Efficient Automated Marshaling of C++ Data Structures for MPI Applications

Wesley Tansey and Eli Tilevich
Center for High-End Computing Systems
Department of Computer Science
Virginia Tech
Email: {tansey,tilevich}@cs.vt.edu

Abstract

We present an automated approach for marshaling C++ data structures in High Performance Computing (HPC) applications. Our approach utilizes a graphical editor through which the user can express a subset of an object's state to be marshaled and sent across a network. Our tool, MPI Serializer*, then automatically generates efficient marshaling and unmarshaling code for use with the Message Passing Interface (MPI), the predominant communication middleware for HPC systems.*

Our approach provides a more comprehensive level of support for C++ language features than the existing state of the art, and does so in full compliance with the C++ Language Standard. Specifically, we can marshal effectively and efficiently non-trivial language constructs such as polymorphic pointers, dynamically allocated arrays, non-public member fields, inherited members, and STL container classes. Additionally, our marshaling approach is also applicable to third party libraries, as it does not require any modifications to the existing C++ source code.

We validate our approach through two case studies of applying our tool to automatically generate the marshaling functionality of two realistic HPC applications. The case studies demonstrate that the automatically generated code matches the performance of typical hand-written implementations and surpasses current state-of-the-art C++ marshaling libraries, in some cases by more than an order of magnitude. The results of our case studies indicate that our approach can be beneficial for both the initial construction of HPC applications as well as for the refactoring of sequential applications for parallel execution.

1 Introduction

Parallel programming for High Performance Computing (HPC) remains one of the most challenging application domains in modern software engineering. At the same time, the programmers who create applications in this complex domain are often non-experts in computing. That is, the primary users of parallel processing tend to be lab scientists and engineers, many of whom have limited experience with parallel programming. The dichotomy between the intrinsic difficulty of this application domain and the non-expert status of its many application developers is a major bottleneck in the scientific discovery process. This provides a strong motivation for research efforts aimed at making parallel processing more intuitive and less error-prone.

Recent studies [16, 15] have indicated that Shared Memory Multiprocessor (SMMP) programming models can lead to higher programmer productivity compared to Distributed Memory Multiple Processor (DMMP) models (i.e., compute clusters). A key difference between these two programming models is the need to pass program state between multiple processors across the network in DMMP. The programming technique for accomplishing this goal is called *marshaling*, also known as *serialization*. The technique entails representing the internal state of a program in an external format. Data in an external format can then be passed across the network. The reverse of this technique is called *unmarshaling*, also known as *deserialization*. These techniques are difficult and error-prone to implement by hand without library or automated tool support and likely contribute to the discrepancies in programmer productivity between the SMMP and DMMP models.

Nevertheless, the relatively lower cost of DMMP hardware makes it more common and accessible to a wider group of parallel programmers [35]. The primary middleware facility for parallel programming on a compute cluster is the Message Passing Interface (MPI) [24]. This middleware facility is ubiquitous, being available on virtually every parallel cluster-based computing system and has a standardized application programming interface (API). In addition, MPI aims at providing a uniform API for multiple programming languages such as Fortran, C, and C++. Consequently, this API has to be tailored to the lowest common denominator, providing only low-level marshaling fa-

cilities. As a result, C++ programmers are forced to provide tedious and error-prone code to interface their C++ application code with the MPI marshaling facilities. Therefore, efforts to map higher-level features of C++ to lower-level marshaling facilities of MPI (either via libraries or automatic tools) have the potential to significantly improve programmer productivity.

C++ is one of the foremost higher-level programming language that provides support for object oriented, procedural, and generic programming models. While large portions of engineering and scientific code have been written in Fortran, C++ is often a preferred language for programming complex, performance-conscious models due to its availability, portability, efficiency, and generality [36, 4, 23]. As C++ is being used more and more in scientific and engineering computing, the importance of alleviating the burden of implementing marshaling logic in C++ for MPI grows due to the number of non-expert users in these domains. However, marshaling C++ data structures is non-trivial due to the inherent complexity of the language.

This paper presents a novel approach to overcoming the challenges of mapping higher-level features of C++ to the lower-level marshaling facilities of MPI. Our approach eliminates the need for the programmer to write any marshaling code by hand. Instead, our tool uses compiler technology to parse C++ code and to obtain an accurate description of a program's data structures. It then displays the resulting description in a graphical editor that enables the programmer to simply "check-off" the subset of a class's state to be marshaled and unmarshaled. Our tool then automatically generates efficient MPI marshaling code, whose performance is comparable to that of hand written code.

Our approach provides a more comprehensive level of support for C++ language features than the existing state of the art. Specifically, we can marshal effectively and efficiently non-trivial language constructs such as polymorphic pointers, dynamically allocated arrays, non-public member fields, inherited members, and STL container classes. A distinguishing characteristic of our approach is that it provides support for a significant subset of the C++ language without requiring any modifications to the existing C++ source code. This makes our approach equally applicable to third party libraries as well as to programmer-written code. Our tool generates code that uses standard MPI calls and requires no additional libraries, thereby simplifying deployment.

The rest of this paper is structured as follows. Section 2 overviews directly and indirectly related work. Section 3 presents a motivating example and shows the deficiencies of prior approaches. Section 4 details our automatic marshaling techniques and describes our automated tool. Section 5 validates our approach by presenting two case studies. Section 6 outlines future work, and Section 7 concludes.

2 Motivation and Related Work

A wealth of existing research literature deals with some aspect of marshaling program state. Nevertheless, as we argue next, our domain presents a unique set of challenges which have not been fully addressed by prior approaches. We first explain the unique challenges of automatically generating C++ marshaling functionality. Then we examine indirectly related research on implementing and improving serialization in Java, C, and C++. Finally, we detail prior efforts to automate marshaling C++ data structures for MPI and compare the existing state of the art with our approach.

2.1 Design Objectives

In creating an automated tool for generating marshaling functionality for C++ data structures, we set the following objectives:

1. The tool should provide abstractions to automate the handling of low-level marshaling details, to make our approach appealing to experts and non-experts alike.
2. The generated marshaling code should be highly efficient, as performance is a crucial requirement for HPC applications.
3. The tool should be able to marshal any subset of an object's state to minimize the amount of network traffic and enable multiple marshaling strategies per object rather than providing only one strategy per type (i.e., C++ class) to allow maximum flexibility.
4. The generated marshaling functionality should not require any modification to the existing code, to make our approach work with third-party libraries.
5. The generated code should use only standard MPI calls and not require any runtime library to ease deployment and to ensure cross-platform compatibility.
6. The marshaling technique should be able to support a subset of the C++ language that contains commonly used features including:
   (a) Non-primitive fields
   (b) Pointers
   (c) Non-public fields (i.e., private and protected)
   (d) Static and dynamic arrays
   (e) Inheritance
   (f) Standard Template Library (STL) containers

In the following discussion of related work, we will refer to the above six objectives to motivate our approach and to explain why existing approaches are insufficient.

2.2 Challenges of Automatic C++ Marshaling

Ultimately, automatic marshaling requires the ability for an external program entity to traverse an object graph. The marshaling functionality has to be able to access all the fields of a C++ object irrespective of their access protection level, including private and protected fields. Additionally, the marshaling functionality must be able to determine type and size information about each marshaled field. In the case of pointers and dynamically allocated structures, this information is not generally available at runtime.

The standard RunTime Type Information (RTTI) [18] of C++ is not sufficient to provide support for automatic marshaling. However, efforts have been made to provide advanced facilities for C++ that enable access to runtime type information. The most notable of which is the Metaobject Protocol (MOP) for OpenC++ [7], which enhances the language with reflective capabilities providing an effective approach to dealing with the challenges outlined above. Nevertheless, standard C++ implementations do not posses reflective capabilities. Thus, an approach using any non-standard extensions of C++ would be inappropriate, as it would violate objective four in the previous section.

2.3 Indirectly Related Work

An example of a mainstream, commercial object oriented programming language with built-in reflective capabilities is Java. Ever since the capability to "pickle" the state of a Java object [29] was added to the language in the form of Java Object Serialization [32], numerous approaches for optimizing this capability have been proposed. Java reflection[31] provides capabilities for runtime inspection of Java object graphs and also for modifying object state. Many of the proposed approaches of optimizing Java serialization had the aim of making the language more amenable for HPC. The proposals for faster serialization eliminated the overheads of reflection by providing custom code for individual objects [27], using native code [21, 22], and finally using runtime code generation to specialize serialization functionality [1]. Because C++ does not have built-in reflective capabilities, none of these approaches are applicable to us. It is worth noting that the vision of Java as a language for HPC has not taken hold in mainstream high performance computing.

Remote Procedure Call (RPC) systems such as Sun RPC [33] and DCE RPC [34] provide marshaling capability for C structures. The programmer provides an IDL specification for a remote procedure call and its parameters, and a stub compiler automatically generates marshaling and unmarshaling code. However, the marshaling/unmarshaling functionality of RPC systems is not suitable for our approach. First, their target language is C rather than C++, which violates objective six. Second, the generated code does not use standard MPI calls, which violates objective five. Finally, it is not easy to specify a subset of an object state to marshal, which violates objective three.

CORBA [26] and DCOM [6] are object-oriented extensions of the RPC systems above. While they do provide C++ language mappings, they do not cover all of the C++ language features in objective six. Specifically, CORBA marshaling facilities support only single class inheritance and have no notion of protected-fields [25]. Furthermore, to satisfy the programming conventions of CORBA, existing C++ code either has to be modified or special factory classes must be written [26], violating objectives four and one, respectively. Neither CORBA or DCOM marshaling provide special handling for STL container classes. DCOM is platform-dependent, being mainly supported on the Windows platform. Lastly, CORBA and DCOM require runtime libraries, violating objective five.

Adaptive parameter passing [20] aims at optimizing RPC marshaling by sending a subset of an object's state graph. The approach introduces a domain specific language and does not support MPI, violating objective five and not sufficiently meeting objective one.

2.4 Directly Related Work

Several prior approaches have attempted to provide automatic serialization of C/C++ data structures for MPI applications. These approaches include both automatic tools and special purpose libraries. The automatic tools that we consider closely related work include AutoMap/AutoLink [11], C++2MPI [14], and MPI Pre-Processor [28]. In addition, Boost.MPI and Boost.Serialization [17] provide library support for seamless marshaling and unmarshaling. Next we describe and compare the features of each of these approaches to motivate our approach and demonstrate how it improves on existing state-of-the-art.

AutoMap/AutoLink [11] provides automatic marshaling of C structures by enabling the user to annotate fields to be marshaled. While the annotations provide a level of abstraction, its granularity might be too low-level particularly for non-expert users, possibly violating objective one. Obviously, this approach will fail if the fields are a part of a structure in a third-party un-modifiable library, which violates objective five. In addition, AutoMap/AutoLink does not support C++, which violates objective six.

C++2MPI [14] and MPI Pre-Processor [28] provide automatic creation of `MPI_Datatype`'s for C/C++ data structures. An `MPI_Datatype` is a list of memory offsets describing the data to be marshaled given the base offset of a structure. However, the approach makes an implicit assumption that all memory in a structure has been allocated

Objectives	AM/AL	C++2MPI	MPIPP	Boost	MPI Serializer
1. High-level abstractions	+/-	+	+	+	+
2. Efficient Code	+	+	+	+/-	+
3. Multiple partial object marshaling	-	-	-	-	+
4. No source modification required	-	-	+	-	+
5. No run-time library required	+	+	+	-	+
6. Support C++	-	+	-	+	+
a. Non-primitive fields	-	-	-	+	+
b. Pointers	-	-	-	+	+
c. Non-public fields	-	+	-	+	+
d. Static and dynamic arrays	-	+/-	-	+	+
e. Inheritance	-	-	-	+	+
f. STL containers	-	-	-	+	+

Table 1. Comparison to directly related state-of-the-art approaches

statically, which violates objective six. In addition, the tools do not support marshaling subsets of an object state, which violates objective three.

Finally, the Boost.MPI and the Boost.Serialization libraries [17] aim at modernizing the C++ interface to MPI by utilizing advanced generic programming techniques [8]. These libraries provide support for automatic marshaling of primitive data types, user-defined classes, and STL container classes. In order for a user-defined class to use the services of the Boost libraries, it has to supply a `serialize` method either as a class member or as an external method. To use a member method requires changes to the original source code of the class, which violates objective four. This violation can be avoided by supplying `serialize` as an external method. However, in this case `serialize` would not be able to access non-public fields of a class, without the class declaring the method as `friend`, which violates objective six. A `friend` declaration, of course, would once again violate objective four. Moreover, the Boost libraries follow the per-type marshaling strategy. That is, all objects of a given C++ class are marshaled based on the functionality of the corresponding `serialize` method, violating objective three. While the logic inside such a `serialize` method might marshal objects differently depending on some of their properties (e.g., field values), this offers only limited flexibility and might incur significant performance overheads, violating objective two.

Table 1 illustrates how well each directly related approach satisfies our stated objectives, thereby motivating our approach.

3 Motivating Example

In the previous section, we gave a general explanation about why the existing state-of-the-art does not fully address the needs of MPI C++ programmers. To further elucidate this claim, we present a concrete example of a C++ class and show why existing approaches are insufficient to enable this class with efficient marshaling functionality. In the process, we also pinpoint the additional capabilities required to accomplish this goal.

Our example comes from a well-known computational problem in astrophysics, the *N-Body* problem [19]. This problem belongs to an important class of parallel algorithms utilizing long range data interactions. In layman's terms, the algorithm computes the gravitational force exerted on a single mass based upon its surrounding masses. To express this problem in C++, a programmer might create a class `Mass` as follows:

```
class Mass {
private:
  vector<Mass> surrounding; //nearby masses
protected:
  float force_x, force_y; //resulting force
public:
  float mass; //mass value
  float x, y; //position
  ...
};
```

As part of the algorithm, we need to send an object of this class from one process to another. Furthermore, only a subset of the object's state is needed at any given time. Specifically, when sending the `Mass` object from the Master process to the Worker process, only fields `mass`, `x`, `y`, and `surrounding` are required. Conversely, when sending the object in the opposite direction, only fields `force_x` and `force_y` are needed.

Let us consider how these simple marshaling tasks can be successfully accomplished using the existing state-of-the-art approaches to C++ marshaling for MPI. In particular, we evaluate the four directly related approaches

discussed in the previous section: AutoMap/AutoLink, C++2MPI, MPIPP, and Boost.

The marshaling functionality required for sending the object from the Master to the Worker process obviously cannot be accomplished by using any of the first three approaches, as they provide no support for marshaling fields of dynamically-determined size such as STL containers. Additionally, because these three approaches do not support partial object marshaling, they could only provide marshaling functionality for sending the object in the opposite direction by defining a new structure and copying the `force_x` and `force_y` fields to a temporary instance of this structure.

The Boost libraries do provide support for the required marshaling functionality, but this support is not adequate to address all of the requirements. The programmer could add a serialize member method to the `Mass` class to marshal the required fields for the Master to Worker communication. Notice that this approach would fail if class `Mass` could not be modified (e.g., being a part of a third-party library). However, the marshaling functionality implemented by such a serialize method would be per-type. As a result, it would be non-trivial to use different marshaling logic when sending the object back from the Worker to the Master. Recall that in this case we only want to send back the fields `force_x` and `force_y`. One way to enable such custom marshaling would be to add an extra `boolean` direction field to the class and to set it to an appropriate value before marshaling the object. Nevertheless, even if code modifications were possible, this solution might not be acceptable, as it incurs performance overheads by preventing Boost from optimizing the marshaling functionality through `MPI_Datatype`'s.

To summarize, the primary reasons why the existing state-of-the-art approaches fell short of meeting the demands of this marshaling task are that they either failed to provide adequate support for common C++ language features (i.e., non-public fields, STL container) or required extensive code modification and restructuring. Thus, we feel that this simple example sufficiently motivates the need for a better approach to automatic C++ MPI marshaling. While this was only a simple example, the case studies that we present in Section 6 further justify the need for our new approach and automatic tool.

4 A Generalized Automatic C++ MPI Marshaling Approach

The domain of HPC applications is computationally intensive. Even while utilizing parallel hardware resources, HPC applications often run for prolonged periods of time before arriving at a solution. Lab scientists and engineers, who are the primary developers of HPC applications, care first and foremost about decreasing this "time-to-answer."

This presents a unique set of challenges to software engineering researchers who aim at providing novel programming tools in support of HPC application developers. These tools must provide a high degree of usability while still producing highly efficient code. These two goals are often irreconcilable. Ensuring good usability entails providing abstractions, which are commonly detrimental to performance. In order to create a user-friendly automated tool for high performance applications, we make the following design assumptions:

- Any changes to the marshaling functionality will be made via the GUI of our tool, which will re-generate the marshaling code.
- The generated marshaling code will not be modified by hand.
- The generated marshaling code places a higher priority on performance than readability.

We believe that these assumptions are reasonable, as the primary focus for HPC developers is on performance, usability, and time-to-answer.

Next we describe the main steps of our approach in turn. Figure 1 shows an overview of the control flow of our approach and automated tool. The programmer interacts with the tool through a GUI. As the first step, the tool uses a compilation technology (i.e., parsing) to extract information from the C++ data structures (e.g., classes and structs) for which marshaling code can be generated. Then the programmer uses the GUI to select the subset of an object state to be marshaled. Figure 2 displays class `Mass` from Section 3, with fields `mass`, `x`, `y`, and `surrounding` selected for marshaling. This visual input is the only action required from the programmer to parameterize the backend code generator of our tool, which then generates marshaling functionality. The programmer can then simply include the automatically generated marshaling code with the rest of their HPC application.

4.1 User Interface

The programmer starts by selecting a C++ source file using a standard file browse dialog. In response, the tool invokes a C++ parsing utility called GCCXML [10]. This utility taps into the platform's C++ compiler (e.g., GCC on UNIX or Visual C++ on Windows) to create a structured XML description of a given C++ compilation unit; this XML description can be used by other language processing tools. Our tool then parses the XML file and displays the extracted information to the user through a GUI.

The GUI employs a tree-view visualization [13] to display C++ object graphs. The tree-view provides an intuitive

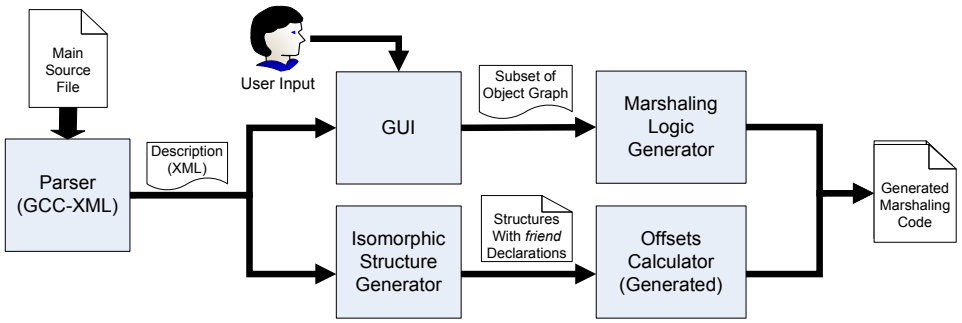

Figure 1. An overview of the control flow for MPI Serializer.

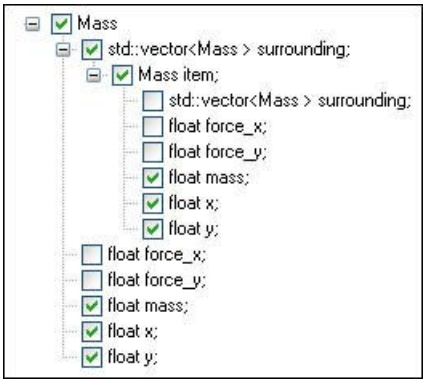

Figure 2. Selecting the fields of `Mass` to marshal.

interface for the programmer to explore the structure of an object graph and to select the fields to be included into the subset of the marshaled object's state. The root node of the tree-view represents the main class selected for marshaling. All other nodes represent fields transitively reachable from the main class. To prevent circular references from causing infinite node expansion, the tree-view implements a lazy node visualization strategy, expanding a node only when it is selected for marshaling.

4.2 Handling C++ Language Features

Automatic marshaling of C++ data structures presents many challenges arising as a result of the sheer complexity of this mainstream programming language. In designing our tool, we had to address a number of these challenges to support a reasonable subset of the language. In the following discussion, we highlight the novel insights we have gained from designing our approach and automated tool.

4.2.1 Dynamic arrays

In C++, the meaning of a pointer variable is somewhat ambiguous. A pointer may be pointing to the memory location of a single element, or it may be pointing to the first element in a dynamic array. In the latter case, the language provides no standard way to determine the array's size. By convention, when designing a class, many C++ programmers define an additional member variable to keep track of the size of its corresponding dynamic array member variable. Our strategy for marshaling dynamic arrays assumes that the programmer is following this convention. To this end, the tool's GUI enables the programmer to visually disambiguate the meaning of a pointer variable to be marshaled. If the variable is indeed a dynamic array, the programmer can select the numeric field representing the array's size.

4.2.2 Polymorphic fields

Another complication that pointer fields present for marshaling is polymorphism. A pointer to a base class may actually be assigned to an instance of any of its subclasses. Our tool automatically determines whether the possibility for polymorphism exists by examining the inheritance graph of each field. In lieu of a precise static analysis, the programmer is expected to specify object graph subsets for all possible derived instances to which a pointer field could be pointing at runtime. Thus, the programmer implicitly disambiguates the range of polymorphic possibilities for a pointer field. In the case of multiple possibilities, runtime type information (RTTI) is employed in the generated code to determine the appropriate marshaling functionality for the polymorphic field.

4.2.3 Non-public and Inherited fields

C++ supports the encapsulation principle of object-oriented programming by disallowing outside entities (i.e., other classes and methods) from accessing non-public fields of a class. However, non-public fields may be part of the subset of an object's state selected for marshaling. Previous directly related approaches employed two different strategies for accessing non-public fields for marshaling.

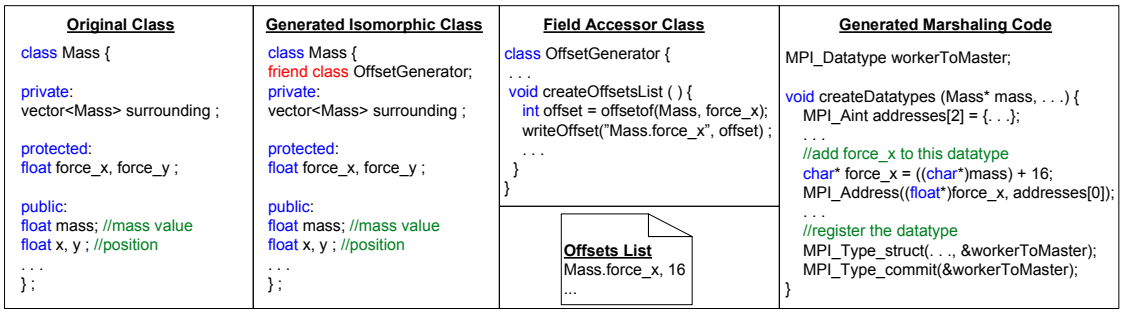

Figure 3. Accessing non-public fields using our C++ standard-compliant strategy.

The Boost libraries require declaring their library `Archive` class as a `friend` to all marshaled classes. In C++, a `friend` entity is granted access to the non-public fields of a befriended class. This strategy is not acceptable for our approach, as it requires modification to the existing source, thereby violating our design objectives. C++2MPI uses a different strategy by creating an isomorphic copy of all marshaled classes, replacing all non-public declarations with public ones in the replicated classes. The non-public fields' offsets in the original classes can subsequently be obtained by consulting the corresponding offsets in the replica classes at runtime. While this strategy does not require changes to the existing source code, the C++ standard [18] only requires that non-bit-fields of a class or structure without an intervening access-specifier (i.e., `private`, `protected`, or `public`) be laid out in non-decreasing address order. In other words, a C++ compiler is free to allocate blocks of fields with different access specifiers in an arbitrary order. As such, the C++2MPI strategy is not guaranteed to work for all C++ standard-compliant compilers. This strategy also results in code bloat: every marshaled class has to be deployed with its isomorphic copy. Such code bloat could be detrimental for the locality of reference in the processor's cache, resulting in performance overhead, which is unacceptable for our approach. Thus, solving the challenge of accessing non-public fields requires a new strategy capable of maintaining the existing source code without sacrificing performance.

To this end, we have designed an approach to calculate the offsets of all fields of a C++ class that ensures cross-platform and compiler independence while still strictly adhering to the C++ standard. This design is enabled by adding an additional code generation layer on top of the existing infrastructure of our tool. This layer generates metadata about the marshaled C++ data structures using a technique that combines the strategies of C++2MPI and the Boost libraries.

For each class, our approach generates an isomorphic copy that preserves the order of fields and access specifiers but also includes a `friend` declaration for a generated "field accessor" class. The field accessor class creates a list of all fields' offsets in a class using the standard `offsetof` macro. In C++, `friend` declarations are strictly compile-time concepts and have no affect on the memory layout of a class or its instances. Therefore, the field offsets obtained from a generated isomorphic class are guaranteed to be the same as the corresponding field offsets in the original class.[1]

The obtained offsets are then inserted into the generated marshaling code to access an object's non-public fields. A slightly simplified example of accessing the `protected` field `force_x` in class `Mass` is shown in Figure 3. This approach minimizes runtime overhead by using generated constant offset values to access non-public fields, maintains a cache-friendly memory footprint by using isomorphic classes only at generation-time, and is guaranteed to be cross-platform and compiler independent by following a C++ standard-compliant strategy.

One could argue that our solution for accessing non-public and inherited fields violates the encapsulation principle by allowing an outside entity (i.e., the marshaling method) to access the non-public state of an object. However, the current solution is acceptable for the HPC domain, which has the requirements of preserving the existing source code and not incurring performance overheads take priority over strict adherence to object-oriented principles.

4.3 Generating Efficient Marshaling Code

In addition to the ability to support a large subset of the C++ language, our approach also implements a fast and memory-efficient buffer packing strategy. This strategy minimizes the required size of the buffer, reducing the

[1] Our approach relies on two assumptions. First, either the source or a corresponding GCCXML descriptor file must be provided for all third-party libraries. Note that the latter option is reasonable for closed-source libraries since the GCCXML file describes only the structure of the library and not any proprietary logic. Second, the C++ compiler must be deterministic. That is, the memory layout of two identically-named isomorphic classes will be the same. Although this property is not strictly guaranteed, it would likely be infeasible for a non-deterministic C++ compiler to be standards-compliant.

bandwidth needed to transmit marshaled data. The key to the efficiency of our approach is generating code that leverages the mature marshaling facilities of MPI. Therefore, we first give a quick overview of these facilities before describing the novel insights of our strategy.

MPI provides a variety of API facilities to support marshaling. However, for this discussion, we focus on `MPI_Datatype` and `MPI_Pack`. The `MPI_Datatype` construct provides a reusable cached collection of static memory offsets for efficient marshaling of arbitrary data structures. The programmer first initializes an instance of a data structure and calculates the memory offsets and sizes of the fields to be marshaled. These offsets are then stored in an `MPI_Datatype` for later use.

To pack data into a buffer, MPI provides an API function `MPI_Pack`. The user passes the data structure to be marshaled and the buffer to store the marshaled data. However, `MPI_Pack` supports only primitive C++ types and user-defined `MPI_Datatype`'s. For objects containing dynamically allocated fields, using `MPI_Pack` is not trivial, as it requires static offsets for all the marshaled data.

To guide the generation of efficient marshaling code, our approach first performs a bounds-checking operation on the marshaled object graph. Bounds-checking traverses the object graph in order to determine if the user-specified subset of the object's state is "bounded." A bounded object state is a transitive property, signifying whether memory offsets and sizes of all marshaled fields can be determined statically. An object graph containing dynamic array, pointer, or STL container fields violates this property.

Since using `MPI_Datatype` is known to be more efficient for packing bounded objects than packing fields individually [24], our approach utilizes this construct for marshaling all bounded subsets of an object graph. Figure 4 illustrates how our approach utilizes `MPI_Datatype`.

If the programmer selected an unbounded subset of an object graph, our approach then employs static polymorphism to provide a global method `MPI_Pack` with a similar signature to that of the standard `MPI_Pack` method. While the standard method takes an `MPI_Datatype` as a parameter, the generated polymorphic method takes an automatically-generated `MPI_Descriptor` enum type specifying the marshaling strategy to use.

This approach is particularly useful for non-experts, who can follow a uniform convention by calling both the standard `MPI_Pack` on a bounded object with an `MPI_Datatype` and by calling the generated `MPI_Pack` on an unbounded object using an `MPI_Descriptor`. The distinction between the standard and the generated version of `MPI_Pack` is resolved at compilation time, resulting in zero runtime performance overhead.

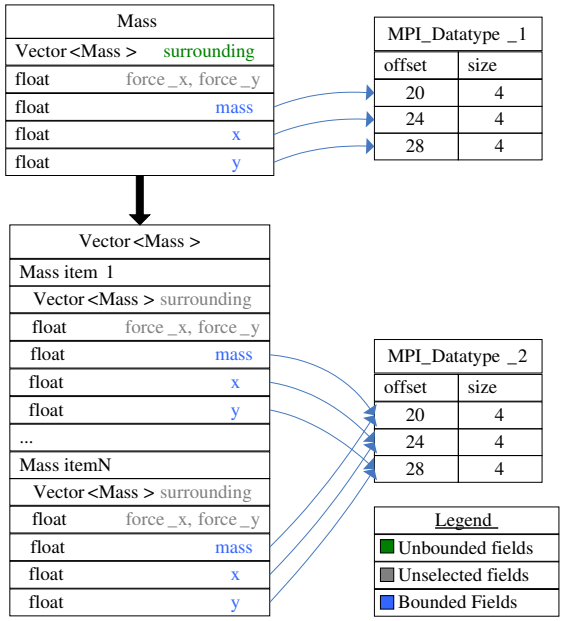

Figure 4. An example of mapping bounded subsets of an object graph to `MPI_Datatype`'s. The offsets and sizes are for a 32-bit architecture using GCC.

4.4 Using Generated Code

To further illustrate how the programmer uses generated marshaling code, consider the Master-Worker communication that needs to pass instances of class `Mass`. In the case of passing an instance of `Mass` from Worker to Master, the subset of the object graph consists of `force_x` and `force_y`, and as such is entirely bounded. To maximize efficiency, the tool generates an `MPI_Datatype` (`workerToMaster`) for this subset. However, passing an instance of `Mass` from Master to Worker, the marshaled subset is unbounded, as it contains an STL container field, `surrounding`. This subset, therefore, requires custom marshaling code, capable of traversing unbounded subset efficiently. The generated code consists of a custom `MPI_Pack` function and an `enum` type, `masterToWorker`. However, the marshaling interface exposed to the programmer is almost identical in bounded and unbounded cases, as shown below:

```
void foo() {
  Mass mass; char* buf;
  ...
  //pack for Master to Worker
  MPI_Pack(&mass,1,masterToWorker,buf,...);
  ...
  //pack for Worker to Master
  MPI_Pack(&mass,1,workerToMaster,buf,...);
}
```

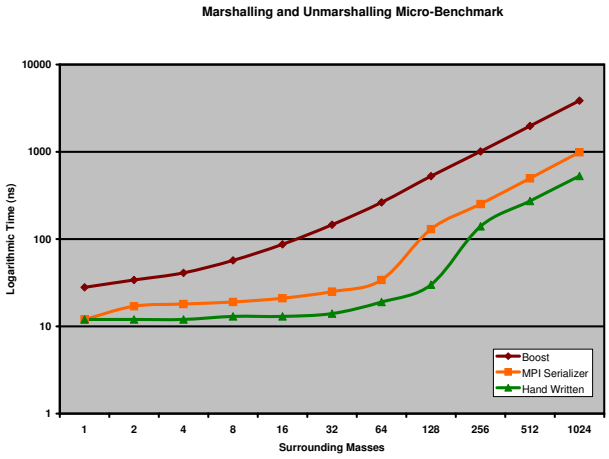

Figure 5. Benchmark results comparing the total marshaling and unmarshaling time required for the `Mass` class example.

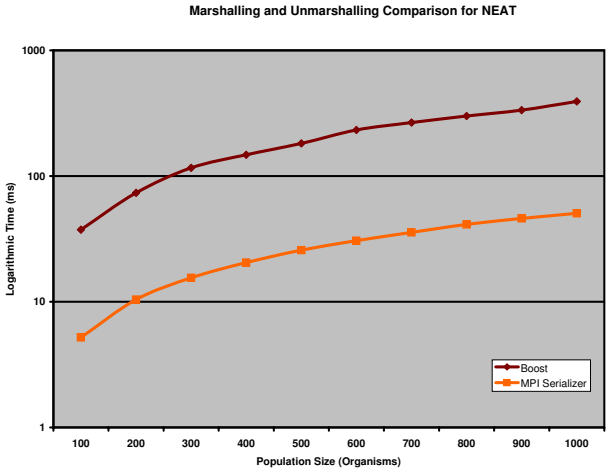

Figure 6. Benchmark results comparing the marshaling time required for a population of organisms (i.e. potential solutions) for NEAT.

The two separate packing calls are exposed to the programmer as if they were the same `MPI_Pack` call, even though the call for `masterToWorker` actually invokes a specialized generated packing method. The advantage of this approach is that it enables the programmer to use the generated code almost identically to how they use the standard MPI interface.

5 Case Studies and Performance Results

The purpose of the presented case studies and the associated performance comparisons is to validate our approach against the requirements stated in Section 2.1 as well as against the existing state of the art. To recap, our goal is to support a sufficiently large subset of the C++ language, while ensuring high performance in the generated marshaling code.

The evaluation of MPI Serializer that we have conducted consists of micro-benchmarks and case studies. The micro-benchmarks enable us to pinpoint the fine-grained performance advantages and limitations of our approach. The two case studies involve the automatic generation of the marshaling functionality for real HPC applications. The first case study showcases the use of our tool as a refactoring aid in the often difficult task of parallelizing a sequential program. The second case study demonstrates the fitness of our tool to provide efficient marshaling functionality for an existing high-performance application.

In making the performance related arguments, we compare our work against the Boost libraries and hand-written code on a 3.0GHz dual-core Pentium 4 with 2GB of RAM running Debian GNU/Linux, GCC version 4.1.2. However, a comparison of our work with other directly related approaches, such as MPI Pre-Processor [28], AutoMap/AutoLink [11], or C++2MPI [14], is impossible due to their lack of support for C++ language features or even C++ itself.

One additional advantage of our approach that is easy to overlook by focusing on performance numbers is that it reduces the amount of maintained hand-written source code. Since the complexity of software grows exponentially in relation to the size of a program [5], every line of source code that the programmer has to write by hand contributes to the software maintenance burden. Addressing changed requirements or fixing program defects requires a program maintenance effort that is directly proportional to the size of a program [12]. By reducing the amount of maintained source code, our approach has the potential to ease the software maintenance burden. Therefore, while the following case studies highlight performance gains, our approach also provides software engineering benefits to the target applications.

5.1 Micro-benchmarks

For the micro benchmark, we used the `Mass` class described in Section 3. Figure 5 shows the total combined marshaling, sending, and unmarshaling time taken for the Master to Worker communication as described earlier. The x-axis represents the number of surrounding `Mass` objects (i.e., being marshaled and unmarshaled). The figure demonstrates the differences in performance between hand-written code utilizing the Boost libraries and code automatically generated by our tool. The results show a speed-up between 2x and 4x for our approach compared to Boost, with

the rate of speedup increasing as the size of the transmitted data structure grows.

While the generated code is highly efficient, the programmer can still write fine-tuned marshaling code by hand, which would yield better performance. This benchmark explores an optimization technique for marshaling collections called striding. It refers to using a heuristic to reduce the number of collection elements to be marshaled. The results show that using a basic striding increment of 3 (i.e., selecting only 1/3 of the elements in the vector of `Mass` objects) can still beat our approach.

5.2 Parallelizing NEAT

NeuroEvolution through Augmenting Topologies [30] (NEAT) is an artificial intelligence algorithm for training a collection of artificial neural networks. NEAT is a genetic algorithm [3] that mimics Darwinian evolution by repeatedly competing potential solutions against each other and then selecting and breeding the fittest individual solutions. We used MPI Serializer to automatically generate marshaling code for an on-going research project that parallelizes NEAT to run on a supercomputer [37].

Figure 6 shows the time required to marshal and umarshal a NEAT population set using the Boost libraries compared to using our approach. The x-axis in Figure 6 represents the number of potential solutions (i.e., population elements) to be marshaled. The performance numbers indicate similar scalability for both approaches. However, the abstractions provided by the Boost libraries result in a performance overhead causing our approach to be as much as an order of magnitude faster in some cases. Additionally, this particular application requires the marshaling and unmarshaling of several nested fields (i.e., object fields with other object fields). Providing Boost `serialize` methods by hand is much more difficult for this case, as several classes must be modified. By contrast, our approach requires the programmer to manipulate only a single visual hierarchy.

5.3 mpiBLAST 2.0

mpiBLAST [9] is a widely-used, open-source, parallel application for aligning genome sequences. It solves the problem of finding the closest known matching sequence for a given genome. mpiBLAST has been known to assist in the process of scientific discovery in domains as diverse as new drug development and classifying new virus species. mpiBLAST 2.0 is written in C++ [2], with MPI as the communication middleware. In this case study, we reimplemented the hand-written marshaling functionality of mpiBLAST by using both the Boost libraries and our automated tool. For benchmarking, we chose the phase of

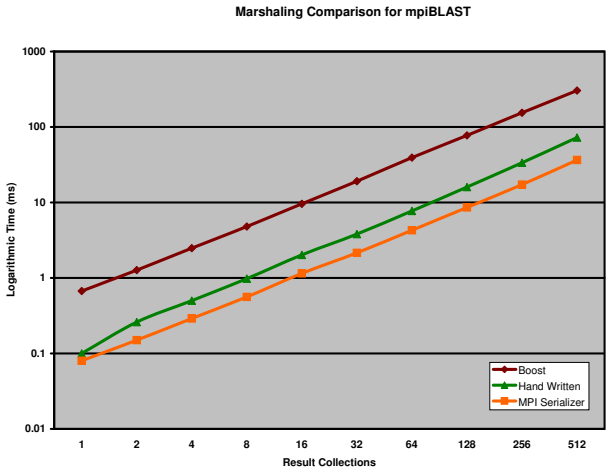

Figure 7. Benchmark results comparing the marshaling time required for a collection of genome alignment results.

the mpiBLAST algorithm when the Worker processes report back their search results to the Master.

Figure 7 compares the performance between the original hand-written code, the code using the Boost libraries, and the code automatically generated by our tool. The x-axis represents the number of sequence result collections marshaled. The results show that the automatically generated code is slightly more efficient than hand-written code and nearly an order of magnitude faster than the Boost implementation in some cases.

Similar to the NEAT case study, Boost's abstractions once again account for the overhead incurred while marshaling and unmarshaling. More surprisingly, the generated code yielded better performance than the original hand-written implementation. The cause of this is that the original implementation currently uses a less-efficient stream-based strategy to marshaling and umarshaling, to aide in the debugging process. However, the code generated automatically by our tool should require no debugging (provided that the tool's implementation is mature enough), as long as the visual input specified by the programmer correctly reflects the subset of the object graph to be marshaled.

6 Future Work

The target audience of our approach and automated tool is non-expert HPC programmers. Therefore, to further increase the usability of our tool, we would like to focus on improving the software engineering quality of the generated code. First, by replacing packing calls with size calculations, the same approach can be used to automatically determine the exact size of a destination buffer. Integrating this technique will make it possible for the generated code

to provide a more programmer-friendly external interface, similar to that of Boost.

Our approach is not necessarily divergent from Boost and other marshaling libraries. The clean, object-oriented interface that the Boost libraries provide to MPI programmers may in some cases be worth the accompanying overhead. Consequently, as a future extension, we plan to provide the capability to generate Boost `serialize` methods for data structures. This demonstrates that our approach's intuitive visual interface does not need to generate marshaling code which is difficult to read or modify by hand if necessary.

Finally, although our tool generates code specifically for HPC applications using MPI, our generalized approach extends beyond this specialized domain. Examples of other applicable software development scenarios that could benefit from having automatically generated marshaling code packed to a generic buffer include transmission, permanent storage, and check-pointing.

7 Conclusions

Our approach provides automatic synthesis of efficient marshaling functionality for HPC applications based entirely on visual input supplied by the programmer. This aspect of our approach is particularly appealing to the many lab scientists and engineers who have limited parallel programming experience. By automatically generating low-level, high-performance marshaling code, our approach eliminates the need to write tedious and error-prone code by hand, thereby facilitating the process of scientific discovery.

Acknowledgments

The authors would like to thank Jeremy Archuleta for his help with the mpiBLAST 2.0 benchmark and the anonymous reviewers for useful comments that helped improve the paper. This research was supported by the Department of Computer Science at Virginia Tech.

References

[1] B. Aktemur, J. Jones, S. Kamin, and L. Clausen. Optimizing marshalling by run-time program generation. In *4th International Conference on Generative Programming And Component Engineering (GPCE'05)*, Tallinn, Estonia, October 2005.

[2] J. S. Archuleta, E. Tilevich, and W. Feng. A Maintainable Software Architecture for Fast and Modular Bioinformatics Sequence Search. In *23rd IEEE International Conference on Software Maintenance*, Paris, France, October 2007.

[3] N. Barricelli. Esempi Numerici di Processi di Evoluzione. *Methodos*, 6(21-22):45–68, 1954.

[4] J. Barton and L. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1994.

[5] D. Berry. *Academic Legitimacy of the Software Engineering Discipline*. Carnegie-Mellon University, Software Engineering Institute, 1992.

[6] N. Brown and C. Kindel. *Distributed Component Object Model Protocol–DCOM/1.0*. Microsoft Corporation, November 1998.

[7] S. Chiba. A metaobject protocol for C++. *ACM SIGPLAN Notices*, 30(10):285–299, 1995.

[8] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 2000.

[9] A. E. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. In *ClusterWorld Conference & Expo and the 4th International Conference on Linux Cluster: The HPC Revolution 2003*, San Jose, California, June 2003. Best Paper: Applications Track.

[10] GCC-XML, the XML output extension to GCC, 2007. http://www.gccxml.org/.

[11] D. Goujon, M. Michel, J. Peeters, and J. Devaney. Automap and autolink: Tools for communicating complex and dynamic data-structures using MPI. In *Lectures Notes in Computer Science*, volume 1362, page 98, 1998. Presented at CANPC98.

[12] L. Gremillion. Determinants of program repair maintenance requirements. *Communications of the ACM*, 27(8):826–832, 1984.

[13] C. Guzak, J. Bogdan, G. Pitt III, and C. Chew. Tree view control, November 1999. US Patent 5,977,971.

[14] R. Hillson and M. Iglewski. C++2MPI: A software tool for automatically generating MPI datatypes from C++ classes. In *International Conference on Parallel Computing in Electrical Engineering (PARELEC'00)*, Washington, DC, USA, 2000. IEEE Computer Society.

[15] L. Hochstein and V. R. Basili. An empirical study to compare two parallel programming models. In *SPAA '06: Proceedings of the eighteenth annual ACM*

symposium on Parallelism in algorithms and architectures, pages 114–114, New York, NY, USA, 2006. ACM.

[16] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. Hollingsworth, and M. Zelkowitz. HPC programmer productivity: A case study of novice HPC programmers. In *ACM/IEEE Supercomputing Conference (SC'05)*, 2005.

[17] P. Kambadur, D. Gregor, A. Lumsdaine, and A. Dharurkar. Modernizing the C++ interface to MPI. In *13th European PVM/MPI Users Group Meeting (EuroPVM/MPI'06)*. Springer, 2006.

[18] A. Koenig. *The C++ Language Standard. Report ISO/IEC 14882: 1998*, 1998. http://www.nctis.org/cplusplus.htm.

[19] B. Lester. *The Art of Parallel Programming*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, USA, 1993.

[20] C. Lopes. Adaptive parameter passing. In *Object Technologies for Advanced Software: Second JSSST International Symposium (ISOTAS'96)*, Kanazawa, Japan, March 1996. Springer.

[21] J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[22] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *The seventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 173–182, New York, New York, USA, 1999. ACM Press.

[23] U. Mello and I. Khabibrakhmanov. On the reusability and numeric efficiency of C++ packages in scientific computing. In *The ClusterWorld Conference and Expo*, pages 23–26, June 2003.

[24] Message Passing Interface Forum (MPIF). MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, 1996.

[25] Object Management Group. *Objects by Value Specification*, 1998. ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf.

[26] Object Management Group. *C++ Language Mapping Specifications*, 2003. http://www.omg.org/docs/formal/03-06-03.pdf.

[27] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency Practice and Experience*, 12(7):495–518, 2000.

[28] E. Renault and C. Parrot. MPI Pre-Processor: Generating MPI derived datatypes from C datatypes automatically. In *The 2006 International Conference Workshops on Parallel Processing*, pages 248–256, Washington, DC, USA, 2006. IEEE Computer Society.

[29] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling state in the Java™system. In *COOTS'96: Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 19–19, Berkeley, CA, USA, 1996. USENIX Association.

[30] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[31] Sun Microsystems. *Java Core Reflection API and Specification*, 1997.

[32] Sun Microsystems. *Java Object Serialization Specification*, 2001.

[33] Sun Microsystems. *RPC: Remote Procedure Call Protocol Specification*, 2004.

[34] The Open Group. *DCE 1.1 RPC Specification*, 1997. http://www.opengroup.org/onlinepubs/009629399/.

[35] TOP500. Top 500 supercomputing sites - architecture. http://top500.org/stats/28/archtype/.

[36] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, pages 49–56, London, UK, 1997. Springer-Verlag.

[37] Virginia Tech. Virginia Tech terascale computing facility. http://www.arc.vt.edu/arc/SystemX/.