1 The DDalphaAMG Solver Library

1.1 Introduction

This MPI-C-Code can be used for solving Wilson-Dirac equations

$$D_W(U, m_0)\psi = \eta$$

with an aggregation-based algebraic multigrid (AMG) method. Herein the smoother was chosen as the Schwarz alternating procedure (SAP). The Dirac operator depends on a configuration U which therefore is required as input. The mass parameter $m_0 = 1/(2\kappa) - 4$ and other parameters can be adjusted in the parameter file, e.g., `sample.ini` in the main directory. We expect that the user of this code is familiar with the required basics (usage of C-compilers, MPI libraries as well as lattice QCD, configurations and so on). This code also supports openmp-threading and SSE-optimization. For understanding the method behind this C-code or even for a brief introduction into the structure of the Wilson Dirac operator from lattice QCD, we refer the interested reader to [1, 2, 3]. Implementation details can be found in [3].

1.2 Compilation

The main directory contains a makefile example. The standard `Makefile` should work with a few adjustments (compiler, MPI library, etc.) on the machine of your choice. Once the necessary adjustments of the makefile are done, the command

`make -f` *yourmakefile* `-j` *numberofthreads* `wilson`

should compile the whole entire Wilson solver code, such that it is ready to be run. The makefile also contains additional rules `library` and `documentation` for compiling the code as a library and for compiling the user documentation. The library interface can be found in the `include` folder. Additional CFLAGS are described in Section 1.5.

1.3 Running the Code

Once the code has been compiled, the executables `dd_alpha_amg` and `dd_alpha_amg_db` can be found in the main directory and run with the typical mpirun commands. The latter executable runs a (slower) debug version with additional check routines. The executables accept a user specified parameter file as optional command line, i.e., the code can be run with

`mpirun -np` *numberofcores* `dd_alpha_amg` *yourinputfile*

The path corresponding to your input file can be stated either relatively to the main directory or as an absolute path. For easy usage we recommend to use a run-script. An example is already included in main directory and can be easily modified for your personal needs.

In order to run the code successfully you will need a *configuration* and a *parameter file*. In the upcoming section you will be guided through the most important aspects of the parameter file in order to make it valid and run the code with your configuration and parameters.

1.4 Adjusting Parameters

In this section we take a closer look at the parameter files `sample.ini` and `sample_devel.ini`. The first is a shorter user oriented version, the latter a longer one for developers. Parameters with (absolutely) obvious meaning are omitted here.

1.4.1 Configurations

First of all the path of your configuration has to be specified (again relative to main directory of absolute). The desired data layout is illustrated in the Appendix section. In the folder `conf/convert` one can find a converter which converts OPENQCD/DD-HMC configurations into the desired layout. The code is also able to read configurations in lime format, therefore use the parameter `format: 1`, for compilation see Section 1.5. Note that you can store the paths of different configurations in the parameter file, the one which is used in the code is always the first parameter file that carries the prefix "`configuration:`". This property also holds for other parameters. In case you want to use two different configurations, one for the hopping term and one for the clover term, you have to enter the respective paths with the prefixes "`hopp cnfg:`" and "`clov cnfg:`" in your input file. In this case, the prefix "`configuration:`" should not appear in your input file.

A multi file IO support was added in version v1701. When using `format: 2`, each process reads its own part of the configuration which was stored in a seperate file previously. A tool for splitting configurations (stored in DDalphaAMG format) can be found in `conf/split`. For debugging reasons we added another `CFLAG` as compile option, see Section 1.5.

1.4.2 Geometry

In the geometry part, the geometry of the lattice (in `depth0`) and the coarser lattices (in `depth1`, `depth2`, ...) as well as the parallelization have to be defined. Therefore we have to know the following details and respect the following restrictions:

- `depth0`
  - `d0 global lattice`(μ) describes the $\mu = 1,\ldots,4$ lattice dimensions of your configuration.
  - `d0 local lattice`(μ) determines the number of lattice sites in every direction μ on a single processor. We assume di `global lattice`(μ)/di `local lattice`(μ) to be positive integers. For the number of MPI processes np we have

$$np = \prod_{\mu=1}^{4} \texttt{d0 global lattice}(\mu)/\texttt{d0 local lattice}(\mu).$$

- `depth`i, $i \geq 0$.

- d*i* `block lattice`(μ) determines the size of the Schwarz blocks. We assume d*i* `local lattice`(μ)/d*i* `block lattice`(μ) to be positive integers. Furthermore we need at least two blocks per processor. If possible, we propose a block size of 4^4.
- The numbers d*i* `global lattice`(μ)/d$i+1$ `global lattice`(μ) $=: agg_i(\mu)$ have to be positive integers. These quantities determine the coarsening ratio/the aggregate size. Thus we also assume that d*i* `local lattice`(μ)/$agg_i(\mu)$ and $agg_i(\mu)$/d*i* `block lattice`(μ) are positive integers. If possible, we propose $agg_0(\mu) = 4$ and $agg_i(\mu) = 2$ for $i > 0$ and for all μ. When using the `SSE`-optimized version, we assert $agg_i(\mu) =$d*i* `block lattice`(μ), i.e., the aggregates have to match the Schwarz blocks on every level.
- We also assume d*i* global lattice(μ)/d*i* local lattice(μ) as a function in i to be monotonically decreasing for all μ. The code allows processes to idle on coarser grids. This can happen since we have less workload on coarser grids. The stated assumption means that the number of processors that are idle can not decrease as we go to a coarser lattice, and once a processor idles on a certain level, he will idle on all coarser levels.
- d*i* `preconditioner cycles`: number of preconditioner cycles in every preconditioner call on level i (is ignored on the coarsest level).
- d*i* `post smooth iter`: number of post smoothing iterations applied on level i (is ignored on the coarsest level).
- d*i* `block iter`: number of iterations for the block solver in SAP on level i (is ignored on the coarsest level).
- d*i* `test vectors`: number of test vectors used on level i. We propose using 20 test vectors for $i = 0$ and 30 for $i > 0$ (is ignored on the coarsest level).
- d*i* `setup iter`: number of setup iterations for AMG on the level i (is ignored on the coarsest level).

Please note that further information about how to tune the method sufficiently can be found in [1, 2, 3]. When running the code for the first time it is enough to adjust the global, local and block lattice for `depth0` as well as $m0$ and c_{sw}.

1.4.3 Dirac Operator

Adjust the parameters `m0` and `csw` according to your m_0 and c_{sw} for which you want to solve the Dirac equation. In the header file `src/clifford.h` you can adjust the Clifford algebra basis representation. The basis representations of BMW-c, OPENQCD/DD-HMC and QCDSF are pre-implemented. In case you want to implement your own representation, please pay attention to our conventions for the Wilson-Dirac operator in [2, 3].

1.4.4 Multilevel Parameters

In the multilevel part most of the parameters do not require any additional tuning. Some additional remarks are given here:

- `odd even preconditioning`: the coarsest grid can be solved with odd-even preconditioned GMRES and the Schwarz blocks can be solved via odd-even preconditioned minimal residual iteration. If you switch on this parameter, i.e., set it to any other value except `0`, you have to make sure that $\prod_\mu \texttt{dc local lattice}(\mu) \geq 2$ and $\texttt{dc}$ `global lattice`(μ) is even for every μ where c denoted the coarsest level. The `SSE` implementation only supports the odd-even preconditioned version of the code.

- `mixed precision`: set it to `0` to use the whole method in double precision (not supported by SSE). The value `1` provides a preconditioner in single precision, the outer FGMRES method is still in double precision. In addition to the value `1`, the value `2` provides a mixed precision outer FGMRES routine (caution: for this one the relative residual estimation in FGMRES might be less accurate).

- `kcycle`: set it to the value `1` to switch it on or `0` to switch it off. If `kcycle` is switched on the standard multigrid V-cycle is replaced by a K-cycle. This cycling strategie can be explained as follows: in a two-level method with GMRES as a coarse grid solver, the coarse grid solver is replaced by an FGMRES method preconditioned with another two-level method of the same kind. It can be viewed as a W-cycle where each level is wrapped by FGMRES. This FGMRES wrapper can be adjusted with a restart length `kcycle length`, a number restart cycles `kcycle restarts` and a tolerance for the relative residual `kcycle tolerance`. Note that all tolerances in this code are concidered as relative and non-squared.

1.4.5 Tracking Parameters

The code offers the possibility to track a parameter. In order to switch this feature on, set `evaluation` to any value except 0. If an update of setup or shift is required, set the respective parameter to any other value than `0`.

1.4.6 Default Values

Most of the parameters in the input file are pre-defined with default values which can be checked or even modified in `src/init.c`. We provide an input file called `sample.ini` with a quite small number of parameters and another one called `sample_devel.ini` with all parameters that can be used. You can extend the short version by any parameter from the long version.

1.5 Additional CFLAGS for compilation

There are additional `CFLAGS` in the makefile, mostly for debugging and IO, that you can switch on/off:

- `-DPARAMPOUTPUT`: prints a summary of the input parameters.
- `-DPROFILING`: prints a bunch of profiling information, useful for optimization.
- `-DTRACK_RES`: prints relative residual norms during the solve.
- `-DCOARSE_RES`: prints all coarser GMRES final relative residuals during setup and solve.
- `-DFGMRES_RESTEST`: computes the true residual after the FGMRES solve and prints it. This is particularly useful when using the parameter `mixed precision: 2` since estimated and true residual norms can differ.
- `-DSCHWARZ_RES`: prints all SAP final relative residuals during setup and solve.
- `-DTESTVECTOR_ANALYSIS`: computes the eigenvalue-ishness of all test vectors during the setup phase.
- `-DSINGLE_ALLREDUCE_ARNOLDI`: modifies the Arnoldi iteration such that just on allreduce per iteration is needed. Therefore the norm of the next iterate is computed from inner product results. Please note that this can cause numerical instabilities in ill-conditioned cases. For further information, see e.g. [3] and references therein.
- `-DHAVE_LIME`: enables the code to read configurations in lime format. It requires an installed version of the c-lime library by USQCD and the enviroment variable `LIMEDIR` to be set with the installation directory. The functions in the header file `lime_io.h` can be used to manage the io required for reading and saving the vectors.
- `-DREAD_CONF_MULTI_CHECKFILE` prints which rank opens which file when reading a multi file DDalphaAMG format configuration, this might be useful for investigating I/O problems.

2 Appendix

In this section we offer additional information which might be helpful.

2.1 Configuration Layout

The configuration layout for our AMG solver has the following structure

Algorithm 1 read conf

```
for t = 1 to n_t do
  for z = 1 to n_s do
    for y = 1 to n_s do
      for x = 1 to n_s do
        for mu = 1 to 4 do
          read U_{t,z,y,x}(mu)
        end for
      end for
    end for
  end for
end for
```

where $U_{t,z,y,x}(mu)$ has to be stored row major.

Algorithm 2 read $U_{t,z,y,x}(mu)$

```
for i = 1 to 3 do
  for j = 1 to 3 do
    read real((U_{t,z,y,x}(mu))_{i,j})
    read imag((U_{t,z,y,x}(mu))_{i,j})
  end for
end for
```

Herein the matrices $U_{t,z,y,x}(mu)$ have to be stored with 18 `double` values.

References

[1] Andreas Frommer, Karsten Kahl, Stefan Krieg, Björn Leder, and Matthias Rottmann. An adaptive aggregation based domain decomposition multilevel method for the lattice Wilson Dirac operator: Multilevel results. 2013. arXiv:1307.6101.

[2] Andreas Frommer, Karsten Kahl, Stefan Krieg, Björn Leder, and Matthias Rottmann. Adaptive aggregation based domain decomposition multigrid for the lattice Wilson Dirac operator. *SIAM J. Sci. Comp.*, 36(4):A1581–A1608, 2014.

[3] Matthias Rottmann. *Adaptive Domain Decomposition Multigrid for Lattice QCD*. PhD thesis, Bergische Universität Wuppertal, 2016.