Enabling Code_Saturne for Multi-Petaflop/Exascale with MPI 3.0 one sided Communication

Chandan Basu, Soon-Heum Ko, Charles Moulinec, Yvan Fournier

Abstract

Code_Saturne is a popular open-source computational fluid dynamics package. We have carried out a study of applying MPI 2.0 / MPI 3.0 one sided communication routines to Code_Saturne and its impact on improving the scalability of the code for future peta/exascaling. We have developed modified versions of the halo exchange routine in Code_Saturne. Our modifications showed that MPI 2.0 one sided calls give some speed improvement and less memory overhead compared to the original version. The MPI 3.0 version on the other hand is unstable and could not run.

Application Code: Code_Saturne

1. Introduction

The computational capabilities of large supercomputers are increasing rapidly and are expected to reach exascale range by year 2020 [1]. This development raises significant challenge for the Computational Fluid Dynamics (CFD) applications for utilizing such resources. Code_Saturne [2] is a well known open-source code for solving Navier-Stokes equations in 2D, 2D-axisymmetric and 3D flows. It can handle steady or unsteady, laminar or turbulent, incompressible, compressible or weakly dilatable, isothermal or non-isothermal cases. Code_Saturne has been chosen for the short list of software gathered in the PRACE Benchmark Suite as one of the two CFD application tools [3] and extensive work has been undertaken within PRACE [5-6] to optimize Code_Saturne.

Code_Saturne is an MPI parallelized code. Its scalability depends on the MPI communication efficiency and how loaded each MPI task is. In the newly released MPI 3.0 standard [7] many new features have been introduced for better scalability of codes, e.g., improved remote memory access (RMA) one-sided communication routines, non-blocking collectives. This is expected to greatly enhance the scalability and performance of MPI-3.0 compared to MPI-2.0 [8].

Substantial communication overhead of Code_Saturne comes from halo exchange between neighbouring subdomains/processors. Halo exchange in Code_Saturne is implemented by MPI_Isend, MPI_Irecv and MPI_Wait routines. In this project we have investigated the impact of replacing these MPI point to point routines with MPI 2.0 / MPI 3.0 one sided routines. The lower overhead associated with one-sided communication as compared to two-sided communication has the potential to increase the performance at peta/exascale by increasing the effective network bandwidth and reducing synchronization overheads.

2. Profiling of Code_Saturne

Code_Saturne is a very large software package and it is difficult to know at first which routine is the most suitable for applying MPI one sided communication. For this purpose we have done profiling of the code to
identify communication intensive routines. This has given us an idea about which routines are using MPI point-to-point calls and what their overhead is.

Profiling with TAU & gprof

Profiling tools of gprof and TAU [9] have been used for finding the hot-spots of the code. TAU is famous for providing detailed information about computation and communication intensive routines. However in our profiling of Code_Saturne we observed that TAU-compiled executable failed to produce any profiling result. We believe this is due to some bug in TAU. This issue has been also addressed in another PRACE task [10]. Therefore we based on gprof tool for detecting hot-spots of the code.

The GNU gprof profiling tool is a simple utility for code profiling. GNU-provided gprof is frequently used for light-weight profiling since it is portable with most application codes. Code_Saturne is compiled and linked with additional -g and -pg flag for using gprof. Code_Saturne is then run as usual. After the run is over profile files are generated. For parallel jobs we set the environment variable GMON_OUT_PREFIX=name, where 'name' is the prefix of the name of the profile files. The generated profile names are, e.g., name.xxx where xxx is the rank number of a process. The profile files are then converted to text files using the command

```bash
$gprof cs_solver name.xxx > profile.xxx
```

The profile.xxx files contain the timing information for different routines. It also shows the call graphs of different routines, e.g., for the halo exchange routine cs_halo_sync_var the call graph looks like:

<table>
<thead>
<tr>
<th>index</th>
<th>%time</th>
<th>self</th>
<th>child called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>time</td>
<td>time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1/50342</td>
<td>cs_mesh_quantities_compute [104]</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>40/50342</td>
<td>cs_mesh_sync_var_scal_ext [318]</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>120/50342</td>
<td>cs_grid_coarsen [50]</td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td>0.00</td>
<td>292/50342</td>
<td>cs_mesh_sync_var Scal [273]</td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td>0.00</td>
<td>353/50342</td>
<td>cgcel_ [18]</td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td>0.00</td>
<td>414/50342</td>
<td>cs_mesh_sync_varVect_ni [222]</td>
<td></td>
</tr>
<tr>
<td>1.29</td>
<td>0.01</td>
<td>49122/50342</td>
<td>cs_halo_sync_component [42]</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>1.5</td>
<td>1.32</td>
<td>0.01 50342</td>
<td>cs_halo_sync_var [41]</td>
</tr>
<tr>
<td>0.01</td>
<td>0.00</td>
<td>1006840/1041650</td>
<td>MPI_Isend [270]</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>1006840/1041650</td>
<td>MPI_Irecv [401]</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>50342/52084</td>
<td>MPI_Waitall [406]</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: call graph of cs_halo_sync_var

In Figure 1 the 1st column is the function index, the 2nd column gives the percentage of elapsed time out of total runtime. Columns 3 and 4 show the percentage of elapsed time in self & child routines. The 5th column in Figure 1 shows how many times these routines are called and the total number of calls. The 6th column gives the name of the routines. In figure 1, routines above cs_halo_sync_var are the routines calling it and the routines below cs_halo_sync_var are the routines called by it. We can see that cs_halo_sync_var calls MPI_Isend, MPI_Irecv and MPI_Waitall. From Figure 1 we can see that cs_halo_sync_var is called 50342 times while cs_halo_sync_var calls MPI_Isend 1006840 times out of total 1041650 calls of MPI_Isend. Similarly cs_halo_sync_var calls MPI_Irecv 1006840 times out of total 1041650 calls of MPI_Irecv and MPI_Waitall 50342 times out of total 52084 calls of MPI_Waitall. This shows that most of the MPI_Isend, MPI_Irecv and MPI_Waitall calls are from the cs_halo_sync_var routine. However timings shown in columns 2-4 are not accurate as gprof can not show the MPI timings correctly as the MPI libraries are not compiled with -g and -pg flags. That is why self and child times of MPI routines are shown as zero in this table. To get the MPI timing correctly we use a simple tool developed by us.

We have a precompiled library of MPI_Wrapper routines. An example of MPI_Waitall wrapper routine is given below:
int MPI_Waitall_prof(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses)
{
    int res;
    TIMER1;
    res = MPI_Waitall(count, array_of_requests, array_of_statuses);
    TIMER2;
    t_waitall = t_waitall + TIME_DIFF;
    n_waitall = n_waitall + 1;
    return res;
}

Most other MPI routines are wrapped in a similar way. The original library calls are substituted by symbolic replacement during compilation. As an example to replace MPI_Waitall with our wrapper we pass additional command line flags “-DMPI_Waitall=MPI_Waitall_prof”. This replaces all the occurrences of MPI_Waitall by MPI_Waitall_prof. Finally while linking we add our MPI wrapper library. When this binary is run every MPI call is instrumented and the cumulative time for each MPI routine is printed at the end of the run. This tool is a simple and adaptable tool which can be used as a starting tool before trying more rigorous profiling tools. For Code_Saturne run the MPI profile is shown in Figure 2.

```
rank 0 MPI_Total(s) 152.499371 (100.000000% of tot) for 1 times
rank 0 MPI_Waitall(s) 3.775951 (6.476044% of tot) for 52084 times
rank 0 MPI_Reduce(s) 0.378624 (0.248279% of tot) for 6 times
rank 0 MPI_Allreduce(s) 18.978851 (12.445200% of tot) for 100419 times
rank 0 MPI_Sendrecv(s) 0.025146 (0.016489% of tot) for 36 times
rank 0 MPI_Send(s) 0.022175 (0.014541% of tot) for 2 times
rank 0 MPI_Recv(s) 0.002828 (0.001854% of tot) for 16 times
```

Figure 2: MPI profiling results for rank 0

In Figure 2 MPI_Total is the total run time between MPI_Init and MPI_Finalize. We can see in Figure 2 that MPI_Waitall amounts for 6.47 % of the total run time. In the above Figure 2 MPI_Isend, MPI_Irecv times are not seen as these are non blocking calls and consequently do not take any significant time. We also see that MPI_Allreduce is the most expensive MPI routine in the run, due to the use of linear solvers. However in this work we do not focus on collective communications as we are interested in replacing MPI point to point calls by MPI one sided calls.

From the discussions above we see that the routine cs_halo_sync_var is the heaviest user of MPI_Isend, MPI_Irecv and MPI_Waitall routines. Hence we have implemented MPI one sided communication the routine cs_halo_sync_var.

### 3. Development activities

From our profiling results we see that the cs_halo_sync_var routine is the best candidate for trying one sided MPI 3.0 communication routines. However not many implementations of MPI 3.0 standard are available as of now. To the best of our knowledge open-source mvapich2 [11] and MPICH [12] have full support for MPI 3.0 standard while OpenMPI [13] has partial support of MPI 3.0. We have chosen mvapich2 as our choice of MPI 3.0 implementation as it is highly optimized for infiniband clusters. For our development activities we use the Triolith cluster [14] at the National Supercomputer Centre (NSC), Sweden.

**Structure of original cs_halo_sync_var routine**

For modifying the cs_halo_sync_var we first need to understand the original routine. The cs_halo_sync_var routine performs halo exchange between neighbouring ranks. The receiver ranks first post non-blocking receive request for data from the neighbouring ranks. The pseudo code is given in Figure 3.
for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    start = halo->index[2*rank_id];
    length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];
    if (halo->c_domain_rank[rank_id] != local_rank) {
        MPI_Irecv(var + halo->n_local_elts + start,
                   length,
                   CS_MPI_REAL,
                   halo->c_domain_rank[rank_id],
                   halo->c_domain_rank[rank_id],
                   cs_glob_mpi_comm,
                   &(_cs_glob_halo_request[request_count++]));
    } else
        local_rank_id = rank_id;
}

Figure 3 Pseudo code for receive request in original cs_halo_sync_var

Then sender ranks assembles send buffers for halo exchange into a contiguous buffer called build_buffer as shown in Figure 4.

for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    start = halo->send_index[2*rank_id];
    length = halo->send_index[2*rank_id + end_shift] - halo->send_index[2*rank_id];
    for (i = 0; i < length; i++)
        build_buffer[start + i] = var[halo->send_list[start + i]];
}

Figure 4: Pseudo code for data copy in a contiguous buffer in original cs_halo_sync_var

Then the sender ranks send data from build_buffer to neighbouring ranks which is shown in Figure 5.

for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    start = halo->send_index[2*rank_id];
    length = halo->send_index[2*rank_id + end_shift] - halo->send_index[2*rank_id];
    MPI_Isend(build_buffer + start,
              length,
              CS_MPI_REAL,
              halo->c_domain_rank[rank_id],
              local_rank,
              cs_glob_mpi_comm,
              &(_cs_glob_halo_request[request_count++]));
}

MPI_Waitall(request_count, _cs_glob_halo_request, _cs_glob_halo_status);

Figure 5: Pseudo code for data copy in a contiguous buffer in original cs_halo_sync_var
The copy is completed when MPI_Waitall call finishes as shown in Figure 5.

**Modified cs_halo_sync_var (version 1)**

We first implement a version of the cs_halo_sync_var routine using MPI-2.0 one sided communication routines. We use the MPI one sided routine MPI_Get. In our implementation we first assemble send buffers for halo exchange into a contiguous buffer as shown in Figure 6

```c
same_as_last = 1;
for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    start = halo->send_index[2*rank_id];
    if (start_indx_of_send_buff[halo->c_domain_rank[rank_id]] == start)
        same_as_last = same_as_last * 1;
    else{
        start_indx_of_send_buff[halo->c_domain_rank[rank_id]] = start;
        same_as_last = 0;
    }
    length = halo->send_index[2*rank_id + end_shift] - start;
    for (i = 0; i < length; i++){
        build_buffer[start + i] = var[halo->send_list[start + i]];
        buffer_end = start + i;
    }
}
```

Figure 6: Pseudo code for buffer copying in cs_halo_sync_var version 1.

The steps in Figure 6 are similar to the steps in Figure 4 of the original cs_halo_sync_var. We have introduced additional variables start_indx_of_send_buff and same_as_last in this section. The start_indx_of_send_buff stores the starting index in build_buffer for different ranks. same_as_last = 1 tells that the send buffer lengths are the same between current and previous calls of cs_halo_sync_var, while same_as_last = 0 tells that the send buffer lengths are different between current and previous calls of cs_halo_sync_var. This is a rank local operation. Next we check whether same_as_last is the same globally as shown in Figure 7.

```c
MPI_Allreduce(&same_as_last,&same_as_last_global,1,MPI_INT,MPI_PROD,
              cs_glob_mpi_comm);
```

Figure 7: Pseudo code for checking whether the send buffer lengths are same between current and previous calls globally in cs_halo_sync_var version 1.

The calculation of same_as_last_global helps to minimize number of times of new MPI window is created. We create a new communication window only when same_as_last_global is 0. In one sided communication the receiver rank should know the memory address in the source rank to fetch data from that location. For this we explicitly receive send buffer address which is stored in start_indx_of_send_buff in the sender side. On the receiver side this is stored in start_indx_of_recv_buff.
if (!same_as_last_global) {
    MPI_Win_create(start_indx_of_send_buff, sz_array * sizeof(int),
                    sizeof(int), MPI_INFO_NULL, cs_glob_mpi_comm, &win1);
    MPI_Win_fence(0, win1);
    for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++)
        if (halo->c_domain_rank[rank_id] != local_rank) {
            MPI_Get(&start_indx_of_recv_buff[halo->c_domain_rank[rank_id]],
                     1, MPI_INT,
                     halo->c_domain_rank[rank_id], local_rank, 1, MPI_INT,
                     win1);
            MPI_Win_create(build_buffer, (buffer_end) * sizeof(cs_real_t),
                            sizeof(cs_real_t), MPI_INFO_NULL, cs_glob_mpi_comm, &win);
        }
}

Figure 8: Pseudo code for creating MPI window for one sided communication and exchange of memory addresses in cs_halo_sync_var version 1.

A procedure of creating an MPI window in Figure 8 is expensive. On the other hand, the send buffer lengths do not change frequently during the simulation run. Hence `same_as_last_global` is 1 in most iterations and consequently the above overhead is minimal. After this operation, MPI_Get is posted to receive data.

for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    start = halo->index[2*rank_id];
    length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];
    if (halo->c_domain_rank[rank_id] != local_rank) {
        MPI_Get(var + halo->n_local_elts + start, length, CS_MPI_REAL,
                halo->c_domain_rank[rank_id], start_indx_of_recv_buff[halo->c_domain_rank[rank_id]], length, CS_MPI_REAL, win);
    }
}

Figure 9: Pseudo code for halo exchange in cs_halo_sync_var version 1.

The MPI_get call in Figure 9 uses `start_indx_of_recv_buff` as the memory address to fetch data from the source rank.

**Modified cs_halo_sync_var (version 2)**

In this version of the routine we try to use a MPI-3.0 routine called MPI_Rget [7]. MPI_Rget is similar to MPI_Get, except that it allocates a communication request object and associates it with the request handle that can be used to wait or test for completion. The use of MPI_Rget requires minimal code changes from our version 1 of cs_halo_sync_var routine. At the same time MPI_Rget gives the possibility of overlapping communication from different ranks. This can reduce the overall communication time. The code remains much the same as cs_halo_sync_var version 1. We only modify the data receiving section by replacing MPI_Get by MPI_Rget and then putting the appropriate MPI_Waitall statement at the end as is shown in figure 10.
for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    start = halo->index[2*rank_id];
    length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];
    if (halo->c_domain_rank[rank_id] != local_rank) {
        MPI_Rget(var + halo->n_local_elts + start, length , CS_MPI_REAL,
                  halo->c_domain_rank[rank_id], start_indx_of_recv_buff[halo>
                    c_domain_rank[rank_id]], length, CS_MPI_REAL, win, req[rank_id]);
    }
}
MPI_Waitall(halo->n_c_domains, req, status);

Figure 10: Pseudo code for halo exchange using MPI_Rget in cs_halo_sync_var version 2

4. Results & discussion

For our tests we have taken a test case called ONE_TUBE. The configuration corresponds to the flow in a staggered bundle of tubes, used by Simonin and Barcouda to conduct their experiment [15]. All our tests are done on the Triolith cluster [14] which is an Intel Sandybridge, infiniband cluster.

**cs_halo_sync_var (version 1)**

We have run our test case on 64, 128, 256 and 512 MPI ranks. We have done correctness checking by comparing results of original run and the modified run. The scaling results are presented in Table 1.

Table 1: Scaling results for One Tube test case runs

<table>
<thead>
<tr>
<th>ranks</th>
<th>Total run time (s)</th>
<th>cs_halo_sync_var time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original</td>
<td>version 1</td>
</tr>
<tr>
<td>64</td>
<td>264.41</td>
<td>264.12</td>
</tr>
<tr>
<td>128</td>
<td>152.49</td>
<td>152.11</td>
</tr>
<tr>
<td>256</td>
<td>105.89</td>
<td>105.96</td>
</tr>
<tr>
<td>512</td>
<td>91.04</td>
<td>91.03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>original</th>
<th>version 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>6.6</td>
<td>6.1</td>
</tr>
<tr>
<td>128</td>
<td>7.3</td>
<td>6.8</td>
</tr>
<tr>
<td>256</td>
<td>9.2</td>
<td>8.5</td>
</tr>
<tr>
<td>512</td>
<td>11.3</td>
<td>10.2</td>
</tr>
</tbody>
</table>

As we can see in Table 1 the modified version 1 is slightly faster than the original version of Code_Saturne. We have also observed that the modified version has 1 - 2% less memory overhead. This is probably because in the modified version as we are using MPI one sided communication it does not require MPI to allocate internal buffer for communication. This effect will be more prominent in very wide jobs running over several thousand cores. In our tests the routine cs_halo_sync_var is called more than 50000 times. The send buffer lengths in cs_halo_sync_var are changed only a few times during the whole run. As we do not have a clear understanding of when the buffer lengths are changing from the previous iteration we are checking every time whether the send buffer lengths have changed from the previous iteration. This brute force method introduces some extra overhead which can be optimized with better knowledge of the behaviour of the code.

**cs_halo_sync_var version 2**

In this version we have used MPI-3.0 function MPI_Rget. With mvapich2 we could compile the code but the run crashes as soon as it encounters the MPI_Rget function. We have no clear explanation why this is happening and we could not get any hint from the mvapich2 user guide. We believe the implementation of MPI_Rget function is not yet very robust in mvapich2.

5. Conclusion and Future Work
Our results show that MPI-2.0 one sided communications are slightly faster than point to point communications in Code_Saturne. However improvements are marginal for our tests. The impact can be more prominent for jobs running on very large number of CPU cores as MPI one sided communication is supposed to give better communication latency. There is some improvement in memory consumption also while using one sided communication. After our modifications Code_Saturne consumes 1-2% less memory. For runs on large number of core counts especially at exascale the memory advantage of MPI one sided communication can be quite significant.

The MPI-3.0 features are still difficult to test because of lack of availability of optimized MPI implementations. Even those implementations that support MPI 3.0 standard are not robust yet. In our tests we used mvapich2 which supports full MPI 3.0 standard, but all the simulations crashed when tested on Code_Saturne.

References


[10] Profiling of Code_Saturne with TAU and autotuning of kernels with Orio, Bjorn Lindi et al (NTNU), Thomas Ponweiser (JKU), Petar Jovanovic (IPB)


Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493.
Appendix A: Code listing for cs_halo_sync_var

```c
void cs_halo_sync_var(const cs_halo_t *halo,
                      cs_halo_type_t sync_mode,
                      cs_real_t var[])
{
    cs_lnum_t i, start, length;

    int local_rank_id = (cs_glob_n_ranks == 1) ? 0 : -1;
    const cs_lnum_t end_shift = (sync_mode == CS_HALO_STANDARD) ? 1 : 2;

    #if defined(HAVE_MPI)
        if (cs_glob_n_ranks > 1) {
            int rank_id;
            int request_count = 0;
            cs_real_t *build_buffer = (cs_real_t *)_cs_glob_halo_send_buffer;
            const int local_rank = cs_glob_rank_id;
            #ifdef CBASU_AT_NSC  //Changes by cbasu@nsc.liu.se
                #define VERSION 1
            #ifdef VERSION 1  // 2nd version, working, speed lightly better than original
                int buffer_end, ii;
                static int first_call = 1;
                static MPI_Group group;
                // the new global variables are now moved to a header file
                /* Assemble buffers for halo exchange; */
                if (first_call) {  // this is called only once
                    for (i = 0; i < sz_array; i++) {
                        start_indx_of_send_buff[i] = 0;
                        start_indx_of_recv_buff[i] = 0;
                    }
                    first_call = 0;
                    MPI_Comm_group(cs_glob_mpi_comm, &group);
                }
                same_as_last = 1;
                MPI_Comm_group(cs_glob_mpi_comm, &group);
            }
            same_as_last = 1;
            for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
                if (halo->c_domain_rank[rank_id] != local_rank) {
                    start = halo->send_index[2*rank_id];
                    if (start == start_indx_of_send_buff[halo->c_domain_rank[rank_id]])
                        same_as_last = same_as_last * 1;
                    else{
                        start_indx_of_send_buff[halo->c_domain_rank[rank_id]] = start;
                        same_as_last = 0;
                    }
                    length = halo->send_index[2*rank_id + end_shift] - start;
                    for (i = 0; i < length; i++){
                        build_buffer[start + i] = var[halo->send_list[start + i]];
                        buffer_end = start + i;
                    }
                }
            }
        }
    ```
```
TIMER1:
MPI_Allreduce(&same_as_last, &same_as_last_global, 1, MPI_INT, MPI_PROD, cs_glob_mpi_comm);
if (!same_as_last_global) {
    if (win1 != MPI_WIN_NULL)
        MPI_Win_free (&win1);
    MPI_Win_create(start_indx_of_send_buff, sz_array*sizeof(int), sizeof(int),
        MPI_INFO_NULL, cs_glob_mpi_comm, &win1);
    MPI_Win_fence(0, win1);
    for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++)
        if (halo->c_domain_rank[rank_id] != local_rank) {
            MPI_Get(&start_indx_of_recv_buff[halo->c_domain_rank[rank_id]], 1, MPI_INT,
                halo->c_domain_rank[rank_id], local_rank, 1, MPI_INT, win1);
        }
    MPI_Win_fence(0, win1);
}
if (win != MPI_WIN_NULL)
    MPI_Win_free (&win);
MPI_Win_create(build_buffer, (buffer_end)*sizeof(cs_real_t), sizeof(cs_real_t),
    MPI_INFO_NULL, cs_glob_mpi_comm, &win);
} #else
TIMER2:
t_cs_halo_sync_var = t_cs_halo_sync_var + TIME_DIFF;
n_cs_halo_sync_var = n_cs_halo_sync_var + 1;
MPI_Win_fence(0, win);
/* Receive data from distant ranks */
for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    start = halo->index[2*rank_id];
    length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];
    if (halo->c_domain_rank[rank_id] != local_rank) {
        MPI_Get(var + halo->n_local_elts + start, length , CS_MPI_REAL,
            halo->c_domain_rank[rank_id], start_indx_of_recv_buff[halo->c_domain_rank[rank_id]], length, CS_MPI_REAL, win);
    } else
        local_rank_id = rank_id;
}
MPI_Win_fence(0, win);
#endif
} // MPI 3.0 version, crashes
int buffer_end, ii;
static int first_call = 1;
//static MPI_Group group;
// the new global variables are now moved to a header file
/* Assemble buffers for halo exchange: */
if (first_call) {
    for (i = 0; i < sz_array; i++) {
        start_indx_of_send_buff[i] = 0;
        start_indx_of_recv_buff[i] = 0;
    }
first_call = 0;
    MPI_Comm_group(cs_glob_mpi_comm, &group);
}

same_as_last = 1;
for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    if (halo->c_domain_rank[rank_id] != local_rank) {
        start = halo->send_index[2*rank_id];
        if (start_indx_of_send_buff[halo->c_domain_rank[rank_id]] == start) {
            same_as_last = same_as_last * 1;
        } else {
            start_indx_of_send_buff[halo->c_domain_rank[rank_id]] = start;
            same_as_last = 0;
        }
        length = halo->send_index[2*rank_id + end_shift] - start;
        for (i = 0; i < length; i++) {
            build_buffer[start + i] = var[halo->send_list[start + i]];
            buffer_end = start + i;
        }
    }
}

TIMER1:
    MPI_Allreduce(&same_as_last, &same_as_last_global, 1, MPI_INT, MPI_PROD, cs_glob_mpi_comm);
    if (!same_as_last_global) {
        if (win1 != MPI_WIN_NULL)
            MPI_Win_free(&win1);
        MPI_Win_create(start_indx_of_send_buff, sz_array*sizeof(int), sizeof(int), MPI_INFO_NULL, cs_glob_mpi_comm, &win1);
        MPI_Win_fence(0, win1);
        for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++)
            if (halo->c_domain_rank[rank_id] != local_rank) {
                MPI_Get(&start_indx_of_recv_buff[halo->c_domain_rank[rank_id]], 1, MPI_INT, halo->c_domain_rank[rank_id], local_rank, 1, MPI_INT, win1);
            }
        MPI_Win_fence(0, win1);
        MPI_Win_free(&win);
        MPI_Win_create(build_buffer, (buffer_end)*sizeof(cs_real_t), sizeof(cs_real_t), MPI_INFO_NULL, cs_glob_mpi_comm, &win);
    }
    MPI_Request req[1000];           // statically allocated, change later
    MPI_Status status[1000];
    /* Receive data from distant ranks */
    for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
        start = halo->index[2*rank_id];
        length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];
        if (halo->c_domain_rank[rank_id] != local_rank) {
            MPI_Rget(var + halo->n_local_elts + start, length, CS_MPI_REAL,
```c
halo->c_domain_rank[rank_id],
> c_domain_rank[rank_id]], length, CS_MPI_REAL, win, &req);
} else
   local_rank_id = rank_id;
} MPI_Waitall( halo->n_c_domains, req, status);
#else
/* Receive data from distant ranks */
for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    start = halo->index[2*rank_id];
    length = halo->index[2*rank_id + end_shift] - halo->index[2*rank_id];
    if (halo->c_domain_rank[rank_id] != local_rank) {
        MPI_Irecv(var + halo->n_local_elts + start,
                  length,
                  CS_MPI_REAL,
                  halo->c_domain_rank[rank_id],
                  halo->c_domain_rank[rank_id],
                  cs_glob_mpi_comm,
                  &(_cs_glob_halo_request[request_count++]));
    } else
        local_rank_id = rank_id;
}
/* Assemble buffers for halo exchange;
   with threading, use dynamic scheduling , as this is probably a small loop */
#pragma omp parallel for private(start, length, i) schedule(dynamic)
for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
    if (halo->c_domain_rank[rank_id] != local_rank) {
        start = halo->send_index[2*rank_id];
        length = halo->send_index[2*rank_id + end_shift] - halo->send_index[2*rank_id];
        for (i = 0; i < length; i++)
            build_buffer[start + i] = var[halo->send_list[start + i]];
    }
}
/* We wait for posting all receives (often recommended) */
if (_cs_glob_halo_use_barrier)
```
MPI_Barrier(cs_glob_mpi_comm);

/* Send data to distant ranks */

for (rank_id = 0; rank_id < halo->n_c_domains; rank_id++) {
  if (halo->c_domain_rank[rank_id] != local_rank) {
    start = halo->send_index[2*rank_id];
    length = halo->send_index[2*rank_id + end_shift] - halo->send_index[2*rank_id];

    MPI_Isend(build_buffer + start,
              length,
              CS_MPI_REAL,
              halo->c_domain_rank[rank_id],
              local_rank,
              cs_glob_mpi_comm,
              &(_cs_glob_halo_request[request_count++]));
  }
}

/* Wait for all exchanges */

MPI_Waitall(request_count, _cs_glob_halo_request, _cs_glob_halo_status);
#endif

/* Copy local values in case of periodicity */

if (halo->n_transforms > 0) {
  if (local_rank_id > -1) {
    cs_real_t *recv_var
    = var + halo->n_local_elts + halo->index[2*local_rank_id];

    start = halo->send_index[2*local_rank_id];
    length = halo->send_index[2*local_rank_id + end_shift] - halo->send_index[2*local_rank_id];

    for (i = 0; i < length; i++)
      recv_var[i] = var[halo->send_list[start + i]];
  }
}
}