Automatic Datatype Generation and Optimization

Fredrik Kjolstad
CSAIL
Massachusetts Institute of Technology

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

Abstract

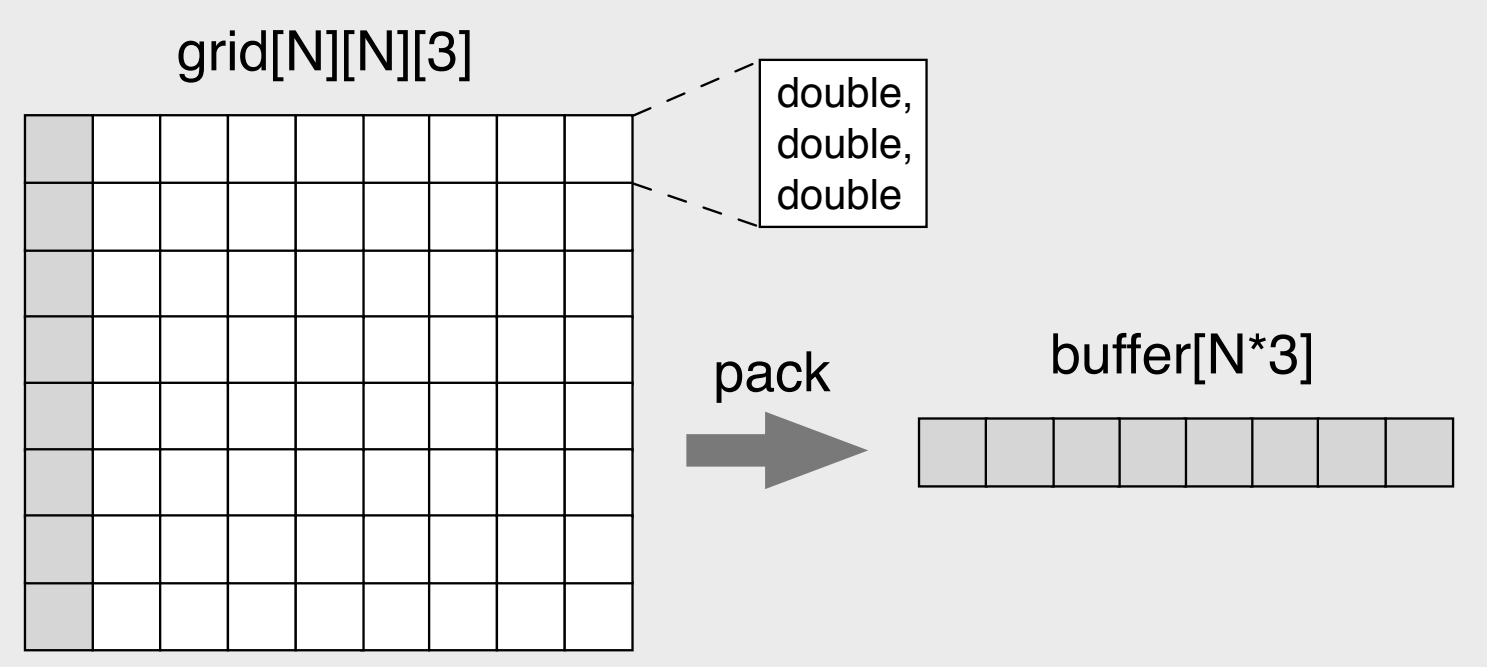
Many high performance applications spend considerable time packing noncontiguous data into contiguous communication buffers. MPI Datatypes provide an alternative by describing noncontiguous data layouts. This allows sophisticated hardware to retrieve data directly from application data structures. However, 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 refactoring tool
- Study of efficiency of compact versus non-compact datatypes and regularity of packing codes in the NAS Parallel Benchmarks

Border Exchange Example

```
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, 0, tag, COMM);
```



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/C++/Fortran primitives
- Derived datatypes describe advanced data layout such as strided and indexed
- Derived datatypes can be composed to describe arbitrary layouts

Constructors

Datatypes are initialized using predefined constructors. For example:

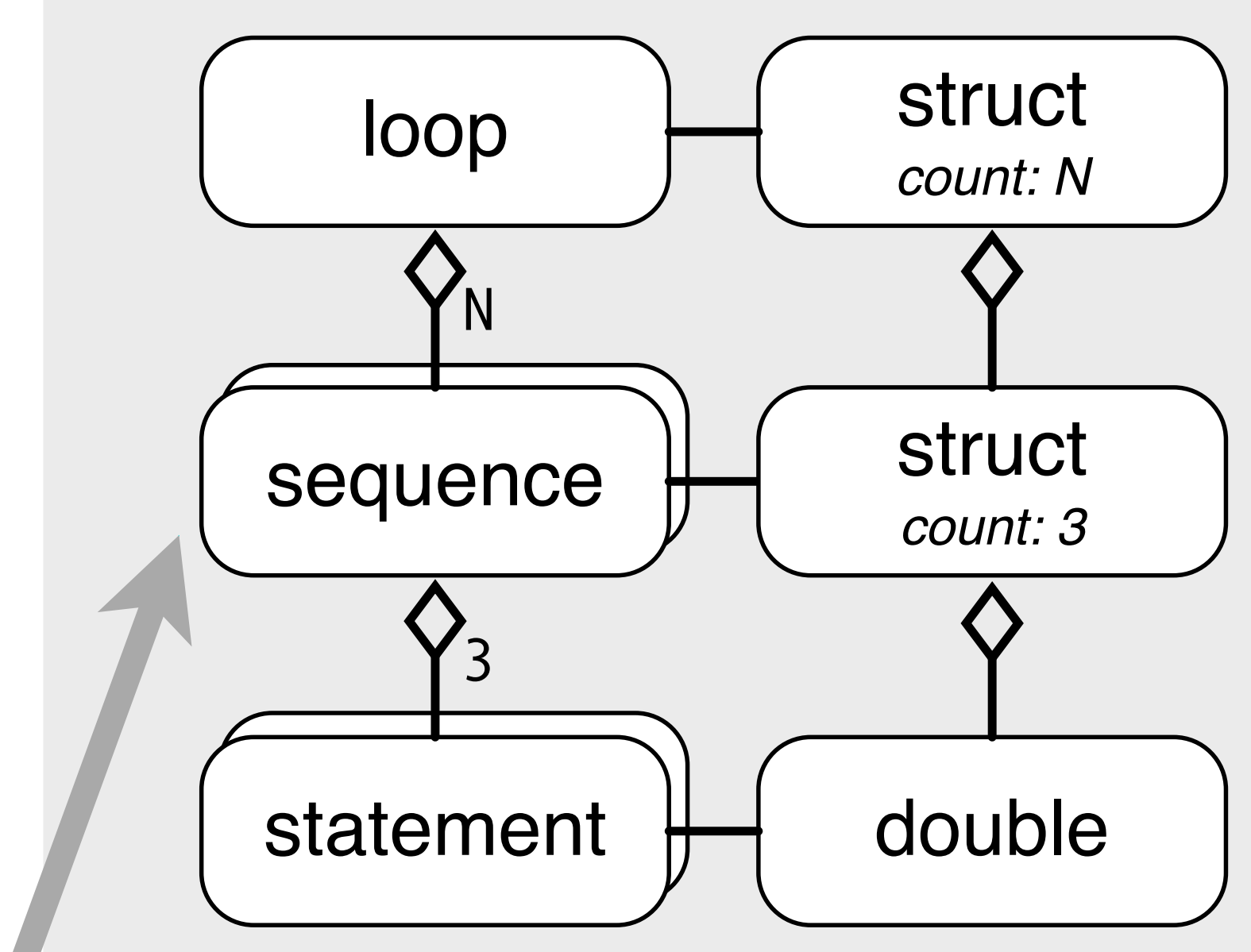
```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype);

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_Aint array_of_displacements[],
                           MPI_Datatype array_of_types[],
                           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:

```
MPI_Datatype vector;
MPI_Type_vector(N/2, 1, 2, MPI_DOUBLE, &vector);
MPI_Type_commit(&vector);
MPI_Send(data, 1, vector, dest, tag, MPI_COMM_WORLD);
```

Generate IR



Intermediate code

```
MPI_Datatype struct_t;
MPI_Aint displacements[N];
int blocklen[N];
MPI_Datatype types[N];
MPI_Aint first_addr;
MPI_Get_address(&grid[0][0], &first_addr);
for(int i=0; i<N; i++) {
  MPI_Datatype struct1_t;
  MPI_Aint displacements1[3];
  int blocklen1[3];
  MPI_Datatype types1[3];
  MPI_Aint first_addr1;
  MPI_Get_address(&grid[i][0][0], &first_addr1);
  MPI_Get_address(&grid[i][0][0], &displacements1[0]);
  displacements1[0] -= first_addr1;
  blocklen1[0] = 1;
  types1[0] = MPI_DOUBLE;
  MPI_Get_address(&grid[i][0][1], &displacements1[1]);
  displacements1[1] -= first_addr1;
  blocklen1[1] = 1;
  types1[1] = MPI_DOUBLE;
  MPI_Get_address(&grid[i][0][2], &displacements1[2]);
  displacements1[2] -= first_addr1;
  blocklen1[2] = 1;
  types1[2] = MPI_DOUBLE;
  MPI_Type_create_struct(3, blocklen1,
    displacements1, types1, &struct1_t);

  displacements[i] = first_addr1 - first_addr;
  blocklen[i] = 1;
  types[i] = struct1_t;
}
MPI_Type_create_struct(N, blocklen, displacements,
  types, &struct_t);
MPI_Type_commit(&struct_t);
MPI_Send(&grid[0][0][0], 1, struct_t, 0, tag, COMM);
```

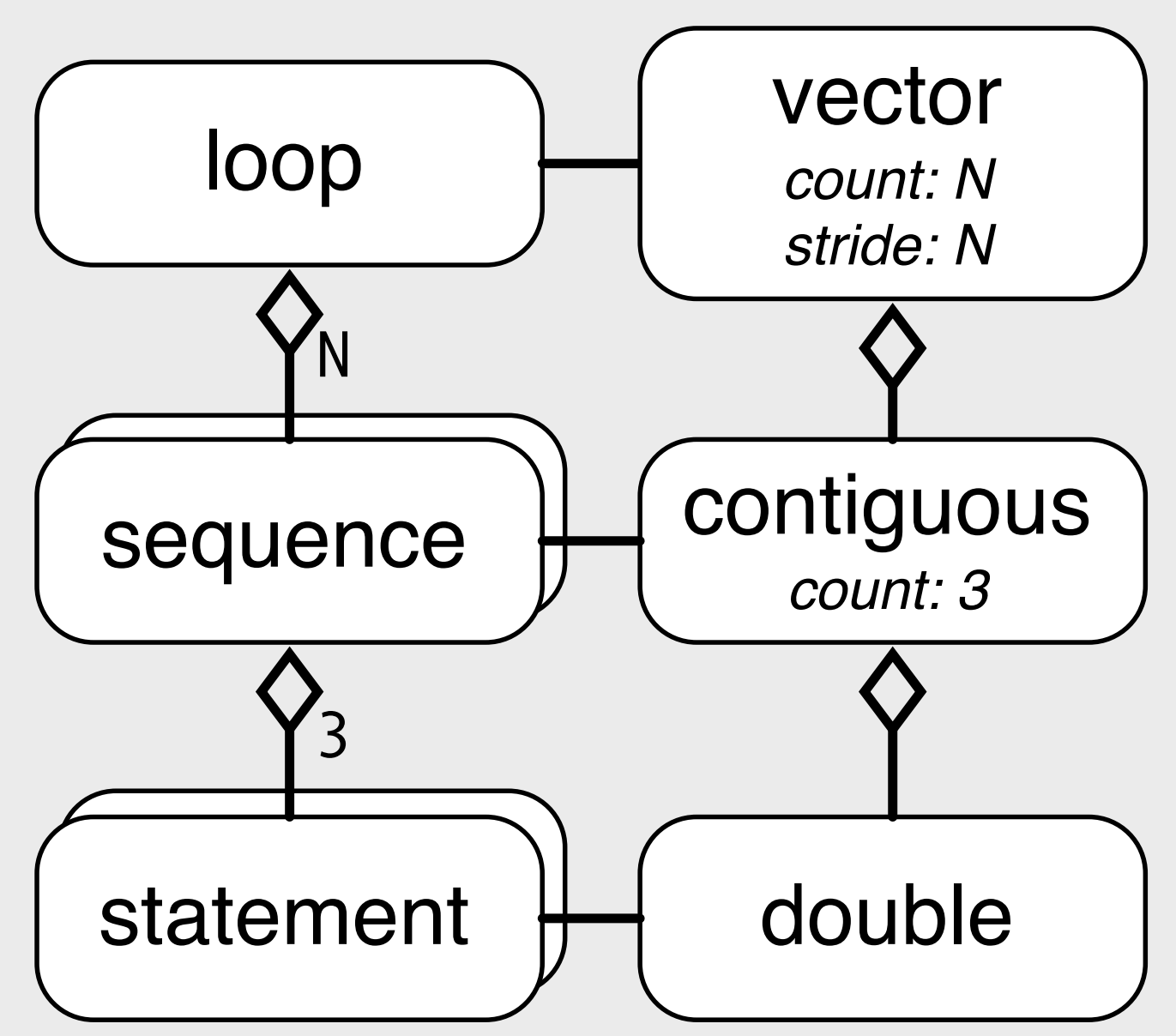
Preconditions

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

Lazy Datatype Construction Boilerplate

```
static MPI_Datatype vec_t;
static int init = 0;
static int _N;
if (N != _N || !init) {
  if (init) MPI_Type_free(&vec_t);
  init = 1;
  _N = N;
  MPI_Type_vector(N, 3, N * 3, MPI_DOUBLE, &vec_t);
  MPI_Type_commit(&vec_t);
}
```

Specialize

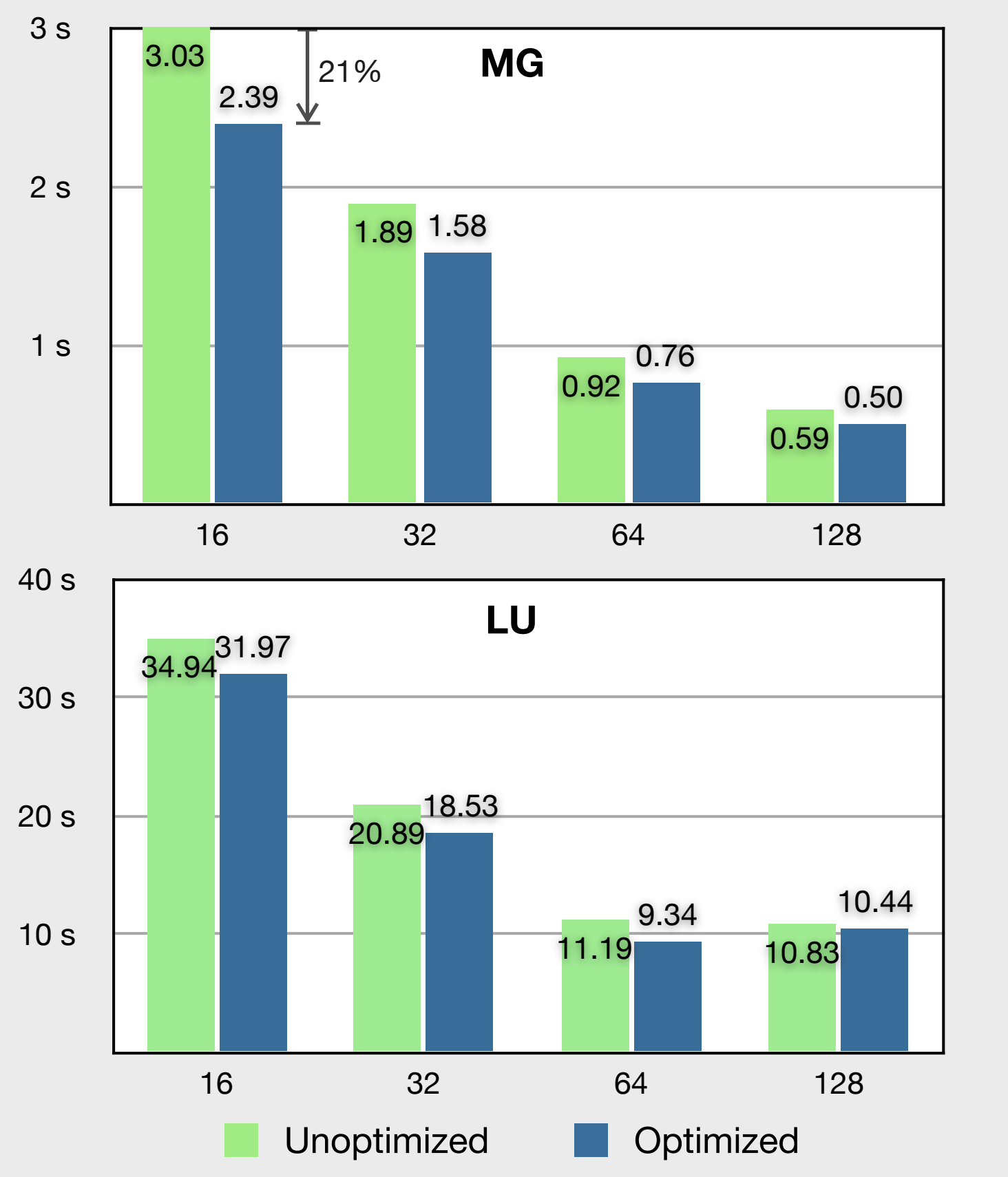


Intermediate code

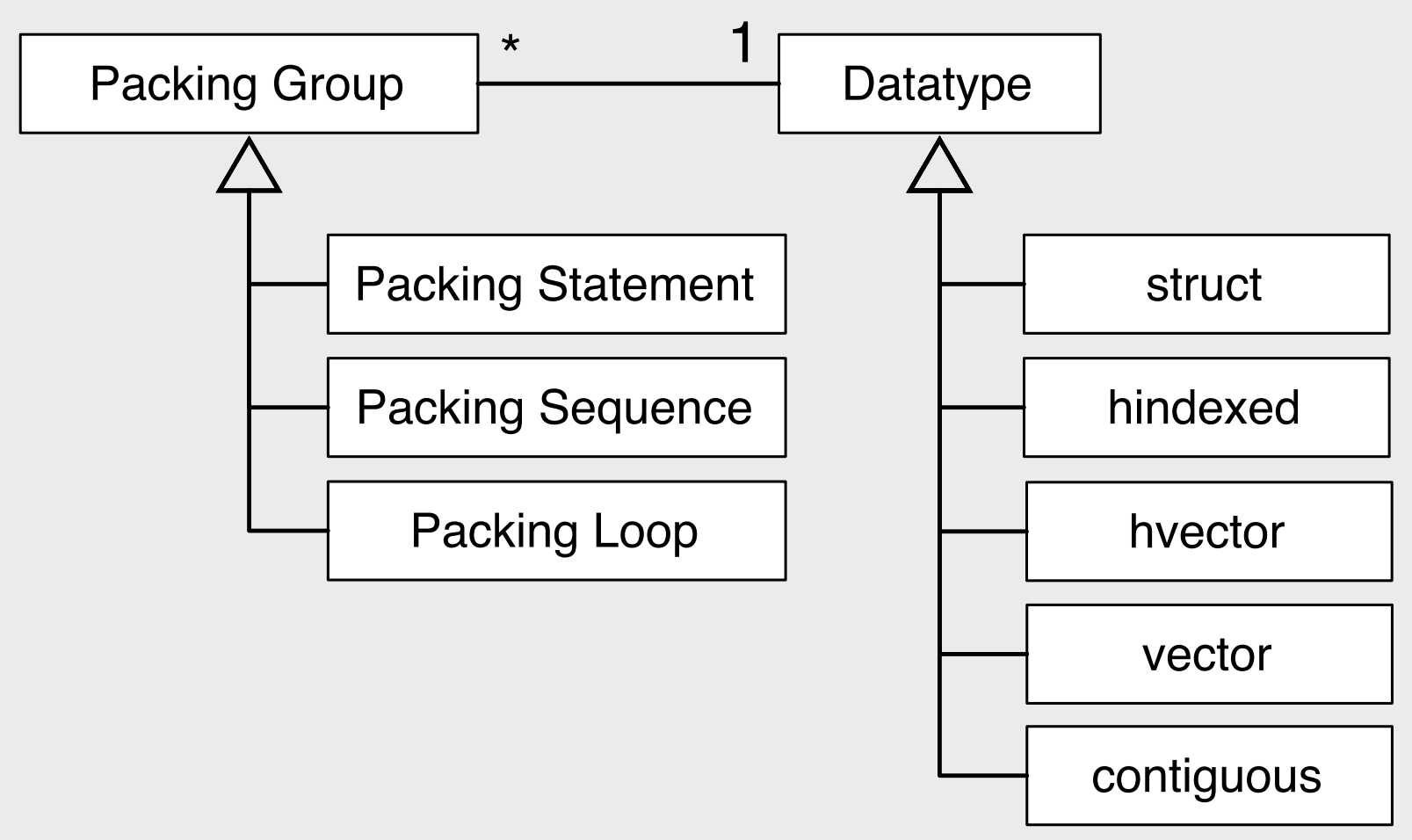
```
MPI_Datatype cont_t;
MPI_Type_contiguous(3, MPI_DOUBLE, &cont_t);
MPI_Datatype vec_t;
MPI_Type_vector(N, 1, N, cont_t, &vec_t);
MPI_Type_commit(&vec_t);
MPI_Send(&grid[0][0][0], 1, vec_t, 0, tag, COMM);
```

Compact = Efficient Datatypes

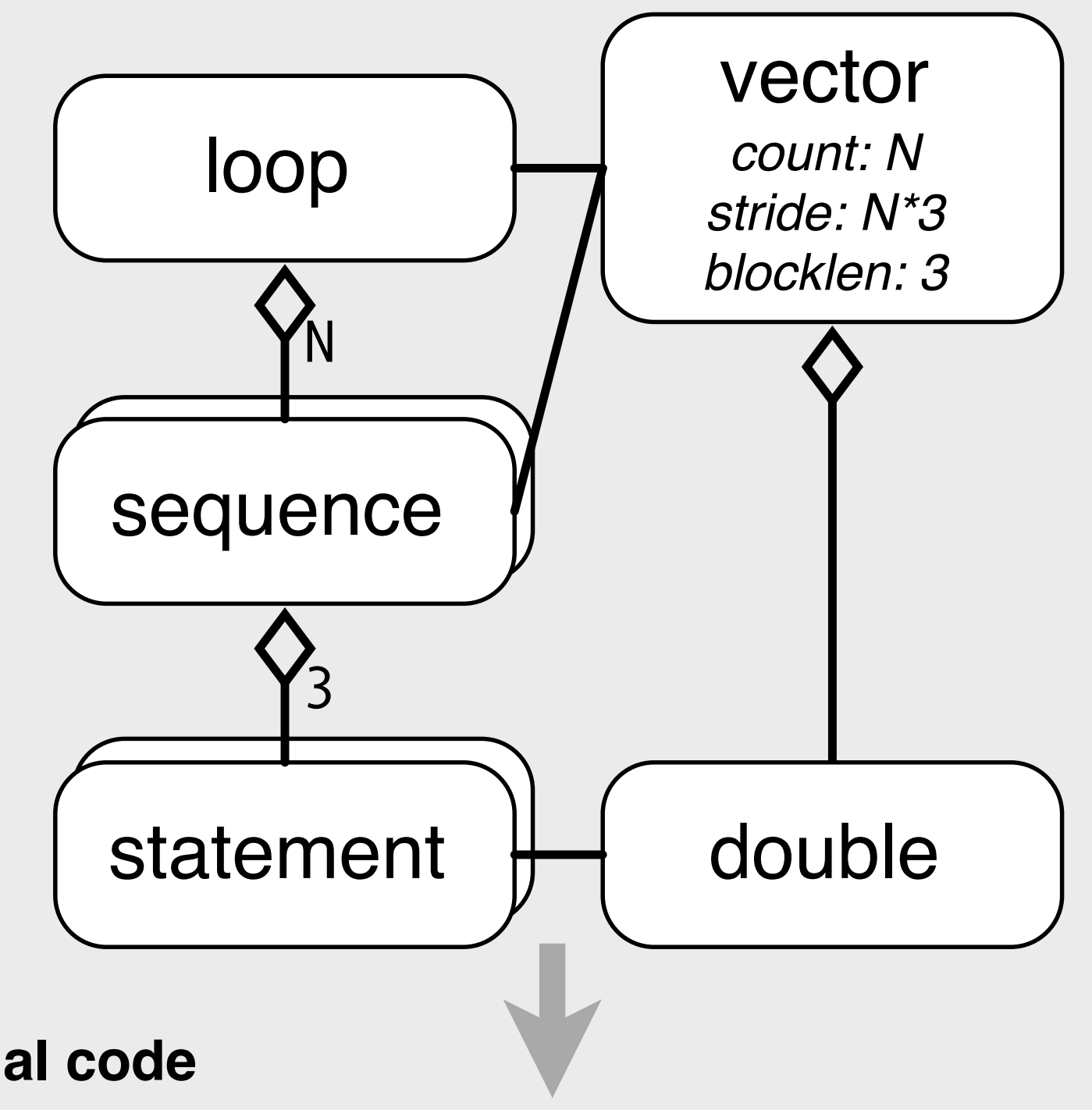
Full application run-times on problem set B on Jaguar. Communication is only a small fraction of total time.



Datatype IR



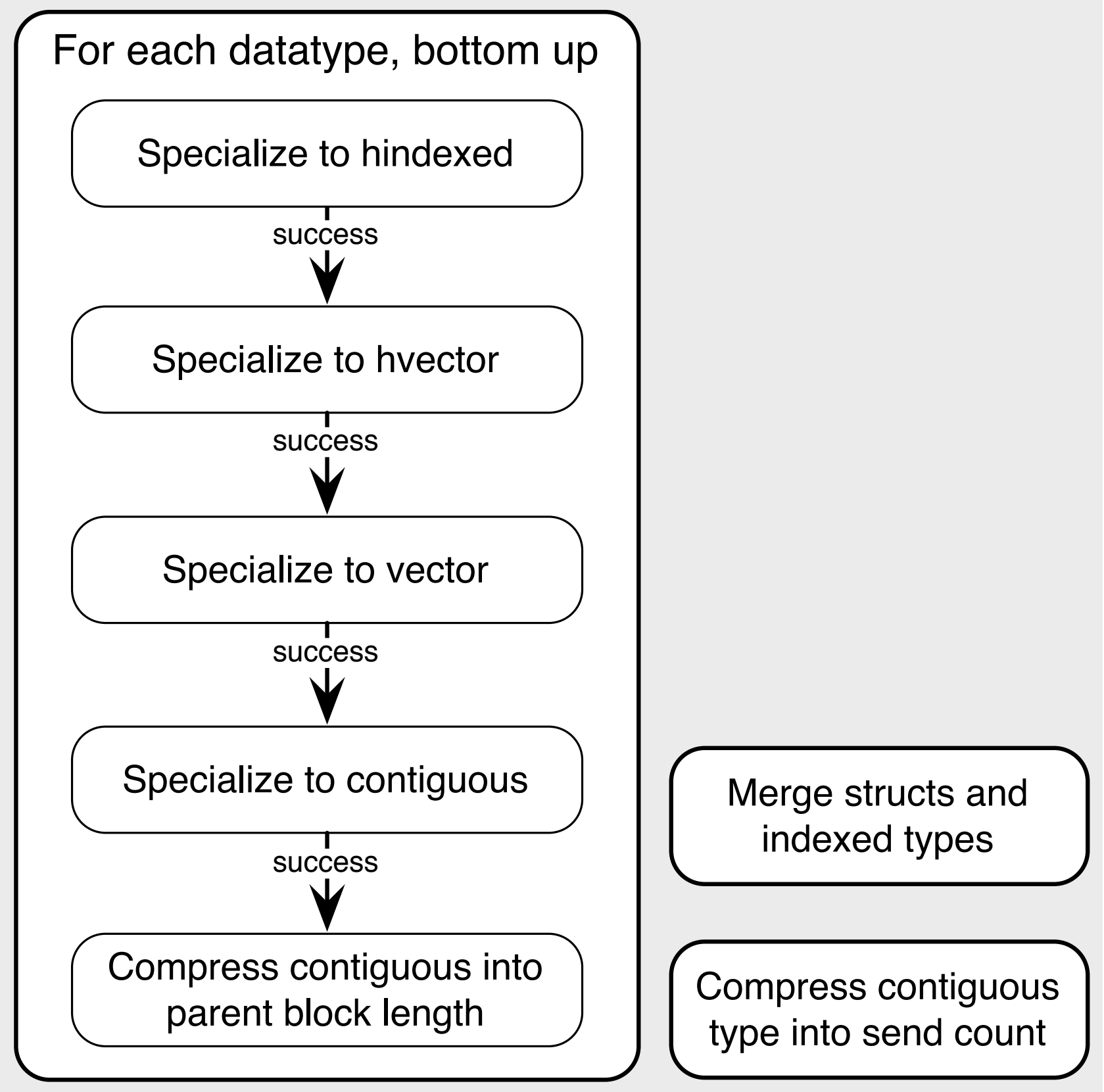
Compress



Final code

```
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, 0, tag, COMM);
```

Optimization Overview


For each datatype, bottom up
Specialize to hindexed
success
Specialize to hvector
success
Specialize to vector
success
Specialize to contiguous
success
Compress contiguous into parent block length
Merge structs and indexed types
Compress contiguous type into send count

Example: Specialize to hvector

1. Every packing statement must access the same source array, struct or scalar variable
2. All block lengths must be the same size
3. The packing group must not contain a conditional
4. There must be a fixed distance between the memory locations accessed by every pair of consecutive packing statements

Requires reasoning about loop induction variables and expression simplification:

$$((3Ni+1)-(3Ni),(3Ni+2)-(3Ni+1))=(1,1)$$

$$(3Ni|\text{stride of } i \text{ is } 1)=3N$$

Fast Communication = Datatypes in Future

Researchers	Approach	Speedup
Wu, et al. [26]	Infiniband non-contiguous remote load/store	3.6x
Santhanaraman, et al. [20]	Infiniband non-contiguous channel communication	4.8x
Worringen, et al. [25]	SCI non-contiguous copy to global memory	2.1x
Tanabe and Nakajo [22]	DIMMnet-2 support for non-contiguous RDMA	6.8x

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

[22] N. Tanabe and H. Nakajo. Acceleration for MPI derived datatypes using an enhancer of memory and network. In *IPDPS Workshops*, 2010.

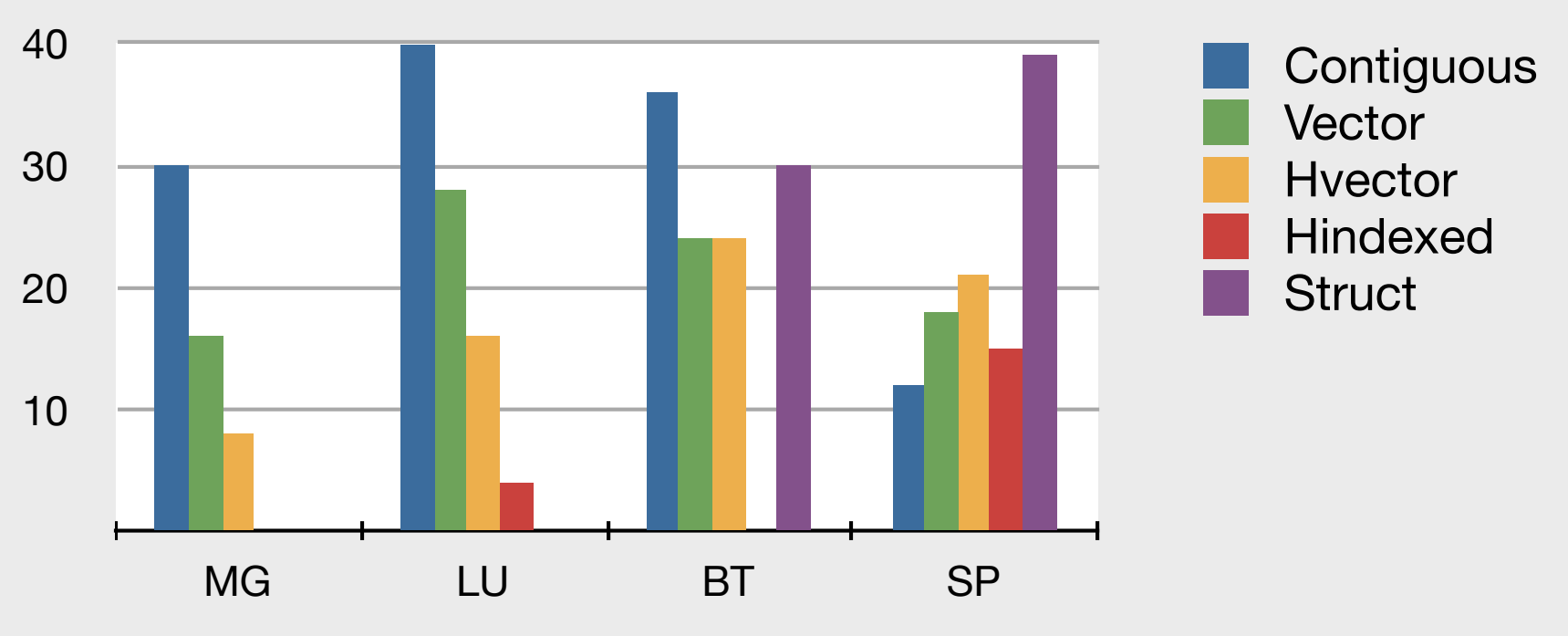
[25] J. Worringen, A. Gaer, and F. Reker. Exploiting transparent remote memory access for non-contiguous and one-sided-communication. In *Workshop on Communication Architecture for Clusters*, 2002.

[26] J. Wu, P. Wyckoff, and D. Panda. High performance implementation of MPI derived datatype communication over infiniband. In *IPDPS*, 2004.

90 datatypes generated from the NAS Parallel Benchmarks

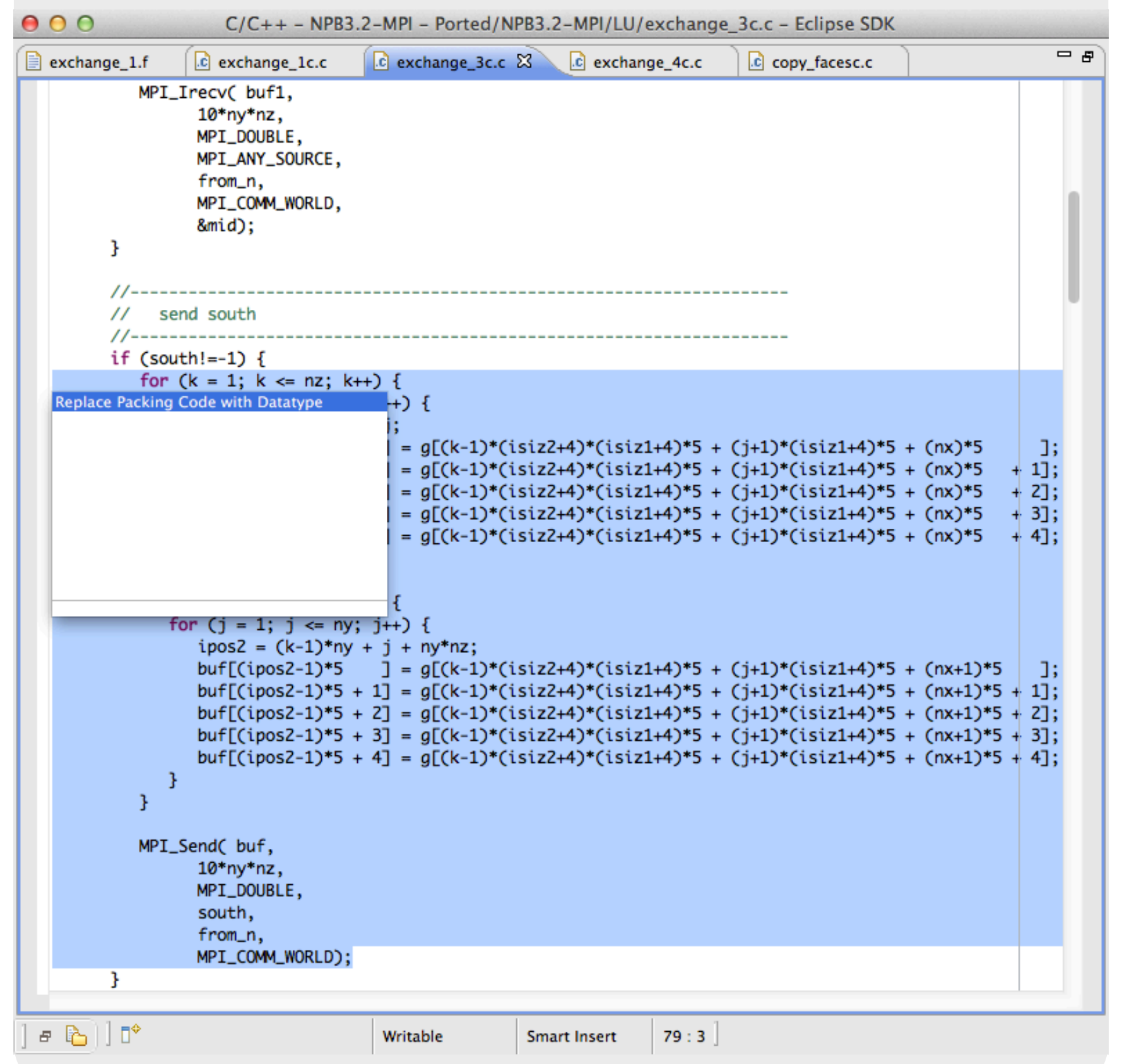
- NPB contains 90 packing/unpacking codes
- Our algorithm can convert 60 (67%) out of the box
- In the remaining 30 cases precondition 2 is not met
  - The code must pack to consecutive buffer locations
- In each case the precondition can easily be established through *a single loop split* (can be automated)

Datatype IR Nodes



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

Datatype Generation Refactoring Tool


C/C++ – NPB3.2-MPI – Ported/NPB3.2-MPI/LU/exchange_3c.c – Eclipse SDK
exchange_1.f
exchange_1c.c
exchange_3c.c
exchange_4c.c
copy_facesc.c
Replace Packing Code with Datatype
Writable
Smart Insert
79 : 3

```
    MPI_Irecv( buf1,
               10*ny*nz,
               MPI_DOUBLE,
               MPI_ANY_SOURCE,
               from_n,
               MPI_COMM_WORLD,
               &mid);
  }

  //---------------------------------------------------------------------
  //   send south
  //---------------------------------------------------------------------
  if (south!=-1) {
    for (k = 1; k <= nz; k++) {
                                    +) {
                                    ;
                                    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5     ];
                                    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5   + 1];
                                    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5   + 2];
                                    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5   + 3];
                                    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx)*5   + 4];


                                    {
      for (j = 1; j <= ny; j++) {
        ipos2 = (k-1)*ny + j + ny*nz;
        buf[(ipos2-1)*5    ] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5    ];
        buf[(ipos2-1)*5 + 1] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 1];
        buf[(ipos2-1)*5 + 2] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 2];
        buf[(ipos2-1)*5 + 3] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 3];
        buf[(ipos2-1)*5 + 4] = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (j+1)*(isiz1+4)*5 + (nx+1)*5 + 4];
      }
    }

    MPI_Send( buf,
              10*ny*nz,
              MPI_DOUBLE,
              south,
              from_n,
              MPI_COMM_WORLD);
  }
```