Fortran packing code from the file exchange_1.f in the NAS LU Benchmark, opened in Eclipse. The selected code shows a simple packing loop that packs the south border of the g array into the dum packing buffer. g is a 4-dimensional array that represents 3-dimensional space. The inner dimension is used to store five values per cell.
The packing code was manually ported to C, as our tool does not currently support Fortran. Note that the 2- and 4-dimensional arrays are flattened to 1-dimensional arrays, with one term indexing into each dimension.
We invoke the Quick Assist feature of Eclipse by pressing ctrl+1 (cmd+1). The quick assist menu is displayed showing context-sensitive transformation suggestions for the selected code.

We select “Replace Packing Code with Datatype” from the quick assist menu, and the tool analyses and transforms the packing code to datatype code using our algorithm. The analysis and transformation takes less than 0.5s and therefore feels instantaneous.
The tool converts the packing code to a vector datatype. The count (arg1) was derived from the loop bounds, and the stride (arg3) was derived from the arithmetic progression of the loop index j.

The tool also created an intermediate contiguous datatype that represented the five packing statements inside the packing loop, but this type could be merged into the vector’s block length (arg2) before code was emitted.
We will now perform a similar transformation for the unpacking code for the east border exchange.
if( iex == 2 ) {
    if( south != -1 ) {
        MPI_Type_vector(1 + jend + -1 * jst, 5, 20 + 5 * isiz1, MPI_DOUBLE, &vec_t);
        MPI_Type_commit(&vec_t);
        MPI_Send(&g[5 + 5 * nx + 5 * (1 + jst) * (4 + isiz1) + 5 * (-1 + k) * (4 + isiz1) * (4 + isiz2)],
                 MPI_Type_free(&vec_t);
    }
}
if( east != -1 ) {
    for(i=ist; i <= iend; ++i) {
        dum[i-1]*5 = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (ny+1)*(isiz1+4)*5 + (i+1)*5 ];
        dum[i-1]*5 + 1 = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (ny+1)*(isiz1+4)*5 + (i+1)*5 + 1];
        dum[i-1]*5 + 2 = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (ny+1)*(isiz1+4)*5 + (i+1)*5 + 2];
        dum[i-1]*5 + 3 = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (ny+1)*(isiz1+4)*5 + (i+1)*5 + 3];
        dum[i-1]*5 + 4 = g[(k-1)*(isiz2+4)*(isiz1+4)*5 + (ny+1)*(isiz1+4)*5 + (i+1)*5 + 4];
    }
}
This time the tool could convert both the five packing statements inside the packing loop, and the packing loop to contiguous datatypes. It could merge these and fold them into the MPI_Recv’s count argument, thereby removing the packing code without introducing any datatypes. This is because the packing code packed completely contiguous data.
We will now look at two more complex examples from the file exchange_3.f in the LU benchmark. The first example is the south border exchange code selected above.

One precondition of our algorithm is that data is packed into contiguous location, but this is not the case here as the first five and last five packing statements in the loop packs into non-contiguous parts of buf.
The contiguous packing precondition can easily be established by the programmer (or a separate tool) by splitting the nested loops.

In the rest of this demo we will assume the programmer has already performed required splits and will not show these.
MPI_Recv( 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++) {
    for (j = 1; j <= ny; j++) {
      ipos2 = (k-1)*ny + j + ny*nz;
      buf[(npos2-1)*5 ] = g[(k-1)*(isz2+4)*(isz1+4)*5 + (j+1)*(isz1+4)*5 + (nx)*5 ];
      buf[(npos2-1)*5 + 1] = g[(k-1)*(isz2+4)*(isz1+4)*5 + (j+1)*(isz1+4)*5 + (nx)*5 + 1];
      buf[(npos2-1)*5 + 2] = g[(k-1)*(isz2+4)*(isz1+4)*5 + (j+1)*(isz1+4)*5 + (nx)*5 + 2];
      buf[(npos2-1)*5 + 3] = g[(k-1)*(isz2+4)*(isz1+4)*5 + (j+1)*(isz1+4)*5 + (nx)*5 + 3];
      buf[(npos2-1)*5 + 4] = g[(k-1)*(isz2+4)*(isz1+4)*5 + (j+1)*(isz1+4)*5 + (nx)*5 + 4];
    }
  }
}

MPI_Send( buf,
    10*ny*nz,
    MPI_DOUBLE,
    south,
    from_n,
    MPI_COMM_WORLD);
MPI_Irecv(buf1,
          10*ny*nz,
          MPI_DOUBLE,
          MPI_ANY_SOURCE,
          from_n,
          MPI_COMM_WORLD,
          &mid);

// send south

if (south!=-1) {
  MPI_Datatype vec_t;
  MPI_Type_vector(ny, 5, 20 + 5 * isiz1, MPI_DOUBLE, &vec_t);
  MPI_Datatype hvec_t;
  MPI_Type_create_hvector(nz, 1, 5 * (4 + isiz1) * (4 + isiz2) * sizeof(double), vec_t, &hvec_t);
  MPI_Datatype hvec_t;
  MPI_Type_create_hvector(2, 1, 5 * sizeof(double), hvec_t, &hvec_t);
  MPI_Type_commit(&hvec_t);
  MPI_Send(&g[40 + 10 * isiz1 + 5 * nx], 1, hvec_t, south, from_n, MPI_COMM_WORLD);
  MPI_Type_free(&hvec_t);
}

// receive from north

if (north!=-1) {
  MPI_Wait(&mid, &status);

  for (k = 1; k <= nz; k++) {
    for (j = 1; j <= ny; j++) {
      ipos1 = (k-1)*ny + j;
      g[(k-1)*(isiz2+4) + (isiz1+4)*5 + (j+1)*(isiz1+4)*5] = buf1[(ipos1-1)*5];
      g[(k-1)*(isiz2+4) + (isiz1+4)*5 + (j+1)*(isiz1+4)*5 + 1] = buf1[(ipos1-1)*5 + 1];
      g[(k-1)*(isiz2+4) + (isiz1+4)*5 + (j+1)*(isiz1+4)*5 + 2] = buf1[(ipos1-1)*5 + 2];
      g[(k-1)*(isiz2+4) + (isiz1+4)*5 + (j+1)*(isiz1+4)*5 + 3] = buf1[(ipos1-1)*5 + 3];
      g[(k-1)*(isiz2+4) + (isiz1+4)*5 + (j+1)*(isiz1+4)*5 + 4] = buf1[(ipos1-1)*5 + 4];
    }
  }
  for (k = 1; k <= nz; k++) {
We will now transform unpacking code of the receive side of the north border exchange (selected) to use a datatype.
The north border exchange receive shows a common pattern in the NAS Parallel Benchmarks. An async receive is posted using MPI_Irecv (top), followed by a send that is overlapped with the receive (middle), followed by a wait (selected) and then unpacking code (bottom).

Our tool does not currently recognize this pattern, so some work is required by the programmer. This demonstrates the power of an interactive tool when automated transformations overlap with manual transformations.
The programmer first selects the packing code. Note that the receive is not selected, as it comes before the ‘send south’ block.
When quick assist is invoked while packing code, but no send/receive, is selected our tool provides the programmer with two options. One option to convert packing code and one to convert unpacking code to a datatype. The programmer knows that this is unpacking code, and selects that option.
The unpacking code is automatically converted to a datatype. Since no MPI_Recv/MPI_Irecv was selected, our tool could not rewrite it and instead provides an MPI_Recv stub with the three arguments that must be changed in the receive statement.
The MPI_Irecv is shown on the top, and the corresponding wait and datatype is shown at the bottom.
The programmer copies the datatype code to the receive site, leaving the MPI_Wait in place.
if (north!=-1) {
  MPI_Datatype vec_t;
  MPI_Type_vector(ny, 5, 20 + 5 * isiz1, MPI_DOUBLE, &vec_t);
  MPI_Datatype hvec1_t;
  MPI_Type_create_hvector(nz, 1, 5 * (4 + isiz1) * (4 + isiz2) * sizeof (double), vec_t, &hvec1_t);
  MPI_Datatype hvec_t;
  MPI_Type_create_hvector(2, 1, 5 * sizeof (double), hvec1_t, &hvec_t);
  MPI_Type_commit(&hvec_t);
  // MPI_Recv(&g[40 + 10 * isiz1], 1, hvec_t, NA, NA, NA, NA, MPI_COMM_WORLD);
  MPI_Irecv(buf1,
             10*ny*nz,
             MPI_DOUBLE,
             MPI_ANY_SOURCE,
             from_n,
             MPI_COMM_WORLD,
             &mid);
  MPI_Type_free(&hvec_t);
}

// send south
//
if (south!=-1) {
  MPI_Datatype vec_t;
  MPI_Type_vector(ny, 5, 20 + 5 * isiz1, MPI_DOUBLE, &vec_t);
  MPI_Datatype hvec1_t;
  MPI_Type_create_hvector(nz, 1, 5 * (4 + isiz1) * (4 + isiz2) * sizeof (double), vec_t, &hvec1_t);
  MPI_Datatype hvec_t;
  MPI_Type_create_hvector(2, 1, 5 * sizeof (double), hvec1_t, &hvec_t);
  MPI_Type_commit(&hvec_t);
  MPI_Send(&g[40 + 10 * isiz1 + 5 * nx], 1, hvec_t, south, from_n, MPI_COMM_WORLD);
  MPI_Type_free(&hvec_t);
}

// receive from north
//
if (north!=-1) {
  MPI_Wait(&mid, &status);
}
The programmer must now change the first three arguments of the MPI_Irecv constructor, and can do so by copying the first three argument from the MPI_Recv stub provided by the tool.
This completes our north border exchange receive transformation. This transformation required some work by the programmer, but the most time-consuming operation by far, the conversion of the receive unpacking code to a datatype, was still automated.
We will now demonstrate how our tool handles packing code that packs from more than one array.
// receive from south
if (ifin1==nx) {
  MPI_Irecv( dum,
             2*ny2,
             MPI_DOUBLE,
             MPI_ANY_SOURCE,
             from_s,
             MPI_COMM_WORLD,
             &msgid1);

  MPI_Wait( &msgid1, &status );

  for (j = 0; j <= ny+1; j++) {
    g[j*(isiz2+2) + nx+1] = dum[j ];
  }
  for (j = 0; j <= ny+1; j++) {
    h[j*(isiz2+2) + nx+1] = dum[j+ny2];
  }
}

// send north
//
if (ibeg==1) {
  for (j = 0; j <= ny+1; j++) {
    g[j*(isiz2+2) + nx+1] = dum[j ];
  }
  for (j = 0; j <= ny+1; j++) {
    h[j*(isiz2+2) + nx+1] = dum[j+ny2];
  }
}

The two packing loops could be represented by the same vector type (vec_t). The two packing loops packed data from two arrays (g and h), and our tool represented this as an hindexed datatype with one index per loop datatype.
In the final example we demonstrate how loops that contain if statements are converted to datatypes.

Note that (1) the outer loop contains an if statement, and (2) the loop index variable of the outer loop (c) is used to determine the loop bounds of the inner loops.
for (c = 1; c <= ncells; c++) {
    for (k = 0; k < cell_size[(c-1)*3 + 2]-1; k++) {
        for (j = 0; j < cell_size[(c-1)*3 + 1]-1; j++) {
            for (i = 0; i < 1; i++) {
                for (m = 1; m < 5; m++) {
                    out_buffer[ss[1]+p1-1] = u[(c-1)*(KMAX+4)*(JMAX+4)*(IMAX+4)*5 + (k+2)*(JMAX+4)*(IMAX+4)];
                }
            }
        }
    }
}
}
Since the outer loop contained an if statement it could not be converted to a vector datatype. Furthermore, since the vector representing the four inner loops depends on the c loop index variable, the vector constructor can not be hoisted from the loop. The outer loop is therefore represented by a struct datatype, and a new vector datatype is constructed for each entry.