Tool for Automated Generation of MPI Datatypes

Andreas Schäfer, Dietmar Fey
Lehrstuhl für Informatik 3, Rechnerarchitektur,
Friedrich-Alexander-Universität Erlangen-Nürnberg
91058 Erlangen, Germany
{andreas.schaefer,dietmar.fey}@cs.fau.de

Adrian Knoth
Lehrstuhl für Rechnerarchitektur und -kommunikation,
Friedrich-Schiller-Universität Jena,
07743 Jena, Germany
Adrian.Knoth@uni-jena.de

Abstract

With the increasing availability of massively parallel high performance computing systems on the one hand, and the growing importance of computer based simulations on the other, MPI has become the industry standard tool for efficient message passing. However, users still have to manually maintain descriptions of their custom datatypes in order to enable MPI to successfully transmit them. We present TypemapGenerator, a flexible, easy to use tool for C/C++ that automatically generates MPI typemaps from user supplied source code. Its key contribution is that it offers both, ease of use and MPI's native performance.

1 Introduction

During the development of MuCluDent [7] (Multi Cluster Dendrite), a parallel simulation software for cooling molten metal alloys, we were facing the problem of having to transmit user-defined C++ datatypes via MPI [2]. Our first approach was to manually specify MPI typemaps for the datatypes, but since the model classes are constantly changing, updating those typemaps soon proved to be tedious and error-prone. Common issues, that are neither detected by the compiler nor by the MPI library include accidentally omitting a new class member or using the wrong MPI datatype. For instance one could use `MPI_FLOAT` even though the member was changed to `double`. In fact, this is a common problem when developing MPI programs: as parallelization and simulation models get increasingly more complex, developers shift from languages like Fortran to other languages offering virtually equivalent efficiency and higher level abstractions. C++ is among the most important languages for parallel programming as it marries speed and expressiveness. However, one of its major drawbacks is the lack of run-time type information. This puts the burden of describing C++ types for MPI on the programmer.

Required features of a tool to solve this problem include low communication/computational overhead, no required user interaction, the ability to work with heterogeneous computing systems (e.g. for grid and cloud systems) and partial serialization.

The last feature is important when a complete typemap cannot be created for a class, as no MPI datatypes are available for some of its members, but transmission of the remaining members would still be beneficial to the user. One example found in our own simulation code are objects which store references to output streams (e.g. a log file) in order to selectively direct debugging information. Instances of such a class cannot be completely serialized via MPI, since the output streams are linked to local resources. Nevertheless we found it useful to automatically keep an MPI typemap maintained for those classes as well and fix the references manually, if need be. This optional omission of members is what we call partial serialization.

2 Related Work

The tools previously available can be roughly broken down into two categories. The first one consists of tools that parse the source code and generate small parts of auxiliary code which in turn can create MPI typemaps [5, 6]. This approach has the advantage that the user's only burden is to appropriately flag classes for which he needs MPI datatypes, but comes with the disadvantage that parsing C++ code is inherently complex and compute intensive. Tools in the second category form a wrapping layer on top of MPI [4, 10, 3]. Due to C++'s weak support for run-time type information they typically require user interaction to correctly serialize user defined datatypes.

C++2MPI [5] works similar to our TypemapGenerator. It parses a user-supplied set of headers and for all classes enclosed by `#pragma MPI_START/END`, a method for MPI typemap creation is generated. C++2MPI's parser can neither handle multi-dimensional arrays [6] nor classes with members for which no MPI datatype could be generated. Apart from that, its code doesn't work with current compilers (e.g. GCC 4.x.x or ICC 10 or later).

MPI Extended C Compiler (MPIECC) [6] works as a drop-in parser which extends an MPI C compiler by an additional stage, located between the preprocessor and the compiler. This way it can intercept user defined datatypes and provide an `MPI_Typeof()` operator which yields the corresponding typemaps. It can handle even complex memory transfers including pointers, but only for C types.

Object Oriented MPI (OOMPI) [10] simplifies the usage of MPI's C-style interface by providing an object oriented C++ facade. It adds an elegant interface to simplify creation of user defined MPI datatypes. However, this interface still requires the user to list all class members manually.

Similarly, TPO++ [4] also acts like a layer on top of MPI. It provides a macro `TPO_TRIVIAL()` to mark classes that have a trivial copy constructor. This relieves the user from manually listing all class members, but since MPI is in that case simply used as a byte stream transport layer, it doesn't work with heterogeneous setups.

Boost.MPI [3] is one of the younger Boost libraries and provides three different ways to serialize objects. The fist one is via Boost.Serialization. Similarly to OOMPI the user has to define a method `serialize()` containing all class members. As Boost.Serialize has to copy all data and translate it into an architecture independent representation, this is also the slowest way. The second way is to additionally flag classes with the macro `BOOST_IS_MPI_DATATYPE()`, which will make Boost.MPI create MPI datatypes from the `serialize()` method. Finally, Boost.MPI will communicate an object similarly to how TPO++ would do with `TPO_TRIVIAL()` enabled, if its class it is flagged with `BOOST_IS_BITWISE_SERIALIZABLE()`.

3 TypemapGenerator

None of the previous works is able to generate valid MPI datatypes for our use case without requiring the programmer to manually describe the classes' members. The best we could get was that our objects would be sent verbatim as byte streams, but that way our codes wouldn't work on heterogeneous systems (e.g. mixed AMD Opteron and IBM Cell BE setup as found in Roadrunner [1]).

Our TypemapGenerator is designed to be a part of LibGeoDecomp [8] (Library for Geometric Decomposition)), a generic framework for high performance stencil codes, specifically targeted at grid systems. LibGeoDecomp was used at the core of MuCluDent and in turn relies on Pollarder [9], a library to detect the topology (e.g. network structure, node configuration) of parallel systems. Since LibGeoDecomp is a generic library and has to run on heterogeneous systems, required features are fully automatic operation without user interaction, even when encountering unknown (user supplied) types and the ability to leverage MPI's support for converting datatypes.

Basically, TypemapGenerator works similarly to C++-2MPI: a parser will gather information about the user defined classes at compile time and a code generator will then create two wrapper files `typemaps.{h,cpp}` which will construct the necessary MPI datatypes at run time. Figure 1 depicts the basic workflow. To avoid C++2MPI's Achilles heel (which is a complex and specialized source code parser, that is hard to maintain and port to future systems), we decided to use Doxygen[1]. Doxygen is originally a tool to automatically create documentation (in HTML, PDF and various other formats) from the source code and specially formatted comments therein. However, it can also produce an XML description of parsed classes. Doxygen has several advantages: it is fast and many projects (including our own) already use it, so it does impose only little overhead during build time. Furthermore it can parse not only C++, but also C, Java, Objective-C, C# and a few others. This is important as another design goal was to keep TypemapGenerator portable to other languages.

Table 1 compares TypemapGenerator's features with C++2MPI and MPIECC. While MPIECC can handle nested pointers, it cannot parse C++ source code. C++2MPI is unable to parse multi-dimensional arrays (which is necessary for some of our simulation codes) and cannot create partial typemaps. Currently we do not support template classes, since our own codes don't use them at the model level, and we furthermore don't support the traversal of pointer structures (as MPIECC does), since this is neither efficient, nor is it required for our models.

3.1 MPIParser

TypemapGenerator itself is written in Ruby[2] and consists of two main classes: `MPIParser` and `MPIGenerator`. The `MPIParser` is a lightweight XML parser based on Ruby's `rexml` library. One of its tasks is to find all classes

[1]`http://www.doxygen.org`
[2]`http://www.ruby-lang.org`

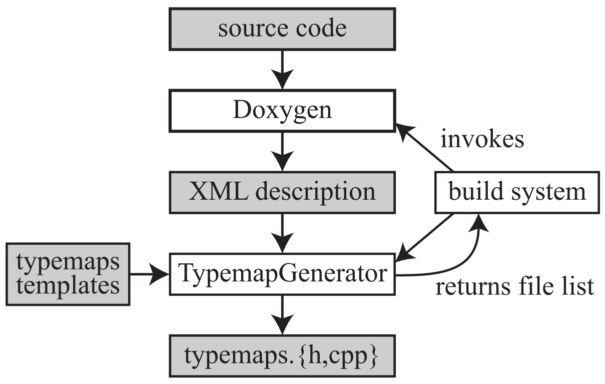

Figure 1. TypemapGenerator workflow and data stream. Shaded boxes represent data, white boxes programs.

Table 1. Features of TypemapGenerator compared with other code parsers and MPI datatype generators.

Feature	C++2MPI	MPIECC	TG
Language	C/C++	C	C/C++
Detect relevant Files	no	yes	yes
Derived Classes	yes	no	yes
Template Classes	yes	no	no
Partial Typemaps	no	no	yes
Array Dimensions	1	any	any
Pointers	no	yes	no

for which MPI typemaps are required. The user has to mark them via `friend Typemaps`. This list is then fed back to the build system so that complete runs of Doxygen and our generator are only invoked if one of those relevant files has changed. This feature is unique to TypemapGenerator. `MPIParser` then has to extract information on all member types and parent classes. For each class a list containing the name of the required members and their types is created. In case of inheritance, parent classes are effectively treated as member types. This corresponds to how C++ implements inheritance and does also capture multiple inheritance nicely.

Since many classes will depend on other classes and MPI datatypes can only be built bottom-up (MPI can only construct new types from already defined ones), `MPIParser` yields a topological sortation according to the *is parent/member of* relation.

3.2 MPIGenerator

Based on the information gained by the parser, it is `MPIGenerator`'s task to create the source code files `typemaps.{h,cpp}`. For this it relies on three template files: the header template will be filled with all necessary header includes, method signatures and MPI datatype variable declarations. The source template contains some auxiliary wire up code and will be filled with all variable and method definitions. For each user defined type, a method to create its MPI datatype will be inserted based on the third template, the method template. Figure 2 outlines how the generated code looks like. Figure 3 illustrates how a user can conveniently access the automatically generated MPI datatypes via template methods. The use of template files makes it easy to adapt TypemapGenerator for new languages.

4 Evaluation

In Section 2 multiple techniques for object serialization have been mentioned. For our use case, serialization has to be platform independent and impose little overhead at runtime. To compare the different techniques, we ran a couple of latency and throughput benchmarks on the Friedrich Schiller University's `omega` cluster[3]. Its nodes are each equipped with two quad core AMD Opteron 2350 processors, 16GB RAM and Mellanox InfiniBand SDR 8Gbit/s controllers. Open MPI 1.3.4[4] was used as the MPI implementation.

Table 2 shows the results of our tests. We have compared performance both for chars (as an example for primitive types) and a complex user defined type as it is used in our

[3]`http://www.uni-jena.de/omega.html`
[4]`http://www.open-mpi.org`

```
namespace MPI
{
    extern Datatype PHASEBORDER;
    extern Datatype COORDPHASEBORDERMAP;
    extern Datatype CELL;
}

class Typemaps
{
public:
    template<typename T>
    static inline MPI::Datatype lookup()
    {
        return lookup((T*)0);
    }

private:
    static MPI::Datatype
        generateMapPhaseBorder();
    static MPI::Datatype
        generateMapPhaseBorderMap();
    static MPI::Datatype
        generateMapCell();

    static inline MPI::Datatype lookup(bool*)
    ...

    static inline MPI::Datatype lookup(Cell*)
    {
        return MPI::CELL;
    }
};
```

Figure 2. Shortened header code as generated by TypemapGenerator for the classes seen in Fig. 4.

```
Cell cells[10];
MPI_Datatype type = Typemaps::lookup<Cell>();
MPI_Send(cells, 10, type, dest, tag,
    MPI_COMM_WORLD);
```

Figure 3. Retrieval of MPI datatypes from the code written by TypemapGenerator.

```
class PhaseBorder
{
    friend Typemaps;

    bool active;
    double value, temperature;
};

class PhaseborderMap
{
    friend Typemaps;

    PhaseBorder phaseBorders[2][2];
    double anisotropyAngle;
};

class Cell
{
    friend Typemaps;

    enum CellState {LIQUID, SOLID,
        SOLIDIFYING, MELTING};
    CellState state;
    double temperature, excessHeatShare;
    PhaseBorderMap phaseBorders;
    bool isEdgeCell;
};
```

Figure 4. Simplified simulation model of MuCluDent.

MuCluDent application (the type Cell and its members are sketched out in Figure 4). For these benchmarks Boost.MPI was used in conjunction with Boost.Serialization, which proved to be very slow for user defined datatypes, when compared with the other methods. There is no measurable overhead for Boost.MPI when sending primitive datatypes. The maximum bandwidth peaks at approximately 92% of the theoretical maximum of the InfiniBand link.

TPO++ is a bit slower for chars, but is by far the fastest when it comes to transmitting Cells. However, this comparison is misleading, since TPO++ did use MPI to simply transmit raw bytes and this method is therefore limited to homogeneous systems, making it unsuitable for our use case.

A native, MPI based implementation with MPI datatypes delivered by TypemapGenerator is the fastest for characters, but slower than TPO++ for Cells. This is because the MPI version we used packs and unpacks non-contiguous datatypes before transmission. The advantage of this method is that it does also work for heterogeneous grids and multi clusters. However, on a homogeneous system like the `omega` cluster, an MPI implementation could savely skip this additional work, thereby delivering better performance.

Finally, as hinted previously, Boost.MPI does achieve benchmark results equivalent to TypemapGenerator and TPO++, if the user flags the classes with the Boost macro `BOOST_IS_BITWISE_SERIALIZABLE()`. (Data not shown.) But that approach of course doesn't work on heterogeneous systems.

5 Summary

We have presented TypemapGenerator, a source code parser/generator to create MPI typemaps for user defined C/C++ datatypes. While it currently does not work with tem-

Table 2. Benchmark results of the serialization strategies. Round trip refers to a ping-pong transmission. It is roughly twice the latency. TG stands for TypemapGenerator.

	Round Trip (μs)		Bandwidth (MiB/s)	
Framework	Cell	char	Cell	char
TPO++	11.56	8.63	930.03	914.64
TG	11.93	7.58	366.74	917.89
Boost.MPI	1077.69	7.72	25.40	917.67

plate classes, it can, unlike previous work, handle both, partial typemaps and multi-dimensional arrays. It can be conveniently integrated into the build system and, once set up, operates fully automatically. Through its flexible architecture it can be extended to work with other languages, too. Benchmark results suggest that MPI datatypes are the most efficient and practical way to serialize user defined datatypes in a platform independent way.

References

[1] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. *SC Conference*, 0:1–11, 2008.

[2] M. P. I. Forum, editor. *MPI: A Message-Passing Interface Standard - Version 2.2.* High-Performance Computing Center Stuttgart, Stuttgart, Germany, 2009.

[3] D. Gregor and M. Troyer. Boost.MPI Library `http://www.boost.org`, 2008. [accessed 22-April-2010].

[4] T. Grundmann, M. Ritt, and W. Rosenstiel. TPO++: An Object-Oriented Message-Passing Library in C++. In *International Conference on Parallel Processing*, pages 43–50, 2000.

[5] 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*, pages 13–17, 2000.

[6] É. Renault. Extended MPICC to Generate MPI Derived Datatypes from C Datatypes Automatically. In F. Cappello, T. Hérault, and J. Dongarra, editors, *PVM/MPI*, volume 4757 of *LNCS*, pages 307–314. Springer, 2007.

[7] A. Schäfer, J. Erdmann, and D. Fey. Simulation of Dendritic Growth for Materials Science in Multi-Cluster Environments. In *Workshop Grid4TS*, volume 3, 2007.

[8] A. Schäfer and D. Fey. Libgeodecomp: A grid-enabled library for geometric decomposition codes. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 285–294, Berlin, Heidelberg, 2008. Springer.

[9] A. Schäfer and D. Fey. Pollarder: An Architecture Concept for Self-adapting Parallel Applications in Computational Science. In *Computational Science - ICCS*, volume 5101 of *LNCS*, pages 174–183. Springer, 2008.

[10] J. M. Squyres, B. McCandless, and A. Lumsdaine. Object Oriented MPI: A Class Library for the Message Passing Interface. In *Proceedings of the POOMA Conference*, 1996.