Channelflow and MPI

Tobias Kreilos

April 2013

1 General stuff

All credits go to Philipp Schlatter from KTH Stockholm, who explained the ideas of parallelizing this type of code to me, and to FFTW, which already provides the transpose and 2D-FFT calls we needed.

The basic idea, when using MPI, is to distribute the data over the available processors. Each processor then works on its local dataset and the intermediate results are combined to obtain a global result. We will hence start by explaining the way we distribute the flowfield-data for the time-integration.

1.1 Idea

The computation to advance a given flowfield in time can (very rudely) be divided into two substeps:

1. Calculate the nonlinear term.
2. Solve the τ-equations

The τ-equations are a set of independent equations for each k_x, k_z Fourier-mode. Calculating the nonlinear term involves doing a Chebyshev-transformation (using all y-gridpoints), followed by a 2D-Fourier-transformation in the xz-plane and backwards. For this, we can't avoid a global communication operation.

1.2 Data distribution

We distribute the data in two directions. This allows the maximum number of processors to be the product of the number of gridpoints in two directions, which is way larger than the maximum number with a 1D parallelization, namely the number of gridpoints.

In spectral state, we want all y-gridpoints to be on the same processor, because then it's easy to solve the τ-equations. That means, both x and z are distributed. We visualize the situation in figure 1(a), where the dark area indicates the information on process 0.

The first step needed for calculating the nonlinear term is now the Chebyshev transform, which can be done without any redistribution. The next step is the 2D-FFT in the xz-planes. FFTW is able to handle 1D-distributed transforms, of which we want to do several in parallel. Therefore, we first need to transpose the data so that in one plane only one direction is distributed. We choose, to do a xy-transpose. Luckily, FFTW also offers the capabilities to that – it's basically an `MPI_Alltoall` with stride. The situation in this intermediate step (which is never available to the end-user) is illustrated in figure 1(b).

Now we are ready to do the 2D-FFT, which is simply a call to FFTW_MPI. This function involves yet another data transpose (the 2D-FFT basically means doing one 1D-FFT, transpose, another 1D-FFT), so that we finally end up with the data distributed along x and y, with all z-values to one x, y-gridpoint on one processor. The physical state is illustrated in figure 1(c).

1.3 A bit more specific

We divide the total number of processors in two directions, $np = np_0 \times np_1$. The number np_0 is the number of processors for the distribution along x in spectral state; this grouping will be used for the xy-transpose. np_0 is also the number of independent parallel FFTs. The number np_1 is the number of processors for the distribution along z in spectral state; np_1 processors work together to do one 2D-FFT.

We require np_0 to divide the number of gridpoints M_x, $M_x \ \% \ np_0 = 0$. In y, we use padding to the next multiple of np_0, the total number of gridpoints hence becomes $M_{y,pad} \geq M_y$. In z we leave the distribution to FFTW, there are no requirements for the number of processors np_1.

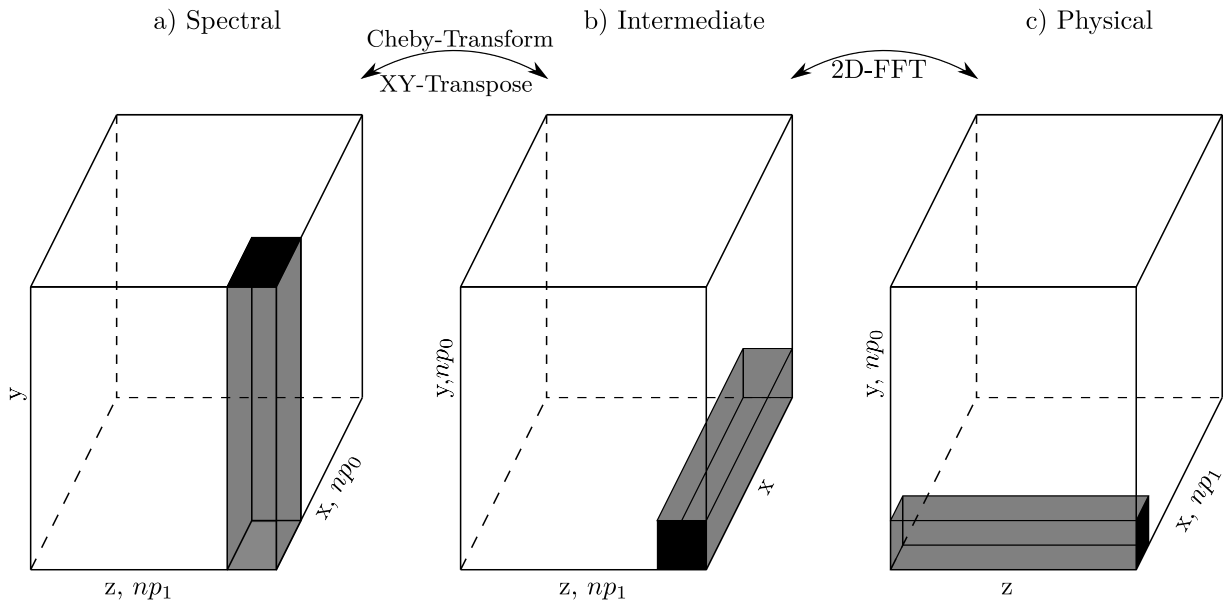

Figure 1: Illustration of the data distribution in spectral and physical space. The shaded are shows the data on process 0. a) In spectral state, the data is distributed along x and z. A cheby-transform can readily be executed, since all information is locally available. b) To do the global FFT, the chebyshev-transform is followed by a xy-transpose. c) To complete the transform, FFTW is called to do a distributed 2D-transform.

1.4 Array alignment

The alignment of the arrays is dictated by the requirements for the transpose operations. In spectral state, the alignment is $z^* x^* yi$, where * indicates that this direction is distributed. In physical state, the alignment is $x^* zy^* i$.

This is completely internal and covered by the flatten functions. But it might be interesting when optimizing array-loops for speed.

2 Usage

2.1 Using MPI within channelflow

All the MPI-stuff is contained in a class called `CfMPI`. FlowField constructors can take a pointer to a `CfMPI`-object – passing a reference would be better, probably, but I got stuck figuring out why so many copies of objects were made that I switched to pointers. That's almost the only part where changes are needed. After that, creation of DNS-objects, time integration etc. works just as before.

The `CfMPI` constructor takes two arguments, namely the number of processes np_0 used in the x-direction (and of simultaneous plane-ffts) and the number of processes np_1 along z (and for processes that work together on the fft of one plane). The total number of processes should be $np = np_0 \cdot np_1$. The default `CfMPI` constructor uses $\sqrt{np}$ for both np_0 and np_1 if np is a square number; otherwise, the distribution must be specified. The only limitation (apart from $np_0 < Ny$ and $np_1 < Nz$) is that np_0 must divide Nx. `CfMPI(1,1)` will create an object that limits flowfields to process 0 and is used for i/o.

In principle it should be possible to replace `MPI_COMM_WORLD` in `CfMPI` by some other communicator (at least, that's what I had in mind when writing the code) but I never tested.

MPI will complain, if you call Finalize before all flowfields are destructed (that is because the fftw plans belonging to the flowfields create MPI-communicators). The easiest way to prevent this is to enclose everything between `MPI_Init` and `MPI_Finalize` in a bracketed block `{}`.

2.2 Writing a parallel program

Usually, there's not much to do. At the beginning put

```
MPI_INIT(&argc, &argv); {
```

and at the end

```
} MPI_FINALIZE();
```

Before creating the first flowfield, create a `CfMPI`-object via:

```
CfMPI* cfmpi = new CfMPI(nproc0, nproc1);
```

and pass the pointer to the flowfield, e.g.

```
FlowField u ("filename", cfmpi);
```

Don't forget to delete cfmpi at the end, before calling finalize:

```
delete cfmpi;
```

2.3 Locally available information

To get the information, which data is available on the current process, there are a couple of functions for a flowfield.

1. `Nloc()` gives the total number of gridpoints
2. `nxlocmin()` gives the first gridpoint in x in spectral state
3. `Nxloc()` gives the local number of gridpoints in x in spectral state. nxloc+nxlocmin can be used as the upper boundary of a loop.
4. `Nyloc()` and `nylocmin()` give the corresponding information in y.
5. To account for the padding, there is also a function `nylocmax()` to give an upper boundary for loops.
6. `Nypad()` gives the padded number of gridpoints (`Ny()` is the unpadded number).
7. For spectral state, there are the functions `Mxloc(), mxlocmin(), Mzloc()` and `mzlocmin()`.

The locally available data is in the range `mxlocmin <= mx < mxlocmin+Mxloc` and `mzlocmin <= mz < mzlocmin+Mzloc` in spectral state and `nxlocmin <= nx < nxlocmin+Nxloc` and `nylocmin <= ny < nylocmax` in physical state.

All these functions are of type `lint`, which is a typedef to `ptrdiff_t` and is an integer that is guaranteed to have 64 bits on a 64bit machine (which is important for large flowfields).

2.4 Deadlocks – and how to avoid them

The parallelization is done so that (almost) all calls are collective calls. A program can hence be written almost as if it was serial. Deadlocks will occur if a function that expects to be called by all processes is only called by one. For example, the following will never return:

```
if (taskid == 0) cout << L2Norm(u) << endl;
```

because the function L2Norm gets called only by process 0. Instead, you should write something like

```
Real l2 = L2Norm(u);
if (taskid == 0) cout << l2 << endl;
```

which will calculate the L2Norm with all processes and then output only on process 0.

For convencience, there's also a function called printout, which takes a string and prints it on process 0. So the following code also works:

```
printout(r2s(L2Norm(u)));
```

2.5 Compiling FFTW with MPI

Channelflow's parallelization depends heavily on the parallel fftw. Enable both openmp and mpi, because a (rudimentary) parallelization with threads is still available.

```
./configure -prefix=$HOME/usr/ -enable-shared -disable-static \
  -enable-openmp -enable-mpi
make && make install
```

There's a problem with the fftw configure script, which overwrites the internal optimization CFLAGS if you define any CFLAGS. If you define any CFLAGS, add `'-O3 -fomit-frame-pointer -mtune=native -malign-double -fstrict-aliasing -ffast-math'` to them, otherwise the performance will go down by almost a factor of 2.

2.6 Testing

All unit tests pass, with two exceptions:

- DNSSinusoid fails with segfault, but also does so without parallelization
- DNSOrrSomm fails, but also does so without parallelization

2.7 Possible errors

I have checked and modified a large part of channelflow, but it would be unrealistic to think that no errors have been introduced. Errors are most likely due to wrong boundaries for loops, e.g. loops going from 0 to Mx instead of $mxlocmin$ to $mxlocmax$. Those errors are mostly captured by the assertions in flatten and complex_flatten.

2.8 Remarks

- Some of the norms I never use are not parallelized.
- The i/o is very slow and rather ineffective. The distributed flowfield gets interpolated to the padded field on process 0, then a spectral to real transform gets applied on process 0 alone and then process 0 stores the field on the harddisk. There are more efficient ways to do it and especially ways that don't require that much memory on one process. The two advantages of this approach are that the data layout doesn't change and it was easy to implement.
- Since the i/o is serial, there's no need for parallel hdf5.