Automatic Datatype Generation and Optimization

Fredrik Kjolstad
CSAIL
Massachusetts Institute of Technology

Torsten Hoefler
Department of Computer Science
University of Illinois at Urbana-Champaign

Marc Snir

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.

Contributions
• Algorithm that converts packing code to datatypes
• Example implementation of the algorithm in an interactive retooling tool
• Study of efficiency of compact versus non-compact datatypes and regularity of packing codes in the NAS Parallel Benchmarks

Borderline Exchange Example

<table>
<thead>
<tr>
<th>Packing</th>
<th>Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact</td>
<td>Efficient</td>
</tr>
<tr>
<td>Unoptimized</td>
<td>Optimized</td>
</tr>
</tbody>
</table>

MPI Datatype Background
• Datatypes are used to send/receive data and to store/retrieve data from files
• Basic datatypes such as MPI_DOUBLE match C++/Fortran primitives
• Derived datatypes describe advanced data layout such as striped and indexed
• Derived datatypes can be composed to describe arbitrary layouts

Constructors
Datatypes are initialized using predefined constructors. For example:

```c
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype newtype);
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype newtype);
int MPI_Type_create_struct(int count, int array_of_block_lengths[], MPI_Datatype array_of_types[], MPI_Datatype oldtype, MPI_Datatype newtype);
```

Vector example
We send every second element of an array `data[N]`. We can use a vector of doubles with a stride of 2. The stride of vectors is specified in number of elements:

```c
MPI_Type_vector(N, 1, 2, MPI_DOUBLE, &vec_t);
MPI_Send(data, 1, MPI_DOUBLE, tag, COMM);
```

Intermediate code

<table>
<thead>
<tr>
<th>Code Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vec_t</code></td>
<td>MPI_DATATYPE_NULL</td>
</tr>
<tr>
<td><code>vec_t = MPI_DOUBLE;</code></td>
<td></td>
</tr>
<tr>
<td><code>blocklen1[3];</code></td>
<td></td>
</tr>
<tr>
<td><code>displacements1[3];</code></td>
<td></td>
</tr>
<tr>
<td><code>struct1_t;</code></td>
<td></td>
</tr>
<tr>
<td><code>(&amp;grid[0][0], &amp;first_addr);</code></td>
<td></td>
</tr>
<tr>
<td><code>(&amp;grid[i][0][0], &amp;displacements1[0]);</code></td>
<td></td>
</tr>
<tr>
<td><code>MPI_Type_create_struct(3, blocklen1, displacements1, types1, &amp;struct1_t);</code></td>
<td></td>
</tr>
<tr>
<td><code>MPI_Type_commit(&amp;struct1_t);</code></td>
<td></td>
</tr>
</tbody>
</table>

Preconditions
1. The code block consists of: nested loops, assignments and conditional statements (if statements)
2. The code block writes to consecutive locations in the packing buffer

Lazy Datatype Construction Boilerplate

```c
MPI_Datatype vec_t;
MPI_Type_vector(N, 1, 2, MPI_DOUBLE, &vec_t);
MPI_Send(data, 1, MPI_DOUBLE, tag, COMM);
```

Compact = Efficient Datatypes

Full application run-times on an problem set B on Jargon Communication is only a small fraction of total time.

Optimization Overview

For each datatype, bottom up

- Specialize to hindexed
- Specialize to<vector
- Specialize to contiguous
- Merge structs and indexed types
- Compress contiguous into parent block length

Example: Specialize to hindexed

```c
MPI_Datatype vec_t;
MPI_Type_vector(N, 1, 2, MPI_DOUBLE, &vec_t);
MPI_Send(data, 1, MPI_DOUBLE, tag, COMM);
```

Fast Communication = Datatypes in Future

<table>
<thead>
<tr>
<th>Environment</th>
<th>Approach</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>MPI sends non-contiguous remote locations</td>
<td>4.8x</td>
</tr>
<tr>
<td>Byna et al. [20]</td>
<td>Intel based non-contiguous channel communication</td>
<td>2.5x</td>
</tr>
<tr>
<td>Wosnack et al. [29]</td>
<td>SCI non-contiguous copy to global memory</td>
<td>6.8x</td>
</tr>
<tr>
<td>Tanabe and Nakajo [22]</td>
<td>DDR2 support for non-contiguous RDMA</td>
<td>6.8x</td>
</tr>
</tbody>
</table>

90 datatypes generated from the NAS Parallel Benchmarks

NPB contains 90 packing/unpacking codes
• Our algorithm can convert 60-65% out of the box
• In the remaining 30 cases precondition 2 is not met
  • The code must pack to consecutive buffer locations
  • In each case the precondition can easily be established through a simple loop split (can be automated)

Datatype Generation Refactoring Tool

38 unpacking codes in NPB were mapped to Irecv statements. In these cases our implementation generated the datatype, but the programmer must rewrite the hrecv to use them.

Datatype Generation Refactoring Tool

```
```

Gojun et al., developed a pre-processor tool called AutoMap that evaluates 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.