DaStGen—A Data Structure Generator for Parallel C++ HPC Software

Hans-Joachim Bungartz, Wolfgang Eckhardt, Miriam Mehl,
and Tobias Weinzierl

Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany
`weinzier@in.tum.de`
`http://www5.in.tum.de`

Abstract. Simulation codes often suffer from high memory requirements. This holds in particular if they are memory-bounded, and, with multicore systems coming up, the problem will become even worse as more and more cores have to share the memory connections. To optimise data structures with respect to memory manually is error-prone and cumbersome.

This paper presents the tool DaStGen translating classes declared in C++ syntax and augmented by new keywords into plain C++ code. The tool automates the record optimisation, as it analyses the potential range of each attribute, and as the user can restrict this range further. Herefrom, the generated code stores multiple attributes within one single primitive type. Furthermore, the tool derives user-defined MPI data types for each class. Using the tool reduces any algorithm's memory footprint, it speeds up memory-bounded applications such as CFD codes, and it hides technical details of MPI applications from the programmer.

1 Introduction

Writing software for the Computational Sciences is a task becoming more and more challenging: For several years, performance has been the only metric measuring the quality of software. Today, scientists insist on fast code that, in addition, is extendable, portable, maintainable and is delivered in time keeping step with the hardware's and algorithms' development. Furthermore, the software is to be embedded seamlessly into the whole simulation pipeline [10].

Many approaches have been developed to tackle these challenges: Problem solving environments [9] address the software's integration into the application landscape. Sophisticated language libraries such as expression templates [12] and well-engineered, standardised application programming interfaces such as MPI allow for an abstraction from hardware and optimisation details. Domain specific languages enable scientists and engineers to concentrate on modelling, and profiling tools and simulators [13] detect bottlenecks during the development. All these examples are chosen arbitrarily, and the list is to be continued.

In this paper, we focus on the question whether the object modelling facilities of C++ are satisfying for scientific software. Several observations can be made:

- The range of a variable’s value frequently is well-known in Scientific Computing. C++ cannot exploit this knowledge which results in a waste of memory: Two integers with range from 0 to 500, e.g., typically consume eight bytes although the whole information could be stored within two bytes.
- Many algorithms don’t have to store all their record’s attributes all the time. Often, some attributes are not needed for example in-between two iterations. In analogy to databases, we either call an attribute persistent or non-persistent if we want to be able to discard it. With C++, there is no way to define attributes that are sometimes not to be stored within containers depending on the object’s life cycle.
- Mapping C++ classes onto MPI data types is an error-prone task as the language has no built-in support for MPI and as developers often are experts in Mathematics, Physics or Engineering, e.g., and not that familiar with MPI interna.

During the development of a new CFD code called Peano [1], these issues came into play. They even influence considerably any realisation discussion up to now: CFD codes often are memory-bounded [4]. High memory requirements for objects that are processed each iteration are thus annoying. As we implement a domain decomposition [2], each developer has to map his records onto MPI data types. All the people involved have a strong mathematical and CFD background. To force them to think about memory layout, techniques saving individual bits and MPI specifics keeps people from their actual work and, sometimes, the resulting code does not meet high quality standards.

Thus, we extend the C++ programming language with new modelling directives and provide a code generator acting as preprocessor and mapping these extensions onto standard C++. The generator is called DaStGen (data structure generator). We implemented it as plain, lightweight, stand-alone tool in Java, i.e. it does not require any third-party components and is easy to extend and tailor. The parser front-end is realised with SableCC [3]. Using DaStGen reduces the amount of memory needed by the application, reduces the development time, and reduces the number of bugs typically introduced by programmers. In this paper, we present the generator and the language extension, as we believe that many C++ applications besides CFD codes can also benefit from the techniques implemented.

Hereby, we turn our attention to the primitive built-in C++ types apart from floating point numbers, and we discuss how to reduce the amount of memory required for them. We have not found a similar, as simple to use and lightweight tool for this issue so far. Also, we touch on the subject inheritance, but we do not take into account all the problems introduced by pointers and dynamic data structures. The idea to generate MPI data types automatically is not new. Actually, it has been discussed several times in more detail including dynamic data structures and pointers (see the tools MPIECC [11] or C++2MPI [5], e.g.). Nevertheless, our restricted but much simpler solution fits perfectly into the concept of a precompiler reducing memory requirements, our solution allows for

exchange of a subset of an object’s attributes, and the resulting code comes along without requiring for additional libraries and tools.

The remainder is organised as follows: First, we present the extension of the C++ class declaration directives, and we specify the mapping onto plain C++ (Sec. 2). Second, we give three small application examples extracted from the CFD code: classes for a solver for the Poisson equation, classes for an adaptive Cartesian grid management, and, finally, a combination of the two into a solver for the Poisson equation on adaptive Cartesian grids (Sec. 3). Third, we show the reduction of memory requirements, a source code extract proving the amount of manual work saved, as well as some preliminary runtime results (Sec. 4). Some concluding remarks in Sec. 5 close the discussion.

2 Specification

In DaStGen, the user defines classes’ attributes in C++ syntax. These attributes have to have a built-in type. Arrays are supported. The generator converts the specification into C++ code where the attributes are hidden and setter and getter operations are provided. Hereby, DaStGen also supports preprocessor macros and compile-time constants. The user just has to declare the latter ones with the keyword `Constant`: before they are used. Afterwards, one can for example define an attribute `double a[D]`, where `D` is a symbol defined when the generated code is compiled.

2.1 Boolean Attributes and Bit Fields

If an attribute’s type is boolean, C++ realises the attribute inefficiently in terms of memory, as the compiler maps the type onto a primitive data type such as byte [6]. The attributes of the instances of the class in Fig. 1, e.g., are mapped to a sequence of $2 + D$ primitives. If a boolean is stored in one byte, this results in a memory need of $2 + D$ bytes for information that could be encoded in $2 + D$ bits. DaStGen tracks all the booleans and arrays of booleans in a class, and stores the resulting information into one single primitive per class — the target bit field.

Fig. 1. Very simple class declaration (left) and corresponding target bit field storing all the $D + 2$ bit values (right). This target bit field would be used if all the attributes were marked as packed.

The type of the bit field is declared via **Packed-Type**: and it typically is a byte or an integer.

In the example in Fig. 1, the target bit field's first bit represents the value of attribute **a**, the bits 1 up to D represent **b**, and the $(D + 2)$ th bit holds **c**. The generated getter and setter extract and modify information within the bit field. They are implemented via fast bit-wise operations. Whether DaStGen packs data into the target bit field or maps it directly on a standard C++ attribute is controlled via the keyword **packed** for each attribute, i.e. the mapping is applied only if the keyword **packed** is written before the type of an attribute. The code in Fig. 1 lacks this keyword. Thus, it would be mapped 1:1 onto a C++ class.

2.2 Enumerations and Integers with Restricted Range

As enumerations are mapped onto numbers by any C++ compiler internally, the idea is obvious to take them into account for the bit packing, too. Otherwise, each enumeration with k variants needs at least one byte, although one could encode the information in $\lceil \log_2 k \rceil$ bits. Hence, DaStGen reserves a fitting number of bits within the target bit field for each enumeration type and stores the information there, if the enumeration attribute is marked by **packed**.

A similar reasoning holds for integers where the range is known a priori. DaStGen allows the programmer to append **from x to y** to any integer attribute marked with **packed** and, afterwards, uses only the actual number of bits required to store the value. This works for an arbitrary number of integers.

Fig. 2. The persistent attributes are stored within an embedded type of their own (left). Furthermore, all the packed attributes of type boolean, array of bit and integer augmented by a range are stored within one bit field. Right-hand side: Adaptive grid resolving a singularity for the Poisson equation.

2.3 Persistent Attributes

Characteristically for Scientific Computing, often not all the attributes of a class are valid throughout the complete object life cycle. For our PDE solver, e.g., we had temporary variables such as the local residual being important during an iteration but useless in-between two iterations. If one stores objects within a container (an array, e.g.), the size of the object determines the memory needed as the container does not know which attributes have semantics and which do not and, thus, are not to be stored. Memory is wasted.

Hence, we introduce the keyword pair **persistent** and **discard**. For the generated C++ class, DaStGen creates an embedded subclass holding only attributes marked as **persistent** (see Fig. 2). Depending on the programmer's needs, one can thus store the complete object or only this embedded type. Getters, setters and a constructor to read and modify the embedded type are created automatically.

2.4 Attributes Sent Via MPI

MPI offers the powerful mechanism of user-defined data types making MPI work with C++ objects. Nevertheless, to write the mapping between C++ classes and the MPI data types is error-prone, and the code has to be updated every time the data structure is modified. This task becomes cumbersome if several **#ifdef** require for different mappings to be written for different compile environments.

DaStGen automatically generates user-defined MPI data types for all **#ifdef** combinations. Similar to Sec. 2.3, users often want to communicate only a subset of the attributes in order to downsize the MPI messages. Thus, DaStGen introduces an additional keyword **parallelize** (or the equivalent **parallelise**) to mark attributes to be taken into account for communication via MPI.

Each generated class then has a public static attribute **Datatype** of type **MPI_Datatype**. The attribute can be passed to MPI operations like any built-in MPI type. Thus, the generated code does neither require any additional libraries or files, nor does DaStGen pose any restrictions on the type of MPI operations. The user can send and receive both (subsets of) objects and arrays of them. A serialisation of data connected by pointers is beyond our scope.

2.5 Inheritance and Hierarchy Flattening

The additional attribute markers of DaStGen allow for a significant reduction of object size by packing information into bit fields and distinguishing between attributes that are persistent and those that are not. Furthermore, DaStGen simplifies the communication via MPI. To facilitate the modelling of data structures, we find it useful to support inheritance. Yet, if the inheritance is mapped directly to C++ inheritance, this implies additional memory requirements to hold the inheritance relationship, and it keeps DaStGen from optimising the memory consumptions aggressively.

If a class holds one bit and one of several subclasses holds another bit, these two bits cannot be packed into one target bit field with standard C++ mechanisms. Although a workaround might be possible, we decided to introduce a flattening operator into DaStGen: Every time DaStGen is passed a (multiple) inheritance with the keyword `Extends:`, the involved classes are merged and the hierarchy is removed. This works for multiple levels of inheritance, too. Afterwards, the merged class is processed as a standard class modelled via DaStGen. The inheritance structure is thus lost in the generated code, but the memory consumption can be reduced. Furthermore, the generation of MPI datatypes works straightforward.

2.6 Name Transformations, MPI Aspects and C++ Internals

Some additional features such as a plugin mechanism to add aspects [7] to the generated code or to validate class and attribute names concerning coding conventions complete the generator. For the application examples, e.g., we provide a plugin adding `send()` and `receive()` operations to the classes, and adding `#pragma pack` directives to tune the memory layout to minimal memory requirements.

3 Application Example

The following three examples extracted from our CFD code Peano [1] demonstrate the DaStGen extensions in action. In the end, we will use them to measure the gain in memory efficiency in Sec. 4.

3.1 A Solver for the Poisson Equation

For a simple solver for the Poisson equation arising from our CFD code's pressure computation, we modelled the unknowns of the PDE as follows:

```
Packed-Type: short int;
class poisson::PoissonDoF {
    enum DoFState { UNDEF, INNER, DIRICHLET };
    persistent packed DoFState state;
    persistent double rhs;
    persistent parallelise double value;
    discard parallelise double residual;
    discard long int vertexNumber;
};
```

Throughout the solver iteration, the local residual is needed for computations and the degree of freedom's number of is required by the visualisation interface. Both values are not to be stored in between iterations. For parallelisation, the solver implements a domain decomposition. Hereby, only the unknown's actual value and the residual have to be interchanged as all the other values can be determined locally.

3.2 An Adaptive Grid Management

Besides the solver, we implemented a grid management for adaptive Cartesian grids [8]. Here, the basic element is the grid vertex holding all local structural information such as refinement properties, refinement status and the information if the vertex is a hanging node. Information on the vertex's position and refinement level is only stored or communicated in the debug mode. Thus, debugging becomes easier.

```
Constant: DIMENSIONS;
class grid::Vertex {
    enum RefinementControl {
        UNREFINED, REFINED, REF_TRIGGERED, REFINING, COARS_TRIGGERED, COARSENING
    };
    persistent packed bool isPersistentVertex;
    persistent parallelise packed RefinementControl refinementControl;
    persistent parallelise packed bool refinementData[DIMENSIONS];
#ifdef Debug
    persistent parallelise packed int level;
    persistent double x[DIMENSIONS];
#endif
};
```

3.3 An Adaptive Poisson Solver

Finally, we combined the code for solving the Poisson equation with the adaptive grid management. The class fitting to this combination has to hold the attributes of both the Poisson solver's and the adaptive grid management's class. To avoid code duplication, we introduced a new vertex class inheriting from the two classes. Hereby, `PoissonDoF.def` and `Vertex.def` are the file names the examples from above are written to.

```
Extends: PoissonDoF.def;
Extends: Vertex.def;
class poisson::PoissonVertex {};
```

4 Results

This section evaluates the influence of the DaStGen features on our CFD code. The experiments were conducted on a Pentium 4, 3.4 GHz processor with 2 GByte RAM. They yield three interesting observations: First, due to packing the bits and splitting up the data into persistent and not persistent attributes, the amount of memory required per object is reduced by a factor of two or more (Table 1). The memory reduction effect especially benefits from the inheritance flattening: The size of `PoissonVertex` is smaller than `Vertex`'s size added `PoissonDoF`'s size.

Second, there's no runtime breakdown to be observed although the getter and setter for the packed records now incorporate several bit shift operations

Table 1. Number of bytes required with different features from Sec. 2 enabled. d represents the constant DIMENSIONS. Memory alignment is switched off.

	poisson PoissonDoF	grid Vertex d=2	poisson PoissonVertex d=2	grid Vertex d=3	poisson PoissonVertex d=3
debug mode	36	19	55	20	56
1:1 mapping	36	7	43	8	44
persistent	20	7	27	8	28
packed	34	2	34	2	34
persistent and packed	18	2	18	2	18
memory improvement	2	3.5	2, 38	4	2, 44

(Fig. 3). Further investigations are necessary, but the result suggests that the amount of additional work to be done is compensated by the speedup due to the reduction of data to be transferred by the memory bus. This does not hold if one artificially increases the computational work to be done per instance, i.e. this property only holds for memory-bounded algorithms.

Fig. 3. Runtime per object instance for 1:1 mapping and the version generated by DaStGen. Measurements result from the two-dimensional case.

Finally, the source code fragment in Fig. 4 shows how many technical details for the mapping of C++ classes onto MPI data types can be hidden from the developer due to the usage of the generator.


```

const int Attributes = 3;
MPI_Datatype subtypes[Attributes] =
{MPI_DOUBLE, MPI_DOUBLE, MPI_UB};
int blocklen[Attributes] =
{1, 1, 1};
MPI_Aint disp[Attributes];
PoissonDoF dummy[2];
MPI_Aint base;
MPI_Address(
    &(dummy[0]),
    &base);
MPI_Address(
    &(dummy[0]._persistent_records._value),
    &disp[0]);
MPI_Address(
    &(dummy[0]._residual),
    &disp[1]);
MPI_Address(
    &(dummy[1]._persistent_records._value),
    &disp[2]);
for (int i=0; i<Attributes; i++)
    disp[i] -= base;
MPI_Type_struct(
    Attributes, blocklen, disp,
    subtypes, &PoissonDoF::Datatype);
MPI_Type_commit(
    &PoissonDoF::Datatype);

```

Fig. 4. Source code extract from the generated MPI data type declaration (left). Typical grid for our CFD experiments (right).

5 Concluding Remarks

DaStGen is a small but powerful tool to map augmented C++ classes to a memory efficient realisation supporting MPI in plain C++. The resulting code requires less memory than a standard implementation, and it runs faster when the application is memory-bounded. Although writing the efficient implementation manually is not of great difficulty for an experienced programmer, using DaStGen is of great value: It reduces the coding time significantly, it reduces the number of bugs in pure technical parts of the code, it allows the programmer to focus on algorithms and semantics instead of the realisation, and it enhances the readability and reusability of the model.

With all the multicore systems coming up, we believe that the performance impact of the records' size will become more and more important. Throughout the discussion of the record size, we have concentrated on integers and enumerations. Yet, the bigger part of a simulation's memory footprint typically stems from floating point numbers. How to apply a packing to these is not clear a priori. Furthermore, it is questionable whether the performance would benefit, as no equivalent for the fast bit-wise operations is available, as many architectures handle floats completely different compared to integers (they are not held in the L1-cache, e.g.), and as floating point operations last long. Although DaStGen's valuable keyword `persistent` is independent of this reasoning, further research on record size reduction might be profitable.

Right now, getting rid of technical details enables faster rapid application prototyping, and, due to simple modelling facilities, people are more willing to

evaluate different algorithmic approaches based upon different record layouts. From a technical point of view, the parallel extension of any application's part becomes simpler. Hence, the tool shows how scientific code—especially memory-bounded code—benefits from application and domain specific language extensions allowing for sophisticated modelling, efficient memory usage and aspect oriented programming.

Acknowledgements. This work has partially been funded by DFG's research unit FOR493, the DFG project HA 1517/25-1/2 and TUM's International Graduate School of Science and Engineering (IGSSE) as part of the Excellence Initiative of the German Federal and State Governments.

References

1. Brenk, M., Bungartz, H.-J., Mehl, M., Muntean, I.L., Neckel, T., Weinzierl, T.: Numerical Simulation of Particle Transport in a Drift Ratchet. *SIAM Journal of Scientific Computing* (2008) (accepted)
2. Bungartz, H.-J., Mehl, M., Weinzierl, T.: A Parallel Adaptive Cartesian PDE Solver Using Space-Filling Curves. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) *Euro-Par 2006*. LNCS, vol. 4128, pp. 1064–1074. Springer, Heidelberg (2006)
3. Gagnon, E., Hendren, L.J.: SableCC, an Object-Oriented Compiler Framework. In: *TOOLS 1998: Proceedings of the Technology of Object-Oriented Languages and Systems*, pp. 140–154. IEEE Computer Society, Los Alamitos (1998)
4. Gropp, W., Kaushik, D., Keyes, D., Smith, B.: Towards Realistic Performance Bounds for Implicit CFD Codes. In: *Proceedings of Parallel CFD 1999*, pp. 241–248. Elsevier, Amsterdam (1999)
5. Hillson, R., Iglewski, M.: C++2MPI: A Software Tool for Automatically Generating MPI Datatypes from C++ Classes. In: *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, pp. 13–17. IEEE Computer Society, Los Alamitos (2000)
6. Hyde, R.: Write Great Code - vol. 2: Thinking Low-Level. In: *Writing High-Level*. No Starch Press (2004)
7. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) *ECOOP 1997*. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
8. Mehl, M., Weinzierl, T., Zenger, C.: A cache-oblivious self-adaptive full multigrid method. *Numerical Linear Algebra With Applications* 13(2-3), 275–291 (2006)
9. Parker, S.G., Weinstein, D.M., Johnson, C.R.: The SCIRun computational steering software system. *Modern Software Tools in Scientific Computing*, 1–40 (1997)
10. President's Information Technology Advisory Committee: *Computational Science: Ensuring America's Competitiveness* (2005)
11. Renault, É.: Extended MPICC to Generate MPI Derived Datatypes from C Datatypes Automatically. In: Cappello, F., Herault, T., Dongarra, J. (eds.) *PVM/MPI 2007*. LNCS, vol. 4757, pp. 307–314. Springer, Heidelberg (2007)
12. Veldhuizen, T.: Expression Templates. *C++ Report* 7(5), 26–31 (1995)
13. Weidendorfer, J., Kowarschik, M., Trinitis, C.: A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) *ICCS 2004*. LNCS, vol. 3038, pp. 440–447. Springer, Heidelberg (2004)