Automatic Datatype Generation and Optimization

Fredrik Kjolstad[1]
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
fred@csail.mit.edu

Torsten Hoefler Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
{htor,snir}@illinois.edu

Abstract

Many high performance applications spend considerable time packing noncontiguous data into contiguous communication buffers. MPI Datatypes provide an alternative by describing noncontiguous data layouts. This allows sophisticated hardware to retrieve data directly from application data structures. However, packing codes in real-world applications are often complex and specifying equivalent datatypes is difficult, time-consuming, and error prone. We present an algorithm that automates the transformation. We have implemented the algorithm in a tool that transforms packing code to MPI Datatypes, and evaluated it by transforming 90 packing codes from the NAS Parallel Benchmarks. The transformation allows easy porting of applications to new machines that benefit from datatypes, thus improving programmer productivity.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming, Parallel programming

General Terms Performance, Design

Keywords MPI, Datatypes, Compiler Technique, Refactoring

1. Introduction

Data movement is the most expensive operation in modern high performance systems. It is therefore essential that we move data as few times as possible. MPI Datatypes are objects that describe arbitrary data layouts. Examples include strided and indexed data. Datatypes let us perform operations such as send, receive, put and write on non-contiguous data with a single call to the runtime.

If the programmer does not use datatypes when he sends noncontiguous data then he must either send it as multiple messages or first copy it to a contiguous buffer. Since it is expensive to send small messages, programmers typically opt for the copy. Such copy code is called packing code, because it packs data into a contiguous buffer. Datatypes allow the programmer to specify non-contiguous sends in one operation, *without* first performing a copy pass.

Traditional MPI systems pack data specified by datatypes into contiguous buffers, thereby forfeiting the performance gains from the removal of application packing code. However, modern network hardware such as InfiniBand provide support for transferring non-contiguous data (scatter/gather). In fact, communication performance improvements of 4.8x have been demonstrated, provided datatypes are specified [5]. Given this performance improvement potential it is no surprise that emerging systems provide increasingly rich datatype support.

```
double buffer[N * 3];
for(int i=0; i < N; i++) {
  buffer[3*i]   = grid[i][0][0];
  buffer[3*i+1] = grid[i][0][1];
  buffer[3*i+2] = grid[i][0][2];
}
MPI_Send(buffer, N * 3, MPI_DOUBLE, left, tag, comm);
```

```
MPI_Datatype vec_t;
MPI_Type_vector(N, 3, N * 3, MPI_DOUBLE, &vec_t);
MPI_Type_commit(&vec_t);
MPI_Send(&grid[0][0][0], 1, vec_t, left, tag, comm);
```

Figure 1. Border exchange communication kernel before and after transformation from packing code to MPI Datatypes.

We present a novel algorithm that transforms packing code to optimized datatypes. The algorithm converts packing code to an internal datatype representation that is optimized through a number of passes before it is used to rewrite the application. The presentation assumes C and MPI, but the techniques generalize to other environments. For evaluation purposes, we implemented the algorithm as an Eclipse CDT refactoring tool. We then used the tool to transform 90 packing codes from the NAS Parallel Benchmarks [2]. Although we chose to implement the transformation in a refactoring tool, it is equally applicable in a compiler setting. For a detailed description of the algorithm including each optimization step, a full evaluation, discussion and two case studies we refer the reader to the accompanying technical report [3]. The tool, experimental data and additional case studies have been made available online [1].

1.1 Motivating Example

Figure 1 shows a very simple 2D left border exchange example before and after it has been transformed from packing code to MPI Datatype code. The example is similar to one of the packing codes in the NAS LU benchmark. The grid array stores three values for each cell of a two dimensional domain. Both kernels send the left border of the grid, which consist of noncontiguous strided data. The first kernel packs the column into a contiguous buffer. The second kernel uses a vector datatype.

1.2 Related Work

Several research groups have investigated techniques for *datatype generation*. These are exemplified by Tansey & Tilevich's tool for generating datatypes for C++ classes [6]. Our approach differs as it transforms packing code to datatypes. Other researchers have explored techniques to improve *datatype performance*. In one strand, researchers optimize runtime datatype parsing and packing code generation [4]. In a second strand, researchers have explored hardware and software support for non-contiguous transfers, achieving large communication speedups such as 4.8x [5]. Our approach statically optimize datatypes to reduce their runtime cost.

[1] The author did his work while he was at the University of Illinois.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

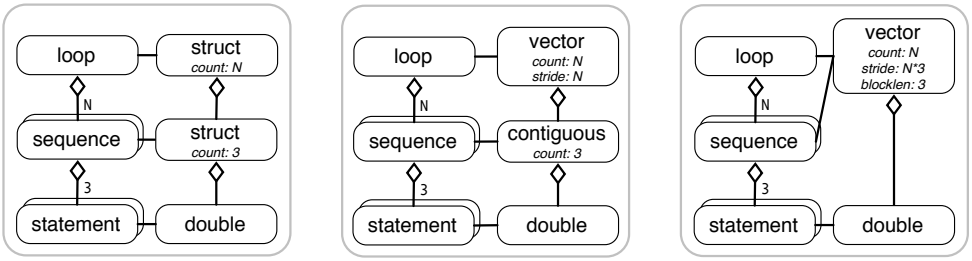

Figure 2. Equivalent Datatype IRs for the figure 1 example: initial IR (left), after specialization (center) and after compression (right).

2. Efficient Datatype Generation

Our algorithm for datatype generation and optimization is centered on an intermediate representation we call the Datatype IR. The Datatype IR captures the important (hierarchical) constructs of the packing code—packing statements, loops and sequences—and links them to equivalent datatypes. Figure 2 contains three Datatype IR examples. Our algorithm supports a number of datatypes. Some of these, listed in order of strictly decreasing generality, are struct, hindexed, hvector, vector and contiguous (see [3] for details). Note that most packing codes can be represented by different equivalent datatypes. For example, the three Datatype IRs in figure 2 are equivalent. We say a datatype representation is more efficient if it is expressed in terms of more specific datatypes, as these require fewer arguments in their construction. That is, a contiguous type can be expressed using fewer arguments than an indexed type. Our goal is therefore not merely to find a datatype that is equivalent to a packing construct, but to find one that is efficient. We achieve this by optimizing the datatype representation.

The Datatype IR can be constructed from any imperative language with packing code, such as C/C++/Fortran with MPI, or UPC. Two preconditions must hold before Datatype IR can be constructed in the current version: (1) the packing code block must only contain nested loops, assignments and if statements, and (2) the code block must write to consecutive locations in the packing buffer. Datatype IR construction requires identifying packing statements and then summarize loop nests containing packing statements, and sequences of packing statements and loops. Every packing statement, packing sequence and packing loop becomes part of a packing group. Once packing groups are summarized, a struct datatype is created to represent each composite group (loops and sequences) and a primitive datatype is set to represent each packing statement. Since struct datatypes can represent any data layout, the initial structs are equivalent to the packing code. The left box in figure 2 shows the initial IR for the example from figure 1.

Once the Datatype IR has been constructed our algorithm performs a number of optimization passes. These fall into two categories: specialization and compression. We currently support four specialization and three compression optimizations. Note that the Datatype IR allows optimization passes to be cleanly expressed in a language-independent manner. The optimizations are summarized below and detailed descriptions, as well as suggestions for additional optimizations, are provided in the technical report [3].

The specialization optimizations passes form a chain that successively specialize datatypes as much as possible. Let $\rightarrow$ denote a specialization pass, then the four specialization optimizations are struct $\rightarrow$ hindexed $\rightarrow$ hvector $\rightarrow$ vector $\rightarrow$ contiguous. That is, each optimization specializes one datatype to the next, provided specialization preconditions are met. Thus, depending on the access pattern of a given packing group, the specialization chain may be aborted at any point. For example, in the center IR in figure 2 the sequence was specialized to a contiguous type, while the loop could only be specialized to a vector. The distribution of the datatypes that replace the 90 packing codes in the NAS Parallel Benchmarks are given in the technical report [3].

Next, three compression optimization passes are performed to clean up the IR and hence the resulting datatype code. These are: (1) compress contiguous into parent block length, (2) merge struct and hindexed types, and (3) compress contiguous into send count. For example, the right IR in figure 1 depicts the center IR after compression optimizations have been applied.

The final step is to emit the datatypes as code. Note that in a compiler setting the target language can be different from the source language. For example, the source language could be UPC while the target language could be C with MPI. Hence it is possible to convert packing code to datatypes and emit these, even if the source language does not support datatypes. In a refactoring tool or a source to source compiler for C with MPI, datatype emit involves three stages. First, the datatype construction code is emitted (optionally with lazy initialization code). For hvector, vector and contiguous types it is sufficient to emit the datatype definition and constructor, as shown in figure 1. For struct and hindexed types we must also emit code that packs the indices. Second, the packing code consumer (e.g. a send call) must be identified and rewritten to use the new datatype. Third, a dead code elimination pass must be run to remove unused packing code.

3. Conclusions

Transfers of noncontiguous data are prevalent in many parallel codes; efficient scatter-gather capabilities that optimize such transfers and reduce the memory traffic generated by such transfers will be essential in future HPC architectures. We expect that future high performance network interfaces and memory controllers will have enhanced scatter-gather capabilities. This will enable direct transfer of noncontiguous data from memory to the network interface or to the CPU. Datatypes provide a means of communication between the executing code and a smart scatter-gather engine. The transformations we outlined will facilitate this communication—as part of a compiler, a refactoring tool or a run-time capability.

References

[1] Automatic datatype generation tool download. URL `http://people.csail.mit.edu/fred/datatypes`. Last Accessed 12/17/2011.

[2] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical report, NASA, 1995.

[3] F. Kjolstad, T. Hoefler, and M. Snir. A transformation to convert packing code to compact datatypes for efficient zero-copy data transfer. Technical report, University of Illinois at Urbana-Champaign, 2011.

[4] R. Ross, N. Miller, and W. Gropp. Implementing fast and reusable datatype processing. In *EuroPVM/MPI*, 2003.

[5] G. Santhanaraman, J. Wu, and D. K. Panda. Zero-copy MPI derived datatype communication over InfiniBand. In *EuroPVM/MPI*, 2004.

[6] W. Tansey and E. Tilevich. Efficient automated marshaling of C++ data structures for MPI applications. In *IPDPS*, 2008.