User manual

PARALUTION

Version 1.1.0

January 19, 2016

PARALUTION - User Manual

This project is funded by

Gefördert durch:
Bundesministerium für Wirtschaft und Energie
aufgrund eines Beschlusses des Deutschen Bundestages

eXIST Existenzgründungen aus der Wissenschaft

ESF Europäischer Sozialfonds für Deutschland

EUROPÄISCHE UNION

Imprint:
PARALUTION Labs UG (haftungsbeschränkt) & Co. KG
Am Hasensprung 6, 76571 Gaggenau, Germany
Handelsregister: Amtsgericht Mannheim, HRA 706051
Vertreten durch PARALUTION Labs Verwaltungs UG (haftungsbeschränkt)
Am Hasensprung 6, 76571 Gaggenau, Germany
Handelsregister: Amtsgericht Mannheim, HRB 721277
Geschäftsführer: Dimitar Lukarski, Nico Trost
www.paralution.com

PARALUTION - User Manual

Chapter 1

Introduction

1.1 Overview

PARALUTION is a sparse linear algebra library with focus on exploring fine-grained parallelism, targeting modern processors and accelerators including multi/many-core CPU and GPU platforms. The main goal of this project is to provide a portable library for iterative sparse methods on state of the art hardware. Figure 1.1 shows the PARALUTION framework as middle-ware between different parallel backends and application specific packages.

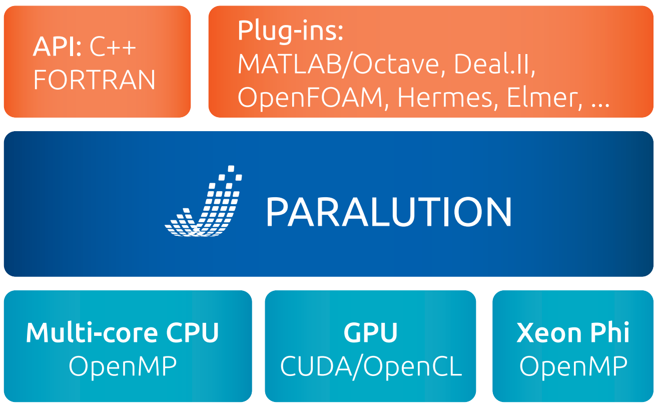


Figure 1.1: The PARALUTION library – middleware between hardware and problem specific packages.

The major features and characteristics of the library are:

- **Various backends**:
  - Host – designed for multi-core CPU, based on OpenMP
  - GPU/CUDA – designed for NVIDIA GPU
  - OpenCL – designed for OpenCL-compatible devices (NVIDIA GPU, AMD GPU, CPU, Intel MIC)
  - OpenMP(MIC) – designed for Intel Xeon Phi/MIC
- **Multi-node/accelerator support** – the library supports multi-node and multi-accelerator configurations via MPI layer.
- **Easy to use** – the syntax and the structure of the library provide easy learning curves. With the help of the examples, anyone can try out the library – no knowledge in CUDA, OpenCL or OpenMP required.
- **No special hardware/library required** – there are no hardware or library requirements to install and run PARALUTION. If a GPU device and CUDA, OpenCL, or Intel MKL are available, the library will use them.
- **Most popular operating systems**:
  - Unix/Linux systems (via cmake/Makefile and gcc/icc)
  - MacOS (via cmake/Makefile and gcc/icc)
  - Windows (via Visual Studio)

- **Various iterative solvers**:
  - Fixed-Point iteration – Jacobi, Gauss-Seidel, Symmetric-Gauss Seidel, SOR and SSOR
  - Krylov subspace methods – CR, CG, BiCGStab, BiCGStab(l), GMRES, IDR, QMRCGSTAB, Flexible CG/GMRES
  - Deflated PCG
  - Mixed-precision defect-correction scheme
  - Chebyshev iteration
  - Multigrid – geometric and algebraic
- **Various preconditioners**:
  - Matrix splitting – Jacobi, (Multi-colored) Gauss-Seidel, Symmetric Gauss-Seidel, SOR, SSOR
  - Factorization – ILU(0), ILU(p) (based on levels), ILU(p,q) (power(q)-pattern method) and Multi-elimination ILU (nested/recursive), ILUT (based on threshold), IC(0)
  - Approximate Inverse - Chebyshev matrix-valued polynomial, SPAI, FSAI and TNS
  - Diagonal-based preconditioner for Saddle-point problems
  - Block-type of sub-preconditioners/solvers
  - Additive Schwarz and Restricted Additive Schwarz
  - Variable type of preconditioners
- **Generic and robust design** – PARALUTION is based on a generic and robust design allowing expansion in the direction of new solvers and preconditioners, and support for various hardware types. Furthermore, the design of the library allows the use of all solvers as preconditioners in other solvers, for example you can easily define a CG solver with a Multi-elimination preconditioner, where the last-block is preconditioned with another Chebyshev iteration method which is preconditioned with a multi-colored Symmetric Gauss-Seidel scheme.
- **Portable code and results** – all code based on PARALUTION is portable and independent of GPU/CUDA, OpenCL or MKL. The code will compile and run everywhere. All solvers and preconditioners are based on a single source code, which delivers portable results across all supported backends (*variations are possible due to different rounding modes on the hardware*). The only difference which you can see for a hardware change is the performance variation.
- **Support for several sparse matrix formats** – Compressed Sparse Row (CSR), Modified Compressed Sparse Row (MCSR), Dense (DENSE), Coordinate (COO), ELL, Diagonal (DIA), Hybrid format of ELL and COO (HYB) formats.
- **Plug-ins** – the library provides a plug-in for the CFD package OpenFOAM [44] and for the finite element library Deal.II [8, 9]. The library also provides a FORTRAN interface and a plug-in for MATLAB/Octave [36, 2].

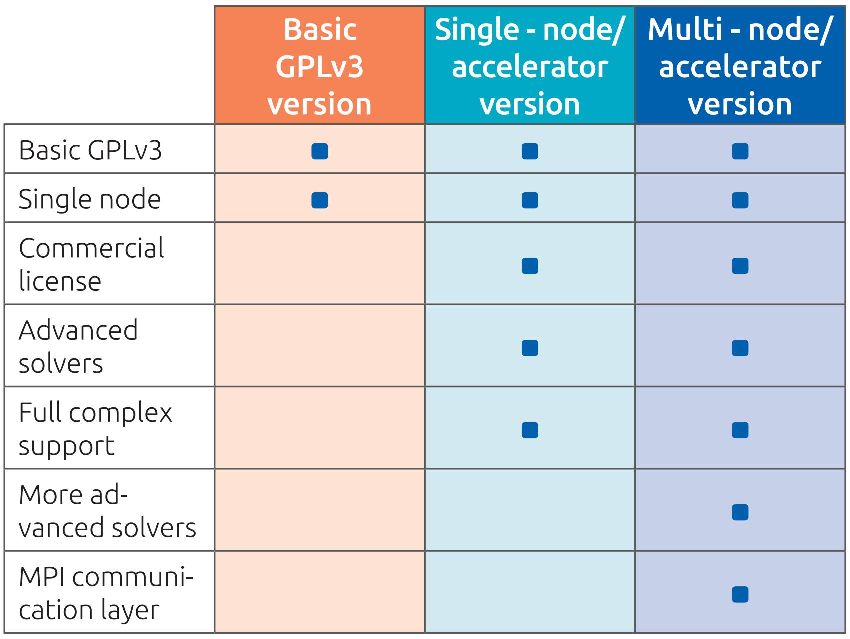
1.2 Purpose of this User Manual

The purpose of this document is to present the PARALUTON library step-by-step. This includes the installation process, internal structure and design, application programming interface (API) and examples. The change log of the software is also presented here.

All related documentation (web site information, user manual, white papers, doxygen) follows the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License [12]. A copy of the license can be found in the library package.

1.3 API Documentation

The most important library's functions are presented in this document. The library's full API (references) are documented via the automatic documentation system - doxygen [55]. The references are available on the PARALUTION web site.



	Basic GPLv3 version	Single - node/ accelerator version	Multi - node/ accelerator version
Basic GPLv3	■	■	■
Single node	■	■	■
Commercial license		■	■
Advanced solvers		■	■
Full complex support		■	■
More advanced solvers			■
MPI communication layer			■

Figure 1.2: The three PARALUTION versions

1.4 Dual License

The PARALUTION library is distributed within a dual license model. Three different version can be obtained – a GPLv3 version which is free of charge and two commercial versions.

1.4.1 Basic Versions

The basic version of the code is released under the GPL v3 license [18]. It provides the core components of the single node functionality of the library. This version of the library is free. Please, note that due to GPLv3 license model, any code which uses the library must be released as Open Source and it should have compliance to the GPLv3.

1.4.2 Commercial Versions

The user can purchase a commercial license which will grant the ability to embed the code into a (commercial) closed binary package. In addition these versions come with more features and with support. The only restriction imposed to the user is that she/he is not allowed to further distribute the source-code.

Single-node/accelerator Version

In addition to the basic version, the commercial license features:

- License to embed the code into a commercial/non-opensource product
- Full complex numbers support on Host, CUDA and OpenCL
- More iterative solvers - QMRCGStab, BiCGStab(l), FCG
- More AMG schemes - Ruge-Stüben, Pairwise
- More preconditioners - VariablePreconditioners

Multi-node/accelerator Version

In addition to the commercial basic version, the multi-node license features:

- License to embed the code into a commercial/non-opensource product
- Same functionality as for the single-node/accelerator version
- MPI layer for HPC clusters

- Global AMG solver (Pairwise)

Note Please note, that the multi-node version does not support Windows/Visual Studio.

1.5 Version Nomenclature

Please note the following compatibility policy with respect to the versioning. The version number x.y.z represents: x is the major (increases when modifications in the library have been made and/or a new API has been introduced), y is the minor (increases when new functionality (algorithms, schemes, etc) has been added, possibly small/no modification of the API) and z is the revision (increases due to bugfixing or performance improvement). The alpha and beta versions are denoted with *a* and *b*, typically these are pre-released versions.

As mentioned above there are three versions of PARALUTION, each release has the same version numbering plus an additional suffix for each type. They are abbreviated with "B" for the Basic version (free under GPLv3), with "S" for the single node/accelerator commercial and with "M" for the multi-node/accelerator commercial version.

1.6 Features Nomenclature

The functions described in this user manual follows the following nomenclature:

- *ValueType* – type of values can be double (D), float (F), integer (I), and complex (double and float). The particular bit representation (8,16,32,64bit) depends on your compiler and operating system.
- *Computation* – on which backend the computation can be performed, the abbreviation follows: Host backend with OpenMP (H); CUDA backend (C); OpenCL backend (O); Xeon Phi backend with OpenMP (X).
- *Available* – in which version this functionality is available: Basic/GPLv3 (B); Single-node/accelerator commercial version (S); Multi-node/accelerator commercial version (M).

The following example states that the function has double, float, integer and complex support; it can be performed on Host backend with OpenMP, CUDA backend, OpenCL backend, Xeon Phi backend with OpenMP; and it is available in all three versions of the library.

ValueType	Computation	Available
D,F,I,C	H,C,O,X	B,S,M

Solvers and preconditioners split the computation into a *Building phase* and *Solving phase* which can have different computation backend performance. In the following example the solver/preconditioner support double, float and complex; the building phase can be performed only on the Host with OpenMP or on CUDA; the solving phase can be performed on all other backends; it is available only in the commercial versions of the library.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O,X	S,M

Note The full complex support is available only in the commercial version, check Section 4.4.

1.7 Cite

If you like to cite the PARALUTION library you can do it as you like with citing our web site. Please specify the version of the software or/and date of accessing our web page, like that:

```
@misc{paralution,
author="{PARALUTION Labs}",
title="{PARALUTION vX.Y.Z}",
year="20XX",
note = {\url{http://www.paralution.com/}}
}
```

1.8 Website

The official web site of the library (including all (free and commercial) versions) is
http://www.paralution.com

Chapter 2

Installation

PARALUTION can be compiled under Linux/Unix, Windows and Mac systems.

Note Please check for additional remarks Sections 11.5.

2.1 Linux/Unix-like OS

After downloading and unpacking the library, the user needs to compile it. We provide two compilation configurations – *cmake* and *Makefile*.

2.1.1 Makefile

In this case, the user needs to modify the Makefile which contains the information about the available compilers. By default PARALUTION will only use gcc [20] compiler (no GPU support). The user can switch to icc [23] with or without MKL [24] support. To compile with GPU support, the user needs to uncomment the corresponding CUDA[1] [43] lines in the Makefile. The same procedure needs to be followed for the OpenCL [29] and for the OpenMP(MIC) backend.

Note During the compilation only one backend can be selected, i.e. if a GPU is available the user needs to select either CUDA or OpenCL support.

The default installation process can be summarized in the following lines:

```
wget http://www.paralution.com/download/paralution-x.y.z.tar.gz

tar zxvf paralution-x.y.z.tar.gz

cd paralution-x.y.z/src

make all
make install
```

where x.y.z is the version of the library.

Note Please note, that the multi-node version of PARALUTION can only be compiled using CMake.

2.1.2 CMake

The compilation with cmake [30] is easier to handle – all compiler specifications are determined automatically.

The compilation process can be performed by

```
wget http://www.paralution.com/download/paralution-x.y.z.tar.gz

tar zxvf paralution-x.y.z.tar.gz

cd paralution-x.y.z

mkdir build
cd build

cmake ..
make
```

[1]NVIDIA CUDA, when mentioned also includes CUBLAS and CUSPARSE

where x.y.z is the version of the library. Advanced compilation can be performed with *cmake -i ..*, you need this option to compile the library with OpenMP(MIC) backend.

The priority during the compilation process of the backends are: CUDA, OpenCL, MIC

You can also choose specific options via the command line, for example CUDA:

```
cd paralution-x.y.z

mkdir build
cd build

cmake -DSUPPORT_CUDA=ON -DSUPPORT_OMP=ON ..
make -j
```

For the Intel Xeon Phi, OpenMP(MIC) backend:

```
cd paralution-x.y.z

mkdir build
cd build

cmake -DSUPPORT_MIC=ON -DSUPPORT_CUDA=OFF -DSUPPORT_OCL=OFF   ..
make -j
```

Note ptk file is generated in the build directory when using cmake.

2.1.3 Shared Library

Both compilation processes produce a shared library file *libparalution.so*. Ensure that the library object can be found in your library path. If you do not copy the library to a specific location you can add the path under Linux in the *LD_LIBRARY_PATH* variable.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/paralution-x.y.z/build/lib/
```

2.2 Windows OS

This section will introduce a step-by-step guide to compile and use the PARALUTION library in a Windows based environment. ***Note*** Please note, that the multi-node version does not support Windows/Visual Studio.

2.2.1 OpenMP backend

PARALUTION with OpenMP backend comes with Visual Studio Project files that support Visual Studio 2010, 2012 and 2013. PARALUTION is built as a static library during this step-by-step guide.

- Open Visual Studio and navigate to *File\Open Project*.

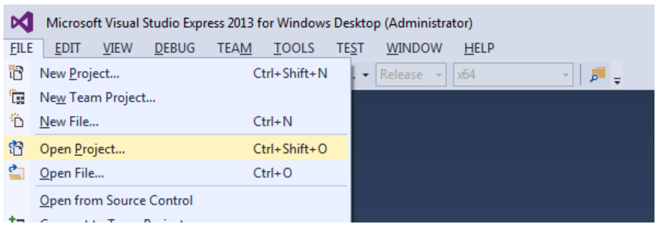

Microsoft Visual Studio Express 2013 for Windows Desktop (Administrator)
FILE EDIT VIEW DEBUG TEAM TOOLS TEST WINDOW HELP
New Project... Ctrl+Shift+N
New Team Project...
New File... Ctrl+N
Open Project... Ctrl+Shift+O
Open File... Ctrl+O
Open from Source Control
Release
x64

Figure 2.1: Open an existing Visual Studio project.

- Load the corresponding PARALUTION project file, located in the *visualstudio\paralution_omp* directory. The *PARALUTION* and *CG* projects should appear.
- Right-click the *PARALUTION* project, and start to build the library. Once finished, Visual Studio should report a successful built.
- Next, repeat the building procedure with the *CG* example project. Once finished, a successful built should be reported.

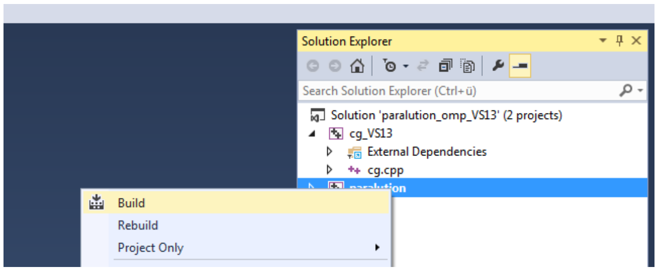
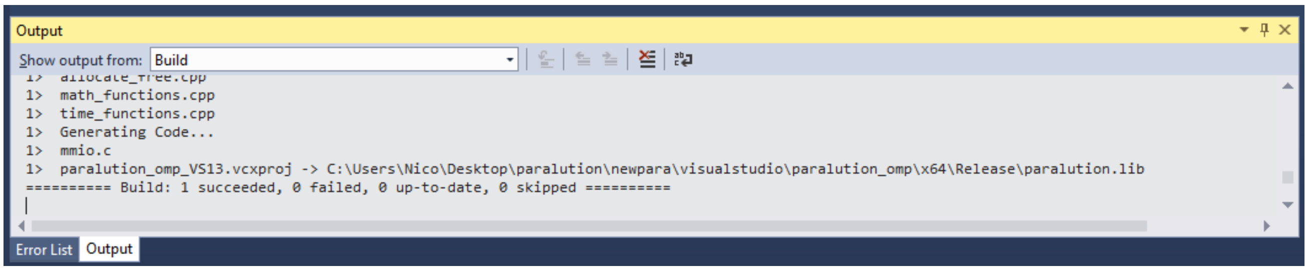


Figure 2.2: Build the PARALUTION library.



```
1>  allocate_free.cpp
1>  math_functions.cpp
1>  time_functions.cpp
1>  Generating Code...
1>  mmio.c
1>  paralution_omp_VS13.vcxproj -> C:\Users\Nico\Desktop\paralution\newpara\visualstudio\paralution_omp\x64\Release\paralution.lib
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Figure 2.3: Visual Studio output for a successful built of the static PARALUTION library.

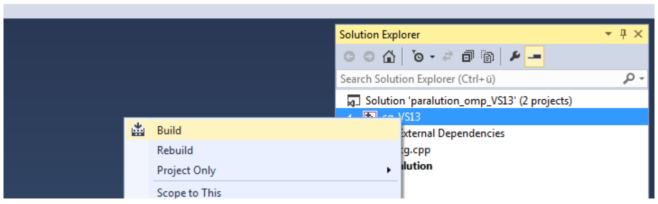
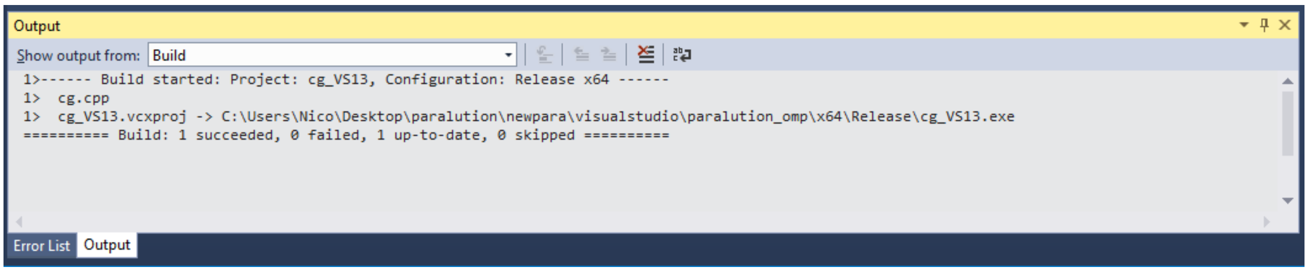


Figure 2.4: Build the PARALUTION CG example.



```
1>------ Build started: Project: cg_VS13, Configuration: Release x64 ------
1>  cg.cpp
1>  cg_VS13.vcxproj -> C:\Users\Nico\Desktop\paralution\newpara\visualstudio\paralution_omp\x64\Release\cg_VS13.exe
========== Build: 1 succeeded, 0 failed, 1 up-to-date, 0 skipped ==========
```

Figure 2.5: Visual Studio output for a successful built of the PARALUTION CG example.

- Finally, the *CG* executable should be located within the *visualstudio\paralution_omp\Release* directory.

Note For testing, Windows 7 and Visual Studio Express 2013 has been used.

Note OpenMP support can be enabled/disabled in the project properties. Navigate to *C++\Language* for the corresponding switch.

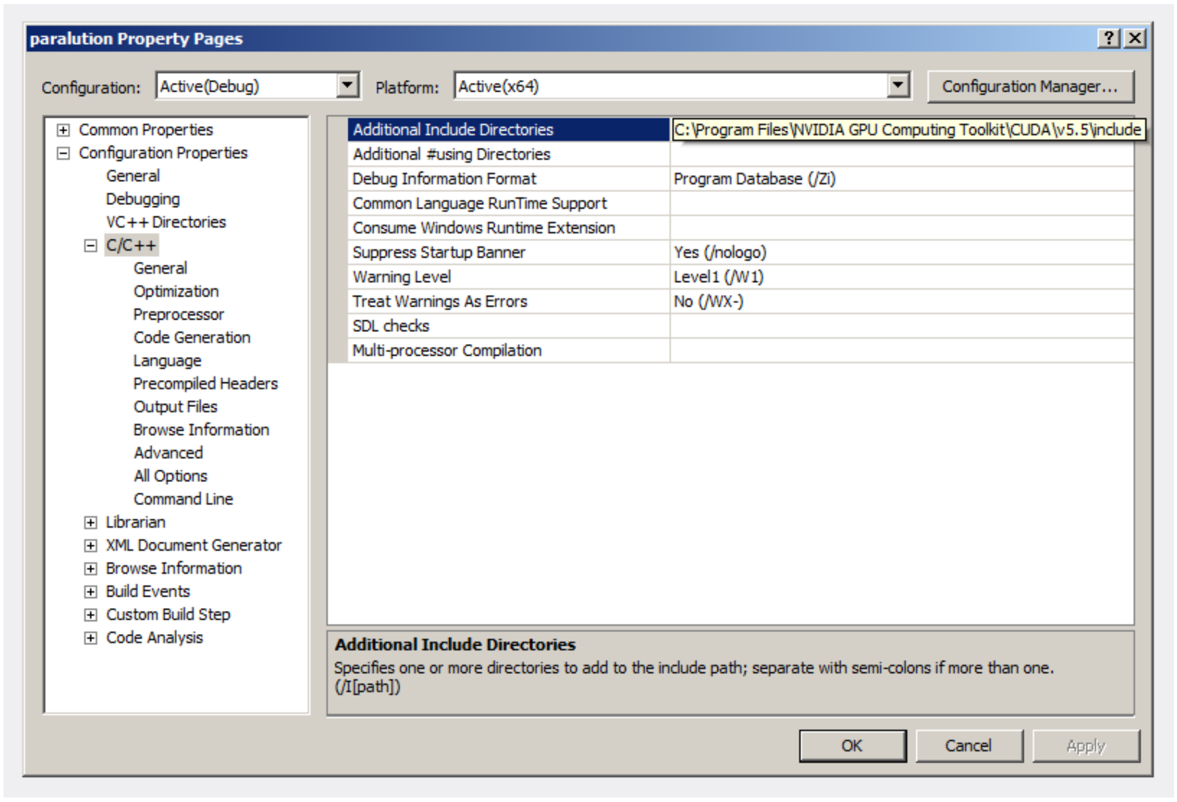
Note The current version of PARALUTION does not support MPI (i.e. the M-version of the library) under Windows.

2.2.2 CUDA backend

PARALUTION with CUDA backend comes with Visual Studio Project files that support Visual Studio 2010, 2012 and 2013. Please follow the same steps as for the OpenMP backend compilation but using the *visualstudio\paralution_cuda* directory.

2.2.3 OpenCL backend

PARALUTION with OpenCL backend comes with Visual Studio Project files that support Visual Studio 2010, 2012 and 2013. Please follow the same steps as for the OpenMP backend compilation but using the *visualstudio\paralution_ocl* directory. Additionally, the OpenCL include directory and OpenCL library directory need to be specified within Visual Studio, as illustrated in Figures 2.6 and 2.7.



Additional Include Directories	C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.5\include
Additional #using Directories	
Debug Information Format	Program Database (/Zi)
Common Language RunTime Support	
Consume Windows Runtime Extension	
Suppress Startup Banner	Yes (/nologo)
Warning Level	Level1 (/W1)
Treat Warnings As Errors	No (/WX-)
SDL checks	
Multi-processor Compilation	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

Figure 2.6: Setting up Visual Studio OpenCL include directory.

2.3 Mac OS

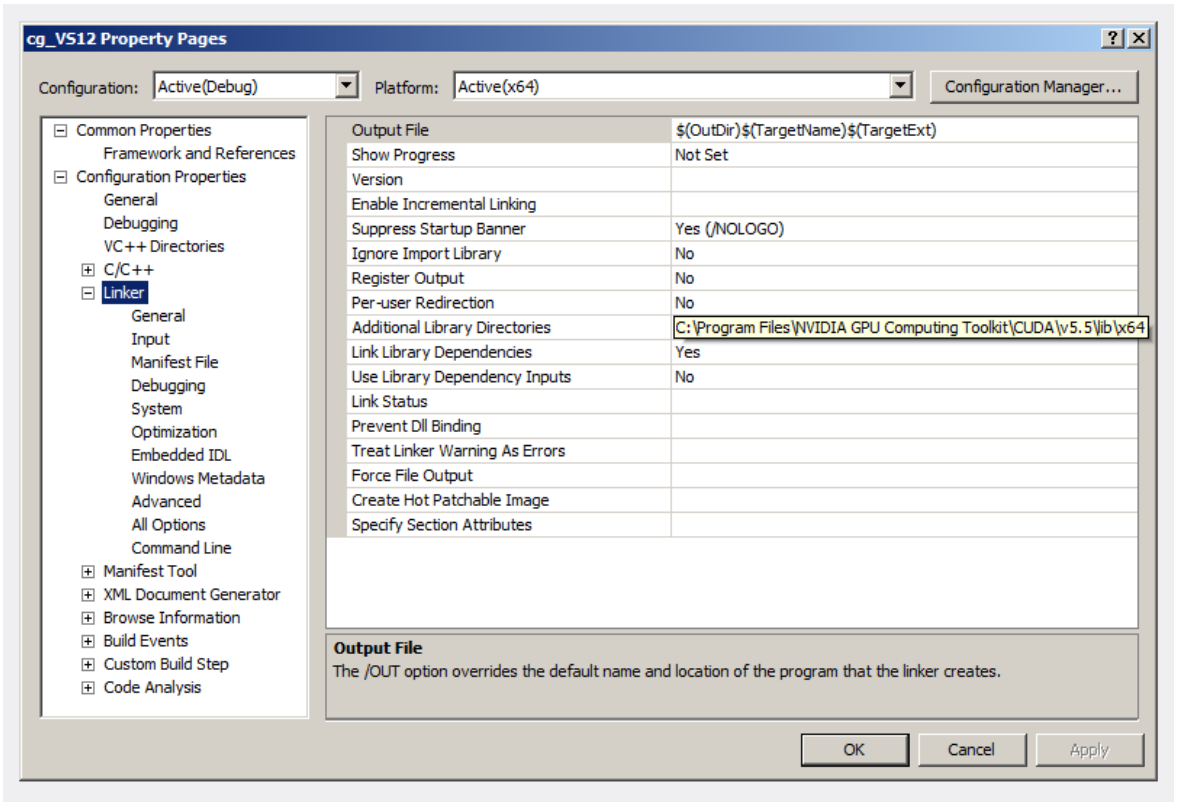
To compile PARALUTION under Mac, please follow the Linux/Unix-like OS instruction for the *CMake* compilation.

2.4 Supported Compilers

The library has been tested with the following compilers:

cmake	2.8.12.2; 3.0.2; 3.1.3; 3.2.0	B,S,M
gcc/g++	4.4.7; 4.6.3; 4.8.2	B,S,M
icc (MKL)	12.0; 13.1; 14.0.4; 15.0.0	B,S,M
CUDA	5.0; 5.5; 6.0; 6.5; 7.0; 7.5	B,S,M
Intel OpenCL	1.2	B,S,M
NVIDIA OpenCL	1.2	B,S,M
AMD OpenCL	1.2; 2.0	B,S,M
Visual Studio	2010, 2012, 2013	B,S
MPICH	3.1.3	M
OpenMPI	1.5.3; 1.6.3; 1.6.5; 1.8.4	M
Intel MPI	4.1.2; 5.0.1	M

Note Please note, that CUDA has limited ICC support.
Note Please note, that Intel MPI >= 5.0.0 is only supported by CMAKE >= 3.2.0.



Configuration: Active(Debug) Platform: Active(x64)

Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.5\lib\x64
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	
Force File Output	
Create Hot Patchable Image	
Specify Section Attributes	

Output File
The /OUT option overrides the default name and location of the program that the linker creates.

Figure 2.7: Setting up Visual Studio OpenCL library directory.

2.5 Simple Test

You can test the installation by running a CG solver on a Laplace matrix [38]. After compiling the library you can perform the CG solver test by executing:

```
cd paralution-x.y.z
cd build/bin

wget ftp://math.nist.gov/pub/MatrixMarket2/Harwell-Boeing/laplace/gr_30_30.mtx.gz
gzip -d gr_30_30.mtx.gz

./cg gr_30_30.mtx
```

2.6 Compilation and Linking

After compiling the PARALUTION library, the user need to specify the include and the linker path to compile a program.

```
g++ -O3 -Wall -I/path/to/paralution-x.y.z/build/inc -c main.cpp -o main.o
g++ -o main main.o -L/path/to/paralution-x.y.z/build/lib/ -lparalution
```

"Compilation and linking"

Before the execution of a program which has been compiled with PARALUTION, the library path need to be added to the environment variables, similar to

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/paralution-x.y.z/build/lib/
```

When compiling with MPI, library and program need to be compiled using *mpic++* or the corresponding MPI C++ compiler.

Chapter 3

Basics

3.1 Design and Philosophy

The main idea of the PARALUTION objects is that they are separated from the actual hardware specification. Once you declare a matrix, a vector or a solver they are initially allocated on the host (CPU). Then every object can be moved to a selected accelerator by a simple move-to-accelerator function. The whole execution mechanism is based on run-time type information (RTTI) which allows you to select where and how you want to perform the operations at run time. This is in contrast to the template-based libraries which need this information at compile time.

The philosophy of the library is to abstract the hardware-specific functions and routines from the actual program which describes the algorithm. It is hard and almost impossible for most of the large simulation software based on sparse computation to adapt and port their implementation in order to use every new technology. On the other hand, the new high performance accelerators and devices have the capability to decrease the computational time significantly in many critical parts.

This abstraction layer of the hardware specific routines is the core of PARALUTION's design, it is built to explore fine-grained level of parallelism suited for multi/many-core devices. This is in contrast to most of the parallel sparse libraries available which are mainly based on domain decomposition techniques. Thus, the design of the iterative solvers the preconditioners is very different. Another cornerstone of PARALUTION is the native support of accelerators - the memory allocation, transfers and specific hardware functions are handled internally in the library.

PARALUTION helps you to use accelerator technologies but does not force you to use them. As you can see later in this chapter, even if you offload your algorithms and solvers to the accelerator device, the same source code can be compiled and executed in a system without any accelerators.

3.2 Operators and Vectors

The main objects in PARALUTION are linear operators and vectors. All objects can be moved to an accelerator at run time – a structure diagram is presented in Figure 3.1. Currently, we support GPUs by CUDA (NVIDIA) and OpenCL (NVIDIA, AMD) backends, and we provide OpenMP MIC backend for the Intel Xeon Phi.

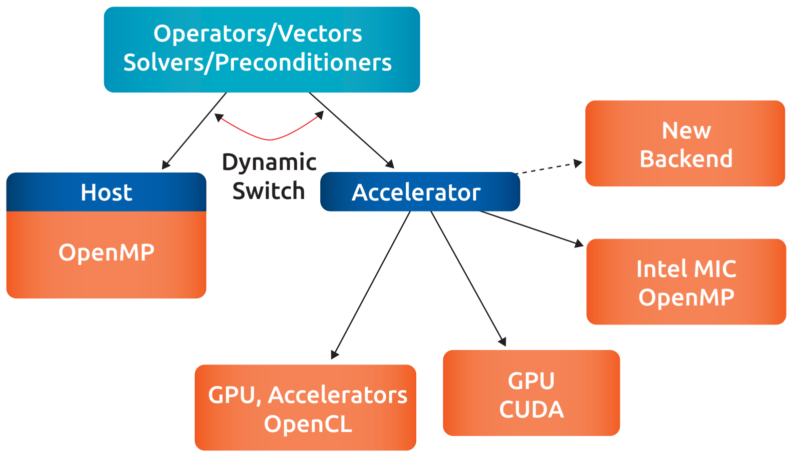


Figure 3.1: Host and backends structure for different hardware

The linear operators are defined as local or global matrices (i.e. on a single node or distributed/multi-node) and local stencils (i.e. matrix-free linear operations).

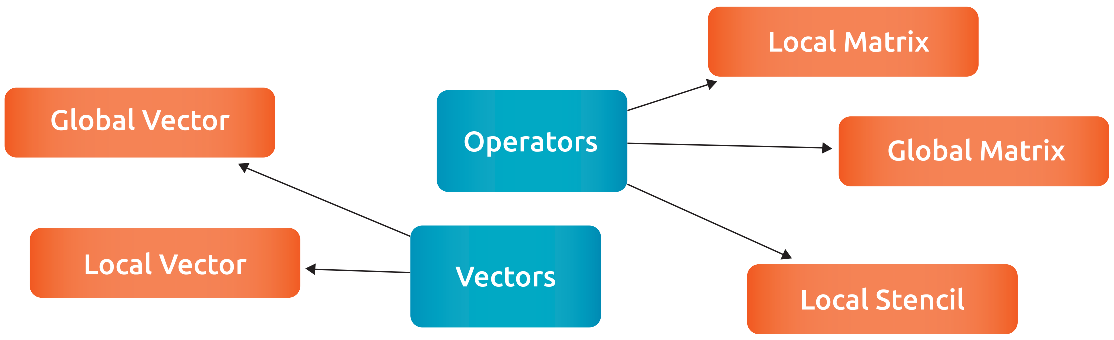


Figure 3.2: Operator and vector classes

The only template parameter of the operators and vectors is the data type (*ValueType*). The operator data type could be *float* or *double*, while the vector data type can be *float*, *double* or *int* (*int* is used mainly for the permutation vectors). In the current version, cross ValueType object operations are not supported.

Each of the objects contains a local copy of the hardware descriptor created by the *init_platform()* function. This allows the user to modify it according to his needs and to obtain two or more objects with different hardware specifications (e.g. different amount of OpenMP threads, CUDA block sizes, etc).

3.2.1 Local Operators/Vectors

By Local Operators/Vectors we refer to Local Matrices and Stencils, and to Local Vectors. By *Local* we mean the fact they stay on a single system. The system can contain several CPUs via UMA or NUMA memory system, it can contain an accelerator.

3.2.2 Global Operators/Vectors

By Global Operators/Vectors we refer to Global Matrix and to Global Vectors. By *Global* we mean the fact they can stay on a single or multiple nodes in a network. For this this type of computation, the communication is based on MPI.

3.3 Functionality on the Accelerators

Naturally, not all routines and algorithms can be performed efficiently on many-core systems (i.e. on accelerators). To provide full functionality the library has internal mechanisms to check if a particular routine is implemented on the accelerator. If not the object is moved to the host and the routine is computed there. This guarantees that your code will run (maybe not in the most efficient way) with any accelerator regardless of the available functionality for it.

3.4 Initialization of the Library

The body of a PARALUTION code is very simple, it should contain the header file and the namespace of the library. The program must contain an initialization call, which will check and allocate the hardware, and a finalizing call which will release the allocated hardware.

```
#include <paralution.hpp>

using namespace paralution;

int main(int argc, char* argv[]) {

  init_paralution();

  info_paralution();

  // ...

  stop_paralution();

}
```

"Initialize and shutdown PARALUTION"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 32
Host thread affinity policy - thread mapping on every core
Number of GPU devices in the system: 2
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: GPU(CUDA)
OpenMP threads:32
Selected GPU device: 0
------------------------------------------------
Device number: 0
Device name: Tesla K20c
totalGlobalMem: 4799 MByte
clockRate: 705500
compute capability: 3.5
ECCEnabled: 1
------------------------------------------------
------------------------------------------------
Device number: 1
Device name: Tesla K20c
totalGlobalMem: 4799 MByte
clockRate: 705500
compute capability: 3.5
ECCEnabled: 1
------------------------------------------------
```

"An example output for *info_paralution()* on a GPU (CUDA) system"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 32
Host thread affinity policy - thread mapping on every core
Number of OpenCL devices in the system: 2
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: OpenCL
OpenMP threads:32
Selected OpenCL platform: 0
Selected OpenCL device: 0
------------------------------------------------
Platform number: 0
Platform name: NVIDIA CUDA
Device number: 0
Device name: Tesla K20c
Device type: GPU
totalGlobalMem: 4799 MByte
clockRate: 705
OpenCL version: OpenCL 1.1 CUDA
------------------------------------------------
------------------------------------------------
Platform number: 0
Platform name: NVIDIA CUDA
Device number: 1
Device name: Tesla K20c
Device type: GPU
totalGlobalMem: 4799 MByte
```

```
clockRate: 705
OpenCL version: OpenCL 1.1 CUDA
------------------------------------------------------
```

"An example output for *info_paralution()* on a GPU (OpenCL) system"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 16
Host thread affinity policy - thread mapping on every core
MIC backed is initialized
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: MIC(OpenMP)
OpenMP threads:16
Number of MIC devices in the system: 2
Selected MIC devices:0
```

"An example output for *info_paralution()* on Intel Xeon Phi (OpenMP) system"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 2
Host thread affinity policy - thread mapping on every core
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: None
OpenMP threads:2
```

"An example output for *info_paralution()* on system without accelerator"

The *init_paralution()* function defines a backend descriptor with information about the hardware and its specifications. All objects created after that contain a copy of this descriptor. If the specifications of the global descriptor are changed (e.g. set different number of threads) and new objects are created, only the new objects will use the new configurations.

For control the library provides the following functions

- *select_device_paralution(int dev)* – this is a unified function which select a specific device. If you have compiled the library with a backend and for this backend there are several available cards you can use this function to select a particular one. This function works for all backends (CUDA, OpenCL, Xeon Phi).
- *set_omp_threads_paralution(int nthreads)* – with this function the user can set the number of OpenMP threads. This function has to be called after the *init_paralution()*.
- *set_gpu_cuda_paralution(int ndevice)* – in a multi-GPU system, the user can select a specific GPU by this function. This function has to be called before the *init_paralution()*.
- *set_ocl_paralution(int nplatform, int ndevice)* – in a multi-platform/accelerator system, the user can select a specific platform and device by this function. This function has to be called before the *init_paralution()*.

3.4.1 Thread-core Mapping

The number of threads which PARALUTION will use can be set via the *set_omp_threads_paralution()* function or by the global OpenMP environment variable (for Unix-like OS this is *OMP_NUM_THREADS*). During the initialization phase, the library provides affinity thread-core mapping based on:

- if the number of cores (including hyperthreading cores) is greater or equal than two times the number of threads – then all the threads can occupy every second core ID (e.g. $0, 2, 4, ...$). This is to avoid having two threads working on the same physical core when hyperthreading is enabled.
- if the number of threads is less or equal to the number of cores (including hyperthreading), and the previous clause is false. Then the threads can occupy every core ID (e.g. $0, 1, 2, 3, ...$).
- if non of the above criteria is matched – the default thread-core mapping is used (typically set by the OS).

Note The thread-core mapping is available only for Unix-like OS. For Windows OS, the thread-core mapping is selected by the operating system.

Note The user can disable the thread affinity by calling *set_omp_affinity(false)* (and enable it with *set_omp_affinity(true)*), before initializing the library (i.e. before *init_paralution()*).

3.4.2 OpenMP Threshold Size

Whenever you want to work on a small problem, you might observe that the OpenMP Host backend is (slightly) slower than using no OpenMP. This is mainly attributed to the small amount of work which every thread should perform and the large overhead of forking/joining threads. This can be avoid by the OpenMP threshold size parameter in PARALUTION. The default threshold is set to 10,000, which means that all matrices under (and equal) this size will use only one thread (disregarding the number of OpenMP threads set in the system). The threshold can be modified via the function *set_omp_threshold()*.

3.4.3 Disable the Accelerator

If you want to disable the accelerator (without recompiling the code), you need to call *disable_accelerator_paralution()* function before the *init_paralution()*.

3.4.4 MPI and Multi-accelerators

When initializing the library with MPI, the user need to pass the rank of the MPI process as well as the number of accelerators which are available on each node. Basically, in this way the user can specify the mapping of MPI process and accelerators – the allocated accelerator will be *rank % num_dev_per_node*. Thus the user can run two MPI process on systems with two GPUs by specifying the number of devices to 2.

```
#include <paralution.hpp>
#include <mpi.h>

using namespace paralution;

int main(int argc, char* argv[]) {

  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;

  int num_processes;
  int rank;
  MPI_Comm_size(comm, &num_processes);
  MPI_Comm_rank(comm, &rank);

  init_paralution(rank, 2);

  info_paralution();

  // ...

  stop_paralution();

}
```

"Initialize and shutdown PARALUTION with MPI"

```
The default OS thread affinity configuration will be used
Number of GPU devices in the system: 2
The default OS thread affinity configuration will be used
PARALUTION ver M0.8.0
PARALUTION platform is initialized
Accelerator backend: GPU(CUDA)
OpenMP threads:1
Selected GPU device: 0
------------------------------------------------------------
Device number: 0
Device name: Tesla K20c
totalGlobalMem: 4799 MByte
clockRate: 705500
compute capability: 3.5
ECCEnabled: 1
```

```
——————————————————————————————————————————————————
——————————————————————————————————————————————————
Device number: 1
Device name: Tesla K20c
totalGlobalMem: 4799 MByte
clockRate: 705500
compute capability: 3.5
ECCEnabled: 1
——————————————————————————————————————————————————
MPI rank:0
MPI size:2
```

"An example output for *info_paralution()* with 2 MPI processes"

3.5 Automatic Object Tracking

By default, after the initialization of the library, it tracks all objects and releasing the allocated memory in them when the library is stopped. This ensure large memory leaks when the objects are allocated but not freed. The user can disable the tracking by editing *src/utils/def.hpp*.

3.6 Verbose Output

PARALUTION provides different levels of output messages. They can be set in the file *src/utils/def.hpp* before the compilation of the library. By setting a higher level, the user will obtain more detailed information about the internal calls and data transfers to and from the accelerators.

3.7 Verbose Output and MPI

To avoid all MPI processes to print information on the screen the default configuration is that only RANK 0 outputs information on the screen. The user can change the RANK or allow all RANK to print by modifying *src/utils/def.hpp*. If file logging is enabled, all ranks write into corresponding log files.

3.8 Debug Output

You can also enable debugging output which will print almost every detail in the program, including object constructor/destructor, address of the object, memory allocation, data transfers, all function calls for matrices, vectors, solvers and preconditioners. The debug flag can be set in *src/utils/def.hpp*.

When enabled, additional *assert()*s are being checked during the computation. This might decrease the performance of some operations.

3.9 File Logging

All output can be logged into a file, the file name will be *paralution-XXXX.log*, where *XXX* will be a counter in milliseconds. To enable the file you need to edit *src/utils/def.hpp*.

3.10 Versions

For checking the version in your code you can use the PARALUTION's pre-defined macros.

```
__PARALUTION_VER_MAJOR -- gives the version major value
__PARALUTION_VER_MINOR -- gives the version minor value
__PARALUTION_VER_REV   -- gives the version revision

__PARALUTION_VER_PRE -- gives pre-releases (as "ALPHA" or "BETA")
```

The final *__PARALUTION_VER* gives the version as $10000 major + 100 minor + revision$.

The different type of versions (Basic/GPLv3; Single-node; Multi-node) are defined as *__PARALUTION_VER_TYPE* *B* for Basic/GPLv3; *S* for Single-node; and *M* for Multi-node.

Chapter 4

Single-node Computation

4.1 Introduction

In this chapter we describe all base objects (matrices, vectors and stencils) for computation on single-node (shared-memory) systems. A typical configuration is presented on Figure 4.1.

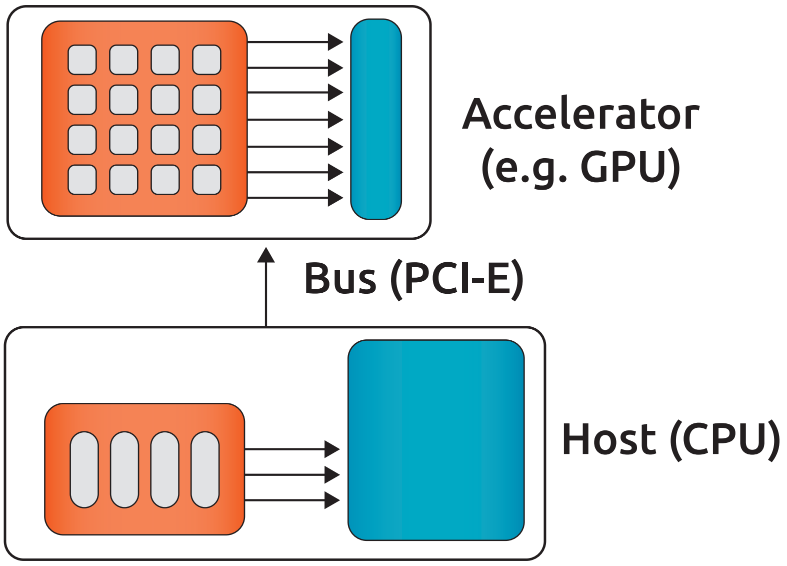


Figure 4.1: A typical single-node configuration, where gray-boxes represent the cores, blue-boxes represent the memory, arrows represent the bandwidth

The compute node contains none, one or more accelerators. The compute node could be any kind of shared-memory (single, dual, quad CPU) system. Note that the memory of the accelerator and of the host can be physically different.

4.2 Code Structure

The *Data* is an object, pointing to the *BaseMatrix* class. The pointing is coming from either a *HostMatrix* or an *AcceleratorMatrix*. The *AcceleratorMatrix* is created by an object with an implementation in each of the backends (CUDA, OpenCL, Xeon Phi) and a matrix format. Switching between host or accelerator matrix is performed in the *LocalMatrix* class. The *LocalVector* is organized in the same way.

Each matrix format has its own class for the host and for each accelerator backend. All matrix classes are derived from the *BaseMatrix* which provides the base interface for computation as well as for data accessing, see Figure4.4. The GPU (CUDA backend) matrix structure is presented in Figure 4.5, all other backend follows the same organization.

4.3 Value Type

The value (data) type of the vectors and the matrices is defined as a template. The matrix can be of type *float* (32-bit), *double* (64-bit) and *complex* (64/128-bit). The vector can be *float* (32-bit), *double* (64-bit), *complex*

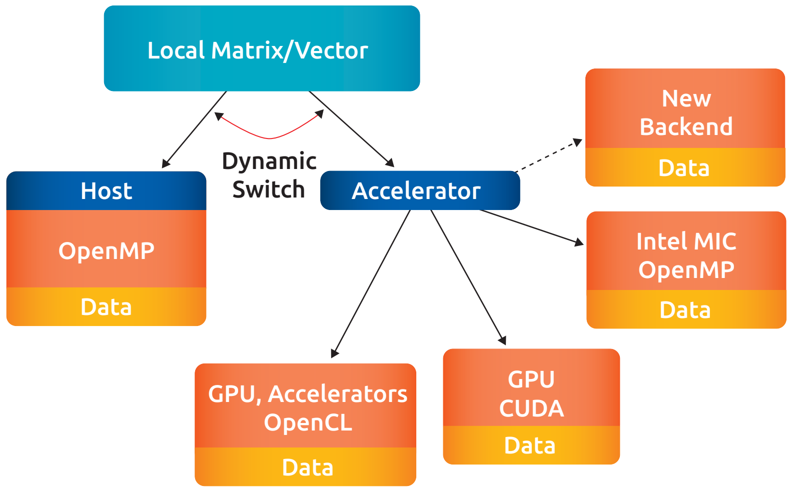


Figure 4.2: Local Matrices and Vectors

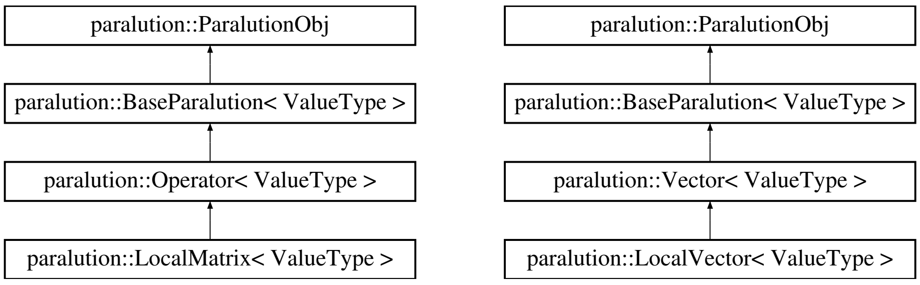

paralution::ParalutionObj
paralution::BaseParalution< ValueType >
paralution::Operator< ValueType >
paralution::LocalMatrix< ValueType >
paralution::ParalutionObj
paralution::BaseParalution< ValueType >
paralution::Vector< ValueType >
paralution::LocalVector< ValueType >

Figure 4.3: LocalMatrix and Local Vector

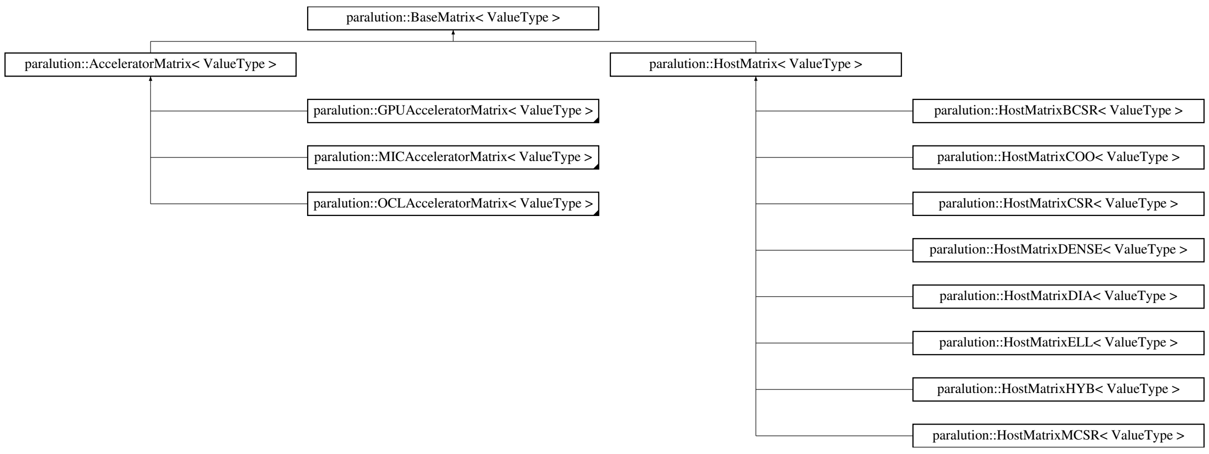

paralution::BaseMatrix< ValueType >
paralution::AcceleratorMatrix< ValueType >
paralution::HostMatrix< ValueType >
paralution::GPUAcceleratorMatrix< ValueType >
paralution::MICAcceleratorMatrix< ValueType >
paralution::OCLAcceleratorMatrix< ValueType >
paralution::HostMatrixBCSR< ValueType >
paralution::HostMatrixCOO< ValueType >
paralution::HostMatrixCSR< ValueType >
paralution::HostMatrixDENSE< ValueType >
paralution::HostMatrixDIA< ValueType >
paralution::HostMatrixELL< ValueType >
paralution::HostMatrixHYB< ValueType >
paralution::HostMatrixMCSR< ValueType >

Figure 4.4: BaseMatrix

(64/128-bit) and *int* (32/64-bit). The information about the precision of the data type is shown in the *Print()* function.

4.4 Complex Support

PARALUTION supports complex computation in all functions due to its internal template structure. The host implementation is based on the *std::complex*. In binary, the data is the same also for the CUDA and for the OpenCL backend. The complex support of the backends with respect to the versions is presented in Table 4.1.

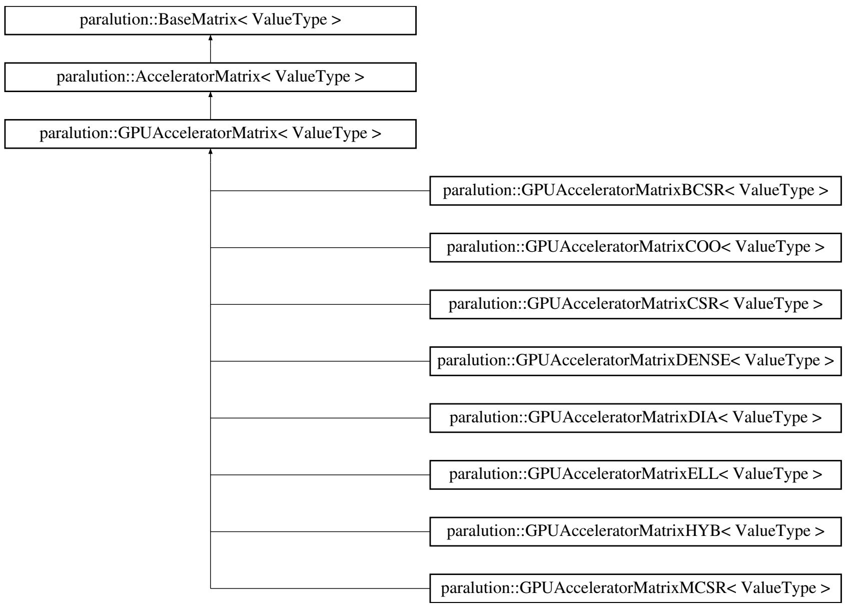

paralution::BaseMatrix< ValueType >
paralution::AcceleratorMatrix< ValueType >
paralution::GPUAcceleratorMatrix< ValueType >
paralution::GPUAcceleratorMatrixBCSR< ValueType >
paralution::GPUAcceleratorMatrixCOO< ValueType >
paralution::GPUAcceleratorMatrixCSR< ValueType >
paralution::GPUAcceleratorMatrixDENSE< ValueType >
paralution::GPUAcceleratorMatrixDIA< ValueType >
paralution::GPUAcceleratorMatrixELL< ValueType >
paralution::GPUAcceleratorMatrixHYB< ValueType >
paralution::GPUAcceleratorMatrixMCSR< ValueType >

Figure 4.5: GPUAcceleratorMatrix (CUDA)

	Host	CUDA	OpenCL	Xeon Phi
B	Yes	No	No	No
S	Yes	Yes	Yes	No
M	Yes	Yes	Yes	No

Table 4.1: Complex support

4.5 Allocation and Free

The allocation functions require a name of the object (this is only for information purposes) and corresponding size description for vector and matrix objects.

```
LocalVector<ValueType> vec;

vec.Allocate("my vector", 100);
vec.Clear();
```

"Vector allocation/free"

```
LocalMatrix<ValueType> mat;

mat.AllocateCSR("my csr matrix", 456, 100, 100); // nnz, rows, columns
mat.Clear();

mat.AllocateCOO("my coo matrix", 200, 100, 100); // nnz, rows, columns
mat.Clear();
```

"Matrix allocation/free"

4.6 Matrix Formats

Matrices where most of the elements are equal to zero are called sparse. In most practical applications the number of non-zero entries is proportional to the size of the matrix (e.g. typically, if the matrix A is $\mathbb{R}^{N \times N}$ then the number of elements are of order $O(N)$). To save memory, we can avoid storing the zero entries by introducing a structure corresponding to the non-zero elements of the matrix. PARALUTION supports sparse CSR, MCSR, COO, ELL, DIA, HYB and dense matrices (DENSE).

To illustrate the different format, let us consider the following matrix in Figure 4.6.

Here the matrix is $A \in \mathbb{R}^{5 \times 5}$ with 11 non-zero entries. The indexing in all formats described below are zero based (i.e. the index values starts at 0, not with 1).

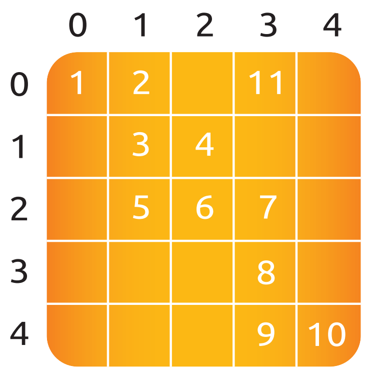


Figure 4.6: A sparse matrix example

Note The functionality of every matrix object is different and depends on the matrix format. The CSR format provides the highest support for various functions. For a few operations an internal conversion is performed, however, for many routines an error message is printed and the program is terminated.

Note In the current version, some of the conversions are performed on the host (disregarding the actual object allocation - host or accelerator).

```
mat.ConvertToCSR();
// Perform a matrix−vector multiplcation in CSR format
mat.Apply(x, &y);

mat.ConvertToELL();
// Perform a matrix−vector multiplcation in ELL format
mat.Apply(x, &y);
```

"Conversion between matrix formats"

```
mat.ConvertTo(CSR);
// Perform a matrix−vector multiplcation in CSR format
mat.Apply(x, &y);

mat.ConvertTo(ELL);
// Perform a matrix−vector multiplcation in ELL format
mat.Apply(x, &y);
```

"Conversion between matrix formats (alternative)"

4.6.1 Coordinate Format – COO

The most intuitive sparse format is the coordinate format (COO). It represent the non-zero elements of the matrix by their coordinates, we need to store two index arrays (one for row and one for column indexing) and the values. Thus, our example matrix will have the following structure:

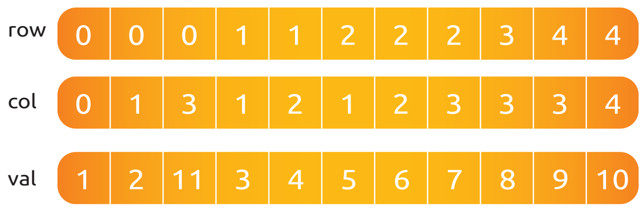

row 0 0 0 1 1 2 2 2 3 4 4
col 0 1 3 1 2 1 2 3 3 3 4
val 1 2 11 3 4 5 6 7 8 9 10

Figure 4.7: Sparse matrix in COO format

4.6.2 Compressed Sparse Row/Column Format – CSR/CSC

One of the most popular formats in many scientific codes is the compressed sparse row (CSR) format. In this format, we do not store the whole row indices but we only save the offsets to positions. Thus, we can easily jump to any row and we can access sequentially all elements there. However, this format does not allow sequential accessing of the column entries.

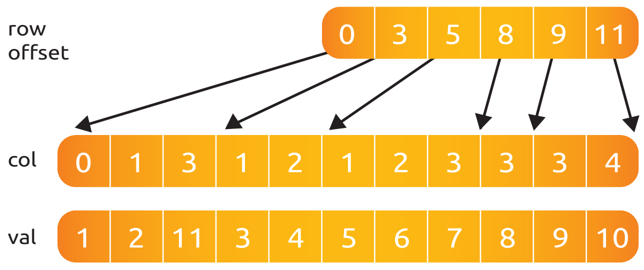


Figure 4.8: Sparse matrix in CSR format

Analogy to this format is the compressed sparse column (CSC), where we represent the offsets by the column – it is clear that we can traverse column elements sequentially.

In many finite element (difference/volumes) applications, the diagonal elements are non-zero. In such cases, we can store them at the beginning of the value arrays. This format is often referred as modified compressed sparse row/column (MCSR/MCSC).

4.6.3 Diagonal Format – DIA

If all (or most) of the non-zero entries belong to a few diagonals of matrix, we can store them with the responding offsets. In our example, we have 4 diagonal, the main diagonal (denoted with 0 offset) is fully occupied while the others contain zero entries.

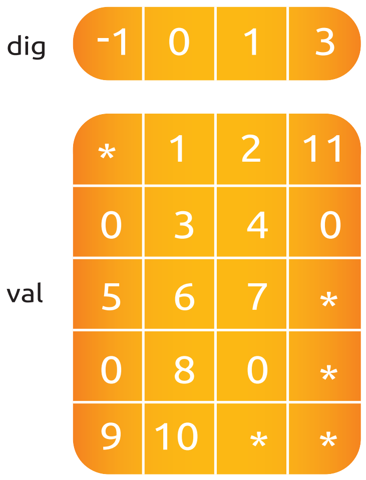


Figure 4.9: Sparse matrix in DIA format

Please note, that the values in this format are stored as array with size $D \times N_D$, where D is the number of diagonals in the matrix and N_D is the number of elements in the main diagonal. Since, not all values in this array are occupied - the not accessible entries are denoted with star, they correspond to the offsets in the diagonal array (negative values represent offsets from the beginning of the array).

4.6.4 ELL Format

The ELL format can be seen as modification of the CSR, where we do not store the row offsets. Instead, we have a fixed number of elements per row.

4.6.5 HYB Format

As you can notice the DIA and ELL cannot represent efficiently completely unstructured sparse matrix. To keep the memory footprint low DIA requires the elements to belong to a few diagonals and the ELL format needs fixed number of elements per row. For many applications this is a too strong restriction. A solution to this issue is to represent the more regular part of the matrix in such a format and the remaining part in COO format.

The HYB format, implemented in PARALUTION, is a mix between ELL and COO, where the maximum elements per row for the ELL part is computed by nnz/num row.

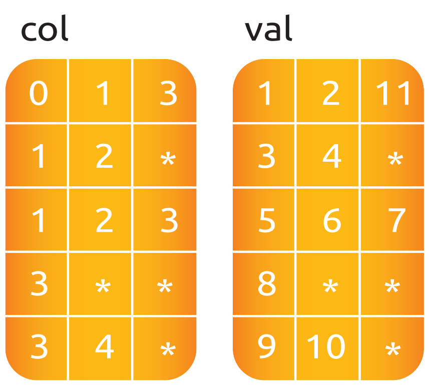


Figure 4.10: Sparse matrix in ELL format

4.6.6 Memory Usage

The memory footprint of the different matrix formats is presented in the following table. Here, we consider a $N \times N$ matrix where the number of non-zero entries are denoted with NNZ.

Format	Structure	Values
Dense	–	$N \times N$
COO	$2 \times NNZ$	NNZ
CSR	$N + 1 + NNZ$	NNZ
ELL	$M \times N$	$M \times N$
DIA	D	$D \times N_D$

For the ELL matrix M characterizes the maximal number of non-zero elements per row and for the DIA matrix D defines the number of diagonals and N_D defines the size of the main diagonal.

4.6.7 Backend support

	Host	CUDA	OpenCL	MIC/Xeon Phi
CSR	Yes	Yes	Yes	Yes
COO	Yes[1]	Yes	Yes	Yes[1]
ELL	Yes	Yes	Yes	Yes
DIA	Yes	Yes	Yes	Yes
HYB	Yes	Yes	Yes	Yes
DENSE	Yes	Yes	Yes[2]	No
BCSR	No	No	No	No

4.7 I/O

The user can read and write matrix files stored in Matrix Market Format [37].

```
LocalMatrix<ValueType> mat;
mat.ReadFileMTX("my_matrix.mtx");
mat.WriteFileMTX("my_matrix.mtx");
```

"I/O MTX Matrix"

Matrix files in binary format are also supported for the compressed sparse row storage format.

```
LocalMatrix<ValueType> mat;
mat.ReadFileCSR("my_matrix.csr");
mat.WriteFileCSR("my_matrix.csr");
```

"I/O CSR Binary Matrix"

[1]Serial version
[2]Basic version

The binary format stores the CSR relevant data as follows

```
out.write((char*) &nrow, sizeof(IndexType));
out.write((char*) &ncol, sizeof(IndexType));
out.write((char*) &nnz,  sizeof(IndexType));
out.write((char*) row_offset, (nrow+1)*sizeof(IndexType));
out.write((char*) col, nnz*sizeof(IndexType));
out.write((char*) val, nnz*sizeof(ValueType));
```

"CSR Binary Format"

The vector can be read or written via ASCII formatted files

```
LocalVector<ValueType> vec;

vec.ReadFileASCII("my_vector.dat");
vec.WriteFileASCII("my_vector.dat");
```

"I/O ASCII Vector"

4.8 Access

ValueType	Computation	Available
D,F,I,C	H	B,S,M

The elements in the vector can be accessed via *[]* operators when the vector is allocated on the host. In the following example, a vector is allocated with 100 elements and initialized with 1 for all odd elements and -1 for all even elements.

```
LocalVector<ValueType> vec;

vec.Allocate("vector", 100);

vec.Ones();

for (int i=0; i<100; i=i+2)
  vec[i] = -1;
```

"Vector element access"

Note Accessing elements via the *[]* operators is slow. Use this for debugging only.

There is no direct access to the elements of matrices due to the sparsity structure. Matrices can be imported by a copy function, for CSR matrix this is *CopyFromCSR()* and *CopyToCSR()*.

```
// Allocate the CSR matrix
int *row_offsets = new int[100+1];
int *col = new int[345];
ValueType *val = new ValueType[345];

// fill the CSR matrix
...

// Create a PARALUTION matrix
LocalMatrix<ValueType> mat;

// Import matrix to PARALUTION
mat.AllocateCSR("my matrix", 345, 100, 100);
mat.CopyFromCSR(row_offsets, col, val);

// Export matrix from PARALUTION
// the row_offsets, col, val have to be allocated
mat.CopyToCSR(row_offsets, col, val);
```

"Matrix access"

4.9 Raw Access to the Data

ValueType	Computation	Available
D,F,I,C	H,C	B,S,M

For vector and matrix objects, you can have direct access to the raw data via pointers. You can set already allocated data with the *SetDataPtr()* function.

```
LocalVector<ValueType> vec;

ValueType *ptr_vec = new ValueType[200];

vec.SetDataPtr(&ptr_vec, "vector", 200);
```

"Set allocated data to a vector"

```
// Allocate the CSR matrix
int *row_offsets = new int[100+1];
int *col = new int[345];
ValueType *val = new ValueType[345];

// fill the CSR matrix
...

// Create a PARALUTION matrix
LocalMatrix<ValueType> mat;

// Set the CSR matrix in PARALUTION
mat.SetDataPtrCSR(&row_offsets, &col, &val,
                  "my matrix",
                  345, 100, 100);
```

"Set allocated data to a CSR matrix"

With *LeaveDataPtr()* you can obtain the raw data from the object. This will leave the object empty.

```
LocalVector<ValueType> vec;

ValueType *ptr_vec = NULL;

vec.Allocate("vector", 100);

vec.LeaveDataPtr(&ptr_vec);
```

"Get (steal) the data from a vector"

```
// Create a PARALUTION matrix
LocalMatrix<ValueType> mat;

// Allocate and fill the PARALUTION matrix mat
...


// Define external CSR structure
int *row_offsets = NULL;
int *col = NULL;
ValueType *val = NULL;

mat.LeaveDataPtrCSR(&row_offsets, &col, &val);
```

"Get (steal) the data from a CSR matrix"

After calling the *SetDataPtr*()* functions (for Vectors or Matrices), the passed pointers will be set to NULL.

Note If the object is allocated on the host then the pointers from the *SetDataPtr()* and *LeaveDataPtr()* will be on the host, if the vector object is on the accelerator then the data pointers will be on the accelerator.

Note If the object is moved to and from the accelerator then the original pointer will be invalid.

Note Never rely on old pointers, hidden object movement to and from the accelerator will make them invalid.

Note Whenever you pass or obtain pointers to/from a PARALUTION object, you need to use the same memory allocation/free functions, please check the source code for that (for Host *src/utils/allocate_free.cpp* and for Host/CUDA *src/base/gpu/gpu_allocate_free.cu*)

4.10 Copy CSR Matrix Host Data

ValueType	Computation	Available
D,F,C	H,C,O	B,S,M

If the CSR matrix data pointers are only accessible as constant, the user can create a PARALUTION matrix object and pass const CSR host pointers by using the *CopyFromHostCSR()* function. PARALUTION will then allocate and copy the CSR matrix on the corresponding backend, where the original object was located at.

4.11 Copy Data

ValueType	Computation	Available
D,F,I,C	H,C	B,S,M

The user can copy data to and from a local vector via the *CopyFromData()* and *CopyToData()* functions. The vector must be allocated before with the corresponding size of the data.

4.12 Object Info

Information about the object can be printed with the *info()* function

```
mat.info();
vec.info();
```

"Vector/Matrix information"

```
LocalMatrix name=G3_circuit.mtx; rows=1585478; cols=1585478; nnz=7660826;
    prec=64bit; asm=no; format=CSR; backends={CPU(OpenMP), GPU(CUDA)};
    current=CPU(OpenMP)

LocalVector name=x; size=900; prec=64bit; host backend={CPU(OpenMP)};
    accelerator backend={No Accelerator}; current=CPU(OpenMP)
```

"Vector/Matrix information"

In this example, the matrix has been loaded, stored in CSR format, double precision, the library is compiled with CUDA support and no MKL, and the matrix is located on the host. The vector information is coming from another compilation of the library with no OpenCL/CUDA/MKL support.

4.13 Copy

ValueType	Computation	Available
D,F,I,C	H,C,O,X	B,S,M

All matrix and vector objects provide a *CopyFrom()* and a *CopyTo()* function. The destination object should have the same size or be empty. In the latter case the object is allocated at the source platform.

Note This function allows cross platform copying - one of the objects could be allocated on the accelerator backend.

```
LocalVector<ValueType> vec1, vec2;

// Allocate and init vec1 and vec2
// ...

vec1.MoveToAccelerator();

// now vec1 is on the accelerator (if any)
// and vec2 is on the host

// we can copy vec1 to vec2 and vice versa

vec1.CopyFrom(vec2);
```

"Vector copy"

Note For vectors, the user can specify source and destination offsets and thus copy only a part of the whole vector into another vector.

Note When copying a matrix - the source and destination matrices should be in the same format.

4.14 Clone

ValueType	Computation	Available
D,F,I,C	H,C,O,X	B,S,M

The copy operators allow you to copy the values of the object to another object, without changing the backend specification of the object. In many algorithms you might need auxiliary vectors or matrices. These objects can be cloned with the function *CloneFrom()*.

```
LocalVector<ValueType> vec;

// allocate and init vec (host or accelerator)
// ...

LocalVector<ValueType> tmp;

// tmp will have the same values
// and it will be on the same backend as vec
tmp.CloneFrom(vec);
```

"Clone"

If the data of the object needs to be kept, then you can use the *CloneBackend()* function to copy (clone) only the backend.

```
LocalVector<ValueType> vec;
LocalMatrix<ValueType> mat;

// allocate and init vec, mat (host or accelerator)
// ...

LocalVector<ValueType> tmp;

// tmp and vec will have the same backend as mat
tmp.CloneBackend(mat);
vec.CloneBackend(mat);

// the matrix-vector multiplication will be performed
// on the backend selected in mat
mat.Apply(vec, &tmp);
```

"Clone backend"

4.15 Assembling

ValueType	Computation	Available
D,F,I,C	H (CSR-only)	B,S,M

In many codes based on finite element, finite difference or finite volume, the user need to assemble the matrix before solving the linear system. For this purpose we provide an assembling function – the user need to pass three arrays, representing the row and column index and the value; similar function is provided for the vector assembling.

$$A_{i,j} = \sum v_{i,j}. \tag{4.1}$$

```
LocalMatrix<ValueType> mat;
LocalVector<ValueType> rhs;

// rows
int i[11] = {0, 1, 2, 3, 4, 5, 1, 2, 4, 3, 3 };

// cols
int j[11] = {0, 1, 2, 3, 4, 5, 2, 2, 5, 2, 2 };

// values
ValueType v[11] = {2.3, 3.5, 4.7, 6.3, 0.4, 4.3, 6.2, 4.4, 4.6, 0.7, 4.8 };

mat.Assemble(i, j, v, // tuple (i,j,v)
             11,      // size of the input array
             "A");    // name of the matrix

rhs.Assemble(i, a, // assembling data
             11,   // size of the input array
             "rhs"); // name of the vector
```

"Matrix assembling"

In this case the function will determine the matrix size, the number of non-zero elements, as well as the non-zero pattern and it will assemble the matrix in CSR format.

If the size of the matrix is know, then the user can pass this information to the assembling function and thus it will avoid a loop for checking the indexes before the actual assembling procedure.

```
mat.Assemble(i, j, a, 11, "A", 6, 6);
```

"Matrix assembling with known size"

The matrix function has two implementations – serial and OpenMP parallel (host only).

In many cases, one might want to assemble the matrix in a loop by modifying the original index pattern – typically for time-dependent/non-linear problems. In this case, the matrix does not need to be fully assembled, the user can use the *AssembleUpdate()* function to provide the new values.

```
LocalVector<ValueType> x, rhs;
LocalMatrix<ValueType> mat;

int i[11] = {0, 1, 2, 3, 4, 5, 1, 2, 4, 3, 3 };
int j[11] = {0, 1, 2, 3, 4, 5, 2, 2, 5, 2, 2 };
ValueType a[11] = {2.3, 3.5, 4.7, 6.3, 0.4, 4.3, 6.2, 4.4, 4.6, 0.7, 4.8 };

mat.Assemble(i, j, a, 11, "A");
rhs.Assemble(i, a, 11, "rhs");
x.Allocate("x", mat.get_ncol());

mat.MoveToAccelerator();
rhs.MoveToAccelerator();
```

```
x.MoveToAccelerator();

GMRES<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.SetOperator(mat);
ls.Build();

for (int k=0; k<10; k++) {

  mat.Zeros();
  x.Zeros();
  rhs.Zeros();

  // Modify the assembling data
  a[4] = k*k*3.3;
  a[8] = k*k-1.3;
  a[10] = k*k/2.0;
  a[3] = k;

  mat.AssembleUpdate(a);
  rhs.Assemble(i, a, 11, "rhs");

  ls.ResetOperator(mat);
  ls.Solve(rhs, &x);
}

x.MoveToHost();
```

”Several matrix assembling with static pattern”

If the information for assembling is available (for calling the *AssembleUpdate()* function), then in the *info()* function will print *asm=yes*

For the implementation of the assembly function we use an adopted and modified code based on [16]. Performance results for various cases can be found in [17].

4.16 Check

ValueType	Computation	Available
D,F,I,C	H (CSR-only)	B,S,M

Checks, if the object contains valid data via the *Check()* function. For vectors, the function checks if the values are not infinity and not NaN (not a number). For the matrices, this function checks the values and if the structure of the matrix is correct.

4.17 Sort

ValueType	Computation	Available
D,F,I,C	H (CSR-only)	B,S,M

Sorts the column values in a CSR matrix via the *Sort()* function.

4.18 Keying

ValueType	Computation	Available
D,F,I,C	H (CSR-only)	S,M

Typically it is hard to compare if two matrices have the same (structure and values) or they just have the same structure. To do this, we provide a key function which generates three keys, for the row index, column index and for the values. An example is presented in the following Listing.

```
LocalMatrix<double> mat;

mat.ReadFileMTX(std::string(argv[1]));

long int row_key;
long int col_key;
long int val_key;

mat.Key(row_key,
        col_key,
        val_key);

std::cout << "Row key = " << row_key << std::endl
          << "Col key = " << col_key << std::endl
          << "Val key = " << val_key << std::endl;
```

"Generating a matrix key"

4.19 Graph Analyzers

ValueType	Computation	Available
D,F,I,C	H (CSR-only)	B,S,M

The following functions are available for analyzing the connectivity in graph of the underlying sparse matrix.

- (R)CMK reordering
- Maximal independent set
- Multi-coloring
- ZeroBlockPermutation
- Connectivity ordering

All graph analyzing functions return a permutation vector (int type) which is supposed to be used with the *Permute()* and *PermuteBackward()* functions in the matrix and vector classes.

The CMK (Cuthill–McKee) and RCMK (Reverse Cuthill–McKee) orderings minimize the bandwidth of a given sparse matrix.

```
LocalVector<int> cmk;

mat.RCMK(&cmk);
mat.Permute(cmk);
```

"CMK ordering"

The Maximal independent set (MIS) algorithm finds a set with maximal size which contains elements that do not depend on other elements in this set.

```
LocalVector<int> mis;
int size;

mat.MaximalIndependentSet(size,
                          &mis);
mat.Permute(mis);
```

"MIS ordering"

The Multi-coloring (MC) algorithm builds a permutation (coloring the matrix) in such way that no two adjacent nodes in the sparse matrix are having the same color.

```
LocalVector<int> mc;
int num_colors;
int *block_colors = NULL;

mat.MultiColoring(num_colors,
                  &block_colors,
                  &mc);

mat.Permute(mc);
```

"MC ordering"

For saddle-point problems (i.e. matrices with zero-diagonal entries) the ZeroBlockPermutation maps all zero-diagonal elements to the last block of the matrix.

```
LocalVector<int> zbp;
int size;

mat.ZeroBlockPermutation(size,
                         &zbp);
mat.Permute(zbp);
```

"Zero-Block-Permutation"

Connectivity ordering returns a permutation which sorts the matrix by non-zero entries per row.

```
LocalVector<int> conn;

mat.ConnectivityOrder(&conn);
mat.Permute(conn);
```

"Connectivity ordering"

Visualization of the graph analyzers can be found in Chapter 13.

4.20 Basic Linear Algebra Operations

There are more functions and routines which matrix and vector objects can perform - for further specifications and API, check the doxygen documentation.

Matrix objects can also perform

- Matrix-vector multiplication $x := Ay$ – *Apply()*
- Matrix-vector multiplication and addition $x := x + Ay$ – *ApplyAdd()*
- Symmetric permutation – *Permute()*; *PermuteBackward()*
- Sub-matrix / diagonal / L / U extractions – *ExtractSubMatrix()*; *ExtractSubMatrices()*; *ExtractDiagonal()*; *ExtractInverseDiagonal()*; *ExtractU()*; *ExtractL()*; *ExtractColumnVector()* ; *ExtractRowVector()*
- Sparse triangular solvers (L,U,LL,LU) – ; *LSolve()*; *LLSolve()*; *LUSolve()*
- Factorizations -
  - ILU(0) – ILU with no fill-ins – *ILU0()*
  - ILU(p) – based on levels – *ILUpFactorize()*;
  - ILU(p,q) – based on power(q)-pattern method – via *ILUpFactorize()*
  - ILUT – based on threshold *ILUTFactorize()*
  - IC0 – Incomplete Cholesky with no fill-ins – *ICFactorize()*
- FSAI computation – *FSAI()*
- Symbolic power – compute the pattern of $|A|^p$, where $|.| = a_{i,j}$ – *SymbolicPower()*
- Sparse matrix-matrix addition (with the same or different sparsity patterns) – *MatrixAdd()*
- Sparse matrix-matrix multiplication – *MatrixMult()*
- Sparse matrix multiplication with a diagonal matrix (left and right multiplication) – *DiagonalMatrixMultL()*; *DiagonalMatrixMultR()*

- Compute the Gershgorin circles (eigenvalue approximations) – *Gershgorin()*
- Compress (i.e. delete elements) under specified threshold, this function also updates the structure – *Compress()*
- Transpose – *Transpose()*
- Create a restriction matrix via restriction map – *CreateFromMap()*
- Scale with scalar all values / diagonal / off-diagonal – *Scale()*; *ScaleDiagonal()*; *ScaleOffDiagonal()*
- Add a scalar to all values / diagonal / off-diagonal – *AddScalar()*; *AddScalarDiagonal()*; *AddScalarOffDiagonal()*

Vector objects can also perform

- Dot product (scalar product) – *Dot()*
- Scaling – *Scale()*
- Vector updates of types: $x := x + \alpha y$, $x := y + \alpha x$, $x := \alpha x + \beta y$, $x := \alpha x + \beta y + \gamma z$ – *ScaleAdd()*; *AddScale()*; *ScaleAddScale()*; *ScaleAdd2()*
- L_1, L_2 and L_∞-norm – *Asum()*; *Norm()*; *Amax()*
- Sum – *Reduce()*
- Point-wise multiplication of two vector (element-wise multiplication) – *PointWiseMult()*
- Permutations (forward and backward) – *Permute()*; *PermuteBackward()*
- Copy within specified permutation – *CopyFromPermute()*; *CopyFromPermuteBackward()*
- Restriction/prolongation via map – *Restriction()*; *Prolongation()*
- Initialized the vector with random values – *SetRandom()*
- Power – *Power()*
- Copy from double or float vector – *CopyFromDouble()*; *CopyFromFloat()*

4.21 Local Stencils

Instead of representing the linear operator as a matrix, the user can use stencil operators. The stencil operator has to be defined manually in each class and backend. This is a necessary step in order to have good performance in terms of spatial-block technique and compiler optimization.

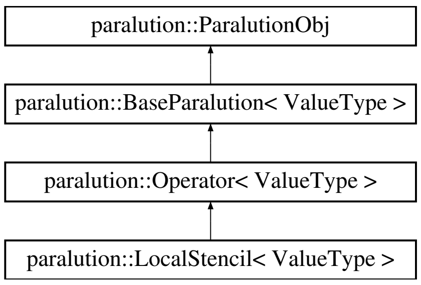

paralution::ParalutionObj
paralution::BaseParalution< ValueType >
paralution::Operator< ValueType >
paralution::LocalStencil< ValueType >

Figure 4.11: Code structure of a host LocalStencil

```
#include <iostream>
#include <cstdlib>

#include <paralution.hpp>

using namespace paralution;

int main(int argc, char* argv[]) {

  init_paralution();

  info_paralution();

  LocalVector<double> x;
  LocalVector<double> rhs;

  LocalStencil<double> stencil(Laplace2D);

  stencil.SetGrid(100); // 100x100

  x.Allocate("x", stencil.get_nrow());
  rhs.Allocate("rhs", stencil.get_nrow());

  // Linear Solver
  CG<LocalStencil<double>, LocalVector<double>, double > ls;

  rhs.Ones();
  x.Zeros();

  ls.SetOperator(stencil);

  ls.Build();

  stencil.info();

  double tick, tack;
  tick = paralution_time();

  ls.Solve(rhs, &x);

  tack = paralution_time();
  std::cout << "Solver execution:" << (tack-tick)/1000000 << " sec" << std::endl;

  stop_paralution();

  return 0;
}
```

"Example of 2D Laplace stencil"

Chapter 5

Multi-node Computation

5.1 Introduction

In this chapter we describe all base objects (matrices and vectors) for computation on multi-node (distributed-memory) systems. Two typical configurations are presented on Figure 5.1 and Figure 5.2.

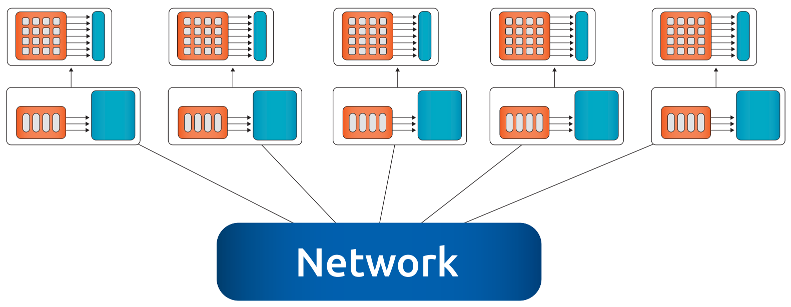


Figure 5.1: An example for a multi-node configurations, all nodes are connected via network – single socket system with a single accelerator.

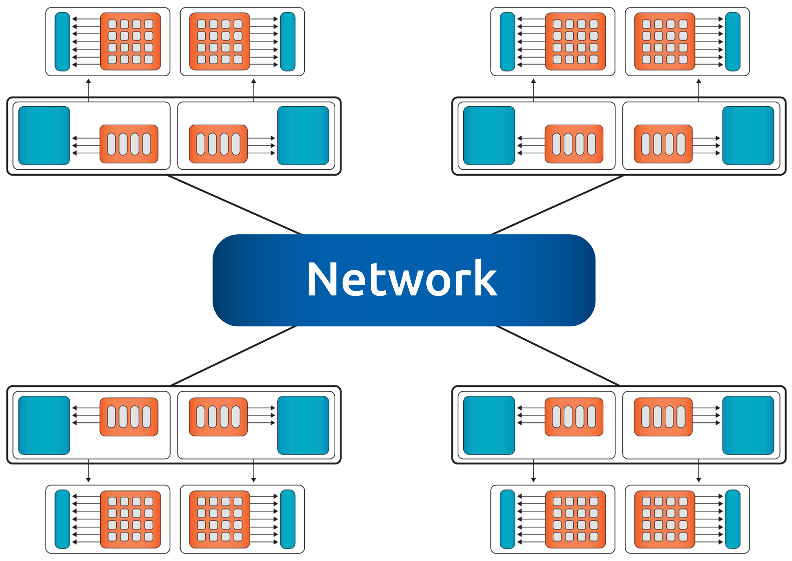


Figure 5.2: An example for a multi-node configurations, all nodes are connected via network – dual socket systems with two accelerators attached to each node

To each compute node, one or more accelerators can be attached. The compute node could be any kind of shared-memory (single, dual, quad CPU) system, details on a single-node can be found in Figure 4.1. Note that the memory of the accelerator and of the host are physically different. All nodes can communicate with each other via network.

For the communication channel between the nodes (and between the accelerators on single or multiple nodes) we use the MPI library. Additional libraries for communication can be added on request.

The PARALUTION library supports non-overlapping type of distribution, where the computational domain is split into several sub-domain with the corresponding information about the boundary and ghost layers. An example is shown on Figure 5.3. The square box domain is distributed into four sub-domains. Each subdomain belongs to a process $P0, P1, P2$ and $P3$.

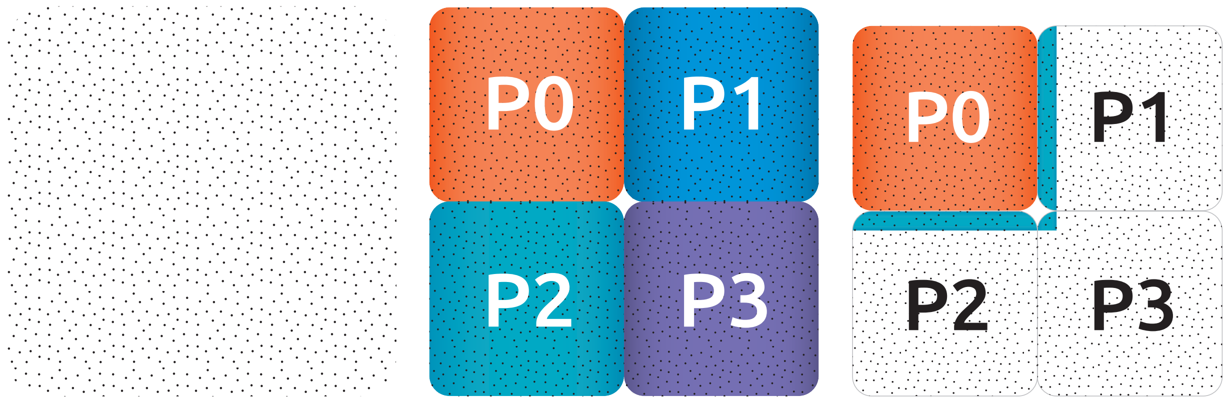


Figure 5.3: An example for domain distribution

To perform a sparse matrix-vector (SpMV) multiplication, each process need to multiply its own portion of the domain and update the corresponding ghost elements. For $P0$ this multiplication reads

$$Ax = y, \tag{5.1}$$
$$A_I x_I + A_G x_G = y_I, \tag{5.2}$$

where I stands for interior and G stands for ghost, the x_G is a vector with three sections coming from $P1, P2$ and $P3$. The whole ghost part of the global vector is used mainly for the SpMV product and this part of the vector does not play any role in the computation of all vector-vector operations.

5.2 Code Structure

Each object contains two local sub-objects. The global matrix stores the interior and the ghost matrix via local objects. The global vector also stores its data via two local objects. In addition to the local data, the global objects have information about the global communication via the parallel manager – see Figure 5.4.

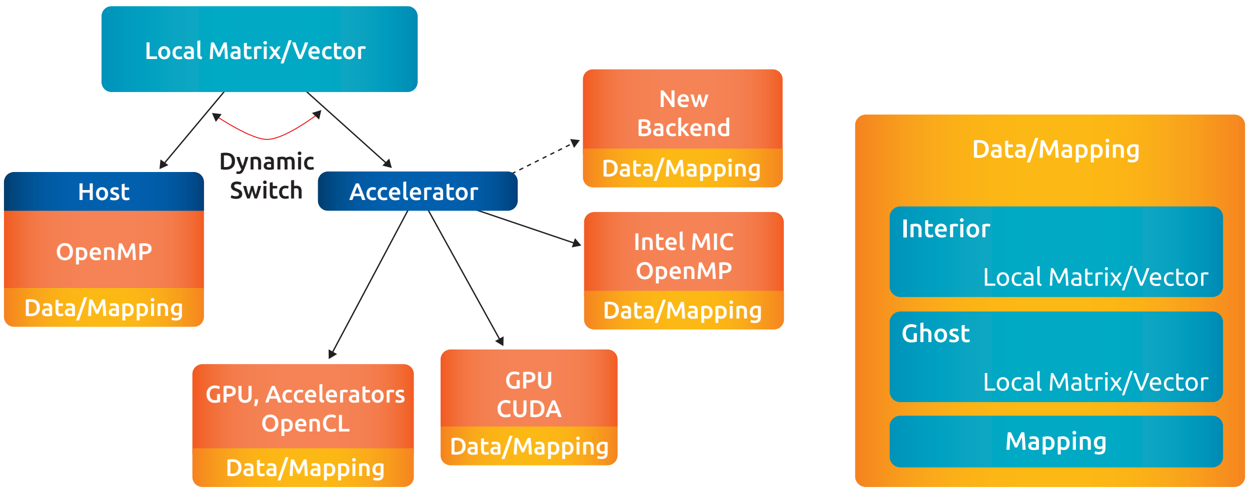


Figure 5.4: Global Matrices and Vectors

5.3 Parallel Manager

The parallel manager class handles the communication and the mapping of the global matrix. Each global matrix and vector need to be initialized with a valid parallel manager in order to perform any operation.

For many distributed simulation, the underlying matrix is already distributed. This information need to be passed to the parallel manager.

The required information is:

- Global size
- Local size of the interior/ghost for each rank
- Communication pattern (what information need to be sent to whom)

5.4 Global Matrices and Vectors

The global matrices and vectors store their data via two local objects. For the global matrix, the interior can be access via the *GetInterior()* and *GetGhost()* functions, which point to two valid local matrices. The same function names exist for the the global vector, which point to two local vector objects.

5.4.1 Value Type and Formats

The value type and the formats can be set in similar manner as in the local matrix. The supported formats are identical to the local case.

5.4.2 Matrix and Vector Operations

Most functions which are available for the local vectors can be performed on the global vector objects as well. For the global matrix, only the SpMV function is supported but all other functions can be performed on the interior of the matrix (without the couplings via the ghost layer).

5.4.3 Asynchronous SpMV

To minimize latency and to increase scalability the PARALUTION library supports asynchronous sparse matrix-vector multiplication. The implementation of the SpMV starts with asynchronous transfer of the needed ghost buffers while at the same time it computes the interior matrix. When the computation of the interior SpMV is done, the ghost transfer is synchronized and the ghost SpMV is performed.

To minimize the PCI-E bus, the OpenCL and CUDA implementation provide a special packaging technique for transferring all ghost data into a contiguous memory buffer.

5.5 Communication Channel

For the communication channel we use MPI but this can be extended, on request, with some other communication mechanics.

5.6 I/O

The user can store and load the full data from and to files. For a solver, the needed data would be:

- Parallel manager
- Sparse matrix
- Vector

The reading/writing of the data can be done fully in parallel without any communication.

To visualize the data we use 4×4 grid which is distributed on 4 MPI processes (organized in 2×2), each local matrix store the local unknowns (with local index) – see Figure 5.5.

The data associated with RANK 0 is presented on Figure 5.6.

5.6.1 File Organization

When the parallel manager, global matrix or global vector are writing to a file, the main file (passed as a file name to this function) will contain information for all files on all ranks.

```
pm.dat.rank.0
pm.dat.rank.1
pm.dat.rank.2
pm.dat.rank.3
```

"Parallel manager (main) file with 4 MPI ranks"

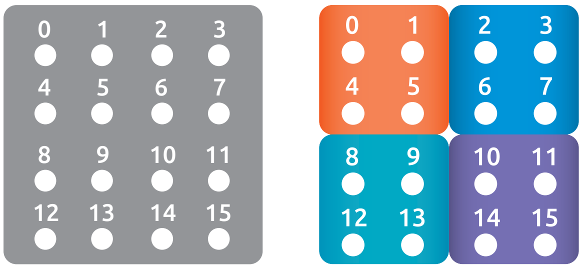
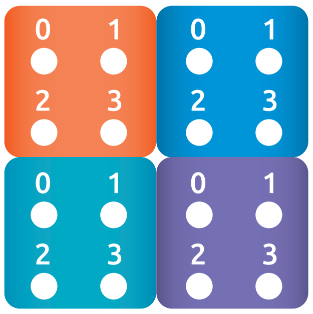


Figure 5.5: An example of 4 × 4 grid, distributed in 4 domains (2 × 2)



RANK = 0
GLOBAL_SIZE = 16
LOCAL_SIZE = 4
GHOST_SIZE = 4
BOUNDARY_SIZE = 4
NUMBER_OF_RECEIVERS = 2
NUMBER_OF_SENDERS = 2
RECEIVERS_RANK = {1, 2}
SENDERS_RANK = {1, 2}
RECEIVERS_INDEX_OFFSET = {0, 2, 4}
SENDERS_INDEX_OFFSET = {0, 2, 4}
BOUNDARY_INDEX = {1, 3, 2, 3}

Figure 5.6: An example of 4 MPI process and the data associate with RANK 0

```
matrix.mtx.interior.rank.0
matrix.mtx.ghost.rank.0
matrix.mtx.interior.rank.1
matrix.mtx.ghost.rank.1
matrix.mtx.interior.rank.2
matrix.mtx.ghost.rank.2
matrix.mtx.interior.rank.3
matrix.mtx.ghost.rank.3
```

”Matrix (main) file with 4 MPI ranks”

```
rhs.dat.rank.0
rhs.dat.rank.1
rhs.dat.rank.2
rhs.dat.rank.3
```

”Vector (main) file with 4 MPI ranks”

Example for the data in each file can be found in Section 15.7.

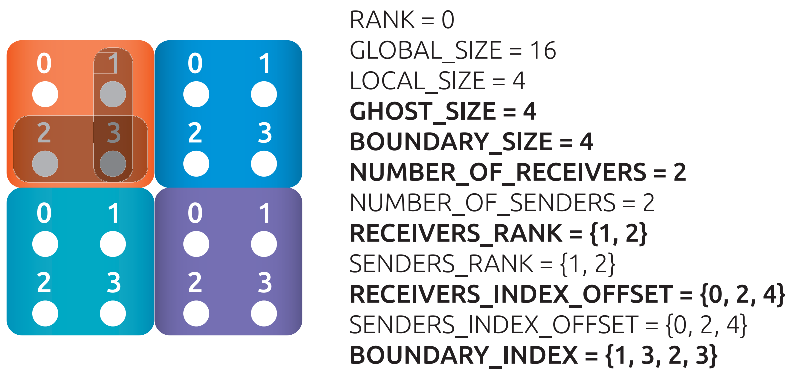
5.6.2 Parallel Manager

The data for each rank can be seen as two type of information – one for receiving data from the neighbors, which is presented on Figure 5.7. There the RANK0 needs information what type of data will be received, and from whom. The second needed data is the sending information – RANK0 needs to know where and what to send, see Figure 5.8.

Receiving data – RANK0 requires:

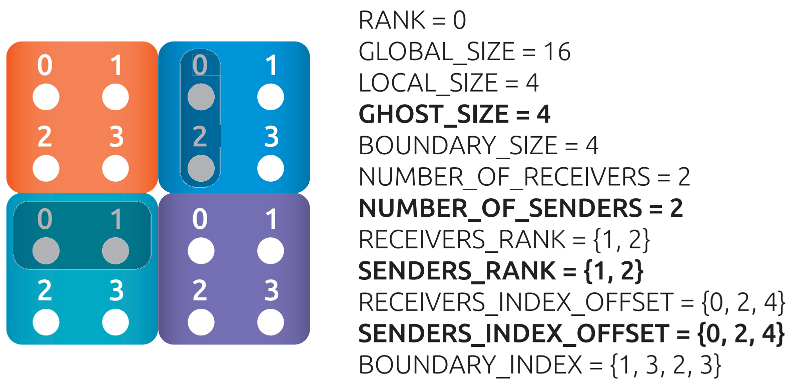
- Total size of the received information (GHOST_SIZE – integer value).
- Number of MPI ranks which will send data to RANK0 (NUMBER_OF_RECEIVERS – integer value).
- Which are the MPI ranks sending the data (RECEIVERS_RANK – integer array).
- How the received data (from each rank) will be stored in the ghost vector (RECEIVERS_INDEX_OFFSET – integer array), in this example the first 30 elements will be received from P1 $[0, 2)$ and the second 30 from P2 $[2, 4)$

Sending data – RANK0 requires:



RANK = 0
GLOBAL_SIZE = 16
LOCAL_SIZE = 4
GHOST_SIZE = 4
BOUNDARY_SIZE = 4
NUMBER_OF_RECEIVERS = 2
NUMBER_OF_SENDERS = 2
RECEIVERS_RANK = {1, 2}
SENDERS_RANK = {1, 2}
RECEIVERS_INDEX_OFFSET = {0, 2, 4}
SENDERS_INDEX_OFFSET = {0, 2, 4}
BOUNDARY_INDEX = {1, 3, 2, 3}

Figure 5.7: An example of 4 MPI process, RANK 0 receives data, the associated data is marked in bold)



RANK = 0
GLOBAL_SIZE = 16
LOCAL_SIZE = 4
GHOST_SIZE = 4
BOUNDARY_SIZE = 4
NUMBER_OF_RECEIVERS = 2
NUMBER_OF_SENDERS = 2
RECEIVERS_RANK = {1, 2}
SENDERS_RANK = {1, 2}
RECEIVERS_INDEX_OFFSET = {0, 2, 4}
SENDERS_INDEX_OFFSET = {0, 2, 4}
BOUNDARY_INDEX = {1, 3, 2, 3}

Figure 5.8: An example of 4 MPI process, RANK 0 sends data, the associated data is marked in bold

- Total size of the sending information (BOUNDARY_SIZE – integer value).
- Number of MPI ranks which will receive data from RANK0 (NUMBER_OF_SENDERS – integer value).
- Which are the MPI ranks receiving the data (SENDERS_RANK – integer array).
- How the sending data (from each rank) will be stored in the sending buffer (SENDERS_INDEX_OFFSET – integer array), in this example the first 30 elements will be sent to P1 $[0, 2)$ and the second 30 to P2 $[2, 4)$
- The elements which need to be send (BOUNDARY_INDEX – integer array). In this example the data which needs to be send to P1 and to P2 is the ghost layer marked as ghost P0. The vertical stripe needs to be send to P1 and the horizontal stripe to P2. The numbering of local unknowns (in local indexing) for P1 (the vertical stripes) are 1, 2 (size of 2) and they are stored in the BOUNDARY_INDEX. After 2 elements, the elements for P2 are stored, they are 2, 3 (2 elements).

5.6.3 Matrices

For each rank, two matrices are used – interior and ghost matrix. They can be stored in separate files, one for each matrix. The file format could be Matrix Market (MTX) or binary.

5.6.4 Vectors

For each rank, the vector object needs only the local (interior) values. They are stored into a single file. The file could be ASCII or binary.

Chapter 6

Solvers

In this chapter, all linear solvers are presented. Most of the iterative solvers can be performed on linear operators *LocalMatrix*, *LocalStencil* and *GlobalMatrix* – i.e. the iterative solvers can be performed locally (on a shared memory system) or in a distributed manner (on a cluster) via MPI. The only exception is the AMG (Algebraic Multigrid) which has two versions (one for the Local and one for the Global type of computation). The only pure local solvers (the one which does not support global/MPI operations) are the mixed-precision defect-correction solver, all direct solvers and the eigenvalue solvers.

All solvers need three template parameters – Operators, Vectors and Scalar type. There are three possible combinations

- *LocalMatrix*, *LocalVector*, *float* or *double*
- *LocalStencil*, *LocalVector*, *float* or *double*
- *GlobalMatrix*, *GlobalVector*, *float* or *double*

where the Operators/Vectors need to use the same ValueType as the scalar for the solver.

6.1 Code Structure

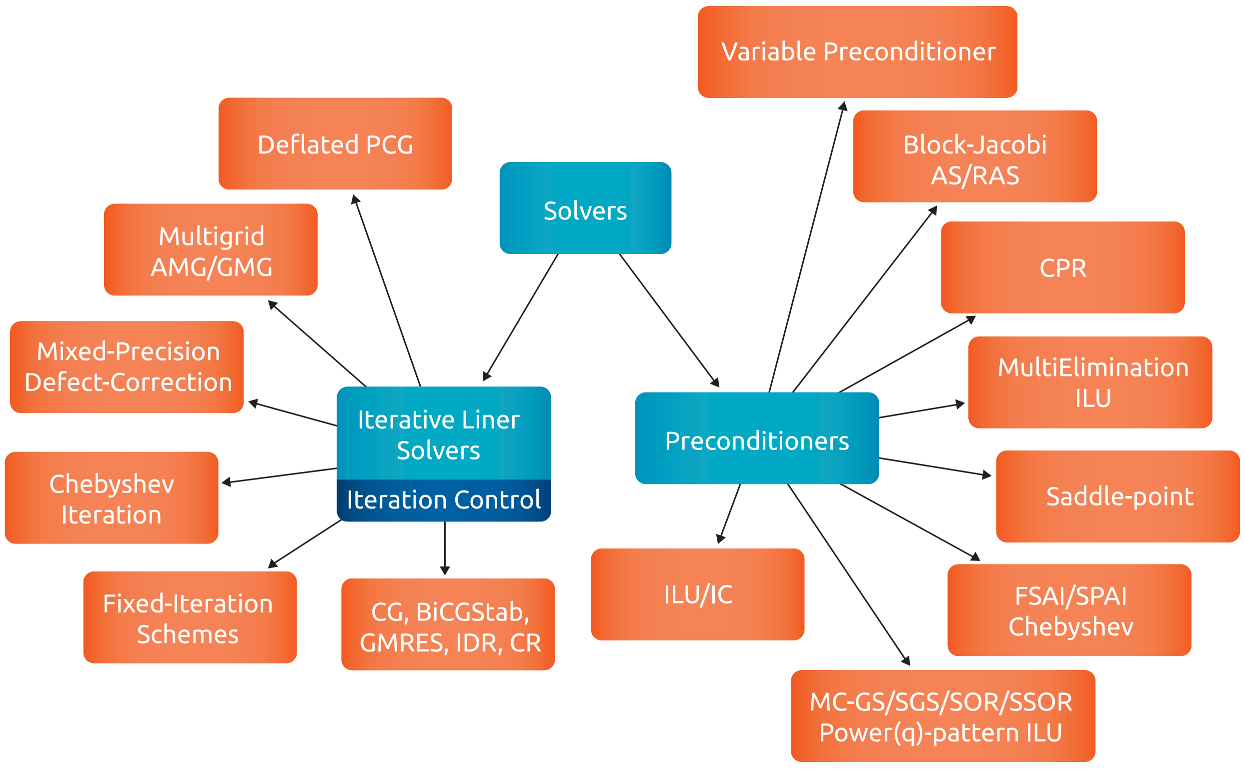


Figure 6.1: Solver's classes hierarchy

The base class Solver is purely virtual, it provides an interface for:

- *SetOperator()* – set the operator A - i.e. the user can pass the matrix here
- *Build()* – build the solver (including preconditioners, sub-solvers, etc), the user need to specify the operator first
- *Solve()* – solve the system $Ax = b$, the user need to pass a right-hand-side b and a vector x where the solution will be obtained
- *Print()* – shows solver information
- *ReBuildNumeric()* – only re-build the solver numerically (if possible),
- *MoveToHost()* and *MoveToAccelerator()* – offload the solver (including preconditioners and sub-solvers) to the host/accelerator.

The computation of the residual can be based on different norms (L_1, L_2 and L_∞). This is controlled by the function *SetResidualNorm()*.

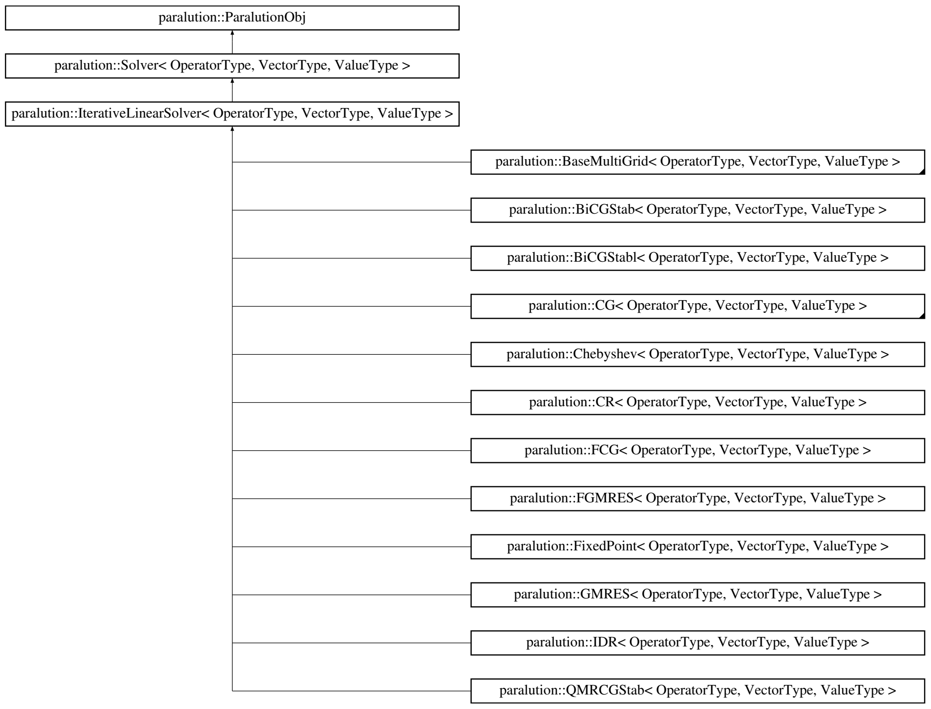

paralution::ParalutionObj
paralution::Solver< OperatorType, VectorType, ValueType >
paralution::IterativeLinearSolver< OperatorType, VectorType, ValueType >
paralution::BaseMultiGrid< OperatorType, VectorType, ValueType >
paralution::BiCGStab< OperatorType, VectorType, ValueType >
paralution::BiCGStabl< OperatorType, VectorType, ValueType >
paralution::CG< OperatorType, VectorType, ValueType >
paralution::Chebyshev< OperatorType, VectorType, ValueType >
paralution::CR< OperatorType, VectorType, ValueType >
paralution::FCG< OperatorType, VectorType, ValueType >
paralution::FGMRES< OperatorType, VectorType, ValueType >
paralution::FixedPoint< OperatorType, VectorType, ValueType >
paralution::GMRES< OperatorType, VectorType, ValueType >
paralution::IDR< OperatorType, VectorType, ValueType >
paralution::QMRCGStab< OperatorType, VectorType, ValueType >

Figure 6.2: Iterative Solvers

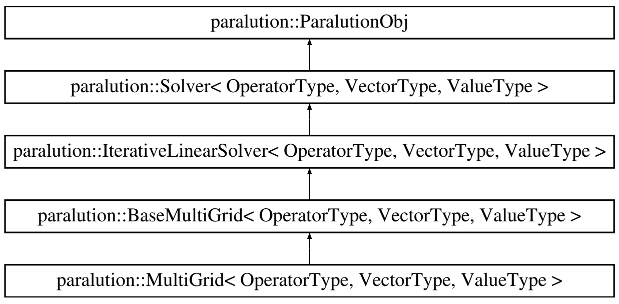

paralution::ParalutionObj
paralution::Solver< OperatorType, VectorType, ValueType >
paralution::IterativeLinearSolver< OperatorType, VectorType, ValueType >
paralution::BaseMultiGrid< OperatorType, VectorType, ValueType >
paralution::MultiGrid< OperatorType, VectorType, ValueType >

Figure 6.3: Multi-Grid

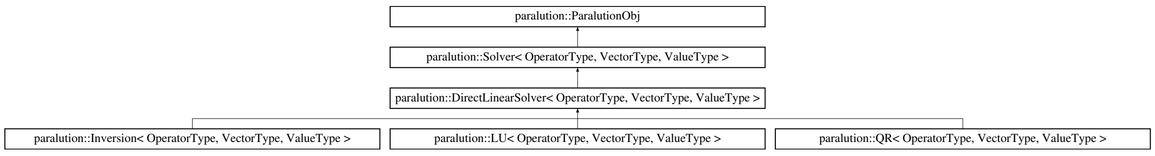

paralution::ParalutionObj
paralution::Solver< OperatorType, VectorType, ValueType >
paralution::DirectLinearSolver< OperatorType, VectorType, ValueType >
paralution::Inversion< OperatorType, VectorType, ValueType >
paralution::LU< OperatorType, VectorType, ValueType >
paralution::QR< OperatorType, VectorType, ValueType >

Figure 6.4: Direct Solvers

6.2 Iterative Linear Solvers

The iterative solvers are controlled by an iteration control object, which monitors the convergence properties of the solver - i.e. maximum number of iteration, relative tolerance, absolute tolerance, divergence tolerance. The iteration control can also record the residual history and store it in an ASCII file.

- *Init()*, *InitMinIter()*, *InitMaxIter()*, *InitTol()* – initialize the solver and set the stopping criteria
- *RecordResidualHistory()*, *RecordHistory()* – start the recording of the residual and store it into a file
- *Verbose()* – set the level of verbose output of the solver (0 – no output, 2 – detailed output, including residual and iteration information)
- *SetPreconditioner()* – set the preconditioning

6.3 Stopping Criteria

All iterative solvers are controlled based on:

- Absolute stopping criteria, when the $|r_k|_{L_p} < eps_{ABS}$
- Relative stopping criteria, when the $|r_k|_{L_p}/|r_1|_{L_p} \leq eps_{REL}$
- Divergence stopping criteria, when the $|r_k|_{L_p}/|r_1|_{L_p} \geq eps_{DIV}$
- Maximum number of iteration N, when $k = N$

where k is the current iteration, r_k is the residual for the current iteration k (i.e. $r_k = b - Ax_k$), r_1 is the starting residual (i.e. $r_1 = b - Ax_{initguess}$). In addition, the minimum number of iteration M can be specified. In this case, the solver will not stop to iterate before $k \geq M$.

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.Init(1e-10,  // abs_tol
        1e-8,   // rel_tol
        1e+8,   // div_tol
        10000); // max_iter

ls.SetOperator(mat);

ls.Build();

ls.Solve(rhs, &x);
```

"Setting different stopping criteria in a CG solver"

The L_p is the norm used for the computation, where p could be 1, 2 and ∞. The norm computation can be set via *SetResidualNorm()* function with 1 for L_1, 2 for L_2 and 3 for L_∞. For the computation with the L_∞, the index of maximum value can be obtained via the *GetAmaxResidualIndex()* function. If this function is called and the L_∞ has not been selected then this function will return -1.

The reached criteria can be obtained via *GetSolverStatus()*, which will return:

- 0 – if no criteria has been reached yet
- 1 – if absolute tolerance has been reached
- 2 – if relative tolerance has been reached
- 3 – if divergence tolerance has been reached
- 4 – if maximum number of iteration has been reached

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.SetOperator(mat);
ls.SetResidualNorm(1);

ls.Build();

ls.Solve(rhs, &x);
```

"CG solver with L_1 norm"

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.SetOperator(mat);
ls.SetResidualNorm(3);

ls.Build();

ls.Solve(rhs, &x);

std::cout << "Index of the L_\infty = " << ls.GetAmaxResidualIndex() << std::end;

std::cout << "Solver status = " << ls.GetSolverStatus() << std::endl;
```

"CG solver with L_∞ norm"

6.4 Building and Solving Phase

Each iterative solver consists of a building step and a solving step. During the building step all necessary auxiliary data is allocated and the preconditioner is constructed. After that the user can call the solving procedure, the solving step can be called several times.

When the initial matrix associated with the solver is on the accelerator, the solver will try to build everything on the accelerator. However, some preconditioners and solvers (such as FSAI and AMG) need to be constructed on the host and then they are transferred to the accelerator. If the initial matrix is on the host and we want to run the solver on the accelerator then we need to move the solver to the accelerator as well as the matrix, the right-hand-side and the solution vector. Note that if you have a preconditioner associate with the solver, it will be moved automatically to the accelerator when you move the solver. In the following Listing we present these two scenarios.

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

mat.MoveToAccelerator();
rhs.MoveToAccelerator();
x.MoveToAccelerator();


ls.SetOperator(mat);
ls.SetPreconditioner(p);

// The solver and the preconditioner will be constructed on the accelerator
ls.Build();

// The solving phase is performed on the accelerator
ls.Solve(rhs, &x);
```

"An example for a preconditioend CG where the building phase and the solution phase are performed on the accelerator"

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

ls.SetOperator(mat);
ls.SetPreconditioner(p);

// The solver and the preconditioner will be constructed on the host
ls.Build();

// Now we move all objects (including the solver and the preconditioner
// to the accelerator

mat.MoveToAccelerator();
rhs.MoveToAccelerator();
```

```
x.MoveToAccelerator();

ls.MoveToAccelerator();

// The solving phase is performed on the accelerator
ls.Solve(rhs, &x);
```

"An example for a preconditioned CG where the building phase is performed on the host and the solution phase is performed on the accelerator"

6.5 Clear function and Destructor

The *Clear()* function clears all the data which is in the solver including the associated preconditioner. Thus, the solver is not anymore associated with this preconditioner. Note that the preconditioner is not deleted (via destructor) only a *Clear()* is called.

When the destructor of the solver class is called, it automatically call the *Clear()* function. Be careful, when declaring your solver and preconditioner in different places - we highly recommend to manually call the *Clear()* function of the solver and not to relay on destructor of the solver.

6.6 Numerical Update

ValueType	Building phase	Available
D,F,C	H,C	S,M

Some preconditioners require two phases in the their construction: an algebraic (e.g. compute a pattern or structure) and a numerical (compute the actual values). In cases, where the structure of the input matrix is a constant (e.g. Newton-like methods) it is not necessary to fully re-construct the preconditioner. In this case, the user can apply a numerical update to the current preconditioner and passed the new operator.

This function is called *ReBuildNumeric()*. If the preconditioner/solver does not support the numerical update, then a full *Clear()* and *Build()* will be performed.

6.7 Fixed-point Iteration

Fixed-point iteration is based on additive splitting of the matrix as $A = M + N$, the scheme reads

$$x_{k+1} = M^{-1}(b - Nx_k). \tag{6.1}$$

It can also be reformulated as a weighted defect correction scheme

$$x_{k+1} = x_k - \omega M^{-1}(Ax_k - b). \tag{6.2}$$

The inversion of M can be performed by preconditioners (Jacobi, Gauss-Seidel, ILU, etc) or by any type of solvers.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	B,S,M

```
FixedPoint<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
FixedPoint<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
FixedPoint<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

"Fixed-Point declaration"

```
FixedPoint<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > fp;
Jacobi<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

fp.SetOperator(mat);
fp.SetPreconditioner(p);
fp.SetRelaxation(1.3);

fp.Build();

fp.Solve(rhs, &x);
```

"Fixed-point iteration based on Jacobi preconditioner - Jacobi iteration"

6.8 Krylov Subspace Solvers

Krylov subspace solvers are iterative methods based on projection. The implemented solvers in PARALUTION are CG, CR, BiCGStab, BiCGStab(l), QMRCGStab, GMRES, as well as Flexible CG and Flexible GMRES. Details of the methods can be found in [46, 10, 15, 48, 54].

6.8.1 CG

CG (Conjugate Gradient) is a method for solving symmetric and positive definite matrices (SPD). The method can be preconditioned where the approximation should also be SPD.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	B,S,M

```
CG<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
CG<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
CG<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

"CG declaration"

6.8.2 CR

CR (Conjugate Residual) is a method for solving symmetric and semi-positive definite matrices. The method can be preconditioned where the approximation should also be SPD or semi-SPD.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	B,S,M

```
CR<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
CR<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
CR<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

"CR declaration"

6.8.3 GMRES

GMRES (Generalized Minimal Residual Method) is a method for solving non-symmetric problems. The pure GMRES solvers is based on restarting technique. The default size of the Krylov subspace for the GMRES is set to 30, it can be modify by *SetBasisSize()* function.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	B,S,M

```
GMRES<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
GMRES<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
GMRES<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

"GMRES declaration"

6.8.4 FGMRES

FGMRES (Flexible Generalized Minimal Residual Method) is a method for solving non-symmetric problems. The Flexible GMRES solvers is based on a window shifting of the Krylov subspace. The default size of the Krylov subspace for the GMRES is set to 30, it can be modify by *SetBasisSize()* function.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	B,S,M

```
FGMRES<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
FGMRES<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
FGMRES<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

”FGMRES declaration”

6.8.5 BiCGStab

BiCGStab (Bi-Conjugate Gradient Stabilized) is a method for solving non-symmetric problems.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	B,S,M

```
BiCGStab<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
BiCGStab<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
BiCGStab<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

”BiCGStab declaration”

6.8.6 IDR

IDR (Induced Dimension Reduction) is a method for solving non-symmetric problems. The dimension of the shadow space for the IDR(s) method can be set by *SetShadowSpace()* function, the default value is 4.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	B,S,M

```
IDR<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
IDR<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
IDR<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

”IDR Solver”

Note The orthogonal system in IDR method is based on random numbers, thus it is normal to obtain slightly different number of iterations every time you run the program.

6.8.7 FCG

FCG (Flexible Conjugate Gradient) is a method for solving symmetric and positive definite matrices (SPD). The method can be preconditioned where the approximation should also be SPD. For additional informations see [41].

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	S,M

```
FCG<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
FCG<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
FCG<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

”FCG declaration”

6.8.8 QMRCGStab

QMRCGStab is a method for solving non-symmetric problems. More details are given in [34].

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	S,M

```
QMRCGStab<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
QMRCGStab<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
QMRCGStab<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

"QMRCGStab declaration"

6.8.9 BiCGStab(l)

BiCGStab(l) (Bi-Conjugate Gradient Stabilized) is a method for solving non-symmetric problems. The degree l can be set via the function *SetOrder()*. More details can be found in [19].

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	S,M

```
BiCGStabl<LocalMatrix<ValueType>,  LocalVector<ValueType>,  ValueType > ls;
BiCGStabl<LocalStencil<ValueType>, LocalStencil<ValueType>, ValueType > ls;
BiCGStabl<GlobalMatrix<ValueType>, GlobalVector<ValueType>, ValueType > ls;
```

"BiCGStab(l) declaration"

Example

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

ls.SetOperator(mat);
ls.SetPreconditioner(p);

ls.Build();

ls.Solve(rhs, &x);
```

"Preconditioned CG solver with ILU(0 1)"

6.9 Deflated PCG

The Deflated Preconditioned Conjugate Gradient Algorithm (DPCG) is a two-level preconditioned CG algorithm to iteratively solve an ill-conditioned linear system, see [51].

ValueType	Building phase	Solving phase	Available
D,F	H	H,C,O,X	B (GPLv3 only)

Deflation attempts to remove the bad eigenvalues from the spectrum of the preconditioned matrix $M^{-1}A$. The preconditioned system can be 'deflated' such that it is transformed into

$$M^{-1}PA\hat{x} = M^{-1}Pb, \tag{6.3}$$

where $P := I - AQ$, $Q := ZE^{-1}Z^T$ and $E := Z^TAZ$. Z is called the deflation sub-space matrix. Its columns are approximations of the eigenvectors corresponding to the bad eigenvalues remaining in the spectrum of the preconditioned system given in $M^{-1}Ax = M^{-1}b$.

There can be many choices for Z. In the current implementation, we are using a construction scheme tuned for matrices arising from discretizing the pressure-correction equation in two-phase flow, details are discussed in [50].

```
DPCG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

// Set the number of deflation vectors
ls.SetNVectors(2);
ls.Build();

ls.Solve(rhs, &x);
```

"Deflated PCG"

Note The number of the deflated vectors can be set via *SetNVectors()* function

Note The dimension of the problem is hard-coded in the *MakeZ_COO()*, *MakeZ_CSR()* function in the solver code (dpcg.cpp, see function *Build()*)

6.10 Chebyshev Iteration

The Chebyshev iteration (also known as acceleration scheme) is similar to the CG method but it requires the minimum and the maximum eigenvalue of the matrix. Additional information can be found in [10].

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	B,S,M

```
Chebyshev<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.SetOperator(mat);
ls.Set(lambda_min, lambda_max);
ls.Build();

ls.Solve(rhs, &x);
```

"Chebyshev iteration"

6.11 Mixed-precision Solver

The library provides mixed-precision solvers based on defect-correction scheme. The current implementation of the library is based on host correction in double precision and accelerator computation in single precision. The solver is based on the following scheme:

$$x_{k+1} = x_k + A^{-1} r_k, \tag{6.4}$$

where the computation of the residual $r_k = b - Ax_k$ and of the update $x_{k+1} = x_k + d_k$ are performed on the host in double precision. The computation of the residual system $Ad_k = r_k$ is performed on the accelerator in single precision. In addition to the setup functions of the iterative solver, the user need to specify the inner (for $Ad_k = r_k$) solver.

ValueType	Building phase	Solving phase	Available
D-F	H,C,O,X	H,C,O,X	B,S,M

```
MixedPrecisionDC<LocalMatrix<double>, LocalVector<double>, double,
                 LocalMatrix<float>, LocalVector<float>, float> mp;

CG<LocalMatrix<float>, LocalVector<float>, float> cg;
MultiColoredILU<LocalMatrix<float>, LocalVector<float>, float> p;

// setup a lower tol for the inner solver
cg.SetPreconditioner(p);
cg.Init(1e-5, 1e-2, 1e+20,
        100000);
```

```

// setup the mixed-precision DC
mp.SetOperator(mat);
mp.Set(cg);

mp.Build();

mp.Solve(rhs, &x);
```

"Mixed-precision solver"

6.12 Multigrid Solver

The library provides algebraic multigrid as well as a skeleton for geometric multigrid methods. The *BaseMultigrid* class itself is not constructing the data for the method. It contains the solution procedure for V, W, K-cycles, for details see [53].

The AMG has two different versions for Local (non-MPI) and for Global (MPI) type of computations.

6.12.1 Geometric Multigrid

ValueType	Building phase	Solving phase	Available
D,F,C	-	H,C,O,X	B,S,M

For the geometric multgrid the user need to pass all information for each level and for its construction. This includes smoothing step, prolongation/restriction, grid traversing and coarsest level solver. This data need to be passed to the solver:

- Restriction and prolongation operations – they can be performed in two ways, based on *Restriction()* and *Prolongation()* function of the LocalVector class, or by matrix-vector multiplication. This is configured by a set function.
- Smoothers – they can be of any iterative linear solver's type. Valid options are Jacobi, Gauss-Seidel, ILU, etc. using a FixedPoint iteration scheme with pre-defined number of iterations. The smoothers could also be a solver such as CG, BiCGStab, etc.
- Coarse grid solver – could be of any iterative linear solver type. The class also provides mechanisms to specify where the coarse grid solver has to be performed on the host or on the accelerator. The coarse grid solver can be preconditioned.
- Grid scaling – computed based on a L2 norm ratio.
- Operator matrices - the operator matrices need to be passed on each grid level
- All objects need to be passed already initialized to the multigrid class.

6.12.2 Algebraic Multigrid

Plain and Smoothed Aggregation

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C,O,X	B,S,M

The algebraic multigrid solver (AMG) is based on the *Multigrid* class. The coarsening is obtained by different aggregation techniques. Currently, we support interpolation schemes based on aggregation [49] and smoothed aggregation [56]. The smoothers could be constructed inside or outside of the class. Detailed examples are given in the examples section.

When building the AMG if not additional information is set, the solver is built based on its default values.

```
AMG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.SetOperator(mat);

ls.Build();

ls.Solve(rhs, &x);
```

"AMG as a standalone solver"

All parameters can in the AMG can be set externally, including smoothers and coarse-grid solver - any type of solvers can be used.

```
LocalMatrix<ValueType> mat;

// Linear Solver
AMG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.SetOperator(mat);

// coupling strength
ls.SetCouplingStrength(0.001);
// number of unknowns on coarsest level
ls.SetCoarsestLevel(300);
// interpolation type for grid transfer operators
ls.SetInterpolation(SmoothedAggregation);
// Relaxation parameter for smoothed interpolation aggregation
ls.SetInterpRelax(2./3.);
// Manual smoothers
ls.SetManualSmoothers(true);
// Manual course grid solver
ls.SetManualSolver(true);
// grid transfer scaling
ls.SetScaling(true);

ls.BuildHierarchy();

int levels = ls.GetNumLevels();

// Smoother for each level
IterativeLinearSolver<LocalMatrix<ValueType>, LocalVector<ValueType>,
                      ValueType > **sm = NULL;
MultiColoredGS<LocalMatrix<ValueType>, LocalVector<ValueType>,
               ValueType > **gs = NULL;

// Coarse Grid Solver
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > cgs;
cgs.Verbose(0);

sm = new IterativeLinearSolver<LocalMatrix<ValueType>, LocalVector<ValueType>,
                               ValueType >*[levels-1];
gs = new MultiColoredGS<LocalMatrix<ValueType>, LocalVector<ValueType>,
                        ValueType >*[levels-1];

// Preconditioner for the coarse grid solver
//  MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType >
    p;
//  cgs->SetPreconditioner(p);

for (int i=0; i<levels-1; ++i) {
  FixedPoint<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > *fp;
  fp = new FixedPoint<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType
      >;
  fp->SetRelaxation(1.3);
  sm[i] = fp;

```

```
    gs[i] = new MultiColoredGS<LocalMatrix<ValueType>, LocalVector<ValueType>,
        ValueType >;
    gs[i]->SetPrecondMatrixFormat(ELL);

    sm[i]->SetPreconditioner(*gs[i]);
    sm[i]->Verbose(0);
  }

ls.SetOperatorFormat(CSR);
ls.SetSmoother(sm);
ls.SetSolver(cgs);
ls.SetSmootherPreIter(1);
ls.SetSmootherPostIter(2);
ls.Init(1e-10, 1e-8, 1e+8, 10000);
ls.Verbose(2);

ls.Build();

ls.Solve(rhs, &x);
```

"AMG with manual settings"

The AMG can be used also as a preconditioner within a solver.

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

// AMG Preconditioner
AMG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

p.InitMaxIter(2);
p.Verbose(0);

ls.SetPreconditioner(p);
ls.SetOperator(mat);
ls.Build();

ls.Solve(rhs, &x);
```

"AMG as a preconditioner"

Ruge-Stueben

The classic Ruge-Stueben coarsening algorithm is implemented following the [49].

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C,O,X	S,M

The solver provides high-efficiency in terms of complexity of the solver (i.e. number of iterations). However, most of the time it has a higher building step and requires higher memory usage.

Pair-wise

The pairwise aggregation scheme is based on [42].

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C,O,X	S,M

The pairwise AMG delivers very efficient building phase which is suitable for Poisson-like equation. Most of the time it requires K-cycle for the solving phase to provide low number of iterations.

Global AMG (MPI)

The global AMG is based on a variation of the pairwise aggregation scheme, using the MPI standard for inter-node communication.

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C,O,X	M

The building and the solving phase are fully MPI parallel. This solver works well with updating technique suitable for time-dependent problems (i.e. time-stepping schemes) which decrease significantly the building phase.

6.13 Direct Linear Solvers

The library provides three direct methods – LU, QR and full inversion (based on QR decomposition). The user can pass a sparse matrix, internally it will be converted to dense and then the selected method will be applied. These methods are not very optimal and due to the fact that the matrix is converted in a dense format, these methods should be used only for very small matrices.

```
Inversion<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.SetOperator(mat);
ls.Build();

ls.Solve(rhs, &x);
```

"A direct solver"

Note These methods works only for Local-type of problems (no distributed problems).

6.14 Eigenvalue Solvers

6.14.1 AMPE-SIRA

The Shift Invert Residual Arnoldi (SIRA) is an eigenvalue solver which solves parts of eigenvalues of a matrix. It computes the closest eigenvalues to zero of a symmetric positive definite (SPD) matrix. Details can be found in [27, 28, 32, 33].

ValueType	Building phase	Solving phase	Available
D,D-F	H,C	H,C,O	B (GPLv3 only)

```
SIRA< LocalMatrix<double>, LocalVector<double>, double > sira;

sira.SetOperator(mat);
// Set the number of target eigenvalues
sira.SetNumberOfEigenvalues(3);

// Set precontidioner of inner linear system
MultiColoredILU<LocalMatrix<double>, LocalVector<double>, double > p;
sira.SetInnerPreconditioner(p);

sira.Build();

sira.Solve(&eigenvalues);
```

"SIRA solver with 3 target eigenvalues"

An adaptive mixed precision (AMP) scheme can be performed on the SIRA method, where the stopping criteria in the inner loop is inspired by Hochstenbach and Notay (HN criteria, see [22]), the declaration takes two different value type.

```
SIRA< LocalMatrix<double>, LocalVector<double>, double,
      LocalMatrix<float>,  LocalVector<float>,  float > ampe_sira;

ampe_sira.SetOperator(mat);
ampe_sira.SetNumberOfEigenvalues(3);

// Set precontidioner of inner linear system
MultiColoredILU<LocalMatrix<double>, LocalVector<double>, double > p1;
MultiColoredILU<LocalMatrix<float>,  LocalVector<float>,  float >  p2;
ampe_sira.SetInnerPreconditioner(p1, p2);

ampe_sira.Build();
ampe_sira.Init(1e-10, 200, 0.8);
ampe_sira.Solve(&eigenvalues);
```

"SIRA solver with adaptive mixed precision scheme"

The $Init(1e{-}10,\ 200,\ 0.8)$ after $ampe\_sira.Build()$ is an optional function to configure the iteration controller in SIRA solver. All the parameters remain default when the value is set zero. The first argument of $Init()$ sets the absolute tolerance for eigenvalues, where the default value is $1e-7$. The second argument sets the maximun number of outer loop iteration for solving one eigenvalue. The default value is 300, and the final number would be $value\ set \times number\ of\ target\ eigenvalues$. The third argument sets the maximum number of inner loop iteration. The default number is 100000. If the value is set on $(0, 1]$, then the maximum number would be $val \times dimension\ of\ operator$, otherwise it would be set directly the value when it is on $(1, \infty)$.

If there is a special starting vector for SIRA solver, say vec, one can assign the vector to SIRA in the first argument in $sira.Solve()$, that is $sira.Solve(vec,\ \&eigenvalues)$.

Note The stopping criteria of SIRA without mixed precision scheme is by default a constant tolerance. If one wants to apply the HN criteria, add $sira.SetInnerStoppingCriterion(1)$ before $sira.Build()$.

Note The preconditioner in the inner linear system of SIRA is set by default FSAI level 1.

Chapter 7

Preconditioners

In this chapter, all preconditioners are presented. All preconditioners support local operators. They can be used as a global preconditioner via block-jacobi scheme which works locally on each interior matrix.

To provide fast application, all preconditioners require extra memory to keep the approximated operator.

7.1 Code Structure

The preconditioners provide a solution to the system $Mz = r$, where either the solution z is directly computed by the approximation scheme or it is iterativily obtained with $z = 0$ initial guess.

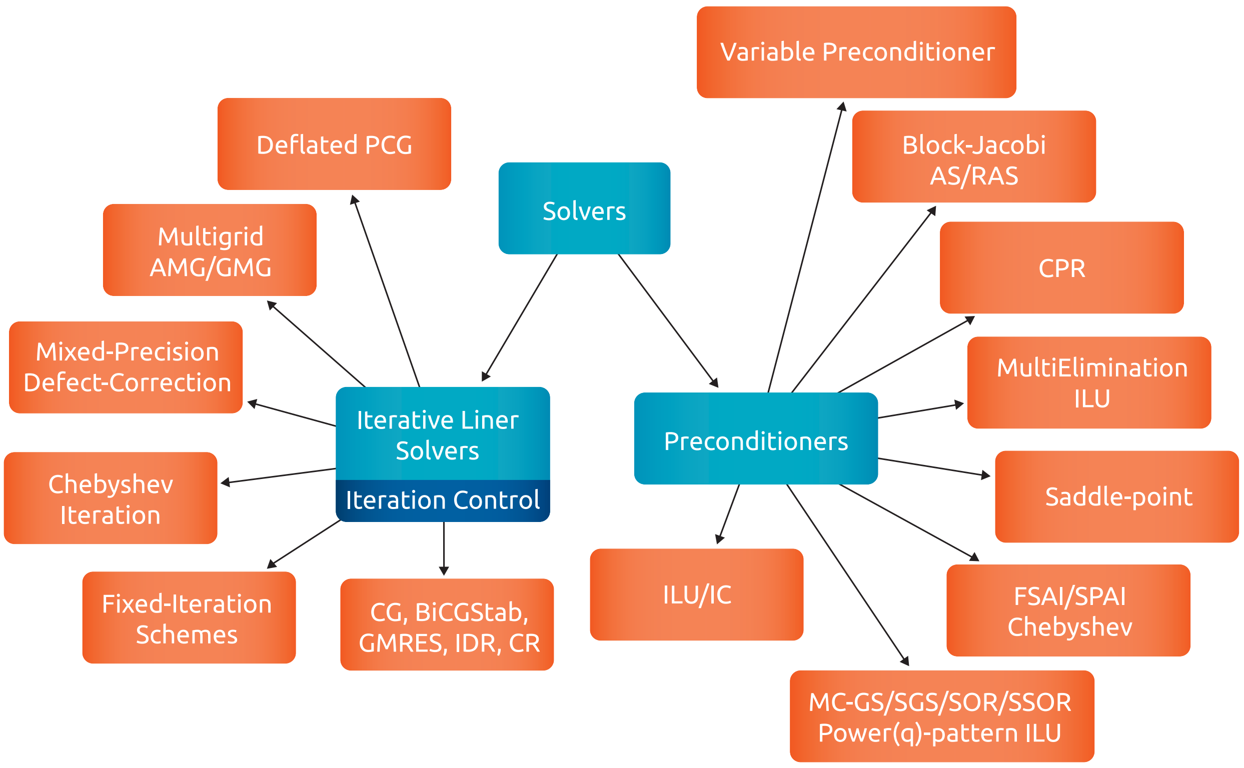


Figure 7.1: Solver's classes hierarchy

7.2 Jacobi

The Jacobi preconditioner is the simplest parallel preconditioner, details can be found in [46, 10, 15].

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O,X	B,S,M

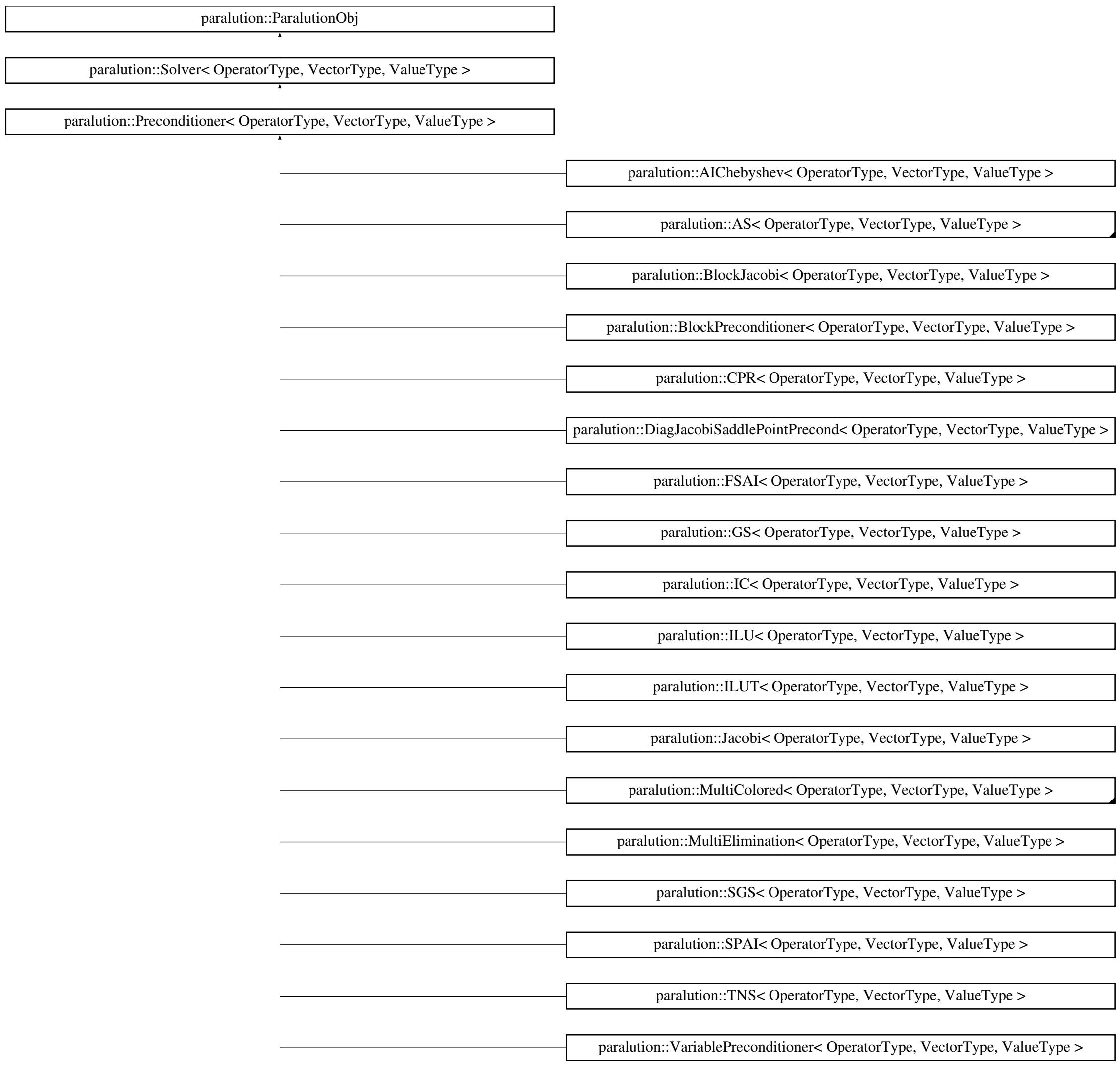

paralution::ParalutionObj
paralution::Solver< OperatorType, VectorType, ValueType >
paralution::Preconditioner< OperatorType, VectorType, ValueType >
paralution::AIChebyshev< OperatorType, VectorType, ValueType >
paralution::AS< OperatorType, VectorType, ValueType >
paralution::BlockJacobi< OperatorType, VectorType, ValueType >
paralution::BlockPreconditioner< OperatorType, VectorType, ValueType >
paralution::CPR< OperatorType, VectorType, ValueType >
paralution::DiagJacobiSaddlePointPrecond< OperatorType, VectorType, ValueType >
paralution::FSAI< OperatorType, VectorType, ValueType >
paralution::GS< OperatorType, VectorType, ValueType >
paralution::IC< OperatorType, VectorType, ValueType >
paralution::ILU< OperatorType, VectorType, ValueType >
paralution::ILUT< OperatorType, VectorType, ValueType >
paralution::Jacobi< OperatorType, VectorType, ValueType >
paralution::MultiColored< OperatorType, VectorType, ValueType >
paralution::MultiElimination< OperatorType, VectorType, ValueType >
paralution::SGS< OperatorType, VectorType, ValueType >
paralution::SPAI< OperatorType, VectorType, ValueType >
paralution::TNS< OperatorType, VectorType, ValueType >
paralution::VariablePreconditioner< OperatorType, VectorType, ValueType >

Figure 7.2: Preconditioners

```
Jacobi<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

ls.SetPreconditioner(p);
ls.Build();
```

"Jacobi preconditioner"

7.3 Multi-colored (Symmetric) Gauss-Seidel and SOR

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,	H,C,O,X	B,S,M

The additive preconditioners are based on the splitting of the original matrix. Higher parallelism in solving the forward and backward substitution is obtained by performing a multi-colored decomposition. Details can be found in [35, 46].

```
MultiColoredSGS<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

ls.SetPreconditioner(p);
ls.Build();
```

"Multi-colored symmetric Gauss-Seidel preconditioner"

```
MultiColoredGS<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

p.SetRelaxation(1.6);

ls.SetPreconditioner(p);
ls.Build();
```

”Multi-colored SOR with relaxation parameter 1.6”

7.4 Incomplete LU with levels – ILU(p)

ILU(p) factorization based on p-levels. Details can be found in [46].

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C	B,S,M

```
ILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

p.Set(1);
ls.SetPreconditioner(p);
ls.Build();
```

”ILU(1) preconditioner - based on levels”

7.5 Incomplete Cholesky – IC

IC factorization without additional fill-ins. Details are given in [46].

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C	B,S,M

```
IC<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

ls.SetPreconditioner(p);
ls.Build();
```

”IC preconditioner”

Note This implementation is still experimental and it is highly recommended to use the ILU preconditioner instead.

7.6 Incomplete LU with threshold – ILUT(t,m)

Incomplete LU (ILUT(t,m)) factorization based on threshold (t) and maximum number of elements per row (m). Details can be found in [46]. The preconditioner can be initialized with the threshold value only or with threshold and maximum number of elements per row.

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C	B,S,M

```
ILUT<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

p.Set(0.01);
ls.SetPreconditioner(p);
ls.Build();
```

”ILUT(0.01) preconditioner”

Note This implementation is still experimental.

7.7 Power(q)-pattern method – ILU(p,q)

ILU(p,q) is based on the ILU(p) factorization with a power(q)-pattern method, the algorithm can be found in[35]. This method provides a higher degree of parallelism of forward and backward substitution compared to the standard ILU(p) preconditioner.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O,X	B,S,M

Note If the preconditioner is initialized with only the first argument (p) then q is taken to be $p + 1$.

```
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

p.Set(1, 2);
ls.SetPreconditioner(p);
ls.Build();
```

"ILU(1 2) preconditioner - based on power(q)-pattern method"

7.8 Multi-Elimination ILU

The multi-elimination ILU preconditioner is based on the following decomposition

$$A = \begin{bmatrix} D & F \\ E & C \end{bmatrix} = \begin{bmatrix} I & 0 \\ ED^{-1} & I \end{bmatrix} \times \begin{bmatrix} D & F \\ 0 & \hat{A} \end{bmatrix}, \tag{7.1}$$

where $\hat{A} = C - ED^{-1}F$.

To make the inversion of D easier, we permute the preconditioning before the factorization with a permutation P to obtain only diagonal elements in D. The permutation here is based on a maximal independent set. Details can be found in [46].

This procedure can be applied to the block matrix $\hat{A}$, in this way we can perform the factorization recursively. In the last level of the recursion, we need to provide a solution procedure. By the design of the library, this can be any kind of solver. In the following example we build a preconditioned CG solver with a multi-elimination preconditioner defined with 2 levels and without drop-off tolerance. The last block of preconditioner is solved using a Jacobi preconditioner.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O	B,S,M

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > cg;
MultiElimination<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;
Jacobi<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > j_p;

p.Set(j_p, 2);

cg.SetOperator(mat);
cg.SetPreconditioner(p);
cg.Build();

cg.Solve(rhs, &x);
```

"PCG solver with multi-elimination ILU preconditioner and Jacobi"

7.9 Diagonal Preconditioner for Saddle-Point Problems

Consider the following saddle-point problem

$$A = \begin{bmatrix} K & F \\ E & 0 \end{bmatrix}. \tag{7.2}$$

For such problems we can construct a diagonal Jacobi-like preconditioner of type

$$P = \begin{bmatrix} K & 0 \\ 0 & S \end{bmatrix} \tag{7.3}$$

with $S = ED^{-1}F$, where D are the diagonal elements of K.

The matrix S is fully constructed (via sparse matrix-matrix multiplication).

The preconditioner needs to be initialized with two external solvers/preconditioners – one for the matrix K and one for the matrix S.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O,X	B,S,M

```
GMRES<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > gmres;

DiagJacobiSaddlePointPrecond<LocalMatrix<ValueType>, LocalVector<ValueType>,
    ValueType > p;
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p_k;
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p_s;

p.Set(p_k, p_s);

gmres.SetOperator(mat);
gmres.SetBasisSize(50);
gmres.SetPreconditioner(p);
gmres.Build();

gmres.Solve(rhs, &x);
```

"GMRES solver with diagonal Jacobi-like preconditioner for Saddle-Point problems"

7.10 Chebyshev Polynomial

The Chebyshev approximate inverse matrix is based on the work [14].

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O,X	B,S,M

```
Chebyshev<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

ls.SetOperator(mat);
ls.Set(lambda_min, lambda_max);
ls.Build();

ls.Solve(rhs, &x);
```

"Chebyshev polynomial preconditioner"

7.11 FSAI(q)

The Factorized Sparse Approximate Inverse preconditioner computes a direct approximation of M^{-1} by minimizing the Frobenius norm $||I - GL||_F$ where L denotes the exact lower triangular part of A and $G := M^{-1}$. The FSAI preconditioner is initialized by q, based on the sparsity pattern of $|A|^q$ [35]. However, it is also possible to supply external sparsity patterns in form of the LocalMatrix class. Further details on the algorithm are given in [31].

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C,O,X	B,S,M

```
FSAI<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

p.Set(2);
ls.SetPreconditioner(p);
ls.Build();
```

”FSAI(2) preconditioner”

Note The FSAI(q) preconditioner is only suited for SPD matrices.

7.12 SPAI

The SParse Approximate Inverse algorithm is an explicitly computed preconditioner for general sparse linear systems. In its current implementation, only the sparsity pattern of the system matrix is supported. The SPAI computation is based on the minimization of the Frobenius norm $||AM - I||_F$. Details can be found in [21].

ValueType	Building phase	Solving phase	Available
D,F,C	H	H,C,O,X	B,S,M

```
SPAI<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

ls.SetPreconditioner(p);
ls.Build();
```

”SPAI preconditioner”

Note The SPAI implementation is still experimental. The current version is based on a original static matrix pattern (similar to ILU0).

7.13 Block Preconditioner

When handling vector fields, typically one can try to use different preconditioners/solvers for the different blocks. For such problems, the library provides a block-type preconditioner. This preconditioner builds the following block-type matrix

$$P = \begin{bmatrix} A_d & 0 & . & 0 \\ B_1 & B_d & . & 0 \\ . & . & . & . \\ Z_1 & Z_2 & . & Z_d \end{bmatrix} \tag{7.4}$$

The solution of P can be performed in two ways. It can be solved by block-lower-triangular sweeps with inversion of the blocks $A_d ... Z_d$ and with a multiplication of the corresponding blocks, this is set by the function *SetLSolver()* (which is the default solution scheme). Alternatively, it can be used only with an inverse of the diagonal $A_d ... Z_d$ (such as Block-Jacobi type) by using the function *SetDiagonalSolver()*.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O,X	B,S,M

```
GMRES<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

BlockPreconditioner<LocalMatrix<ValueType>, LocalVector<ValueType>,
                              ValueType > p;
Solver<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > **p2;

```

```
int n = 2;
int *size;
size = new int[n];

p2 = new Solver<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType >*[n];

  for (int i=0; i<n; ++i) {
    size[i] = mat.get_nrow() / n ;

    MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>,
                    ValueType > *mc;
    mc = new MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>,
                             ValueType >;
    p2[i] = mc;

  }

p.Set(n, size, p2);

ls.SetOperator(mat);
ls.SetPreconditioner(p);

ls.Build();

ls.Solve(rhs, &x);
```

"Block Preconditioner for two MC-ILU"

```
GMRES<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

BlockPreconditioner<LocalMatrix<ValueType>, LocalVector<ValueType>,
                    ValueType > p;
Solver<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > **p2;

int n = 2;
int *size;
size = new int[n];

p2 = new Solver<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType >*[n];


// Block 0
size[0] = mat.get_nrow() / n ;
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > *mc;
mc = new MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>,
    ValueType >;
p2[0] = mc;

// Block 1
size[1] = mat.get_nrow() / n ;
AMG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > *amg;
amg = new AMG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType >;
amg->InitMaxIter(2);
amg->Verbose(0);
p2[1] = amg;


p.Set(n, size, p2);
p.SetDiagonalSolver();

ls.SetOperator(mat);
ls.SetPreconditioner(p);

ls.Build();

ls.Solve(rhs, &x);
```

"Block Preconditioner for MC-ILU and AMG"

7.14 Additive Schwarz and Restricted Additive Schwarz – AS and RAS

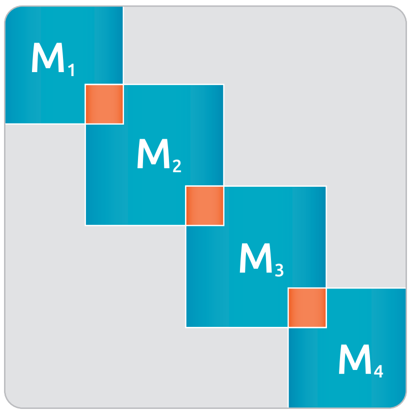
As a preconditioning technique, we can decompose the linear system $Ax = b$ into small sub-problems based on

$$A_i = R_i^T A R_i \tag{7.5}$$

where R_i are restriction operators. Thus, we can define:

- Additive Schwarz (AS) – the restriction operators produce sub-matrices which overlap. This leads to contributions from two preconditioners on the overlapped area, see Figure 7.3 (left). Thus these contribution sections are scaled by 1/2.
- Restricted Additive Schwarz (RAS) – this is a mixture of the pure block-Jacobi and the additive Schwarz scheme. Again, the matrix A is decomposed into squared sub-matrices. The sub-matrices are large as in the additive Schwartz approach – they include overlapped areas from other blocks. After we solve the preconditioning sub-matrix problems, we provide solutions only to the non-overlapped area, see Figure 7.3 (right).

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O,X	B,S,M



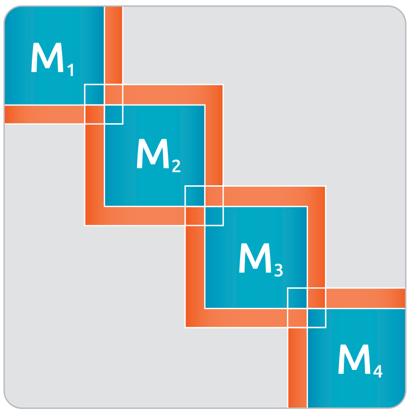


Figure 7.3: Example of a 4 block-decomposed matrix – Additive Schwarz (AS) with overlapping preconditioner (left) and Restricted Additive Schwarz (RAS) preconditioner (right)

Details can be found in [46, 11, 45, 47, 52].

The library provides Additive Schwarz (called *AS*) and Restricted Additive Schwarz (called *RAS*) preconditioners. For both preconditioners, the user need to to specify the number of blocks, the size of the overlapping region and the type of preconditioners which should be used on the blocks.

```
// Linear Solver
GMRES<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;

// Preconditioner
//  AS<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;
RAS<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

// Second level preconditioners
Solver<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > **p2;
```

```

int n = 3;
int *size;
size = new int[n];

p2 = new Solver<LocalMatrix<ValueType>,
                LocalVector<ValueType>, ValueType >*[n];

for (int i=0; i<n; ++i) {
  size[i] = mat.get_nrow() / n ;

  MultiColoredILU<LocalMatrix<ValueType>,
                  LocalVector<ValueType>, ValueType > *mc;
  mc = new MultiColoredILU<LocalMatrix<ValueType>,
                           LocalVector<ValueType>, ValueType >;
  p2[i] = mc;

 }

 p.Set(n,
       20,
       p2);

ls.SetOperator(mat);
ls.SetPreconditioner(p);

ls.Build();

ls.Solve(rhs, &x);
```

"(Restricted) Additive Schwarz defined with 3 blocks and 10 overlapping elements"

7.15 Truncated Neumann Series Preconditioner (TNS)

The Truncated Neumann Series Preconditioner (TNS) is based on

$$M^{-1} = K^T D^{-1} K, \tag{7.6}$$

where $K = (I - LD^{-1} + (LD^{-1})^2)$, D is the diagonal matrix of A and L is strictly the lower triangular part of A.

The preconditioner can be computed in two forms - explicitly and implicitly. In the implicit form, the full construction of M is performed via matrix-matrix operations. Whereas in the explicit from, the application of the preconditioner is based on matrix-vector operations only. The matrix format for the stored matrices can be specified.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C	H,C,O,X	B,S,M

```
CG<LocalMatrix<double>, LocalVector<double>, double > ls;

TNS<LocalMatrix<double>, LocalVector<double>, double > p;

// Explicit or implicit
//  p.Set(false);

 ls.SetOperator(mat);
 ls.SetPreconditioner(p);

 ls.Build();

ls.Solve(rhs, &x);
```

"Truncated Neumann Series Preconditioner (TNS)"

7.16 Variable Preconditioner

The Variable Preconditioner can hold a selection of preconditioners. In this way any type of preconditioners can be combined. As example, the variable preconditioner can combine Jacobi, GS and ILU – then, the first iteration of the iterative solver will apply Jacobi, the second iteration will apply GS and the third iteration will apply ILU. After that, the solver will start again with Jacobi, GS, ILU. It is important to be used with solvers which can handle such type of preconditioners.

ValueType	Building phase	Solving phase	Available
D,F	H,C,O	H,C,O	S,M

7.17 CPR Preconditioner

This preconditioner is still under development and it is not part of the official distributions

The CPR (Constrained Pressure Residual) preconditioner is a special type of preconditioner for reservoir simulations. The preconditioner contains two (or three) stage sub-preconditioners.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O,X	H,C,O,X	S,M

The user has the ability to select any combination of the two (or three) sub-preconditioners, as well as to use the default ILU-like and AMG configuration.

7.18 MPI Preconditioners

The MPI preconditioners are designed to work in parallel on all MPI processes. In this way, any type of preconditioner can be wrapped in a Block-Jacobi type, where the preconditioner will be applied locally on each interior matrix.

ValueType	Building phase	Solving phase	Available
D,F,C	H,C,O	H,C,O	M

Chapter 8

Backends

The support of accelerator devices is embedded in the structure of PARALUTION. The primary goal is to use this technology whenever possible to decrease the computational time.

Note Not all functions are ported and presented on the accelerator backend. This limited functionality is natural since not all operations can be performed efficiently on the accelerators (e.g. sequential algorithms, I/O from the file system, etc).

8.1 Backend and Accelerators

Currently, the library supports CUDA GPU [43], OpenCL [29] and MIC [26] devices. Due to its design the library can be extended to support Cilk [40], Intel TBB [25] backends. The extension of the library will not reflect the algorithms which are based on it.

If a particular function is not implemented for the used accelerator, the library will move the object to the host and compute the routine there. In this a case warning message of level 2 will be printed. For example if we use an OpenCL backend and we want to perform an ILU(0) factorization which is currently not available, the library will move the object to the host, perform the routine there and print the following warning message:

```
*** warning: LocalMatrix::ILU0Factorize() is performed on the host
```

"Warning message for performing the ILU(0) factorization on the host"

8.2 Copy

All matrices and vectors have a *CopyFrom()* function which can be used to transfer data from and to the accelerator.

```
LocalVector<ValueType> vec1, vec2;

// vec1 is on the host
vec1.MoveToHost();

// vec2 is on the accelrator
vec2.MoveToAcclerator();

// copy vec2 to vec1
vec1.CopyFrom(vec2);

// copy vec1 to vec2
vec2.CopyFrom(vec1);
```

"Copying data to and from the accelerator"

8.3 CloneBackend

When creating new objects, often the user has to ensure that it is allocated on the same backend as already existing objects. This can be achieved via the *CloneBackend* function. For example, consider a matrix *mat* and a vector *vec*. If a SpMV operation should be performed, both objects need to be on the same backend. This can be achieved by calling *vec.CloneBackend(mat)*. In this way, the vector will have the same backend as the matrix. Analoguely, *mat.CloneBackend(vec)* can be called. Then, the matrix will end up with the same backend as the vector.

8.4 Moving Objects To and From the Accelerator

All object in PARALUTION can be moved to the accelerator and to the host.

```
LocalMatrix<ValueType> mat;
LocalVector<ValueType> vec1, vec2;

// Performing the matrix-vector multiplication on host
mat.Apply(vec1, &vec2);

// Move data to the accelerator
mat. MoveToAccelerator();
vec1.MoveToAccelerator();
vec2.MoveToAccelerator();

// Performing the matrix-vector multiplication on accelerator
mat.Apply(vec1, &vec2);

// Move data to the host
mat. MoveToHost();
vec1.MoveToHost();
vec2.MoveToHost();
```

"Using an accelerator for sparse matrix-vector multiplication"

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

ls.SetOperator(mat);
ls.SetPreconditioner(p);

ls.Build();

mat.MoveToAccelerator();
rhs.MoveToAccelerator();
x.MoveToAccelerator();
ls.MoveToAccelerator();

ls.Solve(rhs, &x);
```

"Using an accelerator for preconditioned CG solver (building phase on the host)"

```
CG<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > ls;
MultiColoredILU<LocalMatrix<ValueType>, LocalVector<ValueType>, ValueType > p;

mat.MoveToAccelerator();
rhs.MoveToAccelerator();
x.MoveToAccelerator();

ls.SetOperator(mat);
ls.SetPreconditioner(p);

ls.Build();

ls.Solve(rhs, &x);
```

"Using an accelerator for preconditioned CG solver (building phase on the accelerator)"

8.5 Asynchronous Transfers

The PARALUTION library also provides asynchronous transfers of data (currently, only for CUDA backend). This can be done with the *CopyFromAsync()* function or with the *MoveToAcceleratorAsync()* and *MoveToHostAsync()*. These functions return immediately and perform the asynchronous transfer in background mode. The synchronization is done with the *Sync()* function.

```
LocalVector<ValueType> x, y;
LocalMatrix<ValueType> mat;

mat.MoveToAcceleratorAsync();
x.MoveToAcceleratorAsync();
y.MoveToAcceleratorAsync();

// do some computation

mat.Sync();
x.Sync();
y.Sync();
```

"Asynchronous Transfers with *MoveToAcceleratorAsync*"

```
LocalVector<ValueType> x, y;

y.MoveToHost();
x.MoveToAccelerator();

x.CopyFromAsync(y);

// do some computation

x.Sync();
```

"Asynchronous Transfers with *CopyFromAsync*"

When using the *MoveToAcceleratorAsync()* and *MoveToHostAsync()* functions, the object will still point to its original location (i.e. host for calling *MoveToAcceleratorAsync()* and accelerator for *MoveToHostAsync()*). The object will switch to the new location after the *Sync()* function is called.

```
LocalVector<ValueType> x, y;
LocalMatrix<ValueType> mat;

// mat, x and y are initially on the host
...

// Start async transfer
mat.MoveToAcceleratorAsync();
x.MoveToAcceleratorAsync();

// this will be performed on the host
mat.Apply(x, &y);

mat.Sync();
x.Sync();

// Move y
y.MoveToAccelerator();

// this will be performed on the accelerator
mat.Apply(x, &y);
```

"Asynchronous Transfers with *MoveToAcceleratorAsync()*"

Note The objects should not be modified during an active asynchronous transfer. However, if this happen, the values after the synchronization might be wrong.

Note CUDA backend – to use the asynchronous transfers you need to enable the pinned memory allocation. Uncomment *#define PARALUTION_CUDA_PINNED_MEMORY* in file *src/utils/allocate_free.hpp*

8.6 Systems without Accelerators

PARALUTION provides full code compatibility on systems without accelerators - i.e. the user can take the code from the GPU systems, re-compile the same code on a machine without a GPU and it will provide the same results. For example, if one compiles the above matrix-vector multiplication code on a system without GPU support, it will just perform two multiplications on the host - the *MoveToAccelerator()* and *MoveFromAccelerator()* calls will be ignored.

Chapter 9

Advanced Techniques

9.1 Hardware Locking

PARALUTION also provides a locking mechanism based on keys. Each key is generated by the MAC address of the network card and the processor's ID. The library can handle various keys (for running on various computer/nodes). See Figure 9.1 and Figure 9.2.

Available
S,M

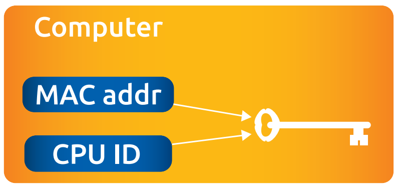


Figure 9.1: Hardware locking based on keys

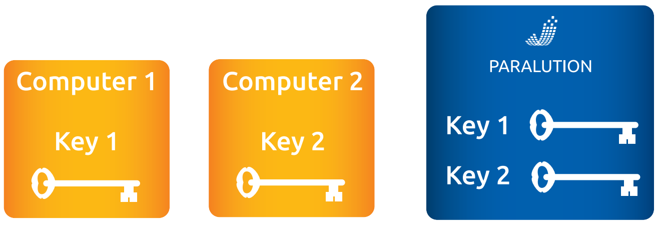


Figure 9.2: Hardware locking based on keys

The user can add the node keys directly in the code by editing *src/utils/hw_check.cpp* file. A detailed description, on the procedure how new keys are added and computed is given in the source file *src/utils/hw_check.cpp*.

Note This feature is available only under Linux OS.

9.2 Time Restriction/Limit

The user can specify also an expiration date. After this date the library cannot be initialized.

Available
S,M

To disable/enable as well as to set the date, edit file *src/utils/time_check.cpp*. In this file, there is a detailed description, on how to edit this feature.

9.3 OpenCL Kernel Encryption

The OpenCL kernels are compiled at runtime by the OpenCL compiler. Due to this fact, the source code of the OpenCl kernels is stored in a readable form in the library. To avoid an easy access to it, PARALUTION provides encryption of the kernels based on RSA mechanism.

Available
S,M

9.4 Logging Encryption

All logging messages (printed by the library) can be encrypted if the developer needs to ensure limited readability of logs. The encryption is based on RSA mechanism.

Available
S,M

9.5 Encryption

The encryption (for the OpenCL kernels or for the log files) can be turn on or off by editing *src/utils/def.hpp.*

The keys for the encryption and for the decryption are based on the RSA mechanism, the user needs to provide the three of them in the code (encryption, decryption, mask), see file *src/utils/enc.cpp*

9.6 Memory Allocation

All data which is passed to and from PARALUTION (via SetDataPtr/LeaveDataPtr) is using the memory handling functions described in the code. By default, the library uses standard *new* and *delete* functions for the host data.

Available
B,S,M

9.6.1 Allocation Problems

If the allocation fails, the library will report an error and exits. If the user requires a special treatment, it has to be placed in the file *src/utils/allocate_free.cpp.*

Available
B,S,M

9.6.2 Memory Alignment

The library can also handle special memory alignment functions. This feature need to be uncommented before the compilation process in the file *src/utils/allocate_free.cpp.*

9.6.3 Pinned Memory Allocation (CUDA)

By default, the standard host memory allocation is realized by *new* and *delete*. For a better PCI-E transfers for NVIDIA GPUs, the user can also use pinned host memory. This can be activated by uncommenting the corresponding macro in *src/utils/allocate_free.hpp.*

Available
B,S,M

Chapter 10

Plug-ins

Note that all plug-ins, except the FORTRAN one, are provided only under GPLv3 license in the Basic version (B), and there is no MPI support.

10.1 FORTRAN

PARALUTION comes with an easy to use Fortran plug-in. Currently it supports *COO* and *CSR* input matrix formats and uses the intrinsic *ISO_BIND_C* to transfer data between Fortran and PARALUTION. The argument passing for the *COO* and *CSR* subroutine calls only differ in the matrix related arrays.

```
call paralution_fortran_solve_coo(                                       &
&            n, m, nnz,                                                   &
&            'GMRES' // C_NULL_CHAR,                                      &
&            'HYB' // C_NULL_CHAR,                                        &
&            'MultiColoredILU' // C_NULL_CHAR,                            &
&            'ELL' // C_NULL_CHAR,                                        &
&            C_LOC(row), C_LOC(col), C_LOC(val), C_LOC(rhs),              &
&            1e-10_C_DOUBLE, 1e-6_C_DOUBLE, 1e+8_C_DOUBLE, 5000,          &
&            30, 0, 1,                                                    &
&            C_LOC(x), iter, resnorm, stat )
```

"Example of Fortran subroutine call using COO matrix format"

```
call paralution_fortran_solve_csr(                                       &
&            n, m, nnz,                                                   &
&            'CG' // C_NULL_CHAR,                                         &
&            'CSR' // C_NULL_CHAR,                                        &
&            'MultiColoredSGS' // C_NULL_CHAR,                            &
&            'CSR' // C_NULL_CHAR,                                        &
&            C_LOC(row_offset), C_LOC(col), C_LOC(val), C_LOC(rhs),       &
&            1e-10_C_DOUBLE, 1e-6_C_DOUBLE, 1e+8_C_DOUBLE, 2000,          &
&            0, 0, 1,                                                     &
&            C_LOC(x), iter, resnorm, stat )
```

"Example of Fortran subroutine call using CSR matrix format"

The arguments include:

- (2) Number of rows, number of columns, number of non-zero elements
- (3) Solver: CG, BiCGStab, GMRES, Fixed-Point
- (4) Operator matrix format: DENSE, CSR, MCSR, COO, DIA, ELL, HYB
- (5) Preconditioner: None, Jacobi, MultiColoredGS, MultiColoredSGS, ILU, MultiColoredILU
- (6) Preconditioner matrix format: DENSE, CSR, MCSR, COO, DIA, ELL, HYB
- (7) Row index (COO) or row offset pointer (CSR), column index, right-hand side
- (8) Absolute tolerance, relative tolerance, divergence tolerance, maximum number of iterations
- (9) Size of the Krylov subspace (GMRES), ILU(p), ILU(q)
- (10) Outputs: solution vector, number of iterations needed to converge, final residual norm, status code

A detailed listing is also given in the header of the PARALUTION Fortran plug-in file.
For a successful integration of PARALUTION into Fortran code a compiled PARALUTION library is necessary. Furthermore, you need to compile and link the Fortran plug-in (located in *src/plug-ins*) because it is not included in the compiled library file. To achieve this, a simple Makefile can be used.

```
CC=g++
FC=gfortran
FLAGS=-O3 -lstdc++ -fopenmp
INC=-I../../../build/inc
LIB=../../../build/lib/libparalution.so
OBJ=paralution_fortran.o fortran_code.o

default: fortran_code

fortran_code: $(OBJ)
  $(FC) -o fortran_code $(OBJ) $(LIB) $(FLAGS)

fortran_code.o: fortran_code.f90
  $(FC) $(FLAGS) -c fortran_code.f90

paralution_fortran.o: ../../plug-ins/paralution_fortran.cpp
  $(CC) $(FLAGS) $(INC) -c ../../plug-ins/paralution_fortran.cpp

clean:
  rm -rf *.o fortran_code
```

"Example Makefile for PARALUTION integration to Fortran"

Note Examples are in *src/examples/fortran*.

10.2 OpenFOAM

OpenFOAM [44] is a C++ toolbox for the development of customized numerical solvers, and pre-/post-processing utilities for the solution of continuum mechanics problems, including computational fluid dynamics (CFD). To use the PARALUTION solvers in an OpenFOAM application you need to compile the library and include it in the application:

- Include the chosen solvers in *Make/files*, for example if you want to add PCG you need to include *PARALUTION_DIR/build/inc/plug-ins/OpenFOAM/matrices/lduMatrix/solvers/paralution_PCG/paralution_PCG.C*
- Add in the *Make/options* file the path to the include files *-IPARALUTION_DIR/build/inc* in EXE_INC part, and in EXE_LIBS the path to the PARALUTION library *PARALUTION_DIR/build/lib/libparalution.so* needs to be added.
- Include the libraries which you want to use (*-fopenmp* for OpenMP, *-lOpenCL* for OpenCL, and *-lcudart -lcublas -lcusparse -L/usr/local/cuda/lib* for CUDA).
- Edit the *fvSolution* configuration file.
- See the example in *src/examples/OpenFOAM*

```
solvers
{
    val
    {
        solver              paralution_PCG;
        preconditioner      paralution_MultiColoredILU;
        ILUp                0;
        ILUq                1;
        MatrixFormat        HYB;
        PrecondFormat       ELL;
        useAccelerator      true;

        tolerance           1e-09;
        relTol              1e-05;
        maxIter             100000;
    };
}
```

"Example configuration for PARALUTION PCG solver"

```
solvers
{
    val
    {
        solver                     paralution_AMG;
        MatrixFormat               HYB;

        smoother                   paralution_MultiColoredGS;
        SmootherFormat             ELL;
        nPreSweeps                 1;
        nPostSweeps                2;

        CoarseGridSolver           CG;
        preconditioner             none;
        PrecondFormat              CSR;

        InterpolationType          SmoothedAggregation;
        AggrRelax                  2.0/3.0;
        nCellsInCoarsestLevel      300;
        couplingStrength           0.01;
        Relaxation                 1.3;
        scaleCorrection            true;

        tolerance                  1e-09;
        relTol                     1e-05;
        maxIter                    10000;

        useAccelerator             true;
    };
}
```

"Example configuration for PARALUTION AMG solver"

The configuration includes:

- Matrix formats: DENSE, CSR, MCSR, COO, DIA, ELL, HYB.
- Operator matrix format: field *MatrixFormat*.
- Preconditioner/smoother matrix format: field *PrecondFormat* and *SmootherFormat*.
- Solvers: paralution_PCG, paralution_PBiCG, paralution_PGMRES, paralution_AMG.
- Preconditioners/smoothers: preconditioners/smoothers are: paralution_Jacobi, paralution_MultiColoredGS (Gauss-Seidel), paralution_MultiColoredSGS (Symmetric Gauss-Seidel), paralution_MultiColoredILU (also with fields *ILUp* and *ILUq*), paralution_FSAI, paralution_MultiElimination (with fields MEp and LastBlockPrecond).
- Using accelerator (GPU): *useAccelerator* true/false.

Note You need to call the *paralution::init_paralution()* and *paralution::stop_paralution()* function in the OpenFOAM solver application file.

Note Optimal configuration with respect to the performance could be different for the host and for the accelerator backends.

Note For details, please check the source files of the solvers.

10.3 Deal.II

Deal.II [8] is a C++ program library targeted at the computational solution of partial differential equations using adaptive finite elements. It uses state-of-the-art programming techniques to offer you a modern interface to the complex data structures and algorithms required. The plug-in for the finite element package Deal.II consists of 3 functions:

- *import_dealii_matrix* – imports a sparse Deal.II matrix to PARALUTION, the user also needs to specify the sparsity pattern of the Deal.II matrix.
- *import_dealii_vector* – converts a Deal.II vector into a PARALUTION vector
- *export_dealii_vector* – exports a PARALUTION vector to a Deal.II vector. For vector functions both of the vectors have to be allocated and they need to have the same sizes.

```
#include <paralution.hpp>
#include <plug-ins/paralution_dealii.hpp>

......

// Deal.II
SparseMatrix<double> matrix;
SparsityPattern      sparsity_pattern;
Vector<double> rhs;
Vector<double> sol;

......

// PARALUTION
paralution::LocalMatrix<double> paralution_matrix;
paralution::LocalVector<double> paralution_rhs, paralution_sol;


paralution::CG<paralution::LocalMatrix<double>,
               paralution::LocalVector<double>,
               double> ls;

paralution::MultiColoredILU<paralution::LocalMatrix<double>,
                            paralution::LocalVector<double>,
                            double> paralution_precond;

......

// Allocate PARALUTION vectors
paralution_rhs.Allocate("rhs", rhs.size());
paralution_sol.Allocate("sol", sol.size());


// Import the Deal.II matrix
import_dealii_matrix(sparsity_pattern,
                     matrix,
                     &paralution_matrix);

paralution_matrix.ConvertToCSR();

// Import the rhs and the solution vectors
import_dealii_vector(rhs, &paralution_rhs);
import_dealii_vector(sol, &paralution_sol);


// Setup the linear solver
ls.Clear();
ls.SetOperator(paralution_matrix);
ls.SetPreconditioner(paralution_precond);

// Build the preconditioned CG solver
ls.Build();

```

```
// Set the matrix to HYB for better performance on GPUs
paralution_matrix.ConvertToHYB();

// Move all the data to the accelerator
paralution_matrix.MoveToAccelerator();
paralution_sol.MoveToAccelerator();
paralution_rhs.MoveToAccelerator();
ls.MoveToAccelerator();


// Solve the linear system
ls.Solve(paralution_rhs, &paralution_sol);

// Get the results back in the Deal.II vector
export_dealii_vector(paralution_sol, &sol);
```

"Integration of a PCG solver into Deal.II"

Note Compatibility problems could occur if the Deal.II library has been compiled with TBB support and the PARALUTION library is with Intel OpenCL (CPU) backend support. Compile the Deal.II code in debug mode for details.

Note Examples are in *src/examples/dealii*.

10.4 Elmer

Elmer [13] is an open source multiphysical simulation software. To use PARALUTION within Elmer, several files need to be added and modified. Please make sure that you have a working and compiled copy of Elmer available. In the following, a step-by-step guide is introduced to integrate PARALUTION into the Elmer FEM package.

1. Compile the PARALUTION library with flags *-m64 -fPIC*.

2. Edit *elmer/fem/src/SolverUtils.src* and search for

```
        IF ( ParEnv % PEs <= 1 ) THEN
          SELECT CASE(Method)
```

   where all the iterative solvers are added. Add the PARALUTION solver by inserting the case

```
              CASE('paralution')
                CALL ParalutionSolve( A, x, b, Solver )
```

3. Copy the PARALUTION plug-in files *Paralution.src* and *SolveParalution.cxx* into *elmer/fem/src*.

4. Add PARALUTION to the Makefile. This can be achieved by editing *elmer/fem/src/Makefile.in*. Here, you need to add the PARALUTION objects *Paralution.$(OBJ_EXT)* and *SolveParalution.$(OBJ_EXT)* to the *SOLVEROBJS* list as well as the rule

```
            Paralution.$(OBJEXT): Paralution.f90 Types.$(OBJEXT)
                Lists.$(OBJEXT)
```

   Furthermore, *Paralution.$(OBJEXT)* need to be added as a dependency to the *Solver.$(OBJEXT)* and *SolverUtils.$(OBJEXT)* rule.

5. In the file *elmer/fem/src/Makefile.am* the list *EXTRA_DIST* need to be extended with *SolveParalution.cxx*.

6. Finally, you need to add the library dependencies of PARALUTION (-fopenmp, -lcudart, etc. dependent on the backends you compiled PARALUTION with) to the Elmer Makefile .

7. Now you should be able to recompile the fem package of Elmer with PARALUTION.

8. Finally, to use the PARALUTION solvers in the supplied Elmer test examples, you will need to change the ASCII .sif file. Simply modify the solver to *Linear System Solver = Paralution*.

The PARALUTION solver, preconditioner, etc. can now be modified in the *SolveParalution.cxx* file using all available PARALUTION functions and classes.

10.5 Hermes / Agros2D

Hermes [4, 1] is a C++ library for rapid development of adaptive hp-FEM / hp-DG solvers. Novel hp-adaptivity algorithms help to solve a large variety of problems ranging from ODE and stationary linear PDE to complex time-dependent nonlinear multiphysics PDE systems.

How to compile Hermes/Agros2D with PARALUTION (step-by-step):

1. Download Hermes and read the quick description on how to compile it with various features
2. Download PARALUTION
3. Compile, build PARALUTION, and copy headers, and libraries so that Hermes's CMake system can find them – On Linux, installing PARALUTION to default install directories is sufficient, on Windows some paths have to be set
4. In your CMake.vars file (or directly in CMakeLists.txt) in the root of Hermes add *set(WITH_PARALUTION YES)* – It is on by default, so by default one has to include PARALUTION to build Hermes
5. That is it, build Hermes, it will automatically link to PARALUTION, include headers and make it usable.

How to use PARALUTION:

1. Check the doxygen manual of the classes
   (a) Hermes::Algebra::ParalutionMatrix
   (b) Hermes::Algebra::ParalutionVector
   (c) Hermes::Preconditioners::ParalutionPrecond
   (d) Hermes::Solvers::IterativeParalutionLinearMatrixSolver
   (e) Hermes::Solvers::AMGParalutionLinearMatrixSolver
   (f) and all classes that these inherit from
2. If you want to see Hermes and PARALUTION readily work together, take any test example in the hermes2d folder in the Hermes root and add one of these lines at the beginning of your main()
   (a) HermesCommonApi.set_integral_param_value(matrixSolverType, SOLVER_PARALUTION_ITERATIVE); // to use iterative solver
   (b) HermesCommonApi.set_integral_param_value(matrixSolverType, SOLVER_PARALUTION_AMG); // to use AMG solver
3. Solver classes of Hermes (NewtonSolver, PicardSolver, LinearSolver, ...) will then take this API setting into account and use PARALUTION as the matrix solver.

10.6 Octave/MATLAB

MATLAB [36] and GNU Octave [2] are computing languages for numerical computations.

How to compile the PARALUTION example for Octave/MATLAB:

1. The Octave/MATLAB example is located in *src/plug-ins/MATLAB*.
2. To compile the example for Octave, the Octave development environment is required. (Ubuntu: *liboctave-dev*)
3. For compilation, open a terminal and run *make octave* to compile the example for Octave or *make matlab* for MATLAB respectively.
4. Once compiled, a file called *paralution_pcg.mex* should be listed.

```
user@pc:~/paralution/src/plug-ins/MATLAB$ make octave
mkoctfile --mex -c paralution_pcg.cpp -o octave_paralution_pcg.o -I../../../build/inc
mkoctfile --mex octave_paralution_pcg.o -L../../../build/lib -lparalution
mv octave_paralution_pcg.mex paralution_pcg.mex
user@pc:~/paralution/src/plug-ins/MATLAB$
```

"The PARALUTION Octave example compile process"

How to run the PARALUTION Octave/MATLAB examples:

1. Start Octave/MATLAB
2. Start the PARALUTION example by the command *pcg_example*.

```
octave:1> pcg_example
pcg: converged in 159 iterations. the initial residual norm was reduced 1.06664e+06 times.
Number of CPU cores: 8
Host thread affinity policy - thread mapping on every second core (avoiding HyperThreading)
Number of GPU devices in the system: 2
LocalMatrix name=A; rows=10000; cols=10000; nnz=49600; prec=64bit; asm=no; format=CSR; host backend={CPU(OpenMP)}; accelerator backend={GPU(CUDA)}; current=GPU(CUDA)
PCG solver starts, with preconditioner:
Jacobi preconditioner
IterationControl criteria: abs tol=0; rel tol=1e-06; div tol=1e+08; max iter=500
IterationControl initial residual = 100
IterationControl RELATIVE criteria has been reached: res norm=9.37523e-05; rel val=9.37523e-07; iter=159
PCG ends
dim =  100
L2 norm |x_matlab - x_paralution| is
 1.9739e-09
octave:2>
```

"The PARALUTION Octave example output"

Create your own solver and preconditioner setup:

1. Open the PARALUTION example file *paralution_pcg.cpp*.
2. Change all PARALUTION relevant structures, similar to the supplied C++ examples.
3. Save *paralution_pcg.cpp* and repeat the compilation steps.

Chapter 11

Remarks

11.1 Performance

- Solvers – typically the PARALUTION solvers perform better than MKL/CUSPARSE based solvers due to faster preconditioners and better routines for matrix and vector operations.
- Solvers – you can also build the solvers (via Build()) on the accelerator. In many cases this is faster than computing it on the host, especially for GPU backends.
- Sizes – small-sized problems tend to perform better on the host (CPU) due to the good caching system, while large-sized problems typically perform better on GPU devices.
- Vectors – avoid accessing via *[]* operators, use techniques based on *SetDataPtr()* and *LeaveDataPtr()* instead.
- Host/Threads – by default the host OpenMP backend picks the maximum number of threads available. However, if your CPU supports HyperThreading, it will allow to run two times more threads than number of cores. This, in many cases, leads to lower performance. If your system supports HyperThreading, you may observe a performance increase by setting the number of threads (via the *set_omp_threads_paralution* function) equal to the number of physical cores.
- Solving a system with multiple right-hand-sides – if you need to solve a linear system multiple times, avoid constructing the solver/preconditioner every time.
- Solving similar systems – if you are solving similar linear systems, you might want to consider to use the same preconditioner to avoid long building phases.
- Matrix formats – in most of the cases, the classical CSR format performs very similar to all other formats on the CPU. On accelerators with many-cores (like GPU), formats like DIA and ELL typically perform better. However, for general sparse matrices one could use HYB format to avoid large memory overhead (e.g. in DIA or ELL formats). The multi-colored preconditioners could be performed in ELL for most of the matrices.
- Matrix formats - not all matrix conversions are performed on the device, the platform will give you a warning if the object need to be moved.
- Integration – if you are deploying the PARALUTION library into another software framework try to design your integration functions to avoid *init_paralution()* and *stop_paralution()* every time you call a solver in the library.
- Compilation – be sure to compile the library with the correct optimization level (*-O3*).
- Info – check if your solver is really performed on the accelerator by printing the matrix information (*my_matrix.info()*) just before calling the *ls.Solve* function.
- Info – check the configuration of the library for your hardware with *info_paralution()*.
- Mixed-precision defect correction – this technique is recommended for accelerators (e.g. GPUs) with partial or no double precision support. The stopping criteria for the inner solver has to be tuned well for good performance.
- Plug-ins – all plug-ins perform an additional copying of the data to construct the matrix, solver, preconditioner, etc. This results in overhead when calling the PARALUTION solvers/schemes. To avoid this, adapt the plug-in to your application as much as possible.
- Verbose level – it is a good practice to use the verbose level 2 to obtain critical messages regarding the performance.

- Xeon Phi – the allocation of memory on the Xeon Phi is slow, try to avoid unnecessary data allocation whenever is possible.

11.2 Accelerators

- Old GPUs - PARALUTION requires NVIDIA GPU with minimum compute capability 2.0, if the GPU is not 2.0 the computation will fall back to the OpenMP host backend.
- Avoid PCI-Express communication – whenever possible avoid extra PCI communication (like copying data from/to the accelerator), check also the internal structure of the functions.
- GPU init time – if you are running your computation on a NVIDIA GPU card where no X Windows is running (for Linux OS), you can minimize the initialization time of the GPUs by *nvidia-smi -pm 1* which eliminates reloading the driver every time you launch your GPU program.
- Pinned memory – pinned memory allocation (page-locked) are used for all host memory allocations when using the CUDA backend. This provides faster transfers over the PCI-Express and allows asynchronous data movement. By default this option is disabled, to enable the pinned memory allocation uncomment *#define PARALUTION_CUDA_PINNED_MEMORY* in file *src/utils/allocate_free.hpp*
- Async transfers – the asynchronous transfers are available only for the CUDA backend so far. If async transfers are called from other backends they will perform simple sync move or copy.
- Xeon Phi – the Intel MIC architecture could be used also via the OpenCL backend. However the performance is not so great due to the fact that most of the kernels are optimize for GPU devices.
- Xeon Phi – you can tune the number OpenCL parameters, after the execution of *cmake* and before compiling the library with *make* edit the OpenCL hardware parameters located in *src/utils/HardwareParameters.hpp*.
- OpenCL (x86 CPU) – the sparse-matrix vector multiplication in COO format is hanging, we are working to fix that.
- OpenCL – after calling the cmake you can set the block size and the warp size by editing *src/utils/HardwareParameters.hpp*. After that you just need to compile the library with make. Alternatively you can modify the *src/utils/ocl_check_hw.cpp* file.

11.3 Plug-ins

- Deal.II – to avoid the initialization time of the accelerator (if used), put the *init_paralution()* and *stop_paralution()* function in the main application file and include the library in the header.

11.4 Correctness

- Matrix – if you are assembling or modifying your matrix, you can check your matrix in octave/MATLAB by just writing it into a matrix-market file and read it via *mmread()* function [39]. You can also input a MATLAB/octave matrix in such way.
- Solver tolerance – be sure you are setting the correct relative and absolute tolerance values for your problem.
- Solver stopping criteria – check the computation of the relative stopping criteria if it is based on $\frac{|b-Ax^k|_2}{|b-Ax^0|_2}$ or $\frac{|b-Ax^k|_2}{|b|_2}$.
- Ill-conditioned problems – solving very ill-conditioned problems by iterative methods without a proper preconditioning technique might produce wrong results. The solver could stop by showing a low relative tolerance based on the residual but this might be wrong.
- Ill-conditioned problems – building the Krylov subspace for many ill-conditioned problems could be a tricky task. To ensure orthogonality in the subspace you might want to perform double orthogonalization (i.e. re-orthogonalization) to avoid rounding errors.
- I/O files – if you read/write matrices/vectors from files, check the ASCII format of the values (e.g. 34.3434 or $3.43434E+01$).

11.5 Compilation

- OpenMP backend – the OpenMP support is by default enabled. To disable it you need to specify *-DSUPPORT_OMP=OFF* in the cmake

- CUDA 5.5 and gcc 4.8 – these compiler versions are incompatible (the compilation will report error *"kernel launches from templates are not allowed in system files"*). Please use lower *gcc* version, you can push the *nvcc* compiler to use it by setting a link in the default CUDA installation directory (*/usr/local/cuda*) - e.g. by running under root *ln -s /usr/bin/gcc-4.4 /usr/local/cuda/bin/gcc*. Or try the *-ccbin* option.
- CUDA backend – be sure that the paths are correctly set (e.g. for Linux *export LD_LIBRARY_PATH= $LD_LIBRARY_PATH:/usr/local/cuda/lib64/* and *export CPLUS_INCLUDE_PATH=$CPLUS_INCLUDE_PATH: /usr/local/cuda/include/*). Then you can run cmake with *make -DSUPPORT_OCL=OFF -DSUPPORT_CUDA=ON ..*
- OpenCL backend – similar for CUDA backend, you need to set the correct paths for the OpenCL library and then you can run cmake with Then you can run cmake with *make -DSUPPORT_OCL=ON -DSUPPORT_CUDA=OFF ..*
- OpenCL backend – when compiling the library with OpenCL (with cmake or with Makefile), during the compilation process you will be asked to select an OpenCL platform and device (if you have more than one). By doing that, the library will select the optimal number of threads and blocks for your hardware. Later on, you can change the platform and device, via the *set_ocl_paralution()* or *select_device_paralution()* function, but the threads configuration will be not changed.
- MIC backend – the Intel Compiler should be loaded (*icc*), then run the camke with *cmake -DSUPPORT_MIC=ON -DSUPPORT_CUDA=OFF -DSUPPORT_OCL=OFF ..*

11.6 Portability

- Different backends – you do not have to recompile your code if you want to run your program with different backends. You just need to load the corresponding dynamic library. As an example, if you compile the library with OpenCL support in */usr/local/paralution-ocl/build/lib* and with CUDA support in */usr/local/paralution-cuda/build/lib*, you will just need to set the path (i.e. *export LD_LIBRARY_PATH= $LD_LIBRARY_PATH:/usr/local/paralution-ocl/build/lib* or *export LD_LIBRARY_PATH= $LD_LIBRARY_PATH: /usr/local/paralution-cuda/build/lib*) and just run your program.

Chapter 12

Performance Benchmarks

12.1 Single-Node Benchmarks

12.1.1 Hardware Description

Hardware	Cores/SP	Memory [GB]	Theoretical Bandwidth [GB/s]	Backend	Notes
2x Xeon E5-2680	2x 8	64	2x 51.2	OpenMP(Host)	NUMA
2x Xeon E5620	2x 4	96	2x 25.6	OpenMP(Host)	NUMA
Core i7 4770K	4	32	25.6	OpenMP(Host)	UMA
Core i7 620M	2	8	17.1	OpenMP(Host)	UMA
MIC/Xeon Phi 5110	60	8	320	OpenMP(MIC)	ECC
K20X	2688	6	250	CUDA/OpenCL	ECC
K20c	2496	5	208	CUDA/OpenCL	ECC
GTX Titan	2688	6	288	CUDA/OpenCL	no ECC
GTX680	1536	2	192	CUDA/OpenCL	no ECC
GTX590	512	1.5	163	CUDA/OpenCL	no ECC
GTX560Ti	384	2	128	CUDA/OpenCL	no ECC
FirePro (FP) W8000	1792	2	176	OpenCL	ECC

Table 12.1: Hardware configurations

The performance values are obtained with the "benchmark" tool from the PARALUTION package. The tool is compiled on various systems, no code modification is applied. The operating system (OS) for all hardware configurations is Linux. All tests are performed in double precision.

The hardware specifications are obtained from Intel [1] [2] [3] [4] [5], AMD [6] and NVIDIA [7] [8] [9] [10] [11] websites.

The configuration is:

- PARALUTION - version 0.4.0
- Host/OpenMP – the number of threads is equal to the number of real cores (no HT), gcc versions 4.4.6, 4.4.7 and 4.6.3
- CUDA – CUDA version 5.0
- OpenCL – NVIDIA OpenCL version 1.1 (comes with CUDA 5.0); AMD OpenCL version 1.2 (comes with AMD SDK 2.8.1.0)
- MIC/OpenMP – for the Xeon Phi, the number of threads for the OpenMP section is not set (the default configuration is used), icc version 13.1

[1] Intel E5620: http://ark.intel.com/de/products/47925/Intel-Xeon-Processor-E5620-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI
[2] Intel E5-2680: http://ark.intel.com/de/products/64583
[3] Intel Core i7 4770K: http://ark.intel.com/products/75123/
[4] Intel Core i7 620M (Notebook): http://ark.intel.com/products/43560
[5] Intel Xeon Phi 5110P: http://ark.intel.com/de/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core
[6] AMD FirePro W8000: http://www.amd.com/us/products/workstation/graphics/ati-firepro-3d/w8000/Pages/w8000.aspx#3
[7] NVIDIA GTX560 Ti: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-560ti/specifications
[8] NVIDIA GTX590: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-590/specifications
[9] NVIDIA GTX680: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680/specifications
[10] NVIDIA GTX TITAN: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications
[11] NVIDIA K20 and K20X: http://www.nvidia.com/object/tesla-servers.html

12.1.2 BLAS1 and SpMV

The vector-vector routines (BLAS1) are performed with size 4,000,000 – Figure 12.1.

- ScaleAdd is $x = \alpha x + y$, where $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$
- Dot product is $\alpha = \sum_{i=0}^{n} x_i y_i$, where $x, y \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$
- Reduce is $\alpha = \sum_{i=0}^{n} x_i$, where $x \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$
- L^2 Norm is $\alpha = \sqrt{\sum_{i=0}^{n} x_i^2}$, where $x \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}$

The backends for all vector-vector routines are CUDA for NVIDIA GPU; OpenCL for AMD GPU; OpenMP for Host/MIC.

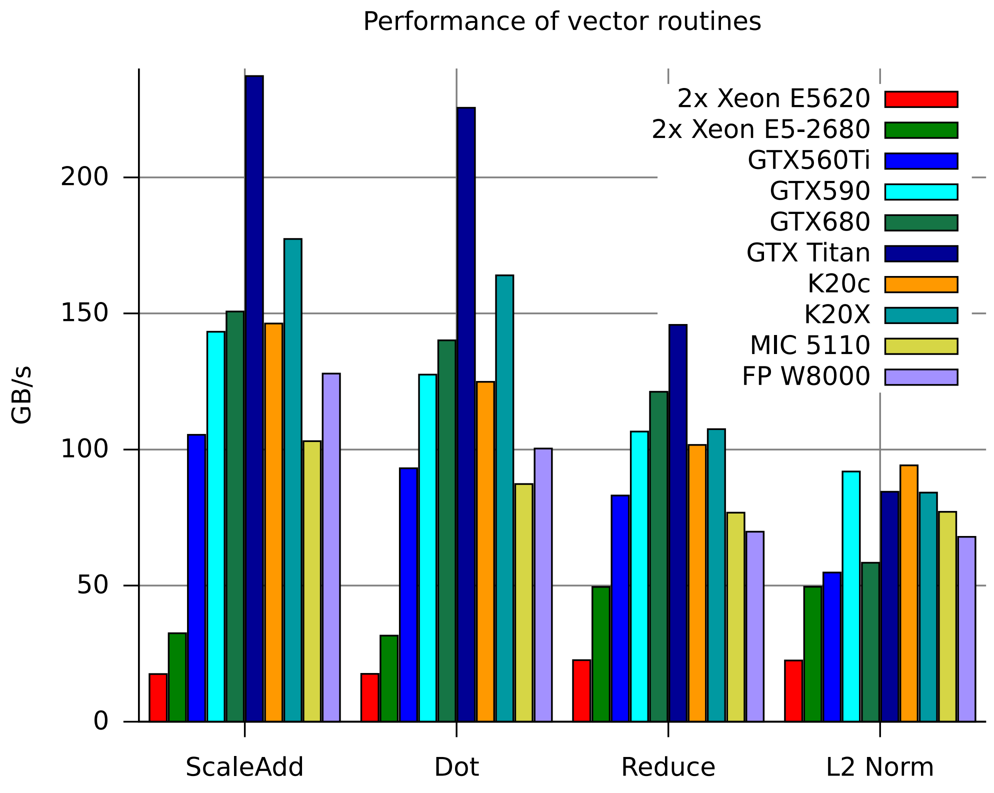


Figure 12.1: BLAS1 routines

The SpMV (sparse matrix-vector multiplication) is computed using a 2D Laplace (structured grid, finite difference) matrix on a grid with 2,000 × 2,000 = 4,000,000 DoFs – Figure 12.2.

All routines are executed 100 times and the average time (in ms resolution) is taken.

12.1.3 CG Solver

Furthermore, a non-preconditioned CG has been performed on a Laplace matrix resulting from a Finite Difference discretization of the unit square with 4.000.000 unknowns (as for the SpMV tests) on all listed architectures – Figure 12.3 and Figure 12.4. The right-hand side is set to one, initial guess to zero. A relative tolerance of 1e-6 based on L2 norm is used.

For the speed-up comparison of the CG method, we compare the best configurations for all setups – Figure 12.5.

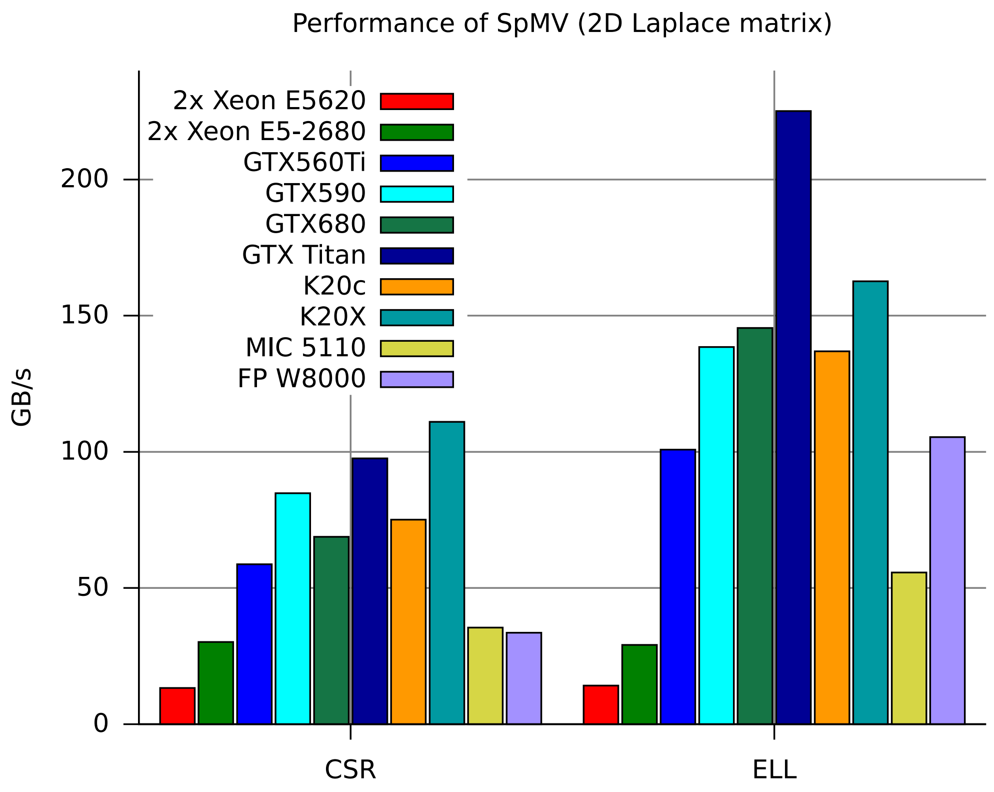


Figure 12.2: SpMV

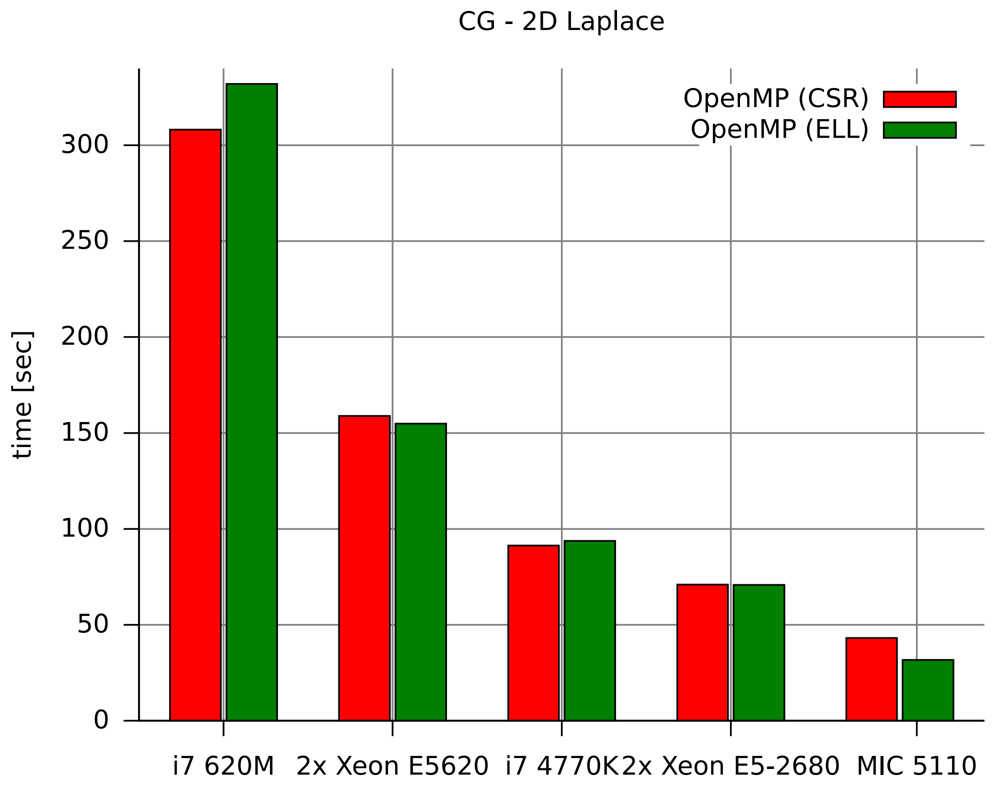


Figure 12.3: CG CPU Performance

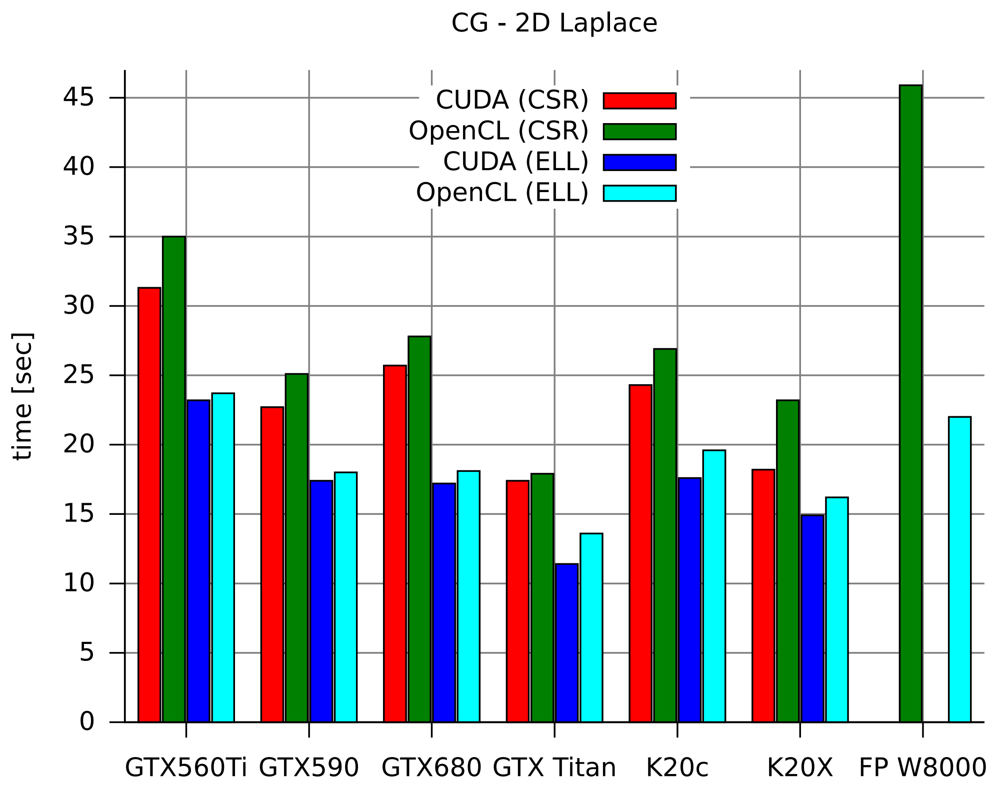


Figure 12.4: CG GPU Performance

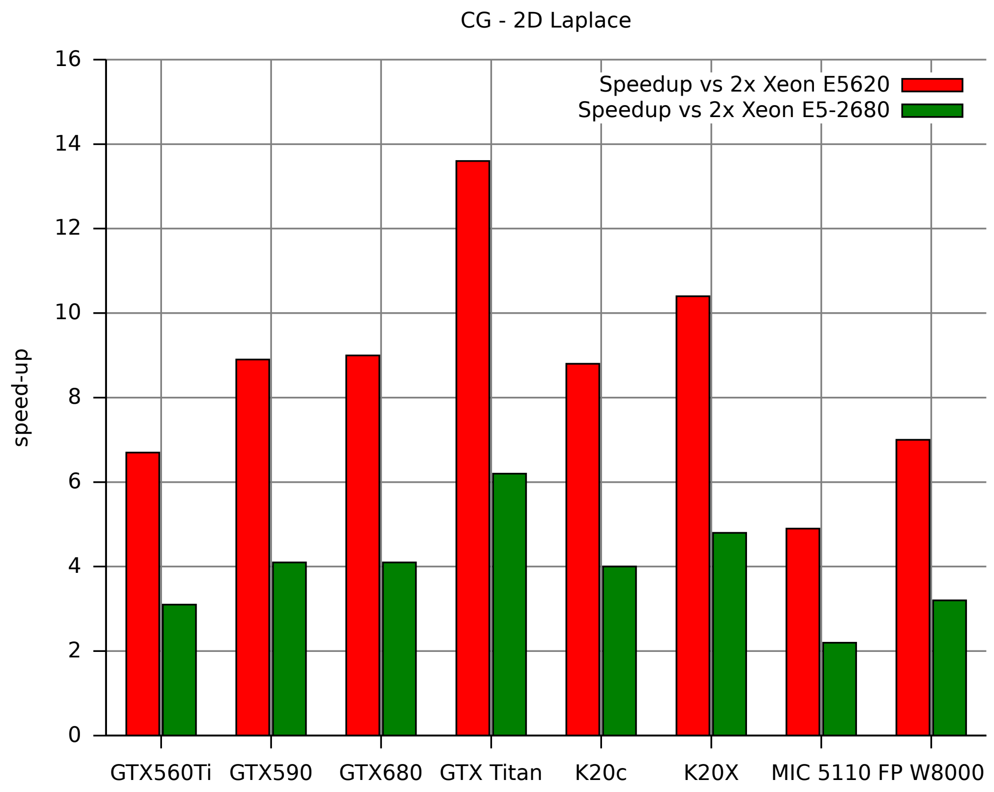


Figure 12.5: CG Speed-up

12.2 Multi-Node Benchmarks

For the multi-node benchmark, we use the following setup

- One MPI rank per GPU (1:1 mapping)
- Solver: CG with FSAI preconditioner
- Solver: CG with Pair-wise (global) AMG

Hardware/Compiler/Software Configuration

- PARALUTION 1.0.0 M
- CPU 2×8-core Xeon per node
- GPU 2×K40 per node
- Intel C++ compiler 15.0.0
- Intel MPI compiler 4.1.0.024
- CUDA 6.5

Benchmark Problems

- Pressure solver only
- 3D Cavity 27M unknowns
- PitzDaily 16M unknowns (LES)

In the following figures we show strong scalability (i.e. the size of the problem is fixed, the number of nodes/GPUs varies). The examples represent the solution of the pressure problem of standard benchmarks, where the data (matrix, right-hand-side, initial guess, normalization factor) is extracted from OpenFOAM [44]. The timings represent the solving phase of the iterative solver.

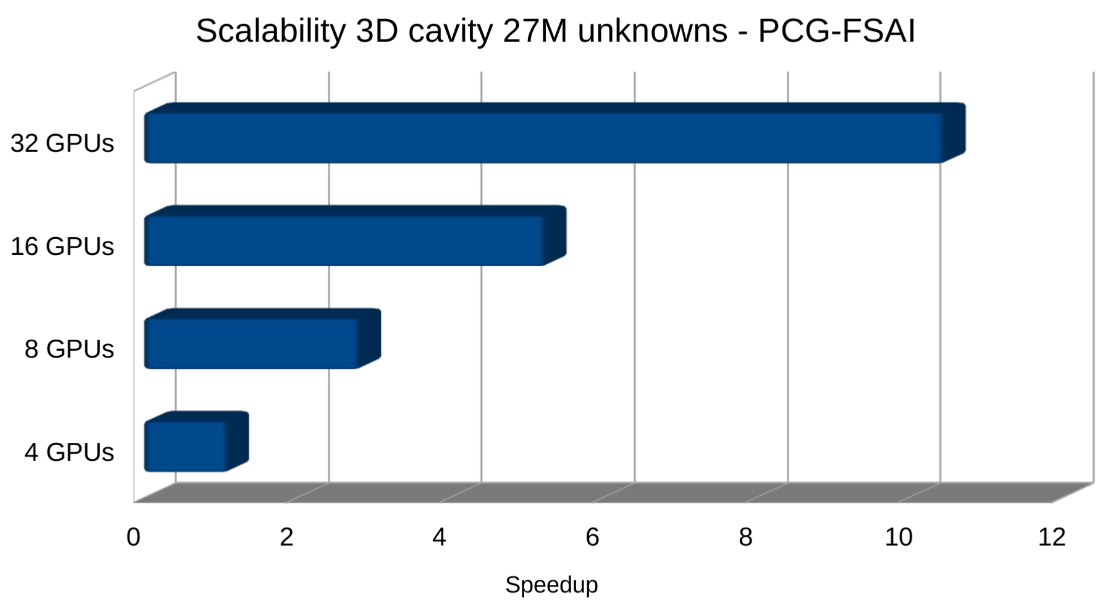


Figure 12.6: 3D Cavity – CG and FSAI, GPU Scalability

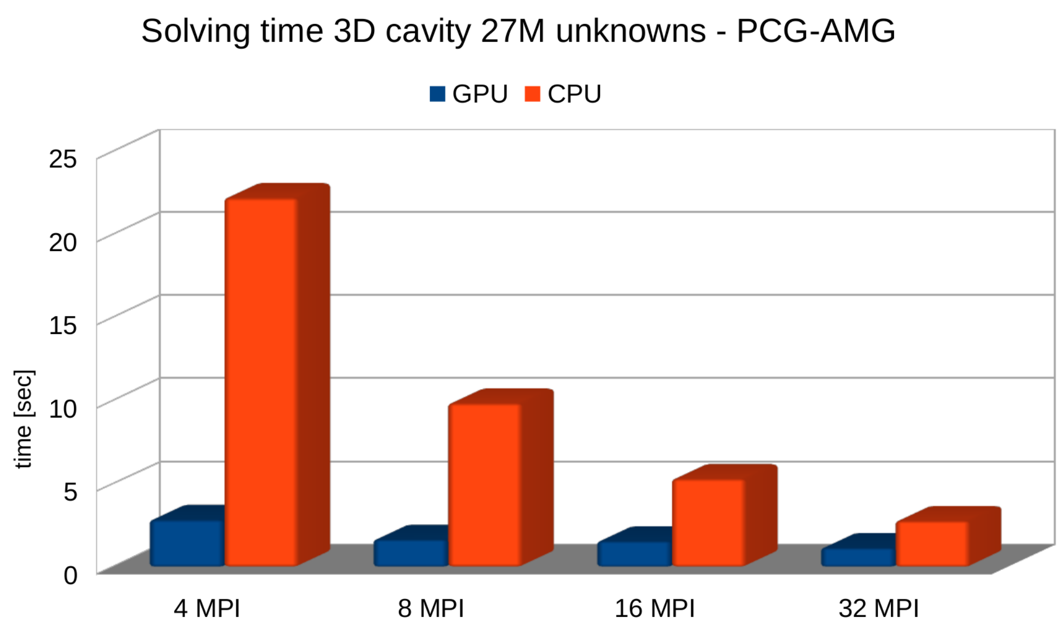


Figure 12.7: 3D Cavity – Pairwise AMG, CPU and GPU Performances

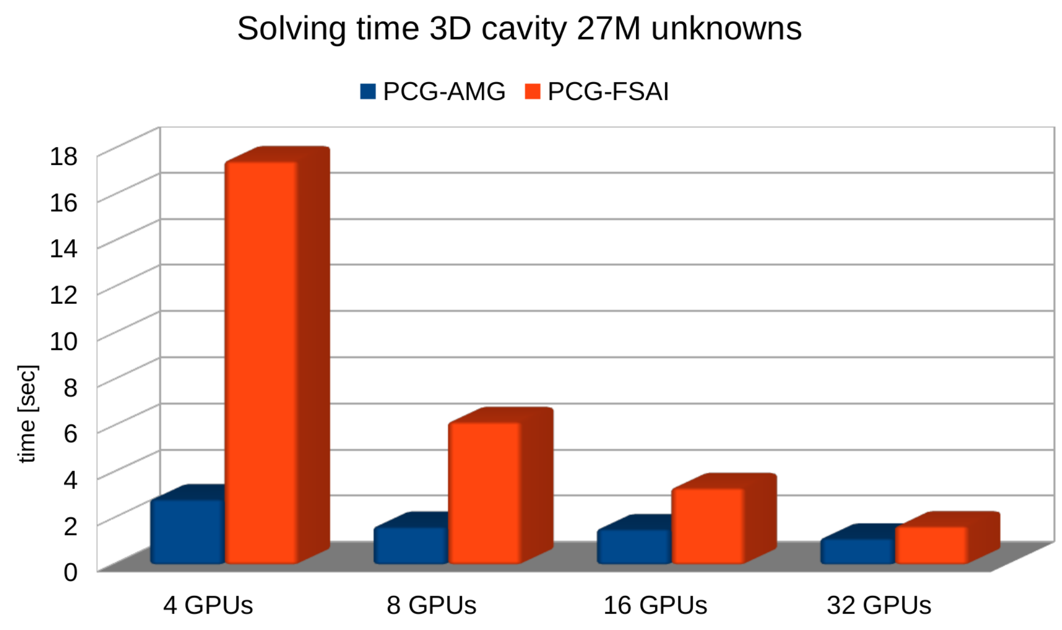


Figure 12.8: 3D Cavity – Pairwise AMG and CG-FSAI Performances

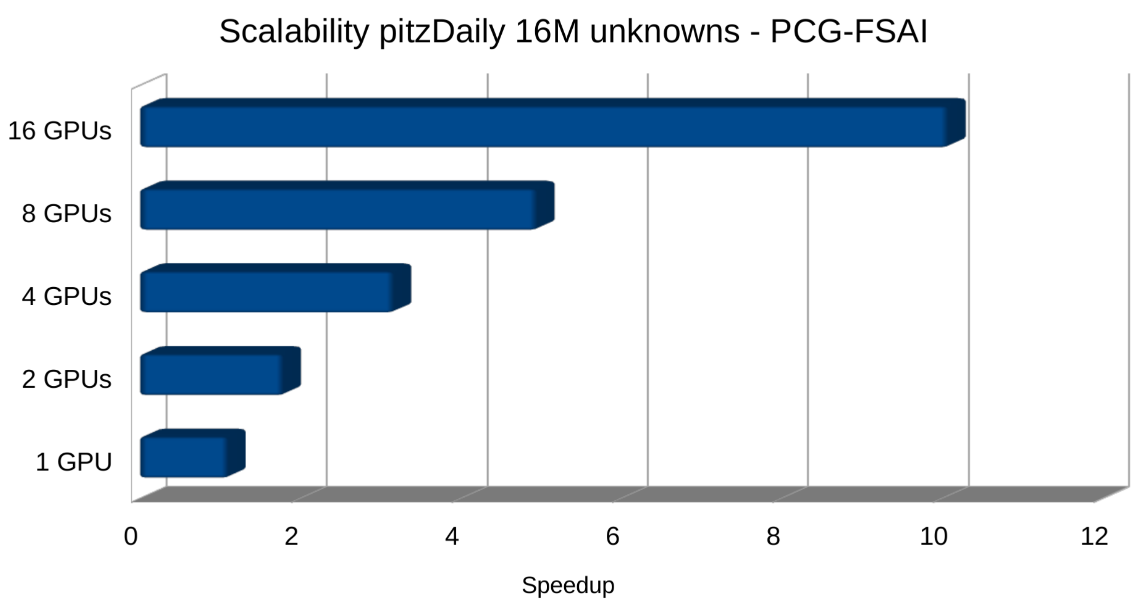


Figure 12.9: Pitzdaily – CG and FSAI, GPU Scalability

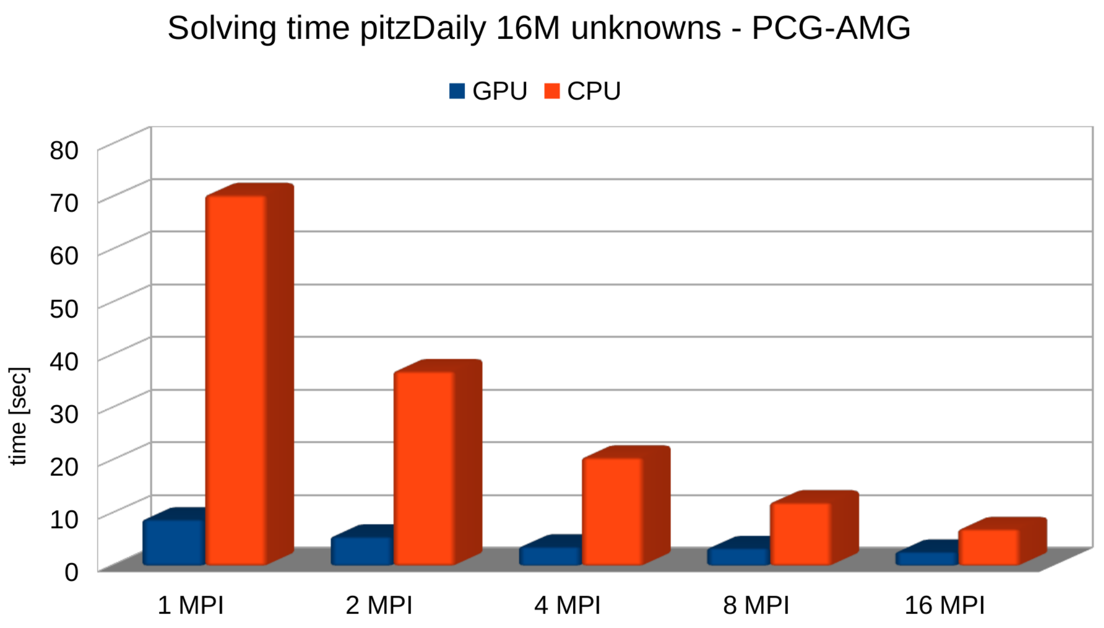


Figure 12.10: Pitzdaily – Pairwise AMG, CPU and GPU Performances

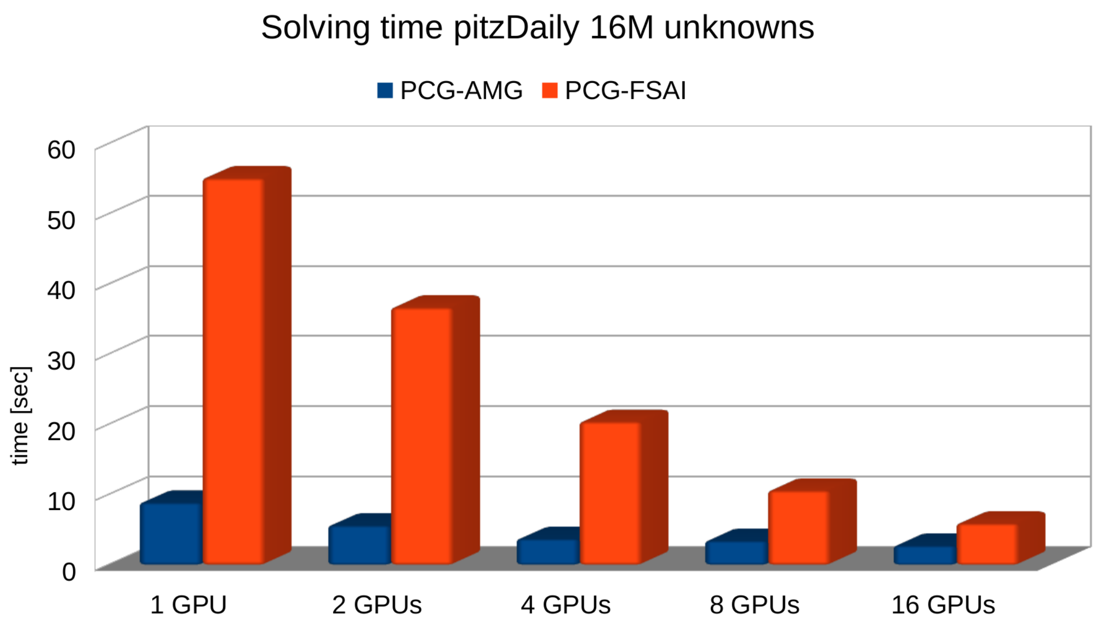


Figure 12.11: Pitzdaily – Pairwise AMG and CG-FSAI Performances

Chapter 13

Graph Analyzers

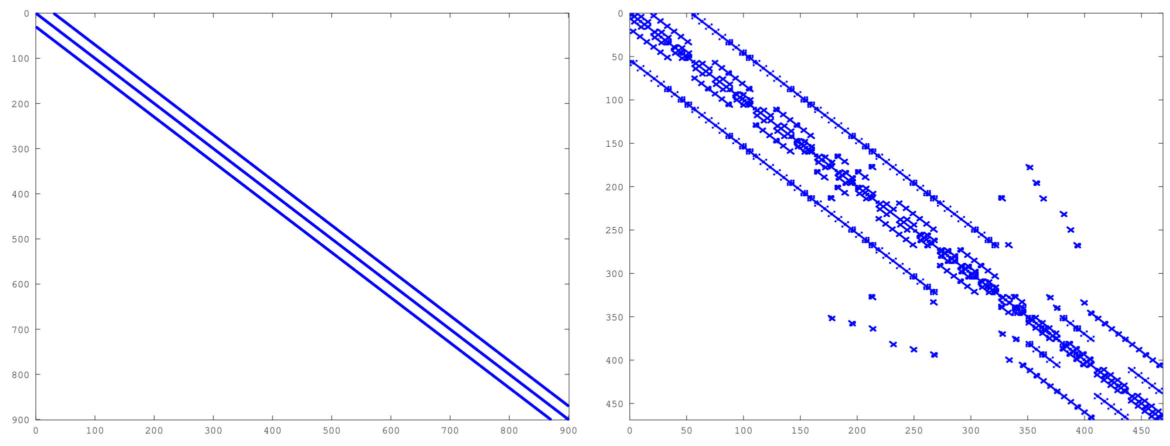


Figure 13.1: gr3030 and nos5 matrices, see [3] and [6]

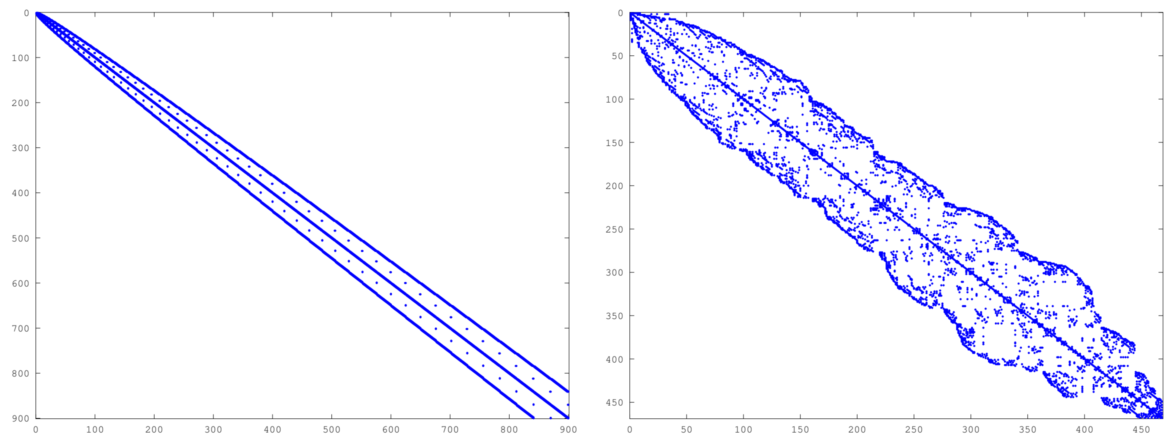


Figure 13.2: CMK permutation of gr3030 and nos5

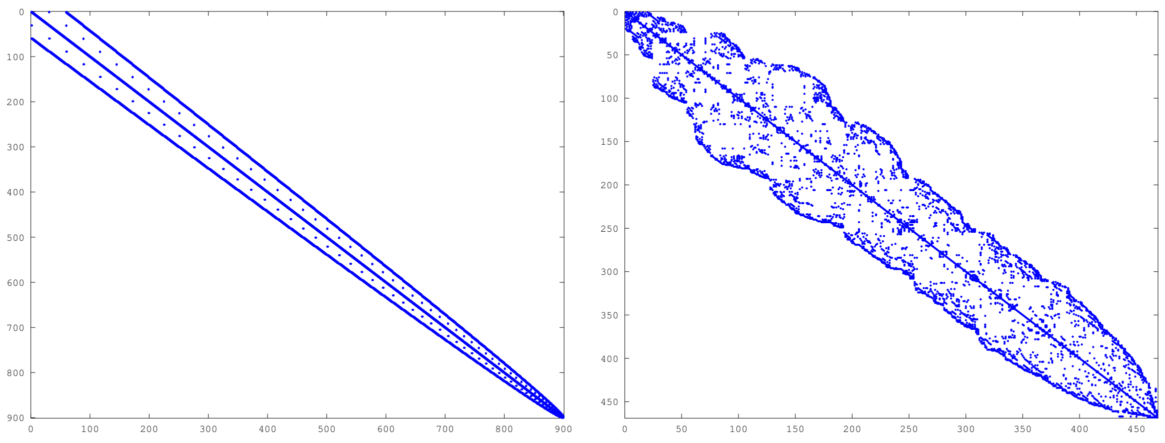


Figure 13.3: RCMK permutation of gr3030 and nos5

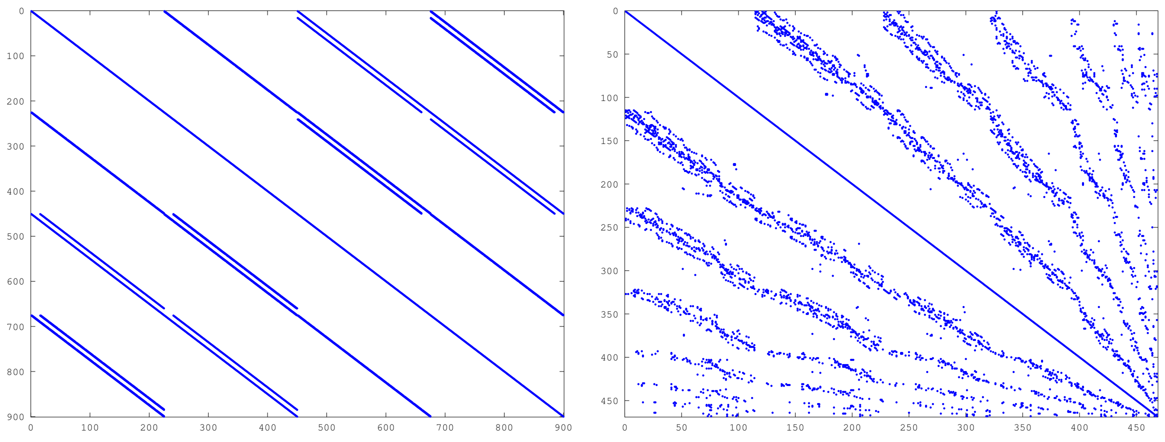


Figure 13.4: Multi-coloring permutation of gr3030 and nos5

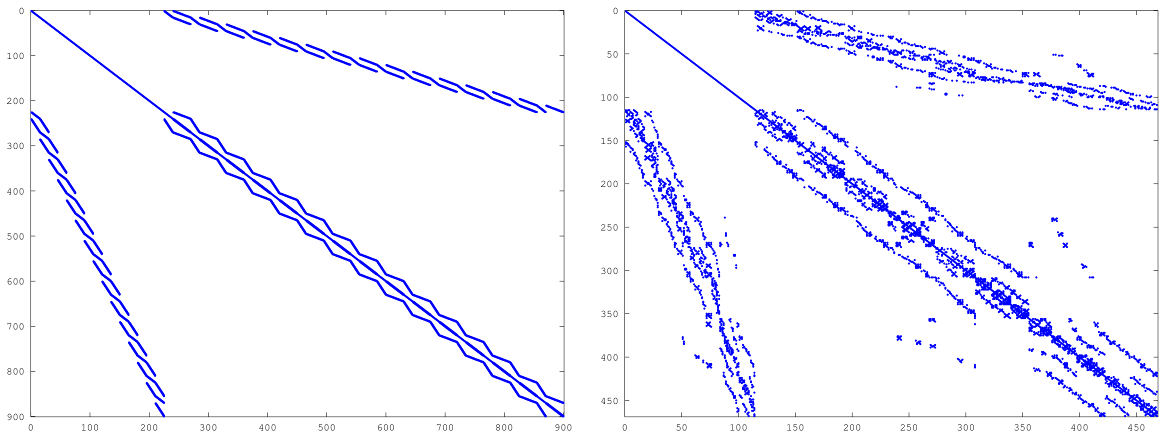


Figure 13.5: MIS permutation of gr3030 and nos5

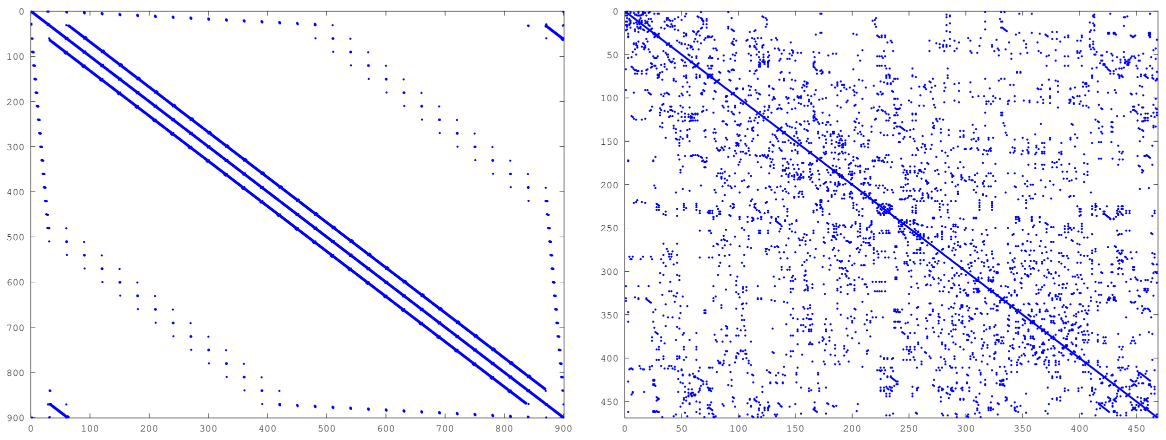


Figure 13.6: Connectivity ordering of gr3030 and nos5

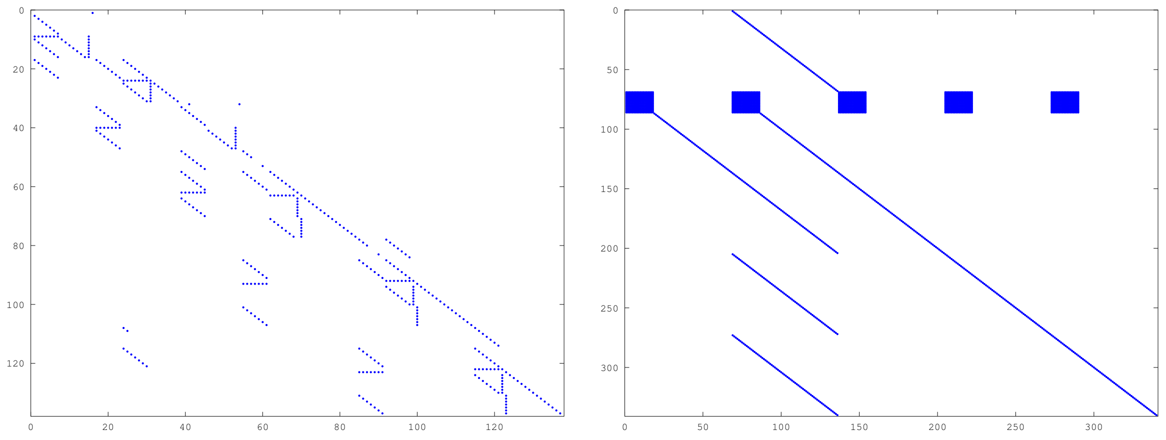


Figure 13.7: impcolc and tols340 matrices, see [5] and [7]

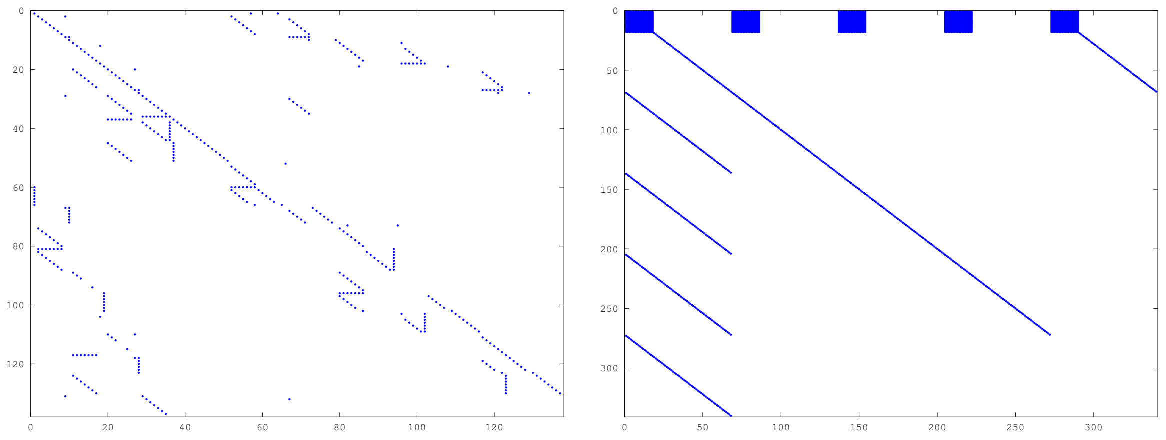


Figure 13.8: Zero-block permutation of impcolc and tols340

Chapter 14

Functionality Table

14.1 Backend Support

Backend name	Host	CUDA	OpenCL	MIC/Xeon Phi
Status	Stable	Stable	Stable	Beta
Target	Intel/AMD CPU	NVIDIA GPU	NVIDIA/AMD GPU	MIC/Xeon Phi
Parallelism	OpenMP	CUDA	OpenCL	OpenMP (offload mode)
Required Lib	None	CUBLAS, CUSPARSE[1]	None	None
Optional Lib	Intel MKL	None	None	None
Compiler	icc, gcc, VS	nvcc	NVIDIA/AMD ocl	icc
OS	Linux,Mac, Windows	Linux,Mac, Windows	Linux,Mac, Windows	Linux

14.2 Matrix-Vector Multiplication Formats

	Host	CUDA	OpenCL	MIC/Xeon Phi
CSR	Yes	Yes	Yes	Yes
COO	Yes[2]	Yes	Yes	Yes[2]
ELL	Yes	Yes	Yes	Yes
DIA	Yes	Yes	Yes	Yes
HYB	Yes	Yes	Yes	Yes
DENSE	Yes	Yes	Yes[3]	No
BCSR	No	No	No	No

All matrix conversions are performed via the CSR format (either as a source or a destitution – e.g. the user cannot directly convert a DIA matrix to an ELL matrix). The conversions can be computed on the host or on the CUDA backend (except back conversions (to a CSR matrix), DENSE, BCSR and MCSR conversions). All other backends can perform the conversions via the host.

	Host	CUDA	OpenCL	MIC/Xeon Phi
B	Yes	No	No	No
S	Yes	Yes	Yes	No
M	Yes	Yes	Yes	No

Table 14.1: Complex support

The base version supports only complex computation on the host.

[1]CUBLAS and CUSPARSE are delivered with the CUDA package
[2]Serial version
[3]Basic version

14.3 Local Matrices and Vectors

All matrix operations (except the sparse matrix-vector multiplication SpMV) require a CSR matrix. Note that if the input matrix is not a CSR matrix, an internal conversion will be performed to the CSR format followed by a back conversion to the current matrix format after the operation. In this case, a warning message on level 2 will be printed.

	Host	CUDA	OpenCL	MIC/Xeon Phi
Local Matrices				
I/O file	Yes	via Host	via Host	via Host
Direct memory access	Yes	Yes	Yes	Yes
Extract sub matrix	Yes	Yes	Yes	via Host
Extract diagonal	Yes	Yes	Yes	via Host
Extract inv diagonal	Yes	Yes	Yes	via Host
Assemble	Yes	via Host	via Host	via Host
Permutation	Yes	Yes	via Host	via Host
Graph Analyzer	Yes	via Host	via Host	via Host
Transpose	Yes	Yes	via Host	via Host
Sparse Mat-Mat Mult	Yes	Yes	via Host	via Host
Local Vectors				
I/O file	Yes	via Host	via Host	via Host
Direct memory access	Yes	Yes	Yes	Yes
Vector updates	Yes	Yes	Yes	Yes
Dot product	Yes	Yes	Yes	Yes
Dot product (conj/complex)	Yes	Yes	Yes	No
Sub copy	Yes	Yes	Yes	Yes
Point-wise mult	Yes	Yes	Yes	Yes
Scaling	Yes	Yes	Yes	Yes
Permutation	Yes	Yes	Yes	Yes

14.4 Global Matrices and Vectors

All Local functions (for Vectors or Matrices) can be applied on the interior of each Global Matrix.

	Host	CUDA	OpenCL	MIC/Xeon Phi
Global Matrices				
I/O file	Yes	via Host	via Host	via Host
Direct memory access	Yes	Yes	Yes	Yes
Global Vectors				
I/O file	Yes	via Host	via Host	via Host
Direct memory access	Yes	Yes	Yes	Yes
Vector updates	Yes	Yes	Yes	Yes
Dot product	Yes	Yes	Yes	Yes
Dot product (conj/complex)	Yes	Yes	Yes	No
Point-wise mult	Yes	Yes	Yes	Yes
Scaling	Yes	Yes	Yes	Yes

Note The OpenCL backend supports only GPUs in the multi-node version.

14.5 Local Solvers and Preconditioners

	Host	CUDA	OpenCL	MIC/Xeon Phi
Local Solvers				
CG – Building	Yes	Yes	Yes	Yes
CG – Solving	Yes	Yes	Yes	Yes
FCG – Building	Yes	Yes	Yes	Yes
FCG – Solving	Yes	Yes	Yes	Yes
CR – Building	Yes	Yes	Yes	Yes
CR – Solving	Yes	Yes	Yes	Yes
BiCGStab – Building	Yes	Yes	Yes	Yes
BiCGStab – Solving	Yes	Yes	Yes	Yes
BiCGStab(l) – Building	Yes	Yes	Yes	Yes
BiCGStab(l) – Solving	Yes	Yes	Yes	Yes
QMRCGStab – Building	Yes	Yes	Yes	Yes
QMRCGStab – Solving	Yes	Yes	Yes	Yes
GMRES – Building	Yes	Yes	Yes	Yes
GMRES – Solving	Yes	Yes	Yes	Yes
FGMRES – Building	Yes	Yes	Yes	Yes
FGMRES – Solving	Yes	Yes	Yes	Yes
Chebyshev – Building	Yes	Yes	Yes	Yes
Chebyshev – Solving	Yes	Yes	Yes	Yes
DPCG – Building	Yes	Yes	via Host	via Host
DPCG – Solving	Yes	Yes	via Host	via Host
Mixed-Precision DC – Building	Yes	Yes	Yes	Yes
Mixed-Precision DC – Solving	Yes	Yes	Yes	Yes
Fixed-Point Iteration – Building	Yes	Yes	Yes	Yes
Fixed-Point Iteration – Solving	Yes	Yes	Yes	Yes
AMG(Plain Aggregation) – Building	Yes	via Host	via Host	via Host
AMG(Plain Aggregation) – Solving	Yes	Yes	Yes	Yes
AMG(Smoothed Aggregation) – Building	Yes	via Host	via Host	via Host
AMG(Smoothed Aggregation) – Solving	Yes	Yes	Yes	Yes
AMG(RS) – Building	Yes	via Host	via Host	via Host
AMG(RS) – Solving	Yes	Yes	Yes	Yes
AMG(Pair-wise) – Building	Yes	via Host	via Host	via Host
AMG(Pair-wise) – Solving	Yes	Yes	Yes	Yes
Local Direct Solvers (DENSE only)				
LU – Building	Yes	via Host	via Host	via Host
LU – Solving	Yes	via Host	via Host	via Host
QR – Building	Yes	via Host	via Host	via Host
QR – Solving	Yes	via Host	via Host	via Host
Inversion – Building	Yes	via Host	via Host	via Host
Inversion – Solving	Yes	Yes	via Host	via Host

Note The building phase of the iterative solver depends also on the selected preconditioner.

	Host	CUDA	OpenCL	MIC/Xeon Phi
Local Preconditioners				
Jacobi – Building	Yes	Yes	Yes	Yes
Jacobi – Solving	Yes	Yes	Yes	Yes
MC-ILU(0,1) – Building	Yes	Yes	via Host	via Host
MC-ILU(0,1) – Solving	Yes	Yes	Yes	Yes
MC-ILU($>0,>1$) – Building	Yes	via Host	via Host	via Host
MC-ILU($>0,>1$) – Solving	Yes	Yes	Yes	Yes
ME-(I)LU – Building	Yes	Yes	via Host	via Host
ME-(I)LU – Solving	Yes	Yes	Yes	Yes
ILU(0) – Building	Yes	Yes	via Host	via Host
ILU(0) – Solving	Yes	Yes	via Host	via Host
ILU(>0) – Building	Yes	via Host	via Host	via Host
ILU(>0) – Solving	Yes	Yes	via Host	via Host
ILUT – Building	Yes	via Host	via Host	via Host
ILUT – Solving	Yes	Yes	via Host	via Host
IC(0) – Building	Yes	Yes	via Host	via Host
IC(0) – Solving	Yes	Yes	via Host	via Host
FSAI – Building	Yes	via Host	via Host	via Host
FSAI – Solving	Yes	Yes	Yes	Yes
SPAI – Building	Yes	via Host	via Host	via Host
SPAI – Solving	Yes	Yes	Yes	Yes
Chebyshev – Building	Yes	Yes	via Host	via Host
Chebyshev – Solving	Yes	Yes	Yes	Yes
MC-GS/SGS – Building	Yes	Yes	via Host	via Host
MC-GS/SGS – Solving	Yes	Yes	Yes	Yes
GS/SGS – Building	Yes	Yes	via Host	via Host
GS/SGS – Solving	Yes	Yes	via Host	via Host
AS/RAS – Building	Yes	Yes	Yes	via Host
AS/RAS – Solving	Yes	Yes	Yes	Yes
Block – Building	Yes	Yes	Yes	via Host
Block – Solving	Yes	Yes	Yes	Yes
SaddlePoint – Building	Yes	Yes	via Host	via Host
SaddlePoint – Solving	Yes	Yes	Yes	Yes

14.6 Global Solvers and Preconditioners

	Host	CUDA	OpenCL	MIC/Xeon Phi
Global Solvers				
CG – Building	Yes	Yes	Yes	Yes
CG – Solving	Yes	Yes	Yes	Yes
FCG – Building	Yes	Yes	Yes	Yes
FCG – Solving	Yes	Yes	Yes	Yes
CR – Building	Yes	Yes	Yes	Yes
CR – Solving	Yes	Yes	Yes	Yes
BiCGStab – Building	Yes	Yes	Yes	Yes
BiCGStab – Solving	Yes	Yes	Yes	Yes
BiCGStab(l) – Building	Yes	Yes	Yes	Yes
BiCGStab(l) – Solving	Yes	Yes	Yes	Yes
QMRCGStab – Building	Yes	Yes	Yes	Yes
QMRCGStab – Solving	Yes	Yes	Yes	Yes
GMRES – Building	Yes	Yes	Yes	Yes
GMRES – Solving	Yes	Yes	Yes	Yes
FGMRES – Building	Yes	Yes	Yes	Yes
FGMRES – Solving	Yes	Yes	Yes	Yes
Mixed-Precision DC – Building	Yes	Yes	Yes	Yes
Mixed-Precision DC – Solving	Yes	Yes	Yes	Yes
Fixed-Point Iteration – Building	Yes	Yes	Yes	Yes
Fixed-Point Iteration – Solving	Yes	Yes	Yes	Yes
AMG(Pair-wise) – Building	Yes	via Host	via Host	No
AMG(Pair-wise) – Solving	Yes	Yes	Yes	No

Note The building phase of the iterative solver depends also on the selected preconditioner.

All local preconditioner can be used on a Global level by Block-Jacobi type of preconditioner, where the local preconditioner will be applied on each node/process interior matrix.

Chapter 15

Code Examples

15.1 Preconditioned CG

```cpp
#include <iostream>
#include <cstdlib>

#include <paralution.hpp>

using namespace paralution;

int main(int argc, char* argv[]) {

  if (argc == 1) {
    std::cerr << argv[0] << " <matrix> [Num threads]" << std::endl;
    exit(1);
  }

  init_paralution();

  if (argc > 2) {
    set_omp_threads_paralution(atoi(argv[2]));
  }

  info_paralution();

  LocalVector<double> x;
  LocalVector<double> rhs;

  LocalMatrix<double> mat;

  mat.ReadFileMTX(std::string(argv[1]));
  mat.MoveToAccelerator();
  x.MoveToAccelerator();
  rhs.MoveToAccelerator();

  x.Allocate("x", mat.get_nrow());
  rhs.Allocate("rhs", mat.get_nrow());

  // Linear Solver
  CG<LocalMatrix<double >, LocalVector<double >, double > ls;

  // Preconditioner
  MultiColoredILU<LocalMatrix<double >, LocalVector<double >, double > p;

  rhs.Ones();
  x.Zeros();

  ls.SetOperator(mat);
  ls.SetPreconditioner(p);

  ls.Build();

//  ls.Verbose(2);

  mat.info();

  double tick, tack;
  tick = paralution_time();

  ls.Solve(rhs, &x);

  tack = paralution_time();
  std::cout << "Solver execution:" << (tack-tick)/1000000 << " sec" << std::endl;

  stop_paralution();

  return 0;
}
```

"Preconditioned CG solver"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 2
Host thread affinity policy - thread mapping on every core
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: None
OpenMP threads:2
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; reading...
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; done
LocalMatrix name=/home/dimitar/matrices/small/sym/gr_30_30.mtx; rows=900;
    cols=900; nnz=7744; prec=64bit; asm=no; format=CSR; host
    backend={CPU(OpenMP)}; accelerator backend={None}; current=CPU(OpenMP)
PCG solver starts, with preconditioner:
Multicolored ILU preconditioner (power(q)-pattern method), ILU(0,1)
number of colors = 4; ILU nnz = 7744
IterationControl criteria: abs tol=1e-15; rel tol=1e-06; div tol=1e+08; max
    iter=1000000
IterationControl initial residual = 30
IterationControl RELATIVE criteria has been reached: res norm=2.532e-05; rel
    val=8.43999e-07; iter=24
PCG ends
Solver execution:0.001982 sec
```

"Output of a preconditioned CG test with matrix gr_30_30.mtx"

15.2 Multi-Elimination ILU Preconditioner with CG

```
#include <iostream>
#include <cstdlib>

#include <paralution.hpp>

using namespace paralution;

int main(int argc, char* argv[]) {

  if (argc == 1) {
    std::cerr << argv[0] << " <matrix> [Num threads]" << std::endl;
    exit(1);
  }

  init_paralution();

  if (argc > 2) {
    set_omp_threads_paralution(atoi(argv[2]));
  }

  info_paralution();

  LocalVector<double> x;
  LocalVector<double> rhs;
  LocalMatrix<double> mat;

  mat.ReadFileMTX(std::string(argv[1]));

  rhs.MoveToAccelerator();
  x.MoveToAccelerator();
  mat.MoveToAccelerator();

  x.Allocate("x", mat.get_nrow());
  rhs.Allocate("rhs", mat.get_nrow());

  x.Zeros();
  rhs.Ones();

  double tick, tack;

  // Solver
  CG<LocalMatrix<double>, LocalVector<double>, double > cg;

  // Preconditioner (main)
  MultiElimination<LocalMatrix<double>, LocalVector<double>, double > p;

  // Last block-preconditioner
  MultiColoredILU<LocalMatrix<double>, LocalVector<double>, double > mcilu_p;

  mcilu_p.Set(0);
  p.Set(mcilu_p, 2, 0.4);

  cg.SetOperator(mat);
  cg.SetPreconditioner(p);

  cg.Build();

  mat.info();
  tick = paralution_time();

  cg.Solve(rhs, &x);

  tack = paralution_time();

  std::cout << "Solver execution:" << (tack-tick)/1000000 << " sec" << std::endl;

  stop_paralution();

  return 0;
}
```

"Multi-Elimination ILU Preconditioner with CG"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 2
Host thread affinity policy - thread mapping on every core
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: None
OpenMP threads:2
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; reading...
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; done
LocalMatrix name=/home/dimitar/matrices/small/sym/gr_30_30.mtx; rows=900;
    cols=900; nnz=7744; prec=64bit; asm=no; format=CSR; host
    backend={CPU(OpenMP)}; accelerator backend={None}; current=CPU(OpenMP)
PCG solver starts, with preconditioner:
MultiElimination (I)LU preconditioner with 2 levels; diagonal size = 225 ; drop
    tol = 0.4 ; last-block size = 675 ; last-block nnz = 4097 ; last-block
    solver:
MultiElimination (I)LU preconditioner with 1 levels; diagonal size = 225 ; drop
    tol = 0.4 ; last-block size = 450 ; last-block nnz = 1320 ; last-block
    solver:
Multicolored ILU preconditioner (power(q)-pattern method), ILU(0,1)
number of colors = 2; ILU nnz = 1320
IterationControl criteria: abs tol=1e-15; rel tol=1e-06; div tol=1e+08; max
    iter=1000000
IterationControl initial residual = 30
IterationControl RELATIVE criteria has been reached: res norm=2.532e-05; rel
    val=8.43999e-07; iter=24
PCG ends
Solver execution:0.001641 sec
```

"Output of a preconditioned CG (ME-ILU) test with matrix gr_30_30.mtx"

15.3 Gershgorin Circles+Power Method+Chebyshev Iteration+PCG with Chebyshev Polynomial

```
#include <iostream>
#include <cstdlib>

#include <paralution.hpp>

using namespace paralution;

int main(int argc, char* argv[]) {

  if (argc == 1) {
    std::cerr << argv[0] << " <matrix> [Num threads]" << std::endl;
    exit(1);
  }

  init_paralution();

  if (argc > 2) {
    set_omp_threads_paralution(atoi(argv[2]));
  }

  info_paralution();

  LocalVector<double> b, b_old, *b_k, *b_k1, *b_tmp;
  LocalMatrix<double> mat;

  mat.ReadFileMTX(std::string(argv[1]));

  // Gershgorin spectrum approximation
  double glambda_min, glambda_max;

  // Power method spectrum approximation
  double plambda_min, plambda_max;

  // Maximum number of iteration for the power method
  int iter_max = 10000;

  double tick, tack;

  // Gershgorin approximation of the eigenvalues
  mat.Gershgorin(glambda_min, glambda_max);
  std::cout << "Gershgorin : Lambda min = " << glambda_min
            << "; Lambda max = " << glambda_max << std::endl;


  mat.MoveToAccelerator();
  b.MoveToAccelerator();
  b_old.MoveToAccelerator();


  b.Allocate("b_k+1", mat.get_nrow());
  b_k1 = &b;

  b_old.Allocate("b_k", mat.get_nrow());
  b_k = &b_old;

  b_k->Ones();

  mat.info();

  tick = paralution_time();

  // compute lambda max
  for (int i=0; i<=iter_max; ++i) {

    mat.Apply(*b_k, b_k1);

    //     std::cout << b_k1->Dot(*b_k) << std::endl;
    b_k1->Scale(double(1.0)/b_k1->Norm());

    b_tmp = b_k1;
    b_k1 = b_k;
    b_k = b_tmp;

  }

  // get lambda max (Rayleigh quotient)
  mat.Apply(*b_k, b_k1);
  plambda_max = b_k1->Dot(*b_k) ;

  tack = paralution_time();
  std::cout << "Power method (lambda max) execution:" << (tack-tick)/1000000 <<
      " sec" << std::endl;

  mat.AddScalarDiagonal(double(-1.0)*plambda_max);


  b_k->Ones();

  tick = paralution_time();

  // compute lambda min
  for (int i=0; i<=iter_max; ++i) {

    mat.Apply(*b_k, b_k1);

    //     std::cout << b_k1->Dot(*b_k) + plambda_max << std::endl;
    b_k1->Scale(double(1.0)/b_k1->Norm());

    b_tmp = b_k1;
    b_k1 = b_k;
    b_k = b_tmp;

  }

  // get lambda min (Rayleigh quotient)
  mat.Apply(*b_k, b_k1);
  plambda_min = (b_k1->Dot(*b_k) + plambda_max);

  // back to the original matrix
  mat.AddScalarDiagonal(plambda_max);

  tack = paralution_time();
  std::cout << "Power method (lambda min) execution:" << (tack-tick)/1000000 <<
      " sec" << std::endl;


  std::cout << "Power method Lambda min = " << plambda_min
            << "; Lambda max = " << plambda_max
            << "; iter=2x" << iter_max << std::endl;

  LocalVector<double> x;
  LocalVector<double> rhs;

  x.CloneBackend(mat);
  rhs.CloneBackend(mat);

  x.Allocate("x", mat.get_nrow());
  rhs.Allocate("rhs", mat.get_nrow());

  // Chebyshev iteration
  Chebyshev<LocalMatrix<double>, LocalVector<double>, double > ls;

  rhs.Ones();
  x.Zeros();

  ls.SetOperator(mat);

  ls.Set(plambda_min, plambda_max);

  ls.Build();

  tick = paralution_time();

  ls.Solve(rhs, &x);

  tack = paralution_time();
  std::cout << "Solver execution:" << (tack-tick)/1000000 << " sec" << std::endl;

  // PCG + Chebyshev polynomial
  CG<LocalMatrix<double>, LocalVector<double>, double > cg;
  AIChebyshev<LocalMatrix<double>, LocalVector<double>, double > p;

  // damping factor
  plambda_min = plambda_max / 7;
  p.Set(3, plambda_min, plambda_max);
  rhs.Ones();
  x.Zeros();

  cg.SetOperator(mat);
  cg.SetPreconditioner(p);

  cg.Build();

  tick = paralution_time();

  cg.Solve(rhs, &x);

  tack = paralution_time();
  std::cout << "Solver execution:" << (tack-tick)/1000000 << " sec" << std::endl;

  stop_paralution();

  return 0;
}
```

"Gershgorin circles + Power method + Chebyshev iteration + PCG with Chebyshev polynomial"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 2
```

```
Host thread affinity policy - thread mapping on every core
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: None
OpenMP threads:2
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; reading...
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; done
Gershgorin : Lambda min = 0; Lambda max = 16
LocalMatrix name=/home/dimitar/matrices/small/sym/gr_30_30.mtx; rows=900;
    cols=900; nnz=7744; prec=64bit; asm=no; format=CSR; host
    backend={CPU(OpenMP)}; accelerator backend={None}; current=CPU(OpenMP)
Power method (lambda max) execution:0.338008 sec
Power method (lambda min) execution:0.310945 sec
Power method Lambda min = 0.0614628; Lambda max = 11.9591; iter=2x10000
Chebyshev (non-precond) linear solver starts
IterationControl criteria: abs tol=1e-15; rel tol=1e-06; div tol=1e+08; max
    iter=1000000
IterationControl initial residual = 30
IterationControl RELATIVE criteria has been reached: res norm=2.98708e-05; rel
    val=9.95692e-07; iter=919
Chebyshev (non-precond) ends
Solver execution:0.028368 sec
PCG solver starts, with preconditioner:
Approximate Inverse Chebyshev(3) preconditioner
AI matrix nnz = 62500
IterationControl criteria: abs tol=1e-15; rel tol=1e-06; div tol=1e+08; max
    iter=1000000
IterationControl initial residual = 30
IterationControl RELATIVE criteria has been reached: res norm=2.55965e-05; rel
    val=8.53216e-07; iter=17
PCG ends
Solver execution:0.002306 sec
```

"Output of the program with matrix gr_30_30.mtx"

15.4 Mixed-precision PCG Solver

```
#include <iostream>
#include <cstdlib>

#include <paralution.hpp>

using namespace paralution;

int main(int argc, char* argv[]) {

  if (argc == 1) {
    std::cerr << argv[0] << " <matrix> [Num threads]" << std::endl;
    exit(1);
  }

  init_paralution();

  if (argc > 2) {
    set_omp_threads_paralution(atoi(argv[2]));
  }

  info_paralution();

  LocalVector<double> x;
  LocalVector<double> rhs;

  LocalMatrix<double> mat;

  // read from file
  mat.ReadFileMTX(std::string(argv[1]));

  x.Allocate("x", mat.get_nrow());
  rhs.Allocate("rhs", mat.get_nrow());

  MixedPrecisionDC<LocalMatrix<double>, LocalVector<double>, double,
                   LocalMatrix<float>, LocalVector<float>, float> mp;

  CG<LocalMatrix<float>, LocalVector<float>, float> cg;
  MultiColoredILU<LocalMatrix<float>, LocalVector<float>, float> p;

  double tick, tack;

  rhs.Ones();
  x.Zeros();

  // setup a lower tol for the inner solver
  cg.SetPreconditioner(p);
  cg.Init(1e-5, 1e-2, 1e+20,
          100000);

  // setup the mixed-precision DC
  mp.SetOperator(mat);
  mp.Set(cg);

  mp.Build();

  tick = paralution_time();

  mp.Solve(rhs, &x);

  tack = paralution_time();

  std::cout << "Solver execution:" << (tack-tick)/1000000 << " sec" << std::endl;

  stop_paralution();

  return 0;
}
```

"Mixed-precision PCG solver"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 2
Host thread affinity policy - thread mapping on every core
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: None
OpenMP threads:2
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; reading...
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; done
MixedPrecisionDC [64bit-32bit] solver starts, with solver:
PCG solver, with preconditioner:
Multicolored ILU preconditioner (power(q)-pattern method), ILU(0,1)
number of colors = 4; ILU nnz = 7744
IterationControl criteria: abs tol=1e-15; rel tol=1e-06; div tol=1e+08; max
    iter=1000000
IterationControl initial residual = 30
MixedPrecisionDC: starting the internal solver [32bit]
PCG solver starts, with preconditioner:
Multicolored ILU preconditioner (power(q)-pattern method), ILU(0,1)
number of colors = 4; ILU nnz = 7744
IterationControl criteria: abs tol=1e-05; rel tol=0.01; div tol=1e+20; max
    iter=100000
IterationControl initial residual = 30
IterationControl RELATIVE criteria has been reached: res norm=0.227825; rel
    val=0.00759417; iter=11
PCG ends
MixedPrecisionDC: defect correcting on the host [64bit]
MixedPrecisionDC: starting the internal solver [32bit]
PCG solver starts, with preconditioner:
Multicolored ILU preconditioner (power(q)-pattern method), ILU(0,1)
number of colors = 4; ILU nnz = 7744
IterationControl criteria: abs tol=1e-05; rel tol=0.01; div tol=1e+20; max
    iter=100000
IterationControl initial residual = 0.227811
IterationControl RELATIVE criteria has been reached: res norm=0.00154375; rel
    val=0.00677646; iter=8
PCG ends
MixedPrecisionDC: defect correcting on the host [64bit]
MixedPrecisionDC: starting the internal solver [32bit]
PCG solver starts, with preconditioner:
Multicolored ILU preconditioner (power(q)-pattern method), ILU(0,1)
number of colors = 4; ILU nnz = 7744
IterationControl criteria: abs tol=1e-05; rel tol=0.01; div tol=1e+20; max
    iter=100000
IterationControl initial residual = 0.00154375
IterationControl RELATIVE criteria has been reached: res norm=1.46245e-05; rel
    val=0.0094733; iter=14
PCG ends
MixedPrecisionDC: defect correcting on the host [64bit]
IterationControl RELATIVE criteria has been reached: res norm=1.46244e-05; rel
```

```
    val=4.87482e-07; iter=4
MixedPrecisionDC ends
Solver execution:0.002661 sec
```

"Output of the program with matrix gr_30_30.mtx"

15.5 PCG Solver with AMG

```
#include <iostream>
#include <cstdlib>

#include <paralution.hpp>

using namespace paralution;

int main(int argc, char* argv[]) {

  double tick, tack, start, end;

  start = paralution_time();

  if (argc == 1) {
    std::cerr << argv[0] << " <matrix> [Num threads]" << std::endl;
    exit(1);
  }

  init_paralution();

  if (argc > 2)
    set_omp_threads_paralution(atoi(argv[2]));

  info_paralution();

  LocalVector<double> x;
  LocalVector<double> rhs;

  LocalMatrix<double> mat;

  mat.ReadFileMTX(std::string(argv[1]));

  x.Allocate("x", mat.get_nrow());
  rhs.Allocate("rhs", mat.get_nrow());

  rhs.Ones();
  x.Zeros();

  tick = paralution_time();

  CG<LocalMatrix<double>, LocalVector<double>, double > ls;
  ls.Verbose(0);

  // AMG Preconditioner
  AMG<LocalMatrix<double>, LocalVector<double>, double > p;

  p.InitMaxIter(1);
  p.Verbose(0);

  ls.SetPreconditioner(p);
  ls.SetOperator(mat);
  ls.Build();

  tack = paralution_time();
  std::cout << "Building time:" << (tack-tick)/1000000 << " sec" << std::endl;

  // move after building since AMG building is not supported on GPU yet
  mat.MoveToAccelerator();
  x.MoveToAccelerator();
  rhs.MoveToAccelerator();
  ls.MoveToAccelerator();

  mat.info();

  tick = paralution_time();

  ls.Init(1e-10, 1e-8, 1e+8, 10000);
  ls.Verbose(2);

  ls.Solve(rhs, &x);

  tack = paralution_time();
  std::cout << "Solver execution:" << (tack-tick)/1000000 << " sec" << std::endl;

  ls.Clear();

  stop_paralution();

  end = paralution_time();
  std::cout << "Total runtime:" << (end-start)/1000000 << " sec" << std::endl;

  return 0;
}
```

"PCG with AMG"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 2
Host thread affinity policy - thread mapping on every core
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: None
OpenMP threads:2
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; reading...
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; done
Building time:0.001901 sec
LocalMatrix name=/home/dimitar/matrices/small/sym/gr_30_30.mtx; rows=900;
    cols=900; nnz=7744; prec=64bit; asm=no; format=CSR; host
    backend={CPU(OpenMP)}; accelerator backend={None}; current=CPU(OpenMP)
PCG solver starts, with preconditioner:
AMG solver
AMG number of levels 2
AMG using smoothed aggregation interpolation
AMG coarsest operator size = 100
AMG coarsest level nnz = 816
AMG with smoother:
Fixed Point Iteration solver, with preconditioner:
Multicolored Gauss-Seidel (GS) preconditioner
number of colors = 4
IterationControl criteria: abs tol=1e-10; rel tol=1e-08; div tol=1e+08; max
    iter=10000
IterationControl initial residual = 30
IterationControl iter=1; residual=13.3089
IterationControl iter=2; residual=1.50267
IterationControl iter=3; residual=0.0749297
IterationControl iter=4; residual=0.00853667
IterationControl iter=5; residual=0.000410892
IterationControl iter=6; residual=2.70269e-05
IterationControl iter=7; residual=1.81103e-06
IterationControl iter=8; residual=2.39013e-07
IterationControl RELATIVE criteria has been reached: res norm=2.39013e-07; rel
    val=7.96709e-09; iter=8
PCG ends
Solver execution:0.010904 sec
Total runtime:0.024303 sec
```

"Output of the program with matrix gr_30_30.mtx"

15.6 AMG Solver with External Smoothers

```cpp
#include <iostream>
#include <cstdlib>

#include <paralution.hpp>

using namespace paralution;

int main(int argc, char* argv[]) {

  double tick, tack, start, end;

  if (argc == 1) {
    std::cerr << argv[0] << " <matrix> [Num threads]" << std::endl;
    exit(1);
  }

  init_paralution();

  if (argc > 2)
    set_omp_threads_paralution(atoi(argv[2]));

  info_paralution();

  LocalVector<double> x;
  LocalVector<double> rhs;

  LocalMatrix<double> mat;

  mat.ReadFileMTX(std::string(argv[1]));

  x.Allocate("x", mat.get_nrow());
  rhs.Allocate("rhs", mat.get_nrow());

  rhs.Ones();
  x.Zeros();

  tick = paralution_time();
  start = paralution_time();

  // Linear Solver
  AMG<LocalMatrix<double >, LocalVector<double >, double > ls;

  ls.SetOperator(mat);

  // coupling strength
  ls.SetCouplingStrength(0.001);
  // number of unknowns on coarsest level
  ls.SetCoarsestLevel(300);
  // interpolation type for grid transfer operators
  ls.SetInterpolation(SmoothedAggregation);
  // Relaxation parameter for smoothed interpolation aggregation
  ls.SetInterpRelax(2./3.);
  // Manual smoothers
  ls.SetManualSmoothers(true);
  // Manual course grid solver
  ls.SetManualSolver(true);
  // grid transfer scaling
  ls.SetScaling(true);

  ls.BuildHierarchy();

  int levels = ls.GetNumLevels();

  // Smoother for each level
  IterativeLinearSolver<LocalMatrix<double>, LocalVector<double>, double > **sm
      = NULL;
  MultiColoredGS<LocalMatrix<double>, LocalVector<double>, double > **gs = NULL;

  // Coarse Grid Solver
  CG<LocalMatrix<double>, LocalVector<double>, double > cgs;
  cgs.Verbose(0);

  sm = new IterativeLinearSolver<LocalMatrix<double>, LocalVector<double>,
      double >*[levels-1];
  gs = new MultiColoredGS<LocalMatrix<double>, LocalVector<double>, double
      >*[levels-1];

  // Preconditioner
  //  MultiColoredILU<LocalMatrix<double>, LocalVector<double>, double > p;
  //  cgs->SetPreconditioner(p);

  for (int i=0; i<levels-1; ++i) {
    FixedPoint<LocalMatrix<double>, LocalVector<double>, double > *fp;
    fp = new FixedPoint<LocalMatrix<double>, LocalVector<double>, double >;
    fp->SetRelaxation(1.3);
    sm[i] = fp;

    gs[i] = new MultiColoredGS<LocalMatrix<double>, LocalVector<double>, double
        >;
    gs[i]->SetPrecondMatrixFormat(ELL);

    sm[i]->SetPreconditioner(*gs[i]);
    sm[i]->Verbose(0);
  }

  ls.SetOperatorFormat(CSR);
  ls.SetSmoother(sm);
  ls.SetSolver(cgs);
  ls.SetSmootherPreIter(1);
  ls.SetSmootherPostIter(2);
  ls.Init(1e-10, 1e-8, 1e+8, 10000);
  ls.Verbose(2);

  ls.Build();

  mat.MoveToAccelerator();
  x.MoveToAccelerator();
  rhs.MoveToAccelerator();
  ls.MoveToAccelerator();

  mat.info();

  tack = paralution_time();
  std::cout << "Building time:" << (tack-tick)/1000000 << " sec" << std::endl;

  tick = paralution_time();

  ls.Solve(rhs, &x);

  tack = paralution_time();
  std::cout << "Solver execution:" << (tack-tick)/1000000 << " sec" << std::endl;

  ls.Clear();

  // Free all allocated data
  for (int i=0; i<levels-1; ++i) {
    delete gs[i];
    delete sm[i];
  }
  delete[] gs;
  delete[] sm;

  stop_paralution();

  end = paralution_time();
  std::cout << "Total runtime:" << (end-start)/1000000 << " sec" << std::endl;

  return 0;
}
```

"AMG with external smoothers"

```
This version of PARALUTION is released under GPL.
By downloading this package you fully agree with the GPL license.
Number of CPU cores: 2
Host thread affinity policy - thread mapping on every core
PARALUTION ver B0.8.0
PARALUTION platform is initialized
Accelerator backend: None
OpenMP threads:2
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; reading...
ReadFileMTX: filename=/home/dimitar/matrices/small/sym/gr_30_30.mtx; done
LocalMatrix name=/home/dimitar/matrices/small/sym/gr_30_30.mtx; rows=900;
    cols=900; nnz=7744; prec=64bit; asm=no; format=CSR; host
    backend={CPU(OpenMP)}; accelerator backend={None}; current=CPU(OpenMP)
Building time:0.041741 sec
AMG solver starts
AMG number of levels 2
AMG using smoothed aggregation interpolation
AMG coarsest operator size = 100
AMG coarsest level nnz = 816
AMG with smoother:
Fixed Point Iteration solver, with preconditioner:
Multicolored Gauss-Seidel (GS) preconditioner
number of colors = 4
IterationControl criteria: abs tol=1e-10; rel tol=1e-08; div tol=1e+08; max
    iter=10000
IterationControl initial residual = 30
IterationControl iter=1; residual=11.9091
IterationControl iter=2; residual=3.13946
IterationControl iter=3; residual=0.853698
IterationControl iter=4; residual=0.232396
IterationControl iter=5; residual=0.0632451
IterationControl iter=6; residual=0.0172144
IterationControl iter=7; residual=0.00468651
IterationControl iter=8; residual=0.00127601
IterationControl iter=9; residual=0.000347431
IterationControl iter=10; residual=9.45946e-05
IterationControl iter=11; residual=2.57537e-05
IterationControl iter=12; residual=7.01112e-06
IterationControl iter=13; residual=1.90858e-06
IterationControl iter=14; residual=5.19535e-07
IterationControl iter=15; residual=1.41417e-07
IterationControl RELATIVE criteria has been reached: res norm=1.41417e-07; rel
   val=4.7139e-09; iter=15
AMG ends
Solver execution:0.012193 sec
Total runtime:0.054093 sec
```

"Output of the program with matrix gr_30_30.mtx"

15.7 Laplace Example File with 4 MPI Process

```
pm.dat.rank.0
pm.dat.rank.1
pm.dat.rank.2
pm.dat.rank.3
```

”Parallel manager (main) file with 4 MPI ranks”

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% PARALUTION MPI ParallelManager output %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RANK
0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#GLOBAL_SIZE
16
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#LOCAL_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#GHOST_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#BOUNDARY_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#NUMBER_OF_RECEIVERS
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#NUMBER_OF_SENDERS
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RECEIVERS_RANK
1
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#SENDERS_RANK
1
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RECEIVERS_INDEX_OFFSET
0
2
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#SENDERS_INDEX_OFFSET
0
2
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#BOUNDARY_INDEX
1
3
2
3
```

”Parallel manager RANK 0 file”

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% PARALUTION MPI ParallelManager output %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
#RANK
1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#GLOBAL_SIZE
16
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#LOCAL_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#GHOST_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#BOUNDARY_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#NUMBER_OF_RECEIVERS
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#NUMBER_OF_SENDERS
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RECEIVERS_RANK
0
3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#SENDERS_RANK
0
3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RECEIVERS_INDEX_OFFSET
0
2
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#SENDERS_INDEX_OFFSET
0
2
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#BOUNDARY_INDEX
0
2
2
3
```

"Parallel manager RANK 1 file"

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% PARALUTION MPI ParallelManager output %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RANK
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#GLOBAL_SIZE
16
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#LOCAL_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#GHOST_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
#BOUNDARY_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#NUMBER_OF_RECEIVERS
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#NUMBER_OF_SENDERS
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RECEIVERS_RANK
0
3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#SENDERS_RANK
0
3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RECEIVERS_INDEX_OFFSET
0
2
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#SENDERS_INDEX_OFFSET
0
2
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#BOUNDARY_INDEX
0
1
1
3
```

"Parallel manager RANK 2 file"

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% PARALUTION MPI ParallelManager output %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RANK
3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#GLOBAL_SIZE
16
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#LOCAL_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#GHOST_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#BOUNDARY_SIZE
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#NUMBER_OF_RECEIVERS
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#NUMBER_OF_SENDERS
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RECEIVERS_RANK
1
2
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#SENDERS_RANK
1
2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#RECEIVERS_INDEX_OFFSET
0
2
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#SENDERS_INDEX_OFFSET
0
2
4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
#BOUNDARY_INDEX
0
1
0
2
```

"Parallel manager RANK 3 file"

```
matrix.mtx.interior.rank.0
matrix.mtx.ghost.rank.0
matrix.mtx.interior.rank.1
matrix.mtx.ghost.rank.1
matrix.mtx.interior.rank.2
matrix.mtx.ghost.rank.2
matrix.mtx.interior.rank.3
matrix.mtx.ghost.rank.3
```

"Matrix file with 4 MPI ranks"

```
%%MatrixMarket matrix coordinate real general
4 4 12
1 1 -2.4e-06
1 2 4e-07
1 3 4e-07
2 1 4e-07
2 2 -2e-06
2 4 4e-07
3 1 4e-07
3 3 -2e-06
3 4 4e-07
4 2 4e-07
4 3 4e-07
4 4 -1.6e-06
```

"Matrix interior RANK 0 file"

```
%%MatrixMarket matrix coordinate real general
4 4 4
2 1 4e-07
3 3 4e-07
4 2 4e-07
4 4 4e-07
```

"Matrix ghost RANK 0 file"

```
%%MatrixMarket matrix coordinate real general
4 4 12
1 1 -2e-06
1 2 4e-07
```

```
1 3 4e-07
2 1 4e-07
2 2 -2.4e-06
2 4 4e-07
3 1 4e-07
3 3 -1.6e-06
3 4 4e-07
4 2 4e-07
4 3 4e-07
4 4 -2e-06
```

"Matrix interior RANK 1 file"

```
%%MatrixMarket matrix coordinate real general
4 4 4
1 1 4e-07
3 2 4e-07
3 3 4e-07
4 4 4e-07
```

"Matrix ghost RANK 1 file"

```
%%MatrixMarket matrix coordinate real general
4 4 12
1 1 -2e-06
1 2 4e-07
1 3 4e-07
2 1 4e-07
2 2 -1.6e-06
2 4 4e-07
3 1 4e-07
3 3 -2.4e-06
3 4 4e-07
4 2 4e-07
4 3 4e-07
4 4 -2e-06
```

"Matrix interior RANK 2 file"

```
%%MatrixMarket matrix coordinate real general
4 4 4
1 1 4e-07
2 2 4e-07
2 3 4e-07
4 4 4e-07
```

"Matrix ghost RANK 2 file"

```
%%MatrixMarket matrix coordinate real general
4 4 12
1 1 -1.6e-06
1 2 4e-07
1 3 4e-07
2 1 4e-07
2 2 -2e-06
2 4 4e-07
3 1 4e-07
3 3 -2e-06
3 4 4e-07
4 2 4e-07
4 3 4e-07
4 4 -2.4e-06
```

"Matrix interior RANK 3 file"

```
%%MatrixMarket matrix coordinate real general
4 4 4
1 1 4e-07
1 3 4e-07
2 2 4e-07
3 4 4e-07
```

"Matrix ghost RANK 3 file"

```
rhs.dat.rank.0
rhs.dat.rank.1
rhs.dat.rank.2
rhs.dat.rank.3
```

"Vector (main) file with 4 MPI ranks"

```
-1.600000e-06
-8.000000e-07
-8.000000e-07
0.000000e+00
```

"Vector RANK 0 file"

```
-8.000000e-07
-1.600000e-06
0.000000e+00
-8.000000e-07
```

"Vector RANK 1 file"

```
-8.000000e-07
0.000000e+00
-1.600000e-06
-8.000000e-07
```

"Vector RANK 2 file"

```
0.000000e+00
-8.000000e-07
-8.000000e-07
-1.600000e-06
```

"Vector RANK 3 file"

Chapter 16

Change Log

1.1.0 - Jan 19, 2016

- New features:
  - CSR to COO conversion (GPU)
  - CSR to HYB conversion (OpenCL)
  - Changeable minimum iteration count for all solvers
- Improvements:
  - Fortran plug-in
  - Improved debug functionality
  - FSAI optimizations
  - PairwiseAMG optimizations
  - Improved OpenCL backend
  - Improvements in matrix conversions
  - Renamed vector function PartialSum() to ExclusiveScan()
- Bugfixes:
  - File logging
  - Fix in Permute
  - Fix in extraction of upper matrix
  - Fix in RS and PairwiseAMG
  - Fix in multicolored preconditioners
  - Fix in CSR to HYB conversion
  - Minor fixes in LocalMatrix class

1.0.0 - Feb 27, 2015

- New features:
  - Dual license model
  - Global operations (via MPI)
  - Pairwise AMG with global operations (via MPI)
  - Hardware/time locking mechanism
- Improvements:
  - Function to disable accelerator
- Bugfixes:
  - Minor fixes in (S)GS preconditioners
  - Residual computation in FixedPoint scheme

0.9.0 - Jan 30, 2015

- New features:
  - Automatic memory PARALUTION object tracking
- Bugfixes:
  - CUDA and OpenCL backends

0.8.0 - Nov 5, 2014

- New features:
  - Complex support
  - Variable preconditioer
  - TNS preconditioner
  - BiCGStab(l); QMRCGStab, FCG solvers
  - RS and PairWise AMG
  - SIRA eigenvalue solver
  - Replace/Extract column/row functions
  - Stencil computation
- Improvements in:
  - Support for OpenFOAM 2.3
  - FSAI building phase
  - Multigrid structure
  - Iteration counter
  - I/O structure
  - Host memory allocation checks
  - New user manual structure
- Bugfixes:
  - Cmake OpenCL path

0.7.0 - May 19, 2014

- New features:
  - Windows support (OpenMP, CUDA, OpenCL backend)
  - MATLAB/Octave plugin
  - Additive/Restricted Schwarz preconditioner
  - Direct solvers LU, QR, inversion (dense)
  - Assembling (matrices and vectors)
  - OpenMP threshold size (avoid OpenMP for very small problems)
- Improvements in:
  - OpenCL backend
  - IC0 (OpenMP, CUDA)
- Bugfixes:
  - FGMRES

0.6.1 - March 2, 2014

- New features:
  - Windows support (CUDA backend)
  - Mac OS support
  - FGMRES in the OpenFOAM plug-in
  - FGMRES and FSAI in the FORTRAN plug-in
- Improvements in:
  - CSR to DIA conversion (CUDA backend)
- Bugfixes:
  - Fixing *Init()* and *Set()* functions in the plug-ins
  - Using absolute and not relative stopping criteria in the Deal.II example

0.6.0 - Feb 25, 2014

- New features:
  - Windows support (OpenMP backend)
  - FGMRES (Flexible GMRES)
  - (R)CMK ordering
  - Thread-core affiliation (for Host OpenMP)
  - Async transfers with *MoveTo*Async()* and with *CopyFromAsync()* (implemented for CUDA backend)
  - Pinned memory allocation on the host when using CUDA backend
  - Debug output
  - Easy to handle timing functions in the examples
- Improvements in:
  - Binary CSR matrix read - now includes also sorting of the CSR
  - Better output for the *init_paralution()* and *info_paralution()*
  - Better CUDA/OpenCL backend initialization phase
  - OpenCL backend - *Compress()*, MIS, initialization step
- Bugfixes:
  - The *Init()* function in all preconditioners, Chebyshev iteration and mixed-precision DC scheme is re-named to *Set()* (due to compiler problems)
  - Chebyshev iteration - Linf indexing norm value
  - *ZeroBlockPermtuation()*

0.5.0 - Dec 11, 2013

- New features:
  - Deflated PCG method
  - CR method
  - BlockPreconditioner
  - Unique select_device_paralution() for all backends
  - Amax (LocalVector class) also returns index position
  - Iterative solvers with Inf norm return also position
  - stop_paralution() restores the original OpenMP thread configuration
  - Invert() function for (small) dense matrices (based on QR decomposition)

  - ExtractL() and ExtractU() function (LocalMatrix class)
  - Write/Read binary CSR files (LocalMatrix class)
  - Importing BlockMatrix/Vector from Deal.II

- Improvements in:
  - The library is built as a shared-library (.so) file
  - Compress (CUDA backend)
  - Faster MIS computation (CUDA backend)
  - Automatic (different) matrix indexing fixed during compilation (MIC/CUDA/OpenCL/CPU)
  - Re-structuring of the MultiGrid class
- Bugfixes:
  - DENSE - ApplyAdd
  - LLSolve(), ILUT factorization
  - IC preconditioner (CUDA backend)

0.4.0 - Oct 3, 2013

- New features:
  - MIC/Xeon Phi OpenMP backend
  - IDR(s) solver for general sparse systems
  - FSAI(q) preconditioner
  - SPAI preconditioner
  - Incomplete Cholesky preconditioner
  - Jacobi-type saddle-point preconditioner
  - L_1, L_2 and L_∞ norms for stopping criteria (Solver class)
  - Asum, Amax functions (LocalVector class)
  - SetRandom value function (LocalVector class)
  - Elmer plug-in
  - Hermes / Agros2D plug-in (directly integrated in Hermes)
- Bugfixes:
  - DENSE - ApplyAdd
  - LLSolve(), ILUT factorization
  - ILU preconditioner (CUDA backend),

0.3.0 - June 1, 2013

- New features:
  - Select a GPU (CUDA and OpenCL backend) function for multi-GPU systems
  - ILUT (Incomplete LU with threshold) - experimental
  - OpenFOAM - direct CSR import function
- Improvements in:
  - Permute CUDA - L1 cache utilization
  - Faster multicoloring with CUDA backend
- Bugfixes:
  - HYB, MCSR - ApplyAdd
  - CUDA backend manager works on systems without GPU (computation on OpenMP)
  - Apple OpenCL include path
  - AMG - Operator mismatch backend

0.2.0 - May 5, 2013

- New features:
  - AMG solver (including OpenFOAM plug-in)
  - ELL, DIA, HYB conversion (CUDA)
  - ResetOperator function (Solver class)
  - SolveZeroSol function (Solver class)
- Improvements in:
  - Multigrid class - restructured
  - Multicoloring analyzer
  - ConvertTo() - internal restructured
  - CMake - FindCUDA files included
  - OpenFOAM plug-in
- Bugfixes:
  - Memory leaks (preconditioners)
  - OpenFOAM solvers - absolute stopping criteria
  - OpenCL backend - fix in memory allocation

0.1.0 - Apr 6, 2013

- New features:
  - GMRES solver
  - Mixed-precision solver
  - Multi-grid solver (solving phase only/V-cycle)
  - Plug-in for Fortran
  - Plug-in for OpenFOAM
  - Geometric-algebraic multi-grid (GAMG) solver for OpenFOAM
- Improvements in:
  - Deal.II plug-in (faster import/export functions)
  - ILU(0) CPU factorization
- Bugfixes:
  - Preconditioned BiCGStab
  - OpenCL ApplyAdd kernel for ELL matrix format
  - ExtractSubMatrix()
  - Cmake when using Intel compiler
  - IterationControl::InitResidual() counter
  - Transpose (for non-squared matrices)

0.0.1 - Feb 24, 2013

- Pilot version of the library
- OpenCL support
- OpenMP/CUDA permutation for CSR matrices
- CMake compilation
- Deal.II plug-in
- Internal handling of functions which are not implemented on the accelerator

0.0.1b - Jan 23, 2013

- Initial pre-release version of the library

Bibliography

[1] Agros2d — an application for solution of physical fields. http://www.agros2d.org/.

[2] Gnu octave. https://www.gnu.org/software/octave/.

[3] gr3030 matrix. http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/laplace/gr_30_30.html.

[4] Hermes — hp-fem group. http://www.hpfem.org/hermes/.

[5] impcolc matrix. http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/chemimp/impcol_c.html.

[6] nos5 matrix. http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/lanpro/nos5.html.

[7] tols340 matrix. http://math.nist.gov/MatrixMarket/data/NEP/mvmtls/tols340.html.

[8] Bangerth, W., Hartmann, R., and Kanschat, G. deal.II – a general purpose object oriented finite element library. *ACM Trans. Math. Softw. 33*, 4 (2007), 24/1–24/27.

[9] Bangerth, W., Heister, T., and Kanschat, G. `deal.II` *Differential Equations Analysis Library, Technical Reference.* `http://www.dealii.org`.

[10] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2 ed. SIAM, Philadelphia, PA, 1994.

[11] Cai, X., and Sarkis, M. A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems. *SIAM J. Sci. Comput 21* (1999), 792–797.

[12] Creative Commons. Attribution-noncommercial-noderivs 3.0 unported license. http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode.

[13] CSC IT Center for Science. Elmer – open source finite element software for multiphysical problems. http://www.csc.fi/english/pages/elmer.

[14] Dag, H., and Semlyen, A. A new preconditioned conjugate gradient power flow. *IEEE Transactions on Power Systems 18*, 4 (2003), 1248–1255.

[15] Demmel, J. W. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, PA, 1997.

[16] Engblom, S. `stenglib:` a collection of Matlab packages for daily use, 2014. *Multiple software components.* http://user.it.uu.se/∼stefane/freeware.html.

[17] Engblom, S., and Lukarski, D. Fast Matlab compatible sparse assembly on multicore computers. *ArXiv e-prints* (June 2014).

[18] Free Software Foundation. GPL v3. http://www.gnu.org/licenses/gpl-3.0.txt.

[19] G, G. L., Sleijpen, G., and Fokkema, D. Bicgstab(l) For Linear Equations Involving Unsymmetric Matrices With Complex Spectrum, 1993.

[20] GNU Compiler Collection. gcc. http://gcc.gnu.org/.

[21] Grote, M. J., and Huckle, T. Effective parallel preconditioning with sparse approximate inverses. *PPSC* (1995), 466–471.

[22] Hochstenbach, M. E., and Notay, Y. Controlling inner iterations in the jacobi-davidson method. *SIAM journal on matrix analysis and applications 31*, 2 (2009), 460–477.

[23] Intel. C++ Compiler. http://software.intel.com/en-us/intel-compilers.

[24] INTEL. Math Kernel Library. http://software.intel.com/en-us/intel-mkl.

[25] INTEL. Threading Building Blocks. http://software.intel.com/en-us/intel-tbb.

[26] INTEL. Xeon Phi / MIC. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[27] JIA, Z., AND LI, C. On Inner Iterations in the Shift-Invert Residual Arnoldi Method and the Jacobi–Davidson Method. *ArXiv e-prints* (2011).

[28] JIA, Z., AND LI, C. The shift-invert residual arnoldi method and the jacobi–davidson method: Theory and algorithms. *arXiv preprint arXiv:1109.5455* (2011).

[29] KHRONOS GROUP. OpenCL. http://www.khronos.org/opencl/.

[30] KITWARE. CMake. http://www.cmake.org/.

[31] KOLOTILINA, L. Y., AND YEREMIN, A. Y. Factorized sparse approximate inverse preconditionings, 1. theory. *SIAM J. Matrix Anal. Appl. 14* (1993), 45–58.

[32] LEE, C.-R. Residual arnoldi method, theory, package and experiments.

[33] LEE, C.-R., AND STEWART, G. Analysis of the residual arnoldi method.

[34] LIU, X., GU, T., HANG, X., AND SHENG, Z. A parallel version of QMRCGSTAB method for large linear systems in distributed parallel environments. *Applied Mathematics and Computation 172*, 2 (2006), 744 – 752. Special issue for The Beijing-HK Scientific Computing Meetings.

[35] LUKARSKI, D. *Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms – Parallel Solvers and Preconditioners.* PhD thesis, Karlsruhe Institute of Technology, 2012.

[36] MATHWORKS. Matlab. http://www.mathworks.se/products/matlab/.

[37] MATRIX MARKET. Format description. http://math.nist.gov/MatrixMarket/formats.html.

[38] MATRIX MARKET. gr_30_30. http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/laplace/gr_30_30.html.

[39] MATRIX MARKET. Matlab interface. http://math.nist.gov/MatrixMarket/mmio/matlab/mmiomatlab.html.

[40] MIT. Cilk. http://supertech.csail.mit.edu/cilk/.

[41] NOTAY, Y. Flexible conjugate gradients. *SIAM J. Sci. Comput 22* (2000), 1444–1460.

[42] NOTAY, Y. An aggregation-based algebraic multigrid method, 2010.

[43] NVIDIA. CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[44] OPENFOAM FOUNDATION. OpenFOAM. http://www.openfoam.org/.

[45] QUARTERONI, A., AND VALLI, A. *Domain Decomposition Methods for Partial Differential Equations*, 1 ed. Numerical Mathematics and Scientific Computation. Clarendon Press, Oxford, 1999.

[46] SAAD, Y. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[47] SMITH, B., BJRSTAD, P., AND GROPP, W. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge Univ. Press, Cambridge, 1996.

[48] SONNEVELD, P., AND VAN GIJZEN, M. B. IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM J. Scientific Computing 31*, 2 (2008), 1035–1062.

[49] STUBEN, K. Algebraic multigrid (AMG): An introduction with applications. *Journal of Computational and Applied Mathematics 128* (2001), 281–309.

[50] TANG, J., AND VUIK, C. Efficient deflation methods applied to 3-D bubbly flow problems. *Electronic Transactions on Numerical Analysis 26* (2007), 330–349.

[51] TANG, J., AND VUIK, C. New variants of deflation techniques for pressure correction in bubbly flow problems. *Journal of Numerical Analysis, Industrial and Applied Mathematics 2* (2007), 227–249.

[52] TOSELLI, A., AND WIDLUND, O. *Domain Decomposition Methods - Algorithms and Theory.* Springer Series in Computational Mathematics ; 34. Springer, Berlin, 2005.

[53] Trottenberg, U., Oosterlee, C., and Schller, A. *Multigrid.* Academic Press, Amsterdam, 2003.

[54] Van Gijzen, M. B., and Sonneveld, P. Algorithm 913: An elegant IDR(s) variant that efficiently exploits biorthogonality properties. *ACM Trans. Math. Softw. 38*, 1 (Dec. 2011), 5:1–5:19.

[55] van Heesch, D. Doxygen. http://www.doxygen.org.

[56] Vanek, P., Mandel, J., and Brezina, M. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing 56*, 3 (1996), 179–196.