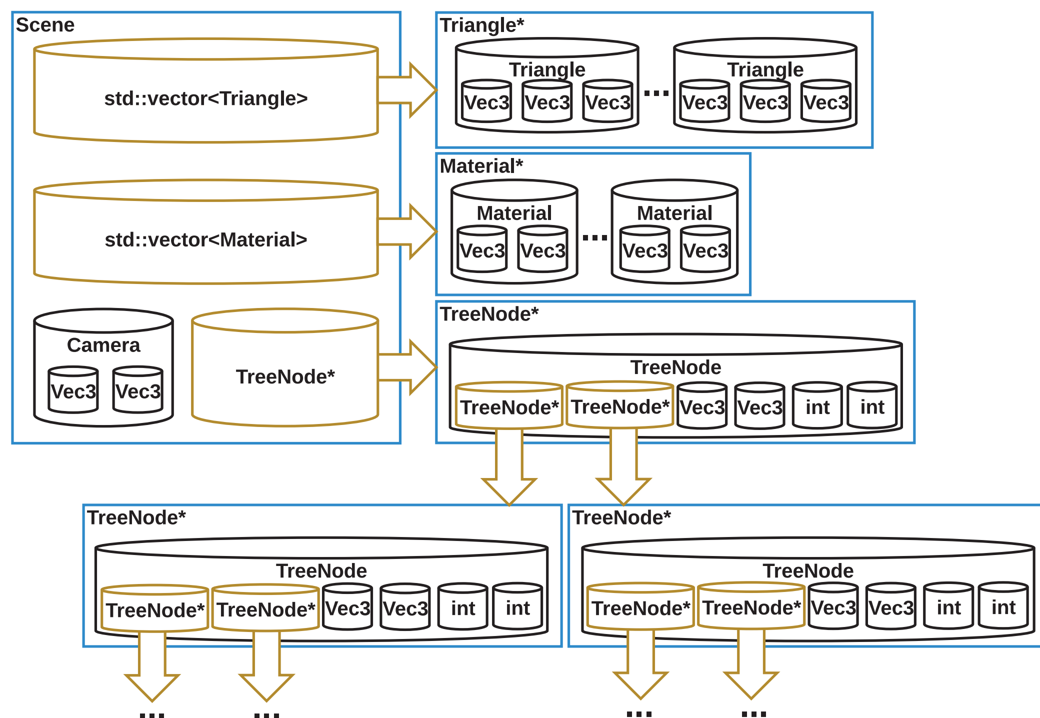

Figure 1 An example of a structure that requires deep copy semantics. Arrows represent pointer traversals to disjoint regions of memory.

copy). Shallow copy is acceptable for shared-memory programming models where it is always legal to dereference a pointer with the underlying assumption that the target of member pointers will be shared among all copies. Users often require deep copy semantics, as illustrated in Fig. 1, where every object in a data structure is transferred. Deep copy requires recursively traversing pointer members in a data structure, transferring all disjoint memory locations, and translating the pointers to refer to the appropriate device location also referred to as object serialization or marshalling, commonly used for sending complex data structures across networks or writing to disk. MPI has basic support for describing the layout of user defined data types and sending user-defined objects between processes (*Message Passing Interface Forum, 2014*).

The directives we propose provide a mechanism to shape and abstract deep copy semantics for MPI programs written in C++. Along with elegantly solving the deep copy problem, this mechanism also reduces the level of difficulty for the programmer who only needs to express the dependencies of an object type, rather than explicitly programming how and when to move the memory behind pointers.

As a motivating example, we show that comparable performance can be achieved when using a simple and generic algorithm to implement deep copy compared to hand coded native MPI implementations. The main contributions of this work are:

- We introduce the MPI Extension Library (MEL), a C++ header-only wrapper around the MPI standard which aims to give a simplified programming interface with

consistent type-safety and compile time error handling, along with providing efficient implementations of higher level parallel constructs such as deep copy.

- As a part of MEL, we provide generic implementations of deep copy semantics that can be easily applied to existing code to enable complex structured data to be deep copied transparently as either a send, receive, broadcast, or file access operation with minimal programmer intervention. The latter can also be used for the purpose of check-pointing when writing fault tolerant MPI code.

RELATED WORK

Message passing as a style of parallel programming enables easy abstraction and code composition of complex inter-process communications. Existing MPI interfacing libraries (*McCandless, Squyres & Lumsdaine, 1996*; *Huang et al., 2006*; *Boost-Community, 2015*) by default rely on the underlying standard shallow copy principle, where data contains no dependencies of memory outside the region directly being copied; and that where dependencies do exist that they are explicitly resolved by the programmer using subsequent shallow copies. However, this simplified model of communication comes at the cost of having to structure computations that require inter-process communication using low-level building blocks, which often leads to complex and verbose implementations (*Friedley et al., 2013*). Similar systems, such as the generic message passing framework (*Lee & Lumsdaine, 2003*) resolve pointers to objects, but do not follow dynamic pointers (data structure traversal) to copy complete complex dynamic structures possibly containing cycles.

MPI works on the principle that nothing is shared between processes unless it is explicitly transported by the programmer. These semantics simplify reasoning about the program's state (*Hoefler & Snir, 2011*) and avoid complex problems that are often encountered in shared-memory programming models (*Lee, 2006*) where automatic memory synchronization becomes a significant bottleneck.

Autolink and Automap (*Goujon et al., 1998*) work together to provide similar functionality. Automap creates objects at the receiver. Autolink tags pointers to determine whether they have been visited or not during traversal. The user must place directives in their code and carry out an additional compilation step to create intermediate files for further compilation. Extended MPICC (*Renault, 2007*) is a C library that converts user-defined data types to MPI data types, and also requires an additional compilation. It can automate the process, but also in some cases requires user input to direct the process. *Tansey & Tilevich (2008)* also demonstrate a method to derive MPI data types and capture user interaction via a GUI to direct the marshalling process.

Autoserial Library (*GNA, 2008*) gives a C++ interface for performing serialization to file as binary or XML; or to a raw network socket as binary data. Their library also offers a set of convenience functions for buffering data to a contiguous array with MPI communications to move the data. Their method makes extensive use of pre-processor macros to generate boilerplate code needed for deep traversal of objects. For MPI, this

library only handles the use case of fully buffered deep copy in the context of `MPI_Send` and `MPI_Recv` communications.

OpenACC (*Beyer, Oehmke & Sandoval, 2014*) tackles the deep copy problem in the context of transferring structured data from host machines to on node hardware such as GPUs and Accelerators. Their approach is based on a compiler implemented `#pragma` notation similar to OpenMP while our method is implemented as a header only template library.

TPO++ (*Grundmann, Ritt & Rosenstiel, 2000*) requires serialize and deserialize functions to be defined. The paper highlights good design goals which we also follow in this work.

Compared to the above approaches, we place much lighter requirements on the user and do not require additional signposting (usually implemented as preprocessor macros wrapped around variable declarations) that other methods require. We do not require an additional compilation step or GUI compared to the above as will be demonstrated in the following sections. We also provide an analysis of our approach. We explicitly demonstrate and analyze our approach on a wide variety of complex dynamic data structures. Our analysis shows that our approach has low time and memory overhead and also requires less user direction to achieve deep copy. It provides this extra functionality at no loss of performance over hand coded approaches. We avoid the in place serialize that some approaches utilize, resulting in our approach having a low memory overhead. We also evaluate our methods in comparison to Boost Serialization Library (*Cogswell, 2005*) and demonstrate that Boost introduces a performance penalty which our method avoids. Boost also requires more intervention from the user/programmer to achieve the same capability. Therefore, the main benefit of our approach over others is that it is a true deep copy approach where the user only has to pass in the root object/node of the data structure.

In CHARM++ (*Kale & Krishnan, 1993*; *Miller, 2015*) messages are by default passed by value, however CHARM++ provides support for deep copy via definition of serialization methods for non-contiguous data structures. It is a user task to define the proper serialization methods including the explicit definition of memory movement and copy operations. If the serialization methods are implemented correctly for a user-defined type, a deep copy will be made of the data being serialized. CHARM++ distinguishes between shared-memory and distributed-memory scenarios, where shared-memory data within a node can be directly passed by pointer. The programmer must explicitly specify the policy to be adopted by indicating if the data should be *conditionally packed* or not. Conditionally packed data are put into a message only when the data leaves the node. In an MPI environment processes within the same node do not share a common address space making such an optimization unavailable.

Generally the more desirable solution is to avoid deep copy operations to maintain efficiency in message transmission. This is straightforward to achieve by converting user-defined types with pointer members to equivalent user-defined types with statically-sized arrays. This approach of restructuring and packing a data structure is often used by shared-memory programming paradigms where structures with pointers are manually

packed and unpacked across the device boundary to reduce transfer costs for data structures used on the device.

When memory isolation (e.g., avoid cross boundary references) is not a requirement other approaches might be possible. For operations executed within sequential or shared memory multi-core processors, hardware can be used more efficiently by avoiding deep copy operations and rely instead on pointer exchange. This requires messages to have an ownership transfer semantics with calls to send (pass) ownership of memory regions, instead of their contents, between processes (*Friedley et al., 2013*). In the context of the present work, we do not focus on ownership passing but on the traditional approach of refactoring code. MEL provides an efficient and intuitive alternative to implementing object packing by hand. Porting an object type to use MEL deep copy only requires adding a member function to the type containing directives that describe the dependencies of the type. In this case, the additional effort to rewrite data structures to allow communication using the standard MPI shallow copy principles is much larger, making refactoring an application to avoid deep copy an undesirable solution.

Deep copy semantics are not only relevant when dealing with inter-process communication. When recovering from process or node failure in fault tolerant MPI, applications often incur problems very similar to the ones dealt by deep copy operations. Fault tolerance plays an important role in high performance computing applications (*Herault & Robert, 2015*) and significant research has focused on its development in MPI (*Gropp & Lusk, 2004*; *Vishnu et al., 2010*; *Bouteiller, 2015*). While the library itself does not provide explicit fault-tolerance support, MPI can provide a standard and well-structured context for writing programs that exhibit significant degrees of fault tolerant behavior. Several approaches have been investigated in literature to achieve fault tolerance in MPI (*Gropp & Lusk, 2004*; *Laguna et al., 2014*), with check-pointing being one of the most commonly used compared to more sophisticated approaches involving direct manipulation of the MPI standard to support fault tolerance (*Fagg, Bukovsky & Dongarra, 2001*; *Fagg & Dongarra, 2004*), or modifying semantics of standard MPI functions to provide resilience to program faults.

In check-pointing, a process will periodically cache its work to disk so that in the event of a crash or node failure, a newly spawned process can load back the last saved state of the failed process and continue the work from there. When the data a process is dependent on is deep in structure, the implementation challenges associated with reading and writing the data to disk are the same ones encountered when handling the communication of such types. MEL provides support for fault-tolerance by leveraging deep copy semantics to transparently target file reads and writes in the same manner it handles the sending and receiving of inter-process communications.

WHEN TO USE DEEP COPY

It is important that programmers be aware of the dangers of shallow-copying deep types without also resolving any dependencies of that type. For example, if an object contains a pointer and is copied by its memory footprint to another MPI process the value of the contained pointer on the receiver is now dangling and accessing the pointed to

memory erroneous. Listing 1 shows an example of performing such an MPI shallow-copy when a deep copy was needed.

Listing 1 User example–error from not resolving the data dependencies of an object when copying with MPI.

```
1  struct SomeStruct {
2      int *ptr = nullptr, len = 0;
3  };
4
5  //-----------------------------------------------------//
6  // On sending process
7  SomeStruct myVar;
8
9  // Allocate sub array
10 myVar.len = 10;
11 MPI_Alloc_mem(myVar.len * sizeof(int), MPI_INFO_NULL, &(myVar.
                                                        ptr));
12
13 // Populate sub array with values...
14
15 MPI_Send(&myVar, sizeof(SomeStruct), MPI_BYTE, dst_rank, tag,
                                                        comm);
16
17 //-----------------------------------------------------//
18 // On receiving process
19 SomeStruct myVar;
20 MPI_Recv(&myVar, sizeof(SomeStruct), MPI_BYTE, src_rank, tag,
                                                        comm);
21
22 // Error! myVar.ptr is now a dangling reference to the memory of the
                                                sending process!
```

While accessing the pointed to memory is invalid, if we declare as a rule that if a pointer is not allocated it will be assigned to `nullptr` (and we strictly adhere to this rule), we can use the value of the dangling pointer to determine if an allocation needs to be made and data received on the receiving process. Listing 2 gives a corrected example of Listing 1, by deep copying a struct containing a pointer safely using native MPI commands.

Listing 2 User example–hand coded deep copy using a dangling pointer from the sending process to determine if data needs to be received.

```
1  struct SomeStruct {
2      int *ptr = nullptr, len = 0;
```

```
3  };
4
5  //---------------------------------------------------//
6  // On sending process
7  SomeStruct myVar;
8
9  // Allocate sub array
10 myVar.len = 10;
11 MPI_Alloc_mem(myVar.len * sizeof(int), MPI_INFO_NULL, &(myVar.
                                                           ptr));
12
13 // Populate sub array with values...
14
15 // Send the footprint of the struct, allowing the receiver to check
16 //if ptr == nullptr or len == 0
17 MPI_Send(&myVar, sizeof(SomeStruct), MPI_BYTE, dst_rank, tag,
                                                           comm);
18
19 // Resolve the dependency of the struct
20 if (myVar.ptr != nullptr && myVar.len > 0) {
21     MPI_Send(myVar.ptr, myVar.len, MPI_INT, dst_rank, tag, comm);
22 }
23
24 //---------------------------------------------------//
25 // On receiving process
26 SomeStruct myVar;
27
28 // Receive the footprint of the struct so we can check if the array
29 // needs receiving
30 MPI_Recv(&myVar, sizeof(SomeStruct), MPI_BYTE, src_rank, tag, comm);
31
32 // Resolve the dependency of the struct
33 if (myVar.ptr != nullptr && myVar.len > 0) {
34     MPI_Alloc_mem(myVar.len * sizeof(int), MPI_INFO_NULL, &(myVar.
                                                           ptr));
35
36     MPI_Recv(myVar.ptr, myVar.len, MPI_INT, src_rank, tag, comm);
37 }
```

If an object which implements its own memory management through copy/move constructors and assignment operators, such as `std::vector`, is used, heap corruption can occur in a manner that can be difficult to debug. An example of this is shown in Listing 3.

If a std::vector is copied by footprint its internal pointer, just like the raw pointer previously, is no longer valid. The vector class works on the assumption that its internal pointer is always valid, and that it needs to be de-allocated or re-allocated if any of the assignment, resize, or destructor functions are called. If the vector goes out of scope and its destructor is called the incurring segfault will often not be caught correctly by a debugger and the error will be reported "nearby," leaving the programmer to hunt down the true source of the error. Short of using the C++ placement-new operator to force the vector to be recreated without calling its destructor there is no way of "safely" recovering in this situation.

Listing 3 User example–the dangers of copying deep types by their footprint in memory without fixing them properly on the receiving processes.

```
1   struct SomeStruct {
2       std::vector<int> someVec;
3   };
4
5   //----------------------------------------------------//
6   // On sending process
7   SomeStruct myVar;
8
9   // push_back into myVar.someVec a few times...
10
11  MPI_Send(&myVar, sizeof(SomeStruct), MPI_BYTE, dst_rank, tag,
                                                          comm);
12
13  // Resolve the dependency of the struct
14  if (myVar.someVec.size() > 0) {
15      MPI_Send(&(myVar.someVec[0]), myVar.someVec.size(), MPI_INT,
                                              dst_rank, tag, comm);
16  }
17
18  //----------------------------------------------------//
19  // On receiving process
20  SomeStruct myVar;
21  MPI_Recv(&myVar, sizeof(SomeStruct), MPI_BYTE, src_rank, tag,
                                                          comm);
22
23  // If myVar goes out of scope we segfault!
24  //myVar.someVec.clear();            // Segfault!
25  //myVar.someVec.resize(10);         // Segfault!
26  //myVar.someVec.reserve(10);        // Segfault!
27  //myVar.someVec = std::vector<int>();// Segfault!
```

```
28  // etc...
29
30  // It is safe to access .size() of the vector even if its internal
31  // pointer is invalid, we can use this to create a new vector in
32  // place, and to determine if we need to receive data.
33
34  // Force a new vector to be constructed at the memory address of the
35  // existing one without calling the existing vector's destructor.
36  new (&(myVar.someVec)) std::vector<int>(myVar.someVec.size());
37
38  // Resolve the dependency of the struct
39  if (myVar.someVec.size() > 0) {
40      MPI_Recv(&(myVar.someVec[0]), myVar.someVec.size(), MPI_INT,
                                                  src_rank, tag, comm);
41  }
```

Buffered vs. non-buffered

So far we have discussed methods for deep copying object types by recursively traversing the data-structure and performing discrete message operations to resolve each dependency. While often small there is a performance cost associated with beginning and ending a communication between processes, and this cost is exacerbated when communication occurs between processes on different physical nodes connected by a network interface. In many cases it is beneficial to pack a deep structure into a contiguous buffer on the sending process and to transport it as a single communication, the buffer can then be received and unpacked to reconstruct the target data structure. Listing 4 demonstrates a variant on Listing 2 where data is packed into a buffer before being transported and unpacked on the receiving process.

While buffered deep copy enables greater performance when communicating large structures made up of many small objects between processes, this speed comes at the cost of increased code complexity and limitations on the size of data that can be transferred. In the scenario where the data to be deep copied occupies more than half of the available system memory buffering into a contiguous buffer is no longer applicable as there is no remaining space in memory to allocate the buffer. Additionally, for programs that make many small allocations and de-allocations during normal execution system memory can become fragmented, leading to a situation where there is more than enough available memory to allocate the buffer but it is split up in many small pieces meaning no one contiguous allocation can be made. In these scenarios there is no alternative but to perform a non-buffered deep copy to move the data.

Listing 4 User example–hand coded buffered deep copy using a dangling pointer from the sending process to determine if data needs to be unpacked.

```
1  struct SomeStruct {
2      int *ptr = nullptr, len = 0;
3  };
4
5  //--------------------------------------------------//
6  // On sending process
7  SomeStruct myVar;
8
9  // Allocate sub array
10 myVar.len = 10;
11 MPI_Alloc_mem(myVar.len * sizeof(int), MPI_INFO_NULL, &(myVar.
                                                          ptr));
12
13 // Calculate buffer size and allocate space
14 int buffer_size = sizeof(SomeStruct);
15 if (myVar.ptr != nullptr && myVar.len > 0) {
16     buffer_size += (sizeof(int) * myVar.len);
17 }
18
19 char *buffer, *pos;
20 MPI_Alloc_mem(buffer_size, MPI_INFO_NULL, &buffer);
21 pos = buffer;
22
23 // Pack the struct itself to move non-deep members
24 memcpy(pos, &myVar, sizeof(SomeStruct));
25 pos += sizeof(SomeStruct);
26
27 // Pack the array of the struct
28 if (myVar.ptr != nullptr && myVar.len > 0) {
29     memcpy(pos, myVar.ptr, sizeof(int) * myVar.len);
30     pos += sizeof(int) * myVar.len;
31 }
32
33 // Send the buffer
34 MPI_Send(buffer, buffer_size, MPI_BYTE, dst_rank, tag, comm);
35
36 // Free the buffer
37 MPI_Free_mem(buffer);
38
39 //--------------------------------------------------//
```

```
40  // On receiving process
41  SomeStruct myVar;
42
43  // Calculate buffer size and allocate space
44  MPI_Status status;
45  MPI_Probe(src_rank, tag, comm, &status);
46
47  int buffer_size;
48  MPI_Get_count(&status, MPI_BYTE, &buffer_size);
49
50  char *buffer, *pos;
51  MPI_Alloc_mem(buffer_size, MPI_INFO_NULL, &buffer);
52  pos = buffer;
53
54  // Receive the buffer
55  MPI_Recv(buffer, buffer_size, MPI_BYTE, src_rank, tag, comm);
56
57
58  // Unpack the struct itself to move non-deep members
59  memcpy(&myVar, pos, sizeof(SomeStruct));
60  pos += sizeof(SomeStruct);
61
62  // Unpack the array of the struct
63  if (myVar.ptr != nullptr && myVar.len > 0) {
64      MPI_Alloc_mem(myVar.len * sizeof(int), MPI_INFO_NULL, &(myVar.
                                                                  ptr));
65
66      memcpy(myVar.ptr, pos, sizeof(int) * myVar.len);
67      pos += sizeof(int) * myVar.len;
68  }
69
70  // Free the buffer
71  MPI_Free_mem(buffer);
```

Buffering may also perform worse than non-buffered methods when the data to be deep copied consists of a small number of large objects, such as a struct containing several pointers to large buffers. In this case it may be detrimental to force the local copying of the large buffers into a single message only to unpack them on the receiving process when it would have been faster to transport them separately while taking the hit on the overheads associated with setting up multiple communications.

MEL–THE MPI EXTENSION LIBRARY

MEL is a C++11, header-only library, being developed with the goal of creating a lightweight and robust framework for building parallel applications on top of MPI. MEL is designed to introduce no (or minimal) overheads while *drastically* reducing code complexity. It allows for a greater range of common MPI errors to be caught at compile-time rather than during program execution when it can be far more difficult to debug.

A good example of this is type safety in the MPI standard. The standard does not dictate how many of the object types should be implemented leaving these details to the implementation vendor. For instance, in Intel MPI 5.1 `MPI_Comm` objects and many other types are implemented as integer handles, `typedef int MPI_Comm`, to opaque data that are managed by the MPI run-time. A drawback with this approach is it causes compile time type-checking of function parameters to not flag erroneous combinations of variables. The common signature `MPI_Send(void*, int, MPI_Datatype, int, int, MPI_Comm)` is actually seen by the compiler as `MPI_Send(void*, int, int, int, int, int)`, allowing any ordering of the last five variables to be compiled as valid MPI code, while potentially causing catastrophic failure at run-time. In contrast, Open MPI 1.10.2 implements these types as structs which are inherently type-safe. With MEL we aim to:

- Remain true to the underlying design of MPI, by keeping to an imperative function interface that does not fundamentally change the way in which the programmer interacts with the MPI run-time.
- To provide a type-safe, consistent, and unified function syntax that allows distributions of MPI from all vendors to behave in a common and predictable way at both compile-time and run-time.
- To be soluble, allowing the compiler to remove the abstractions MEL provides to achieve the same performance as native MPI code.
- To be memory efficient by minimizing the use of intermediate buffers whenever possible.
- To make use of modern C++ language features and advanced template meta programming to both ensure correctness at compile-time and to generate boiler-plate values that programmers have to provide themselves with native MPI code.
- To give higher-level functionality that is not available from the MPI standard such as deep copy Semantics (our focus in this paper).

MEL deep copy

Our algorithm is implemented in four parts, a top-level interface of functions for initiating deep copy as send/receive, broadcast, or file-IO operation; a transport API of functions that describe how data is to be moved within a deep copy operation, a set of transport methods that describe generically how to move a region of memory; and a hash-map interface for tracking which parts of the data structure have already been traversed. Figure 2 shows the architecture of our algorithm.

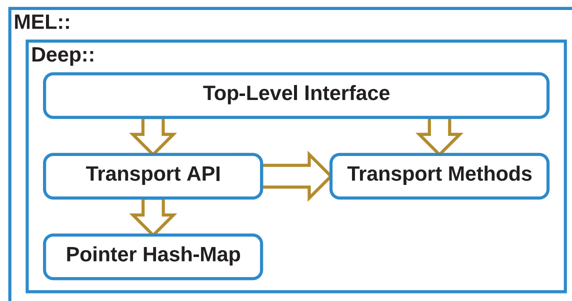

Figure 2 MEL deep copy architecture.

In order to ensure correct memory management for deep structures the user must adhere to:

- Unallocated pointers are initialized to `nullptr`.
- Dynamic Memory must be allocated using `MPI_Alloc_mem` and freed using `MPI_Free_mem`, or the equivalent MEL calls:

```
1  T*   MEL::MemAlloc<T>(int len)
2  T*   MEL::MemAlloc(int len, T &value)
3  void MEL::MemFree(T *ptr)
4  T*   MEL::MemConstruct<T>(Args &&...args)
5  void MEL::MemDestruct(T *ptr, int len = 1)
```

- Pointers refer to distinct allocations. E.g. It is erroneous to have an allocation of the form `char *ptr = new char[100]` in one object, and to then have a weak-pointer into the array in subsequent objects: `char *mySubPtr = &ptr[50]`. In these situations, it is best to store integer offsets into the array, rather than the pointer address itself.

Top-Level interface

The top-level interface for our algorithm (Listing 5) consists of functions for initiating a deep copy as a send, receive, broadcast, or file access operation on a templated pointer (`T*`), a pointer-length pair (`T*, len`), an object reference (`T&`), or an STL container (`std::vector<T>&`, `std::list<T>&`). In the case of receiving methods (`Recv`, `Bcast`, and `FileRead`) the len parameter can either be passed by reference so that it can be modified to reflect the number of elements that were actually received, or captured from an integer literal or constant variable to provide a run-time assertion whether the correct number of elements were received. All methods are blocking and do not return until the entire data-structure has been transferred.

Buffered variants of the top-level interface initiate a local deep copy to a contiguous buffer on the sender, this buffer is then sent as a single transport to the receiving processes where it can be unpacked. By decreasing the number of MPI communications or file

accesses needed to transfer a deep structure significant reductions in latency can be achieved, at the cost of added memory overhead from packing and unpacking data before and after transport. In general, large structures of small objects (i.e. a tree with many nodes that are small in memory) benefit most from buffering while smaller structures of large objects (i.e. a struct containing large arrays) tend to benefit from non-buffered transport.

Another motivating reason for providing a non-buffered mechanism for deep copy is the scenario where the deep structure occupies more than half of the available system memory. In such cases it is not possible to make a single contiguous allocation large enough to pack the structured data. An example of where this can happen is the use of MPI to distribute work to banks of Intel Xeon Phi Coprocessors which are exposed to the host system via a virtual network interface. While such hardware provides a large number of physical processor cores (60) on card memory is reduced (8–16 GB). On larger systems with more available memory this is less likely to occur although the use of non-buffered methods may still be desirable for the reasons outlined above; and in any case, achieving low memory overhead is good practice.

Detecting objects that require deep copy

Determining whether a given object is “deep” or not is performed at compile time using C++ template meta-programming to detect the presence of a member function of the form

```
template<typename MSG> void DeepCopy(MSG &msg)
```

that describes how to resolve the dependencies of a given object type. The template parameter `MSG` is a shorthand for `MEL::Deep::Message<TRANSPORT_METHOD, HASH_MAP>` where `TRANSPORT_METHOD` and `HASH_MAP` are types satisfying the constraints described in sections Transport Method and Hashing Shared Pointers, respectively. A detailed example of the method used to detect the presence of a matching member function is given in section Detecting the Deep Copy Function using Template Meta-Programming.

The use of template meta-programming in C++ allows for the complete set of possible copy operations needed to transport a structure to be known at compile time, allowing the compiler to make optimizations that might otherwise not be possible if inheritance and virtual function calls were used. Template programming also opens up the future possibility of using more advanced C++ `type_traits` such as `std::is_pod<T>` (is-plain-old-data) and other similar type traits to help make informed decisions about how best to move types automatically at compile time.

Listing 5 MEL implementation–MEL deep copy top-level interface.

```
1  // Calculate buffer size needed to pack an object or array of objects.
2  int MEL::Deep::BufferSize(T   &obj)
3  int MEL::Deep::BufferSize(T  *&ptr)
4  int MEL::Deep::BufferSize(T  *&ptr, const int len)
5  int MEL::Deep::BufferSize(STL  &container)
6  // ^ STL can  (currently)  be std::vector, std::list
```

```
7
8  // MPI_Send
9  void MEL::Deep::Send(T    &obj, const int dst, const int tag, const
                                                   MEL::Comm &comm)
10 void MEL::Deep::Send(T   *&ptr, ...)
11 void MEL::Deep::Send(T   *&ptr, const int len, ...)
12 void MEL::Deep::Send(STL  &container, ...)
13
14 // MPI_Recv
15 void MEL::Deep::Recv(T  &obj,  const int src, const int tag, const
                                                   MEL::Comm &comm)
16 void MEL::Deep::Recv(T *&ptr, ...)
17 void MEL::Deep::Recv(T *&ptr, int const &len, ...)
18 // ^ len matches  int literals and constants - runtime assertion
19 // on number of received elements
20 void MEL::Deep::Recv(T *&ptr, int &len, ...)
21 // ^ len matches  int  variables by reference - gets set to the
22 // number of received elements
23 void MEL::Deep::Recv(STL &container, ...)
24
25 // MPI_Broadcast
26 void MEL::Deep::Bcast(T   &obj, const int root, const MEL::Comm
                                                          &comm)
27 void MEL::Deep::Bcast(T   *&ptr, ...)
28 void MEL::Deep::Bcast(T   *&ptr, int const &len, ...)
29 // ^ len matches int literals and constants - runtime assertion on
30 // number of received elements
31 void MEL::Deep::Bcast(T   *&ptr, int &len, ...)
32 // ^ len matches  int  by  reference - set on receivers to the number
33 // of received elements
34 void MEL::Deep::Bcast(STL   &container, ...)
35
36 // STL File Streams
37 void MEL::Deep::FileWrite(T   &obj, std::ofstream &file)
38 void MEL::Deep::FileRead(T    &obj, std::ifstream &file)
39
40 // MPI_File
41 void MEL::Deep::FileWrite(T   &obj, MEL::File &file)
42 void MEL::Deep::FileRead(T    &obj, MEL::File &file)
43
44 // Overloads for buffered methods follow the same pattern
45 void MEL::Deep::BufferedSend(T  &obj, ...)
46 void MEL::Deep::BufferedSend(T  &obj, ..., const int bufferSize)
```

```
47  // ^ Source process can specify the buffer size to allocate for
48  // packing the structure
49
50  void MEL::Deep::BufferedRecv(T  &obj, ...)
51  // ^ Buffer size on the recieving processes is determined from the
52  // sender
53
54  void MEL::Deep::BufferedBcast(T  &obj, ...)
55  void MEL::Deep::BufferedBcast(T  &obj, ..., const int bufferSize)
56  // ^ Buffer size only used on sender, ignored on receiving
57  // processes
58
59  void MEL::Deep::BufferedFileWrite(T  &obj, ...)
60  void MEL::Deep::BufferedFileWrite(T  &obj, ..., const int
                                                      bufferSize)
61
62  void MEL::Deep::BufferedFileRead(T  &obj, ...)
```

Because we use the same function for sending/receiving, buffered/non-buffered, and for point-to-point/collective/ or file access communications we make use of a utility type, `Message`, that tracks which operation is being performed and where data is coming from or going to. The message object is created internally when one of the top-level functions is called and remains unmodified throughout the deep copy.

Message Transport-API

The deep copy function declares to our algorithm how data dependencies of a type need to be resolved in order to correctly rebuild a data structure on the receiving process. To keep the definition of this function simple the `Message` object exposes a small API of functions (Listing 6) that abstract the details of how data is sent and received between processes.

Listing 6 **MEL implementation–message transport-API.**

```
1  // Transfer a deep object. Only needed for deep types!
2  // Non-deep members are transported automatically
3  void Message::packVar(T &obj)
4
5  // Transfer a deep/non-deep pointer to len objects
6  void Message::packPtr(T *&ptr, int len = 1)
7
8  // Transfer a deep/non-deep pointer to len objects where the
9  // pointer may also be referenced in
```

```
10  // other parts of the deep structure. (i.e. A graph structure
11  // where multiple nodes point to a shared neighbour)
12  void Message::packSharedPtr(T *&ptr, int len = 1)
13
14  // Transfer a std::vector of deep/non-deep objects.
15  void Message::packSTL(std::vector<T> &vec)
16  // Or a std::list  (doubly linked list).
17  void Message::packSTL(std::list<T> &lst)
18
19  // Or use the shorthand operators
20  Message& Message::operator&(T  &obj) // <- Calls packVar which is
21                                       // only defined for deep types
22  Message& Message::operator&(std::vector<T> &vec)
23  Message& Message::operator&(std::list<T> &lst)
24
25  // Only used in Top Level interface functions, these variants differ
26  // only from their standard counterparts (above) in that they do not
27  // assume the parent object has been transported as for the root
28  // object there is no parent.
29  void Message::packRootVar(T  &obj)
30  void Message::packRootPtr(T  *&ptr, int len = 1)
31  void Message::packRootSTL(std::vector<T> &vec)
32  void Message::packRootSTL(std::list<T> &lst)
```

Listing 7 gives an example usage of the `Message` transport API to move a complex data-structure. All of the functions provided work transparently with both deep and non-deep types, with the exception of `Message::packVar` which is intended only for the transport of deep types as non-deep member variables will be transported automatically. By comparison, Boost Serialization Library requires that all types except for language defined base types (i.e. `int`, `bool`, `double`) provide serialization functions regardless of whether they contain deep members, and that all member variables within the type (including non-deep members) are explicitly registered with the archive object.

Listing 7 MEL implementation–registering dependencies using the transport-API.

```
1  struct SomeDeepStruct {
2      // Non-deep members will be copied automatically.
3      int a, b, c, len;
4      SomeFlatStruct d;
5
6      // Deep members must be declared in the Deep-copy function
7      AnotherDeepStruct e, f, g;
```

```
8       char *myArray = nullptr;
9       GraphNode *mySharedPointer = nullptr;
10      std::vector<int> v;
11      std::vector<AnotherDeepStruct> w;
12
13      template<typename MSG>
14      void DeepCopy(MSG &msg) {
15          // Pack a deep object by reference.
16          msg.packVar(e);
17          // A lighter syntax for non-pointer members.
18          msg & f & g;
19
20          // Transfer a char array of len elements.
21          msg.packPtr(myArray, len);
22          // Transfer a shared pointer that may also be used
23          // elsewhere in the structure.
24          msg.packSharedPtr(mySharedPointer);
25
26          // Transfer a std::vector.
27          msg.packSTL(v);
28          // We can also transfer a std::vector or std::list
29          // using & syntax.
30          msg & w;
31
32          // In fact, we can simply replace all of the above
33          // code (in this function) with:
34          msg & e & f & g & v & w;
35          msg.packPtr(myArray, len);
36          msg.packSharedPtr(mySharedPointer);
37      }
38  };
```

An example copy

In essence, the deep copy algorithm works by both sending and receiving processes entering a message loop or handshake with one another where they both expect to keep sending and receiving data until the entire structure has been transferred. The sending process determines how much data is to be sent, and this information is conveyed to the receiving processes transparently in such a way that when a receiving process determines there is nothing left to receive the sending process has returned.

Listing 8 shows an example of using the deep copy function to move an array of non-deep objects. Because the type, `int`, does not provide a member function for deep copy

the footprint of the array is sent in a single MPI message. On the receiving process memory is allocated into the pointer provided and the data is received.

Listing 8 User example–MEL deep copy of non-deep type.

```
1  // On sending process
2  int len = 10;
3  int *ptr = MEL::MemAlloc<int>(len);
4
5  // Fill ptr with some values... ptr = [0..len)
6  for (int i = 0; i < len; ++i) ptr[i] = i;
7
8  MEL::Deep::Send(ptr, len, dst_rank, tag, comm);
9
10 //---------------------------------------------------//
11 // On receiving process
12 int len;
13 int *ptr = nullptr;
14 MEL::Deep::Recv(ptr, len, src_rank, tag, comm);
15 // len  = 10 and ptr now equals an address to len integers
16 // ptr  = [0..len)
```

An example of moving an array of structs containing pointers to dynamically allocated memory is given in Listing 9. In order to correctly reconstruct the data on receiving processes a deep copy function has been implemented which tells the algorithm to copy a char array containing len elements. Because the type has a deep copy function the receiving processes will allocate the memory for the array of structs and copy the footprint of the array as a single contiguous chunk resulting in non-deep member variables being transferred automatically. The receiving process makes the necessary allocations to receive its dependencies. Both sending and receiving processes will then loop over each element in their array and call the objects deep copy function to resolve its data dependencies. If the struct contained variables which themselves required a deep copy the algorithm would recurse on them until all dependencies are resolved. In this simple case, however, the struct contains a char array which does not require a deep copy and as such the sub-array is transferred by allocating the needed memory and copying the entire sub-array as one contiguous chunk, as in Listing 8.

Listing 9 User example–MEL deep copy of deep type.

```
1  struct SomeStruct {
2      int len;
3      char *array = nullptr;
4
```

```
5       template<typename MSG> void DeepCopy(MSG &msg) {
6           msg.packPtr(array, len);
7       }
8   };
9
10  //------------------------------------------------------//
11  // On sending process allocate array and subarrays
12  int len = 5;
13  SomeStruct *ptr = MEL::MemAlloc<SomeStruct>(len);
14
15  for (int i = 0; i < len; ++i) {
16      // Allocate sub array
17      ptr[i].len = i + 1;
18      ptr[i].array = MEL::MemAlloc<char>(ptr[i].len);
19
20      // Fill ptr[i].ptr with some values... ptr_i = [0..len)
21      for (int j = 0;  j < ptr[i].len; ++j) ptr[i].array[j] = j;
22  }
23
24  MEL::Deep::Send(ptr, len, dst_rank, tag, comm);
25
26  //------------------------------------------------------//
27  // On receiving process
28  int len;
29  SomeStruct *ptr = nullptr;
30  MEL::Deep::Recv(ptr, len, src_rank, tag, comm);
31  //  len = 5 and ptr equals an address to an array of 5 structures
32  //  each having their respective lengths and subarrays
33  //  ptr = [0..5) : { [0..1), [0..2), [0..3), [0..4), [0..5) }
```

Transport method

The `Message` object represents how our algorithm traverses the deep structure and ensures that both sending and receiving processes independently come to the same conclusion on what order objects are traversed in with minimal communication. This traversal order is independent of, and identical for all deep copy operations. Because of this we template the `Message` object on a type that represents the specific nature of the data transportation we want to perform (i.e. `Message<TransportSend>` to perform deep copy as an `MPI_Send` communication), allowing the same traversal scheme to be reused.

As a part of our implementation we provide transport methods for a wide variety of data movement scenarios:

TransportSend	Performs each transport call as a discrete `MPI_Send` communication.
TransportRecv	Performs each transport call as a discrete `MPI_Recv` communication.
TransportBcastRoot	Performs each transport call as a discrete `MPI_Bcast` communication, as a sender.
TransportBcast	Performs each transport call as a discrete `MPI_Bcast` communication, as a receiver.
TransportFileWrite	Performs each transport call as a discrete `MPI_FileWrite` operation.
TransportFileRead	Performs each transport call as a discrete `MPI_FileRead` operation.
TransportSTLFileWrite	Performs each transport call as a discrete `std::ofstream::write`.
TransportSTLFileRead	Performs each transport call as a discrete `std::ifstream::read`.
TransportBufferWrite	Performs each transport call as a discrete `std::memcpy` to a contiguous memory buffer.
TransportBufferRead	Performs each transport call as a discrete `std::memcpy` from a contiguous memory buffer.
NoTransport	This transport method acts as a sender but does not move any data. This method is used to implement the top-level interface functions for `MEL::Deep::BufferSize` which counts how many bytes need to be moved without performing any transportation.

Adding additional transport methods is as simple as implementing a class with a public-member function of the form

```
template<typename T> inline void transport(T *&ptr, const int len)
```

that describes how to move a region of memory, and a public-static-member variable `static constexpr bool SOURCE` which tells the compiler whether or not this is a sending or a receiving transport method. This boolean is important as it tells the `Message` object whether or not it needs to make allocations as it traverses the deep structure. The transport method should also store any state variables need to maintain the transport over the duration of the deep copy. Such state variables may be but are not limited to an MPI communicator and process rank, a file handle, or a pointer to an array used for buffering.

Hashing shared pointers

When considering large structured data containing duplicate pointers the method used to track which parts of the structure have already been transported can have a significant impact on the traversal time. A hash-map is a natural choice for representing an unordered map between two pointers as it is efficient for random access lookups and insertions.

As with the transport method, the `Message` object is also templated on the hash-map to use for pointer tracking, namely `Message<TRANSPORT_METHOD, HASH_MAP = MEL::Deep::PointerHashMap>`. This allows for the user to provide an adapter to their own implementation of a hash-map specifically optimized for pointers or to provide an adapter type to a third-party hash-map implementation.

To use a custom hash-map with any of the top-level functions simply override the default template parameter when initiating a deep copy operation. E.g. `MEL::Deep::Send<int, MyCustomHashMap>(ptr, len, dst, tag, comm);` where `MyCustomHashMap` exposes public-member functions of the form:

```
template<typename T> inline bool find(T* oldPtr, T* &ptr)
template<typename T> inline void insert(T* oldPtr, T* ptr)
```

These functions are templated on the pointer type, `T*`, so that user provided hash-map adapters are able to use this extra type information to optimize hashing if needed.

External deep copy functions

So far we have discussed the use of deep copy functions and the transport API in cases where the deep copy function was a local member function of the type being considered. In some use cases, a structure may be defined in headers or libraries that cannot be modified easily (or at all). In such cases, we still would like to be able to define the deep copy semantics for the type without directly modifying its implementation. To enable this, we provide an overload of all the functions in the transport API and top-level interface that take an additional template parameter that is a handle to a global-free-function of the form

```
template<typename MSG> inline void MyTypeDeepCopy(MyType &obj, MSG
&msg)
```

that takes by reference an instance of the object to transport and a `Message` object to perform the deep copy.

Listing 10, shows the usage of external free deep copy functions with types needing deep copy. `StructB` contains an internal member function for performing deep copy, while `StructA` does not. Passing an instance of `StructA` to the top-level interface will result in incorrect results as its dependencies will not be resolved. By implementing a global-free-function that defines the deep copy requirements of `StructA`, we can then tell the top-level interface to explicitly use that function to resolve external dependencies of the type. If we provide an external free function for `StructB` which already has an internal deep copy function, the internal function is ignored and the free function explicitly given is used.

Listing 10 User example–using external global-free-functions for deep copy.

```
1  struct StructA {
2      std::vector<int> arr;
3  };
4
```

```
5   struct StructB {
6       std::list<int> lst;
7
8       // Internal - Local member deep-copy function
9       template<typename MSG> void DeepCopy(MSG &msg) {
10          msg & lst;
11      }
12  };
13
14  // External - Global free deep-copy function
15  template<typename MSG> void StructA_DeepCopy(StructA &obj, MSG
                                                                    &msg) {
16      msg & obj.arr;
17  }
18
19  // External - Global free deep-copy function
20  template<typename MSG> void StructB_DeepCopy(StructB &obj, MSG
                                                                    &msg) {
21      msg & obj.lst;
22  }
23
24  // Example usage:
25  StructA sA;
26
27  MEL::Deep::Send(sA, dst, tag, comm);
28  // ^  Error! StructA contains a std::vector but does not
29  // have a deep-copy function
30
31  MEL::Deep::Send<StructA, MEL::Deep::PointerHashMap,
                                 StructA_DeepCopy>(sA, dst, tag, comm);
32  // ^  Correct. Uses external free function to perform the deep-copy
33
34
35  StructB sB;
36
37  MEL::Deep::Send(sB, dst, tag, comm);
38  // ^  Correct. Uses internal member function to perform the deep-copy
39
40  MEL::Deep::Send<StructB, MEL::Deep::PointerHashMap,
                                 StructB_DeepCopy>(sB, dst, tag, comm);
41  // ^ Correct. Uses external free function (overrides
42  // internal function) to perform the deep-copy
```

The same rules apply for providing external free functions to the transport API. Listing 11, shows an example of this, where once again StructA is a deep type that does not provide an internal deep copy function. StructC is also deep and contains a std::list of StructA. If the deep copy function of StructC simply calls the ampersand operator or Message::packSTL function (Listing 11, lines 15, 16) to transport the std::list then the instances of StructA will be transported incorrectly as a non-deep type. In the same manner as with the top-level interface the free function to use to deep copy StructA is given explicitly to Message::packSTL so that it can correctly resolve the dependencies of the deep structure.

Listing 11 **User example–using external global-free-functions for deep copy with the Transport-API.**

```
1   struct StructA {
2       std::vector<int> arr;
3   };
4
5   // External - Global free deep-copy function
6   template<typename MSG> void StructA_DeepCopy(StructA &obj, MSG
                                                                  &msg) {
7       msg & obj.arr;
8   }
9
10  struct StructC {
11      std::list<StructA> lst;
12
13      // Internal - Local member deep-copy function
14      template<typename MSG> void DeepCopy(MSG &msg) {
15          //msg & lst;          // <- Error - StructA has no internal
16                                // deep-copy function
17          //msg.packSTL(lst); // <- Error
18          msg.packSTL<StructA, StructA_DeepCopy>(lst); // <- Correct
19      }
20  };
21
22  // Example usage:
23  StructC sC;
24  MEL::Deep::Send(sC, dst, tag, comm);
25  // ^  Correct. Uses internal member function of StructC and the
26  // external free function StructA_DeepCopy for StructA.
```

The option to use external deep copy functions gives our method flexibility when we need to add deep copy semantics to code that cannot be directly, or easily modified. However, this does not mean it will always be applicable as it requires intimate and low-level knowledge of the object's internal implementation and methods of allocation.

MEL IMPLEMENTATION DETAILS

In the following section we provide a detailed discussion of the implementation of the MEL deep copy algorithm.

Detecting the deep copy function using template meta-programming

To detect whether the type under consideration contains a deep copy function we make use of *SFINAE* (*Substitution Failure Is Not An Error*) to create a compile-time boolean test for the existence of a member function with the desired signature. We encapsulate the usage of this method into a templated shorthand that uses `std::enable_if` to give us a clean and concise method for providing function overloads for deep and non-deep types.

Listing 12, shows an implementation of the technique used to conditionally detect member functions of template types at compile time. The overloads of `void someFunc (T &obj)` for when `T` is or is not a type with a deep copy function allows us specialize our implementation for deep types while allowing them to share identical function signatures.

Listing 12 **MEL implementation–detecting the deep copy function.**

```
1   template<typename T>
2   struct HasDeepCopyMethod {
3       // This pseudo-type does not exist unless type U has a member
4       // function of the desired form:
5       // template<typename MSG> void DeepCopy(MSG &msg)
6       template<typename U, void(U::*)(MEL::Deep::Message
                                  <NoTransport>&)> struct SFINAE {};
7
8       // If this succeeds Test<T> will be a function that returns char
9       template<typename U> static char Test(SFINAE<U, &U::DeepCopy>*);
10      // Otherwise Test<T> will return an int
11      template<typename U> static int Test(...);
12
13      // We can now test if type T has the desired member function by
14      // seeing if the result is the size of a char or an int.
15      static const bool value = sizeof(Test<T>(0)) == sizeof(char);
16  };
17
18  // Shorthands for when implementing functions
19  template<typename T, typename R = void>
20  using enable_if_deep     = typename std::enable_if<
                                  HasDeepCopyMethod<T>::value, R>::type;
21  template<typename T, typename R = void>
22  using enable_if_not_deep = typename std::enable_if
                            <!(HasDeepCopyMethod<T>::value), R>::type;
```

```
23
24  // Example usage in function definitions
25  template<typename T>    enable_if_deep<T> someFunc(T &obj) {
26      std::cout  << "Called with deep type!" << std::endl;
27  }
28
29  template<typename T> enable_if_not_deep<T> someFunc(T &obj) {
30      std::cout  << "Called with non-deep type!" << std::endl;
31  }
32
33  // A deep type
34  struct StructA {
35      template<typename MSG> void DeepCopy(MSG &msg) {}
36  };
37
38  StructA sA;
39  someFunc(sA); // Called with deep type!
40
41  int i;
42  someFunc(i); // Called with non-deep type!
```

Transport-API implementation

Next we describe the implementation of the transport API which specifies the traversal order our algorithm uses when performing deep copy.

Message::packVar

The `Message::packVar` function will call the deep copy function of the given variable to resolve its dependencies. This function works on the assumption that local member variables of the object have already been transported when the parent object was traversed. It is for this reason that `Message::packVar` is only defined for deep types, as a non-deep type will have been transported automatically with the parent. In all of the following listings for the implementations of the transport API the overloads for non-deep types have been omitted for space.

Listing 13 **MEL implementation–`Message::packVar`.**

```
1  // Transport a deep object
2  template<typename D>
3  inline enable_if_deep<D> Message::packVar(D &obj) {
4       // Assumes that the footprint of obj has already been transported
5       obj.DeepCopy(*this); // *this == the Message object
6  }
```

Message::packPtr

When transporting dynamically allocated memory special care must be taken to correctly allocate memory on the receiving processes. Listing 14 shows the implementation of `Message::packPtr` for deep types. This function offloads its work to the `transportAlloc` helper function of the `Message` object. On receiving process, `transportAlloc` will make an allocation of `len` elements of the given type before receiving the data. On the sending process, `transportAlloc` is identical to `transport` and simply moves the requested data. For a deep type, `Message::packPtr` will then loop over all the received elements and call their deep copy functions to resolve any dependencies.

Listing 14 MEL implementation–Message::packPtr.

```
1  // Transport a deep pointer to len objects
2  template<typename D>
3  inline enable_if_deep<D> Message::packPtr(D *&ptr, int len = 1) {
4      // On sender    - If  (len > 0) and (ptr != nullptr) send the memory
5      //
6      // On receiver  - If  (len > 0) and (ptr != nullptr) then overwrite
7      // the dangling ptr  with a new allocation of len elements and
8      // receive the memory
9      transportAlloc(ptr, len);
10
11      // Followed by the recursion for deep types
12      if (ptr != nullptr) {
13          for (int i = 0; i < len; ++i) ptr[i].DeepCopy(*this);
14      }
15 }
```

Message::packSharedPtr

In complex structured data there is often a requirement for data to be self referencing. That is, one part of the deep structure may be pointed to from multiple other points within the structure. In these situations, a naïve deep copy algorithm would traverse the shared object within the structure multiple times allocating a unique copy of it with each visit. If the shared object is deep itself and points to one of its ancestors within the structure, then the deep copy algorithm will become stuck in an infinite cycle within the data, allocating new memory with each loop. To avoid this and to allow complex self-referential data to be transported, we provide the `Message::packSharedPtr` function shown in Listing 15. This method checks the given pointer against a hash-map of type ($pointer \rightarrow pointer$) to determine if the pointed to memory has already been transported.

Listing 15 MEL implementation–`Message::packSharedPtr`.

```
1  // Transport a deep shared pointer to len objects
2  template<typename D>
3  inline enable_if_deep<D> Message::packSharedPtr(D *&ptr, int len = 1) {
4      // Save the original pointer in case we modify it
5      D *oldPtr = ptr;
6  
7      // Is the given pointer already in the hash-map?
8      // If so, set ptr equal to the pointer stored in the hash-map and
9      // return
10     if (pointerMap.find(oldPtr, ptr)) return;
11 
12     // Same as for packPtr
13     transportAlloc(ptr, len);
14 
15     // Insert the (newly allocated, on receiver) ptr into the hashmap
16     // with the original dangling pointer (from the sender) as the key
17     pointerMap.insert(oldPtr, ptr);
18 
19     // Followed by the recursion for deep types
20     if (ptr != nullptr) {
21         for (int i = 0; i < len; ++i) ptr[i].DeepCopy(*this);
22     }
23 }
```

During deep copy, the first time a shared pointer is passed to `Message::packSharedPtr` on both the sending and receiving processes, it is transported in the same manner as in `Message::packPtr` by calling `transportAlloc`. On the sending process, the pointer is then inserted into the hash-map so it can be ignored if it is visited again. On the receiving processes, the call to `transportAlloc` will have caused the dangling pointer from the sender to have been overwritten with the newly allocated pointer. This new pointer is inserted into the hash-map with the original (dangling) pointer as the key, so that next time the receiver is asked to transport the same dangling pointer it can simply lookup and return the existing allocation.

When a shared pointer that has already been visited is passed to `Message::packSharedPtr` and it is found within the hash-map then sending process can simply return as no memory needs to be transported; the receiving process uses the dangling pointer passed to it to retrieve the valid pointer that was previously allocated and transported the last time the shared pointer was visited. All interaction with the hash-map is performed through the `pointerMap.find` and `pointerMap.insert` functions of the `Message` object. These functions are further discussed in Section Hash-map implementation.

A nice property of this scheme is that the hash-map is never communicated and is constructed independently on both the sending and receiving processes. This means that for non-buffered communications the sender and receiver can traverse the structure in parallel (lock-step), and for buffered communications or buffered/non-buffered file-access the processes can traverse the structure independently.

Message::packSTL

As part of the transport API, we provide helper functions for moving common C++ STL containers. Listing 16 shows the implementation of `Message::packSTL` for C++ `std::vector`'s of both deep and non-deep types. This is very similar to the implementation of `Message::packPtr` discussed previously with the slight difference that instead of making a new allocation on the receiving processes via `transportAlloc` we instead repair the internal pointer of the given `std::vector` by calling the placement-new operator to recreate the vector in place (as discussed in Listing 3). The implementations of `Message::packSTL` for other STL containers is conducted in the same way and is omitted here.

Listing 16 MEL implementation–`Message::packSTL` for `std::vector`.

```
1   // Transport a std::vector of deep types
2   template<typename D>
3   inline enable_if_deep<D> packSTL(std::vector<D> &obj) {
4       // std::vector::size() is safe to access even if the internal
5       // pointer is invalid
6       int len = obj.size();
7       // If this is a recieving process then we need to repair the
8       // dangling internal pointer
9       if (!TRANSPORT_METHOD::SOURCE) {
10          // std::vector forces construction of elements
11          new (&obj) std::vector<D>(len, D());
12           // we need to call the destructor explicitly in case any
13           //resources were acquired upon default construction of each
14           // element.
15          for (int i = 0;  i < len; ++i)   (&obj[i])->~D();
16      }
17
18      D *p = &obj[0];
19      if (len > 0) transport(p, len);
20
21      // Followed by the recursion for deep types
22      for (int i = 0; i < len; ++i) {
23          obj[i].DeepCopy(*this);
24      }
25  }
```

Message::packRootVar, Message::packRootPtr, & Message::packRootSTL

Finally, we provide a set of functions to simplify the implementation of the top-level interface. Recall that `Message::packVar` is only defined for deep types and assumes that the object's footprint is always transported with the parent object. This is not the case for the top-level functions as no parent has been transported; in this case we must explicitly transport the object footprint regardless of whether it is deep or not.

A similar scenario occurs for pointers passed to the top-level interface. In order to avoid duplicating all of the top-level functions to account for whether the root pointer is shared we always insert it into the hash-map as this is a small constant overhead that does not affect performance. Recall from the implementation of `Message::packSharedPtr` that on the receiving processes the dangling pointer from the sender is used as the key into the hash-map. Because of this, for the root pointer we must explicitly transport the address-value of the pointer from the sender to the receiving processes so they can insert it into their hash-maps.

Finally, when considering STL containers passed to the top-level interface, receiving processes cannot query `.size()` of the container as its footprint was not previously transported. Instead, we explicitly transport the size of the container and call `.resize()` on the receiving processes.

Listing 17 **MEL implementation–`Message::packRootVar`, `Message::packRootPtr`, & `Message::packRootSTL`.**

```
1   // Transport the footprint of a non-deep object
2   template<typename T>
3   inline enable_if_not_deep<T> Message::packRootVar(T &obj) {
4       transport(obj);        // Transport the footprint
5   }
6
7   // Transport the footprint of a deep object and call its
8   // DeepCopy function
9   template<typename D>
10  inline enable_if_deep<D> Message::packRootVar(D &obj) {
11      transport(obj);       // Transport the footprint
12      obj.DeepCopy(*this); // Recurse on the deep structure
13  }
14
15  // Transport a root pointer to len deep objects
16  template<typename D>
17  inline enable_if_deep<D> Message::packRootPtr(D *&ptr, int len = 1) {
18      // Explicitly transport the pointer value for the root node
19      // so the it can be hashed correctly on recieving processes
```

```
20      size_t addr = (size_t) ptr;
21      transport(addr);
22      ptr  =  (D*) addr;
23
24      // Same as packSharedPtr, except we don't need to check the pointer
25      D *oldPtr = ptr;
26      transportAlloc(ptr, len);
27      pointerMap.insert(oldPtr, ptr);
28
29      // Followed by the recursion for deep types
30      if (ptr != nullptr) {
31          for (int i = 0; i < len; ++i) ptr[i].DeepCopy(*this);
32      }
33  }
34
35  // Transport a root stl container to len deep objects
36  template<typename D>
37  inline enable_if_deep<D> packRootSTL(std::vector<D> &obj) {
38      // Explicitly transport the length of the container
39      int len;
40      if (TRANSPORT_METHOD::SOURCE) {
41          len = obj.size(); transport(len);
42      }
43      else {
44          transport(len); obj.resize(len, D());
45          for (int i = 0; i < len; ++i) (&obj[i])->~D();
46      }
47
48      D *p = &obj[0];
49      if (len > 0) transport(p, len);
50
51      // Followed by the recursion for deep types
52      for (int i = 0; i < len; ++i) {
53          obj[i].DeepCopy(*this);
54      }
55  }
```

Transport method implementation & usage

A transport method is a class which provides a single public-member function of the form

```
template<typename T> inline void transport(T *&ptr, const int len)
```

which defines how to move `len` objects of type `T` from a given pointer `ptr`. Listing 18 shows the implementation of the `TransportSend` transport method, which defines

how to move data using a discrete MPI_Send for each transport. An instance of a transport method carries any state needed to represent the data movement over the duration of the deep copy. In the case of TransportSend the state needed to represent the transfer are the MPI rank of the destination process, a tag to use for the communication, and the MPI communicator over which the data will be transferred. For other transport methods the state may be a file handle, or a pointer to an array used for buffering.

Listing 18 **MEL implementation–transport method for Message.**

```
1   class TransportSend {
2   private:
3       // Members - Store any state or resources needed to maintain
4       // this transport method
5       const int pid, tag;
6       const MEL::Comm comm;
7
8   public:
9       // A transport method is either a source or a destination
10      // This is known at compile time
11      static constexpr bool SOURCE = true;
12
13      TransportSend(const int _pid, const int _tag, const MEL::Comm
                                                              &_comm)
14                          : pid(_pid), tag(_tag), comm(_comm) {}
15
16      // Transport function describes how to move data, in this case by
17      // performing an MPI_Send
18      template<typename T>
19      inline void transport(T *&ptr, const int len) {
20          MEL::Send(ptr, len, pid, tag, comm);
21      }
22  };
```

Listing 19 shows the implementation of one of the top-level interface functions for performing deep copy as an MPI_Send operation. A Message<TransportSend> object is instantiated, and the parameters from the function are transparently forwarded to the instance of the transport method within the Message object using std::forward<Args>(args). After creating the message object the pointer to the deep structure can be transported by calling Message::packRootPtr from the transport API.

Listing 19 MEL implementation–usage of a transport method in the top-level interface.

```
1  template<typename P, typename HASH_MAP = MEL::Deep::PointerHashMap>
2  inline enable_if_pointer<P> Send(P &ptr, const int dst, const int
                                          tag, const Comm &comm) {
3      // Arguments to the Message constructor are std::forward'd to the
4      // TransportSend constructor
5      Message<TransportSend, HASH_MAP> msg(dst, tag, comm);
6
7      // Transport the deep-structure
8      msg.packRootPtr(ptr);
9  }
```

When performing a buffered deep copy the data is first packed into a contiguous buffer on the sending process before being transported as a single operation to the receiving processes where the data can then be expanded back into the deep structure. Listing 20 shows the implementation of `BufferedSend` and `BufferedRecv` which make use of the `TransportBufferWrite` and `TransportBufferRead` transport methods.

Listing 20 MEL implementation–usage of a buffered transport method in the top-level interface.

```
1  template<typename P, typename HASH_MAP = MEL::Deep::PointerHashMap>
2  inline enable_if_pointer<P> BufferedSend(P &ptr, const int dst,
3                                            const int tag,
4                                            const Comm &comm) {
5      // Compute the buffer size for the deep structure and transport it
6      MEL::Deep::BufferedSend(ptr, dst, tag, comm, MEL::Deep::
                                                    BufferSize(ptr));
7  }
8
9  template<typename P, typename HASH_MAP = MEL::Deep::PointerHashMap>
10 inline enable_if_pointer<P> BufferedSend(P &ptr, const int dst,
11                                           const int tag,
12                                           const Comm &comm,
13                                           const int bufferSize) {
14     // Allocate the buffer for packing
15     char *buffer = MEL::MemAlloc<char>(bufferSize);
16
17     // Deep-copy into the buffer
18     Message<TransportBufferWrite, HASH_MAP> msg(buffer, bufferSize);
19     msg.packRootPtr(ptr);
20
21     // Send the buffer in one message. Uses Message<TransportSend>
```

```
22        // bufferSize represents an upperbound on how much data there
23        // is to transport,
24        // msg.getOffset() gives us how much data was actually packed into
25        // the buffer
26        MEL::Deep::Send(buffer, msg.getOffset(), dst, tag, comm);
27
28        // Clean up the buffer
29        MEL::MemFree(buffer);
30    }
31
32    template<typename P, typename HASH_MAP = MEL::Deep::PointerHashMap>
33    inline enable_if_pointer<P> BufferedRecv(P &ptr, const int src,
34                                              const int tag,
35                                              const Comm &comm) {
36        // Recieve the packed buffer in one message. Uses
37        // Message<TransportRecv>
38        int bufferSize;
39        char *buffer  = nullptr;
40        MEL::Deep::Recv(buffer, bufferSize, src, tag, comm);
41        // bufferSize on the receiving processes is equal to msg.getOffset()
42        // on the sending process
43
44        // Deep-copy out of the buffer
45        Message<TransportBufferRead, HASH_MAP> msg(buffer, bufferSize);
46        msg.packRootPtr(ptr);
47
48        // Clean up the buffer
49        MEL::MemFree(buffer);
50    }
```

The last parameter to buffered transport methods on sending processes is an integer value representing the byte size of the contiguous buffer to use for packing the deep structure. If this value is omitted an overloaded version of the function computes the upper-bound of the buffer size needed by calling `MEL::Deep::BufferSize` before forwarding its parameters to the main function overload.

Note that on the sending process for a buffered transport that `msg.getOffset()` is used as the length parameter when transporting the buffer (Listing 20, line 19) and not the `bufferSize` parameter. This means that if the sender blindly requests a large buffer because it does not know the size of the deep structure exactly, but only a part of the buffer is filled, only the used part of the buffer will be transported to the receiving processes. In the scenario where the buffer size given was not large enough to complete the deep copy, a run-time assertion occurs.

Hash-Map implementation

The Message object is templated on a hash-map type that exposes public-member functions of the form:

```
template<typename T> inline bool find(T* oldPtr, T* &ptr)
template<typename T> inline void insert(T* oldPtr, T* ptr)
```

This allows the user to provide an implementation of a hashing scheme optimized for pointers or to provide an adapter to a third-party hash-map implementation. One of the goals of MEL is to be portable and to not introduce external dependencies on the users code; because of this, our default hash-map implementation (Listing 21) is simply a wrapper around a std::unordered_map container between two void pointers.

Listing 21 **MEL implementation–default hash-map interface for MEL::Deep::Message.**

```
1   class PointerHashMap {
2   private:
3       // Hashmaps for storing pointers to types of any size
4       std::unordered_map<void*, void*> pointerMap;
5
6   public:
7       // Pointer hashmap public interface
8
9       // Returns true if oldPtr is found in the hash-map and sets ptr
10       // equal to the stored value
11      // Otherwise returns false and ptr is unaltered
12      template<typename T>
13      inline bool find(T* oldPtr, T* &ptr) {
14          // Is oldPtr already in the hashmap?
15          const auto it = pointerMap.find((void*) oldPtr);
16
17          if (it != pointerMap.end()) {
18              // If so set ptr equal to the value stored in the hashmap
19              ptr = (T*) it->second;
20              return true;
21          }
22          return false;
23      }
24
25      // Insert ptr into the hashmap using oldptr as the key
26      template<typename T>
27      inline void insert(T* oldPtr, T* ptr) {
28        pointerMap.insert(std::make_pair((void*) oldPtr, (void*) ptr));
29      }
30  };
```

BENCHMARKS

For benchmarking we used the Swansea branch of the HPC Wales compute cluster. Nodes contain two Intel Xeon E5-2670 processors for a total of 16 physical cores with 64GB's of RAM per-node, connected with Infiniband 40 Gbps networking. Benchmarks were run using Intel MPI 4.1 and compiling under Intel ICPC 13.0.1.

Case study: ray-tracing scene structure

To evaluate the performance of our algorithms relative to the equivalent hand coded MPI implementations and to other libraries that offer deep copy semantics such as Boost Serialization Library (*Cogswell, 2005*), we used the example of deep copying a large binary-tree structure between processes in the context of a distributed ray-tracer. A 3D scene (Fig. 3) is loaded on one process, consisting of triangular meshes, cameras, materials, and a bounding volume hierarchy to help accelerate ray-triangle intersection tests.

For each experiment, a scene was loaded containing increasing numbers of the classic Utah Teapot mesh. The scene structure was then communicated using the various algorithms and the performance measured by comparing the times spent between `MPI_Barrier`'s before and after the communication.

Broadcast–MPI vs. MEL

For this example, just 4 lines of code calling the transport API were added to the BVH TreeNode and Scene structs (see appendix Scene Object containing MEL Deep Copy Methods) to enable both buffered and non-buffered deep copy using our algorithm.

By comparison, the hand coded MPI non-buffered (see appendix Hand coded Non-Buffered Bcast of Scene Object) method took 34 lines of code, and 70 lines of code for the MPI buffered (see appendix Hand Coded Buffered Bcast of Scene Object) algorithm (not including comments, formatting, or trailing brackets), where pointers, allocations, and object construction had to be managed manually by the programmer. Also, these implementations only handled the case of `Bcast` operations, while the MEL version works transparently with all operations.

Despite its generic interface and minimal syntax, our algorithm performs almost identically with hand coded MPI implementations in fewer lines of code and a fraction of the code complexity. Relevant code for this example is given in appendix Experiment 1: Broadcasting a Large Tree Structure.

Figure 4A shows the resulting times from broadcasting increasingly larger scenes with each algorithm, between 256 nodes on HPC Wales. We can see that the buffered methods that only send a small constant number of messages between processes are faster than non-buffered methods despite the added overheads from packing and unpacking the data. The scalability of our algorithm with respect to the number of MPI processes involved in the communication is only bounded by the scalability of the transport method itself. In the case of a broadcast operation, Fig. 4B shows that varying the number

Figure 3 **Utah teapot mesh used for benchmarks.**

processes is of the same complexity as the underlying `MPI_Bcast` communication (logarithmic).

File Write/Read–MEL vs. Boost

When fault tolerance is a concern one method for recovering from a failed process is to periodically cache the current state of what is being worked on to disk so that in the event of a failure the data can be reloaded on a new process (potentially on a different node) and the work continued from the point at which it was last saved. When the data needed to store the state of a process is deep we incur the same problems that arise during deep copy. MEL implements file read and write operations for both buffered and non-buffered file access, utilizing the same user defined deep copy functions needed for the broadcast, send, and receive methods. For this experiment we also compared our performance to the Boost Serialization Library which is designed for saving and restoring structured data from file.

Figure 5 shows the results of using MEL to write/read a large tree structure to or from file. Unlike with MPI communications where MEL's buffered methods performed considerably faster than non-buffered variants due to the overheads from starting and ending network communications; with file access non-buffered reads perform almost identically to buffered methods. This is due to `std::fstream`'s use of an internal buffer to optimize file access, meaning that cost of starting and ending write/read operations is negligible compared to the cost of traversing the deep structure. While Boost Serialize also uses C++ streams their method of traversing the deep structure incurs significant overheads leading to poor and differing performance when reading and writing data. Finally, non-buffered writes perform slightly poorer then buffered writes due to file system having to allocate additional blocks as the file grows.

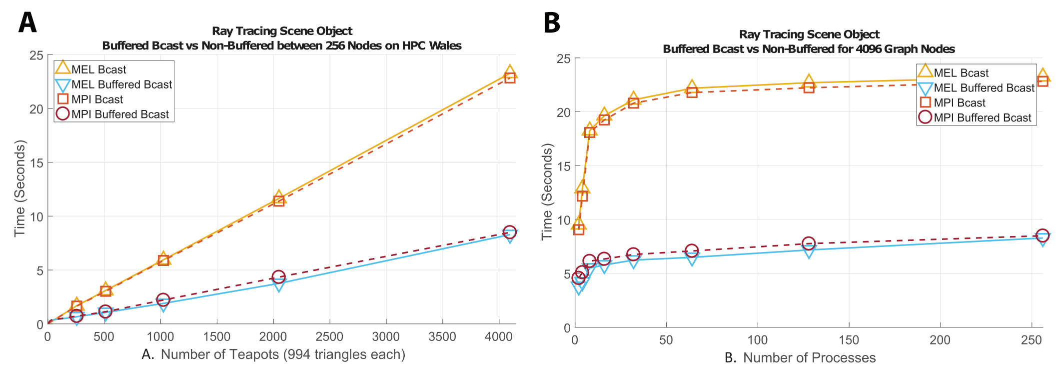

Figure 4 Time comparison of algorithms broadcasting large tree structures between processes within node and on separate nodes. MEL requires the addition of four simple lines of code which greatly accelerate programming time and vastly reduces the chance of user induced bugs.

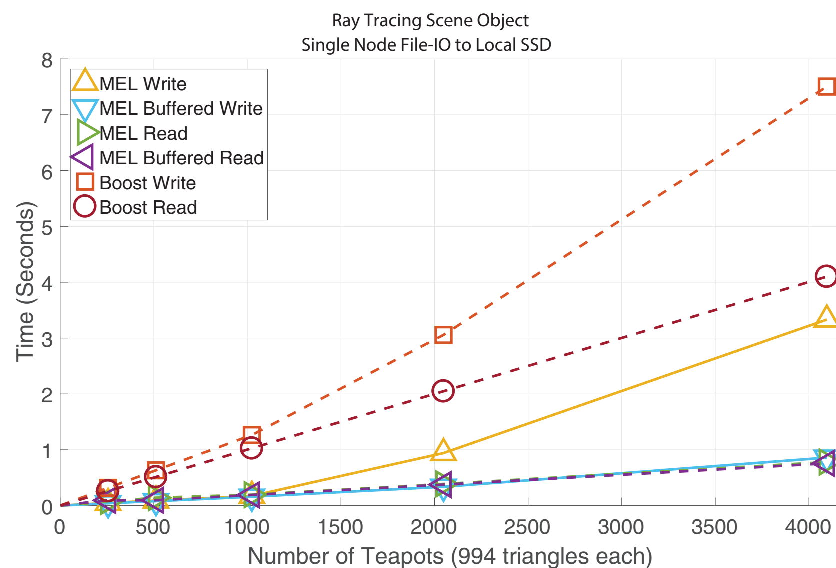

Figure 5 Time comparison of MEL to Boost Serialization Library for File Read/Write on a single node, to a within node Solid State Drive.

Case study: graphs with cycles

In the previous example the implementation of `TreeNode` was simplified by the observation that tree nodes were only pointed to from a single parent. However, in many applications multiple objects may share a common child. To show how MEL copes with structures containing pointers to shared dependents we used the example of communicating generic directed graph structures constructed in various connectivities (see Fig. 6A–6D). Relevant code for this example can be found in appendix Experiment 2: Communicating Generic Directed Graph structures.

Fully connected graphs

Figure 7 shows the results for communicating fully connected graph structures of increasing size in terms of broadcast (Fig. 7A) and writing a checkpoint to file (Fig. 7B).

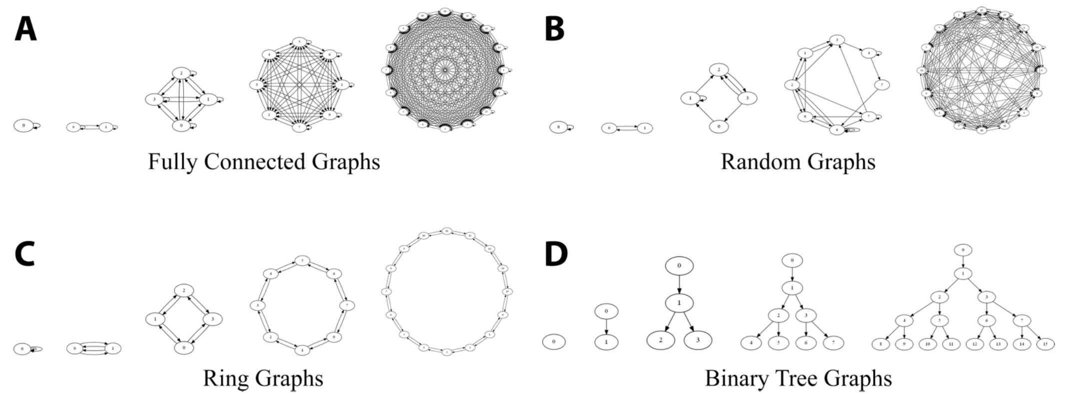

Figure 6 Graph connectivities for $\{2^0, 2^1, 2^2, 2^3, 2^4, \ldots\}$ nodes.

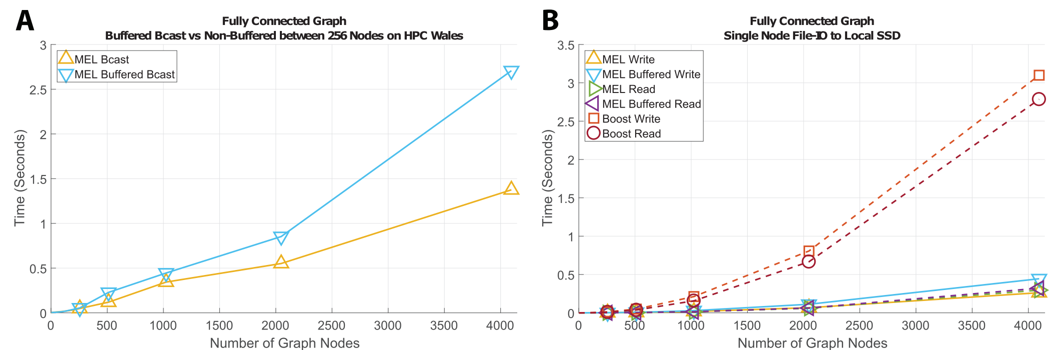

Figure 7 Time comparison for broadcast and file-IO operations on fully connected graph structures.

In this example, n independent graph nodes will be traversed, each containing a list of pointers to all n nodes; during deep copy the hash-map will be queried n^2 times and will grow to contain n entries. Compared to the previous broadcast example for the ray tracing case study (Section Broadcast–MPI vs. MEL) where buffered communication showed better performance, with fully connected graphs we see the opposite effect. Non-buffered communication is consistently faster when the number of shared dependents is high. Internally, shared pointers are tracked using a hash table to ensure that only distinct pointers are transported and duplicates linked correctly. Because of the overheads attached to insert and find operations on the hash table, when the number of shared dependents is high the overhead from sending separate communications for each object in the structure is small compared to that of accessing the hash table. This has the effect of making the overhead from buffering the structure into a contiguous array for transport a bottleneck for deep copy.

A similar trend is observed for file access, where non-buffered access is more efficient than buffered. In this example we also compare MEL to Boost Serialization library. Here shared pointer usage introduces significant overheads for Boost that our method avoids leading to significantly improved performance.

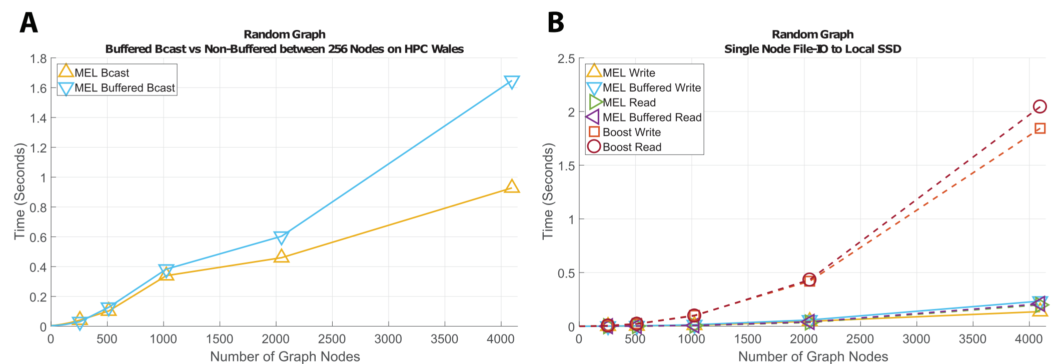

Figure 8 Time comparison for broadcast and file-IO operations on randomly connected graph structures.

Random graph

Next we look at graphs with random connectivities. Figure 8 shows the results of communicating randomly generated graphs of different sizes for broadcasting (Fig. 8A) and writing a checkpoint to file (Fig. 8B). With this example, n independent graph nodes will be traversed, each containing a list of pointers to a random number of nodes (at least one); during deep copy the hash-map will be queried between n and n^2 times and will grow to contain n entries. Again, we see that when the number of shared dependents within the structure is large non-buffered communication performs consistently better than for buffered. We also see slightly better performance than with the fully connected graphs, showing that time complexity scales linearly with the number of graph edges. For file access the same trends emerge, where our method performs considerably faster than Boost Serialization.

Ring graph

A ring graph can be modeled a doubly-linked list where the last element is connected back to the first element in the structure. For this example, n independent graph nodes will be traversed, each containing a list of two pointers to previous and next nodes; during deep copy the hash-map will be queried $2n$ times and will grow to contain n entries. Figure 9 shows the results of communicating large ring structures for broadcasting (Fig. 9A) and writing a checkpoint to file (Fig. 9B). Because the number of shared edges is small we initially see that buffered communication is faster than non-buffered as with Section Broadcast–MPI vs. MEL. As the number of graph nodes in the structure passes 2,400, the amount of time needed to buffer the structure becomes larger than the overhead associated with starting and stopping separate MPI communications making non-buffered method more efficient for larger structures. For file access, we still see that our methods perform consistently faster than Boost's even when the number of shared dependents is low.

Binary tree

Finally, we look at the example of constructing a binary tree shaped graph where there are no shared dependents. The generic container does not know this, and still must use

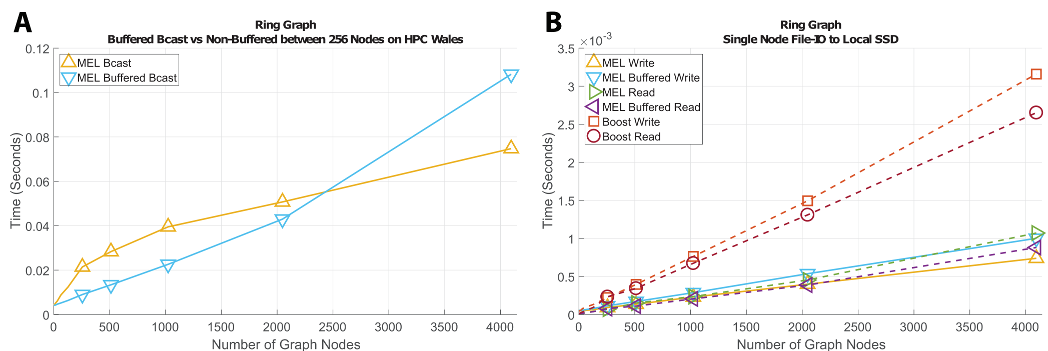

Figure 9 **Time comparison for broadcast and file-IO operations on ring graph structures.**

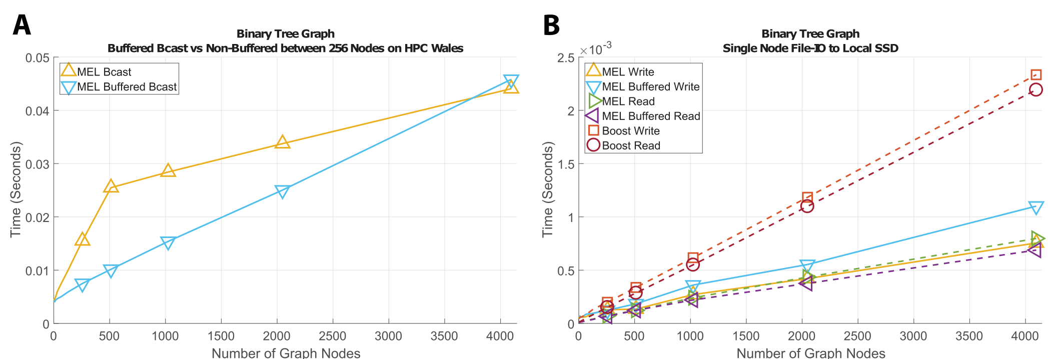

Figure 10 **Time comparison for broadcast and file-IO operations on binary tree graph structures.**

`Message::packSharedPtr` to transport child nodes, meaning it still incurs overheads of pointer lookup. In this example, n independent graph nodes will be traversed, each containing a list of one or two pointers to descending child nodes; during deep copy the hash-map will be queried n times and will grow to contain n entries. Figure 10 shows the results of communicating binary trees of different sizes in terms of broadcast (Fig. 10A) and writing a checkpoint to file (Fig. 10B). Similarly to communicating ring graphs, buffered network communication is significantly faster non-buffered methods until the structure becomes large enough that buffering becomes the main bottleneck.

For file access the opposite is true, with non-buffered file access being slightly faster than buffered. We attribute this to `std::fstream`'s use of internal buffering, which renders the overheads from our fully buffered method unnecessary in this use case.

CONCLUSIONS AND FUTURE WORK

In this paper we have presented our implementation of deep copy semantics that encapsulates both buffered and non-buffered methods for dealing with complex structured data in the context of MPI inter-process communication and file access. Users may choose shared versions for when data structures contain cycles or faster non-shared variants for when they do not. We have shown that a generic implementation of such semantics can achieve like for like performance with hand crafted implementations while

dramatically reducing code complexity and decreasing the chance for programmer error. We also demonstrate the method to be faster than utilizing Boost Serialization Library. MEL non-buffered methods provide a generic, low memory overhead, high performance (equal to hand crafted) solution to the deep copy problem.

In the future we intend to include the implementation of a non-blocking top-level interface for asynchronous deep copy, additional transport methods for communicating deep structured data to CUDA and OpenCL based accelerators, and a hash-map implementation highly optimized for pointers.

The algorithms discussed in this paper are implemented as part of the MEL, which is currently in development with the goal of creating a light weight, header only C++ wrapper around the C-style functions exposed by the MPI-3 standard, with backwards compatibility for systems where only MPI-2 is available. We plan to keep MEL in active development and hope that the research community will join us as we continue to grow the features and capabilities encompassed within the project.

MEL is Open-Source and available on Github under the MIT license at: https://github.com/CS-Swansea/MEL.

APPENDICES

Experiment 1: broadcasting a large tree structure

Full code for this example is available at https://github.com/CS-Swansea/MEL/ under `example–code/RayTracingDeepCopy.cpp`.

Listing 22 Deep copy of ray tracing scene object.

```
1  //------------------------------------------------------------//
2  // Example Usage:                                             //
3  // mpirun -n [number of processes] ./RayTracingDeepCopy [mesh path]
                                                  [method index] //
4  // mpirun -n 8 ./RayTracingDeepCopy "Teapot.obj" 0            //
5  //------------------------------------------------------------//
6  int main(int argc, char *argv[]) {
7      MEL::Init(argc, argv); // Setup
8  

9      // Who are we?
10     MEL::Comm comm = MEL::Comm::WORLD;
11     const int rank = MEL::CommRank(comm),
12               size = MEL::CommSize(comm);
13 
14     // Check param count
15     if (argc != 3) {
16         if (rank == 0) std::cout << "Wrong number of parameters..."<<
                                                             std::endl;
```

```
17          MEL::Exit(-1);
18          // ^^ Equivalent of calling MPI_Finalize() followed by
19          // std::exit(-1)
20      }
21
22      // Which model should we load and which algorithm should we use?
23      const std::string meshPath = std::string(argv[1]);
24      const int         method   = std::stoi(argv[2]);
25
26      // Load the scene on the root process
27      Scene *scene = nullptr;
28      if (rank == 0) {
29          std::cout << "Loading scene..." << std::endl;
30          scene = loadScene(meshPath);
31      }
32
33      MEL::Barrier(comm);
34      auto startTime = MEL::Wtime();// Start the clock!
35
36      // Broadcast the Scene structure with the selected method
37      switch (method) {
38      case 0:
39          MEL::Deep::Bcast(scene, 0, comm);  // Call MEL::Deep method.
40          break;
41      case 1:
42          MEL::Deep::BufferedBcast(scene, 0, comm);
                                                 // Call MEL::Deep method.
43          break;
44      case 2:
45          MPI_NonBufferedBcast_Scene(scene, 0, (MPI_Comm) comm);
                                                 // Hand written below
46          break;
47      case 3:
48          MPI_BufferedBcast_Scene(scene, 0, (MPI_Comm) comm);
                                                 // Hand written below
49          break;
50      default:
51          if (rank == 0) std::cout << "Invalid method index..." << std::
                                                                       endl;
52          MEL::Exit(-1);
53      }
54
55      MEL::Barrier(comm);
```

```
56      auto endTime = MEL::Wtime(); // Stop the clock!
57
58      if (rank == 0) {
59          std::cout << "Broadcast Scene in " << (endTime - startTime)
60                      << " seconds..." << std::endl;
61      }
62
63      // All processes now have a Scene pointer that points to an
64      // equivalent data-structure
65
66      //-------------------------------------------------------//
67      // Now we can do some ray-tracing using the scene object!    //
68      //-------------------------------------------------------//
69
70      // Clean up
71      MEL::MemDestruct(scene);
72      // ^^ Equivalent to explicitly calling the destructor followed by
73      // MPI_Free_mem.
74      // scene->~Scene();
75      // MPI_Free_mem(scene);
76
77      MEL::Finalize(); // Tear down
78      return 0;
79  }
```

Scene object containing MEL deep copy methods

Listing 23 Ray tracing scene object.

```
1   //---------------------------------------------//
2   // Structure representing a node in the BVH Tree //
3   //---------------------------------------------//
4   struct TreeNode {
5       int         startElem,  endElem;      // Start and End indices
                                                into vector of triangles
6       Vec         v0, v1;                   // Vec is non-deep struct
7       TreeNode    *leftChild, *rightChild; // TreeNode is deep struct
8
9       TreeNode() : TreeNode(0, 0) {}
10      TreeNode(const int _s, const int _e) : startElem(_s), endElem(_e),
11                                             leftChild(nullptr),
12                                             rightChild(nullptr),
```

```
13                                           v0{ INF, INF, INF },
14                                           v1{ -INF, -INF, -INF } {}
15
16      // Ensure TreeNode can't be used incorrectly
17      TreeNode(const TreeNode &old)                   = delete;
                                             // Remove CopyConstructor
18      inline TreeNode& operator=(const TreeNode &old) = delete;
                                             // Remove CopyAssignment
19      TreeNode(TreeNode &&old)                        = delete;
                                             // Remove MoveConstructor
20      inline TreeNode& operator=(TreeNode &&old)      = delete;
                                             // Remove MoveAssignment
21
22      ~TreeNode() {
23          MEL::MemDestruct( leftChild);
24          MEL::MemDestruct(rightChild);
25      }
26
27      // Implementation of Ray-TreeNode (Ray-AABB) intersection
28      // omitted for this example
29      bool intersect(const Ray &rayInv, double &tmin, const
                                                 double dist) const;
30
31      template<typename MSG>
32      inline void DeepCopy(MSG &msg) {
33          msg.packPtr( leftChild);
34          msg.packPtr(rightChild);
35      }
36  };
37
38  //---------------------------------------------------//
39  // Structure representing a scene object to be rendered //
40  //---------------------------------------------------//
41  struct Scene {
42      Camera                  camera;     // Camera is non-deep struct
43      std::vector<Material> materials; // Material is non-deep struct
44      std::vector<Triangle> mesh;      // Triangle is non-deep struct
45      TreeNode             *rootNode;  // TreeNode is deep struct
46
47      Scene() : rootNode(nullptr) {}
48
```

```
49      // Ensure Scene can't be used incorrectly
50      Scene(const Scene &old)                  = delete; // Remove
                                                     CopyConstructor
51      inline Scene& operator=(const Scene &old) = delete; // Remove
                                                     CopyAssignment
52
53      // Move Constructor
54      Scene(Scene &&old) : mesh(std::move(old.mesh)),
55                           materials (std::move(old.materials)),
56                           camera(old.camera),rootNode(old.rootNode){
57          old.mesh.clear();
58          old.materials.clear();
59          old.rootNode = nullptr;
60      }
61
62      // Move Assignment Operator
63      inline Scene& operator=(Scene &&old) {
64          mesh      = std::move(old.mesh);
65          materials = std::move(old.materials);
66          rootNode  = old.rootNode;
67          camera    = old.camera;
68          old.mesh.clear();
69          old.materials.clear();
70          old.rootNode = nullptr;
71          return *this;
72      }
73
74      ~Scene() {
75          MEL::MemDestruct(rootNode);
76      }
77
78      // Implementation of Ray-Scene intersection omitted for this example
79      bool intersect(const Ray &ray, Intersection &isect) const;
80
81      template<typename MSG>
82      inline void DeepCopy(MSG &msg) {
83          msg & mesh & materials;
84          msg.packPtr(rootNode);
85      }
86  };
```

Hand coded non-buffered Bcast of scene object

Listing 24 Hand coded non-buffered Bcast of ray tracing scene object.

```
1  inline void MPI_NonBufferedBcast_Scene(Scene *&scene, const int
       root, const MPI_Comm comm) {
2      int rank;
3      MPI_Comm_rank(comm, &rank);
4
5      // Receiving nodes allocate space for scene
6      if (rank != root) {
7          MPI_Alloc_mem(sizeof(Scene), MPI_INFO_NULL, &scene);
8          new (scene) Scene();
9      }
10
11     // Bcast the camera struct
12     MPI_Bcast(&(scene->camera), sizeof(Camera), MPI_CHAR, root,
           comm);
13
14     // Bcast the vector sizes
15     int sizes[2];
16     if (rank == root) {
17         sizes[0] = (int) scene->mesh.size();
18         sizes[1] = (int) scene->materials.size();
19     }
20     MPI_Bcast(sizes, 2, MPI_INT, root, comm);
21
22     // 'Allocate' space for vectors
23     if (rank != root) {
24         scene->mesh.resize(sizes[0]);
25         scene->materials.resize(sizes[1]);
26     }
27
28     // Bcast the vectors
29     MPI_Bcast(&(scene->mesh[0]), sizeof(Triangle) * sizes[0],
           MPI_CHAR, root, comm);
30     MPI_Bcast(&(scene->materials[0]), sizeof(Material) * sizes[1],
           MPI_CHAR, root, comm);
31
32     // Receiving nodes allocate space for rootNode
33     if (rank != root) {
34         MPI_Alloc_mem(sizeof(TreeNode), MPI_INFO_NULL,
               &(scene->rootNode));
```

```
35            new (scene->rootNode) TreeNode();
36        }
37
38        // While the stack is not empty there is work to be done
39        std::stack<TreeNode*> treeStack;
40        treeStack.push(scene->rootNode);
41        while (!treeStack.empty()) {
42            // Get the current node to traverse
43            TreeNode *currentNode = treeStack.top();
44            treeStack.pop();
45
46            // Bcast the current node's values
47            MPI_Bcast((currentNode), sizeof(TreeNode), MPI_CHAR,
                                                      root, comm);
48
49            // Do we need to send/receive children?
50            bool hasChildren = (currentNode->leftChild != nullptr);
51
52            if (hasChildren) {
53                // Allocate space for child nodes on receiving process
54                if (rank != root) {
55                    MPI_Alloc_mem(sizeof(TreeNode), MPI_INFO_NULL,
                                         &(currentNode->leftChild));
56                    MPI_Alloc_mem(sizeof(TreeNode), MPI_INFO_NULL,
                                        &(currentNode->rightChild));
57                    new (currentNode->leftChild) TreeNode();
58                    new (currentNode->rightChild) TreeNode();
59                }
60
61                // Push children onto the stack so they get processed
62                treeStack.push(currentNode->leftChild);
63                treeStack.push(currentNode->rightChild);
64            }
65        }
66    }
```

Hand coded buffered Bcast of scene object

Listing 25 **Hand coded buffered Bcast of ray tracing scene object.**

```
1  inline void MPI_BufferedBcast_Scene(Scene *&scene, const int root,
                                              const MPI_Comm comm) {
2      int rank;
```

```
3       MPI_Comm_rank(comm, &rank);
4
5       // Receiving nodes allocate space for scene
6       if (rank != root) {
7           MPI_Alloc_mem(sizeof(Scene), MPI_INFO_NULL, &scene);
8           new (scene) Scene();
9       }
10
11      // Calculate the byte size of the tree on root process
12      int packed_size = 0;
13      if (rank == root) {
14          packed_size += sizeof(Camera);
15          packed_size += sizeof(int) + ((int) scene->mesh.size()
                                              * sizeof(Triangle));
16          packed_size += sizeof(int) + ((int) scene->materials.size()
                                              * sizeof(Material));
17
18          // While the stack is not empty there is work to be done
19          std::stack<TreeNode*> treeStack;
20          treeStack.push(scene->rootNode);
21          while (!treeStack.empty()) {
22              // Get the current node to traverse
23              TreeNode *currentNode = treeStack.top();
24              treeStack.pop();
25
26              packed_size += sizeof(TreeNode);
27
28              // Do we need to send children?
29              bool hasChildren = (currentNode->leftChild != nullptr);
30
31              if (hasChildren) {
32                  // Push children onto the stack so they get processed
33                  treeStack.push(currentNode->leftChild);
34                  treeStack.push(currentNode->rightChild);
35              }
36          }
37       }
38
39       // Share the buffer size to all processes
40       MPI_Bcast(&packed_size, 1, MPI_INT, root, comm);
41
42       // Allocate the buffer
43       int position = 0;
```

```
44      char *buffer;
45      MPI_Alloc_mem(packed_size, MPI_INFO_NULL, &(buffer));
46
47      // If root then we pack the structure into the buffer
48      if (rank == root) {
49          // Pack the camera struct
50          MPI_Pack(&(scene->camera), sizeof(Camera), MPI_CHAR,
                                buffer, packed_size, &position, comm);
51
52          int mesh_size      = scene->mesh.size(),
53              materials_size = scene->materials.size();
54
55          // Pack the mesh vector
56          MPI_Pack(&(mesh_size), 1, MPI_INT, buffer, packed_size,
                                                        &position, comm);
57          MPI_Pack(&(scene->mesh[0]), mesh_size * sizeof(Triangle),
58                   MPI_CHAR, buffer, packed_size, &position, comm);
59
60          // Pack the materials vector
61          MPI_Pack(&(materials_size), 1, MPI_INT, buffer,
                                    packed_size, &position, comm);
62          MPI_Pack(&(scene->materials[0]), materials_size
                                                  * sizeof(Material),
63                   MPI_CHAR, buffer, packed_size, &position, comm);
64
65          // While the stack is not empty there is work to be done
66          std::stack<TreeNode*> treeStack;
67          treeStack.push(scene->rootNode);
68          while (!treeStack.empty()) {
69              // Get the current node to traverse
70              TreeNode *currentNode = treeStack.top();
71              treeStack.pop();
72
73              // Pack the current node
74              MPI_Pack(currentNode, sizeof(TreeNode),
75                       MPI_CHAR, buffer, packed_size, &position, comm);
76
77              // Do we need to send children?
78              bool hasChildren = (currentNode->leftChild != nullptr);
79
80              if (hasChildren) {
81                  // Push children onto the stack so they get processed
82                  treeStack.push(currentNode->leftChild);
```

```
83                 treeStack.push(currentNode->rightChild);
84             }
85         }
86 
87         // Send the buffer
88         MPI_Bcast(buffer, packed_size, MPI_CHAR, root, comm);
89     }
90 
91     // If not root then we unpack the structure from the buffer
92     else {
93         // Receive the packed buffer
94         MPI_Bcast(buffer, packed_size, MPI_CHAR, root, comm);
95 
96         MPI_Alloc_mem(sizeof(TreeNode), MPI_INFO_NULL,
                                               &(scene->rootNode));
97         new (scene->rootNode) TreeNode();
98 
99         // Unpack the camera struct
100        int mesh_size, materials_size;
101        MPI_Unpack(buffer, packed_size, &position,
                                                  &(scene->camera),
102                    sizeof(Camera), MPI_CHAR, comm);
103 
104        // Unpack mesh vector
105        MPI_Unpack(buffer, packed_size, &position, &(mesh_size),
                                                  1, MPI_INT, comm);
106        scene->mesh.resize(mesh_size);
107        MPI_Unpack(buffer, packed_size, &position, &(scene->mesh
                                                              [0]),
108                    mesh_size * sizeof(Triangle), MPI_CHAR, comm);
109 
110        // Unpack materials vector
111        MPI_Unpack(buffer, packed_size, &position,
                              &(materials_size), 1, MPI_INT, comm);
112        scene->materials.resize(materials_size);
113        MPI_Unpack(buffer, packed_size, &position,
                                          &(scene->materials[0]),
114                  materials_size*sizeof(Material),MPI_CHAR,comm);
115 
116        // While the stack is not empty there is work to be done
117        std::stack<TreeNode*> treeStack;
118        treeStack.push(scene->rootNode);
119        while (!treeStack.empty()) {
```

```
120                // Get the current node to traverse
121                TreeNode *currentNode = treeStack.top();
122                treeStack.pop();
123
124                // Unpack the current node
125                MPI_Unpack(buffer, packed_size, &position, currentNode,
126                           sizeof(TreeNode), MPI_CHAR, comm);
127
128                // Do we need to receive children?
129                bool hasChildren = (currentNode->leftChild != nullptr);
130
131                if (hasChildren) {
132                    // Allocate space for child nodes on receiving process
133                    MPI_Alloc_mem(sizeof(TreeNode), MPI_INFO_NULL,
                                     &(currentNode->leftChild));
134                    MPI_Alloc_mem(sizeof(TreeNode), MPI_INFO_NULL,
                                    &(currentNode->rightChild));
135                    new (currentNode->leftChild) TreeNode();
136                    new (currentNode->rightChild) TreeNode();
137
138                    // Push children onto the stack so they get processed
139                    treeStack.push(currentNode->leftChild);
140                    treeStack.push(currentNode->rightChild);
141                }
142            }
143        }
144
145        // Clean up
146        MPI_Free_mem(buffer);
147    }
```

Experiment 2: communicating generic directed graph structures

Listing 26 Functions for constructing directed graphs in different shapes.

```
1  //------------------------------------------------------------//
2  // Example Usage:                                             //
3  // mpirun -n [num of procs] ./GraphCycles [graph nodes: 0 <= n]
                                       [graph type: 0 <= t <= 3] //
4  // mpirun -n 8 ./GraphCycles 11 0                             //
5  //------------------------------------------------------------//
6  int main(int argc, char *argv[]) {
7      MEL::Init(argc, argv);
```

```
8
9     MEL::Comm comm = MEL::Comm::WORLD;
10    const int rank = MEL::CommRank(comm),
11              size = MEL::CommSize(comm);
12
13    if (argc != 3) {
14        if (rank == 0)
15            std::cout << "Wrong number of parameters..." << std::endl;
16        MEL::Exit(-1);
17    }
18
19    const int numNodes  = 1 << std::stoi(argv[1]), // 2^n nodes
20              graphType =      std::stoi(argv[2]);
21
22    DiGraphNode<int> *graph = nullptr;
23    if (rank == 0) {
24        switch (graphType) {
25        case 0:
26            graph = MakeBTreeGraph(numNodes);
27            break;
28        case 1:
29            graph = MakeRingGraph(numNodes);
30            break;
31        case 2:
32            graph = MakeRandomGraph(numNodes);
33            break;
34        case 3:
35            graph = MakeFullyConnectedGraph(numNodes);
36            break;
37        }
38    }
39
40    MEL::Barrier(comm);
41    auto startTime = MEL::Wtime(); // Start the clock!
42
43    // Deep copy the graph to all nodes
44    MEL::Deep::Bcast(graph, 0, comm);
45
46    MEL::Barrier(comm);
47    auto endTime = MEL::Wtime(); // Stop the clock!
48
49    if (rank == 0) {
50        std::cout << "Broadcast Graph in " << (endTime - startTime)
```

```
51                    << " seconds..." << std::endl;
52      }
53
54      // File name for output
55      std::stringstream sstr;
56      sstr << "rank=" << rank << " type=" << graphType << " nodes="
                                              << numNodes << ".graph";
57
58      // Save the output to disk from each node
59      std::ofstream graphFile(sstr.str(), std::ios::out |
                                                      std::ios::binary);
60      if (graphFile.is_open()) {
61          MEL::Deep::FileWrite(graph, graphFile);
62          graphFile.close();
63      }
64
65      DestructGraph(graph);
66
67      MEL::Finalize();
68      return 0;
69  }
```

Factory functions for building directed graphs in different shaped structures

Listing 27 Functions for constructing directed graphs in different shapes.

```
1   inline DiGraphNode<int>* MakeBTreeGraph(const int numNodes) {
2       /// BTree Graph
3       std::vector<DiGraphNode<int>*> nodes(numNodes);
4       for (int i = 0; i < numNodes; ++i) {
5           nodes[i] = MEL::MemConstruct<DiGraphNode<int>>(i);
6       }
7
8       if (numNodes > 1) nodes[0]->edges.push_back(nodes[1]);
9
10      for (int  i = 1; i < numNodes; ++i) {
11          const int j = ((i - 1) * 2) + 2;
12          nodes[i]->edges.reserve(2);
13          if (j < numNodes)        nodes[i]->edges.push_back(nodes[j]);
14          if ((j + 1) < numNodes)   nodes[i]->edges.push_back(nodes
                                                             [j + 1]);
15      }
```

```
16      return nodes[0];
17  }
18
19  inline DiGraphNode<int>* MakeRingGraph(const int numNodes) {
20      /// Ring Graph
21      std::vector<DiGraphNode<int>*> nodes(numNodes);
22      for (int i = 0; i < numNodes; ++i) {
23          nodes[i] = MEL::MemConstruct<DiGraphNode<int>>(i);
24      }
25
26      for (int i = 0; i < numNodes; ++i) {
27          nodes[i]->edges.reserve(2);
28          nodes[i]->edges.push_back(nodes[(i + 1) % numNodes]);
29          nodes[i]->edges.push_back(nodes[(i == 0) ? (numNodes - 1) :
                                                               (i - 1)]);
30      }
31      return nodes[0];
32  }
33
34  inline DiGraphNode<int>* MakeRandomGraph(const int numNodes) {
35      srand(1234567);
36
37      /// Random Graph
38      std::vector<DiGraphNode<int>*> nodes(numNodes);
39      for (int i = 0; i < numNodes; ++i) {
40          nodes[i] = MEL::MemConstruct<DiGraphNode<int>>(i);
41      }
42
43      for (int i = 0; i < numNodes; ++i) {
44          const int numEdges = rand() % numNodes;
45          nodes[i]->edges.reserve(numEdges);
46          nodes[i]->edges.push_back(nodes[(i + 1) % numNodes]);
47          for (int j = 1; j < numEdges; ++j) {
48              nodes[i]->edges.push_back(nodes[rand() % numNodes]);
49          }
50      }
51      return nodes[0];
52  }
53
54  inline DiGraphNode<int>* MakeFullyConnectedGraph(const int numNodes) {
55      /// Fully Connected Graph
56      std::vector<DiGraphNode<int>*> nodes(numNodes);
57      for (int i = 0; i < numNodes; ++i) {
```

```
58            nodes[i] = MEL::MemConstruct<DiGraphNode<int>>(i);
59        }
60
61        for (int i = 0; i < numNodes; ++i) {
62            nodes[i]->edges.reserve(numNodes);
63            for (int j = 0; j < numNodes; ++j) {
64                nodes[i]->edges.push_back(nodes[j]);
65            }
66        }
67
68        return nodes[0];
69    }
```

Generic implementation of directed graph container

Listing 28 Generic implementation of directed graph container for deep copy.

```
1   template<typename T>
2   struct DiGraphNode {
3       T value;
4       std::vector<DiGraphNode<T>*> edges;
5
6       DiGraphNode() {};
7       explicit DiGraphNode(const T &_value) : value(_value) {};
8
9       template<typename MSG>
10      inline void DeepCopy(MSG &msg) {
11          msg & edges;
12          for (auto &e : edges) msg.packSharedPtr(e);
13      }
14  };
15
16  inline void VisitGraph(DiGraphNode<int> *&root,
17                         std::function<void(DiGraphNode<int>
                                                    *&node)> func) {
18
19      std::unordered_set<DiGraphNode<int>*> pointerMap;
20      std::stack<DiGraphNode<int>*> stack;
21
22      stack.push(root);
23      while (!stack.empty()) {
24          DiGraphNode<int> *node = stack.top();
25          stack.pop();
```

```
26
27          // If node has not been visited
28          if (pointerMap.find(node) == pointerMap.end()) {
29              pointerMap.insert(node);
30              for (auto e : node->edges) stack.push(e);
31              func(node);
32          }
33      }
34  }
35
36  inline void DestructGraph(DiGraphNode<int> *&root) {
37      VisitGraph(root, [](DiGraphNode<int> *&node) -> void {
38          MEL::MemDestruct(node);
39      });
40  }
```

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

Joss Whittle is funded by an EPSRC PhD studentship. The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Grant Disclosures

The following grant information was disclosed by the authors:
EPSRC PhD studentship.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Joss Whittle conceived and designed the experiments, performed the experiments, analyzed the data, contributed reagents/materials/analysis tools, wrote the paper, prepared figures and/or tables, performed the computation work.
- Rita Borgo conceived and designed the experiments, analyzed the data, contributed reagents/materials/analysis tools, wrote the paper, reviewed drafts of the paper.
- Mark W. Jones conceived and designed the experiments, analyzed the data, contributed reagents/materials/analysis tools, wrote the paper, reviewed drafts of the paper.

Data Deposition

The following information was supplied regarding data availability:

Source code available at: https://github.com/CS-Swansea/MEL.

REFERENCES

Beyer J, Oehmke D, Sandoval J. 2014. Transferring userdefined types in OpenACC. In: *Proceedings Cray User Group (CUG'14)*. Lugano: Cray User Group.

Boost-Community. 2015. Boost C++ libraries. Version 1.62. *Available at http://boost.org* (accessed 7 November 2016).

Bouteiller A. 2015. Fault-tolerant MPI. In: Herault T, Robert Y, eds. *Fault-Tolerance Techniques for High-Performance Computing, Chapter 3*. Heidelberg, New York, Dordrecht, London: Springer Publishing Company, Incorporated, 145–228.

Cogswell J. 2005. Adding an easy file save and file load mechanism to your C++ program. *InformIT. Available at http://www.boost.org/doc/libs/release/libs/serialization/* (accessed 7 November 2016).

Fagg GE, Bukovsky A, Dongarra JJ. 2001. Harness and fault tolerant MPI. *Parallel Computing* **27(11):**1479–1495 DOI 10.1016/S0167-8191(01)00100-4.

Fagg GE, Dongarra JJ. 2004. Building and using a fault-tolerant MPI implementation. *International Journal of High Performance Computing Applications* **18(3):**353–361 DOI 10.1177/1094342004046052.

Friedley A, Hoefler T, Bronevetsky G, Lumsdaine A. 2013. Ownership passing: efficient distributed memory programming on multi-core systems. In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Shenzen, China.* New York: ACM, 177–186.

GNA. 2008. Autoserial library. *Available at http://home.gna.org/autoserial/mpi.html*.

Goujon DS, Michel M, Peeters J, Devaney JE. 1998. Automap and autolink tools for communicating complex and dynamic data-structures using MPI. In: Panda DK, Stunkel CB, eds. *Network-Based Parallel Computing Communication, Architecture, and Applications CANPC '98*. Berlin, Heidelberg: Springer, 98–109.

Gropp W, Lusk E. 2004. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications* **18(3):**363–372 DOI 10.1177/1094342004046045.

Grundmann T, Ritt M, Rosenstiel W. 2000. TPO++: an object-oriented message-passing library in C++. In: *Proceedings of the 2000 international conference on parallel processing*. Piscataway: IEEE, 43–50.

Herault T, Robert Y. 2015. *Fault-Tolerance Techniques for High-Performance Computing*. First edition. Switzerland: Springer International Publishing, 3–85.

Hoefler T, Snir M. 2011. Writing parallel libraries with MPI–common practice. In: *Proceedings of the 18th MPI Users' Group Meeting*. Vol. 6960. Berlin, Heidelberg: Springer, 345–355.

Huang C, Zheng G, Kalé L, Kumar S. 2006. Performance evaluation of adaptive MPI. In: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06*. New York: ACM, 12–21.

Kale LV, Krishnan S. 1993. CHARM++: a portable concurrent object oriented system based on C++. In: *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '93*. New York: ACM, 91–108.

Laguna I, Richards DF, Gamblin T, Schulz M, de Supinski BR. 2014. Evaluating user-level fault tolerance for MPI applications. In: *Proceedings of the 21st European MPI Users' Group Meeting, EuroMPI/ASIA '14*. New York: ACM, 57–62.

Lee EA. 2006. The problem with threads. *Computer* **39(5):**33–42 DOI 10.1109/MC.2006.180.

Lee L-Q, Lumsdaine A. 2003. The generic message passing framework. In: *Parallel and Distributed Processing Symposium, 2003. Proceedings. International.* Piscataway: IEEE.

McCandless BC, Squyres JM, Lumsdaine A. 1996. Object oriented MPI (OOMPI): a class library for the message passing interface. In: *MPI Developer's Conference, 1996.* Piscataway: IEEE, 87–94.

Message Passing Interface Forum. 2014. MPI: a message-passing interface standard version 3.1. Technical report. Stuttgart, DE: High Performance Computing Center Stuttgart (HLRS).

Miller P. 2015. Productive parallel programming with CHARM++. In: *Proceedings of the Symposium on High Performance Computing HPC '15.* San Diego: Society for Computer Simulation International, 241–242.

Renault É. 2007. Extended MPICC to generate MPI derived datatypes from C datatypes automatically. In: Cappello F, Herault T, Dongarra J, eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting.* Berlin, Heidelberg: Springer, 307–314.

Tansey W, Tilevich E. 2008. Efficient automated marshaling of C++ data structures for MPI applications. In: *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.* Piscataway: IEEE, 1–12.

Vishnu A, Dam HV, de Jong W, Balaji P, Song S. 2010. Fault-tolerant communication runtime support for data-centric programming models. In: *2010 International Conference on High Performance Computing, HiPC 2010, Dona Paula, Goa, India, December 19–22, 2010.* Piscataway: IEEE, 1–9.