

PLASMA Users' Guide

Parallel Linear Algebra Software for Multicore Architectures

Version 2.0

Electrical Engineering and Computer Science
University of Tennessee

Electrical Engineering and Computer Science
University California at Berkeley

Mathematical & Statistical Sciences
University of Colorado Denver

Emmanuel Agullo
Jack Dongarra
Bilel Hadri
Jakub Kurzak
Julie Langou
Julien Langou
Hatem Ltaief
Piotr Luszczek
Asim YarKhan

Contents

1	Essentials	1
1.1	PLASMA	1
1.2	Problems that PLASMA Can Solve	2
1.3	Computers for which PLASMA is Suitable	2
1.4	PLASMA versus LAPACK and ScaLAPACK	2
1.5	PLASMA and the BLAS	3
1.6	Availability of PLASMA	3
1.7	Commercial Use of PLASMA	4
1.8	Installation of PLASMA	4
1.9	Documentation of PLASMA	4
1.10	Support for PLASMA	5
2	Installing PLASMA	6
2.1	Getting the PLASMA Installer	6
2.2	PLASMA Installer flags	7
2.3	PLASMA Installer Usage	8
2.4	PLASMA Installer Support	9
2.5	Tips and Tricks	9
2.5.1	Tests are slow	9
2.5.2	Installing BLAS on a Mac	9
2.5.3	Processors with Hyper-threading	9
2.5.4	Problems with Downloading	10
2.6	PLASMA under Windows	10
2.6.1	Using the Windows PLASMA binary package	10
2.6.2	Building PLASMA libraries	11

3	PLASMA Testing Suite	12
3.1	Simple Test Programs	12
3.2	Advanced Test Programs	13
3.3	Send the Results to Tennessee	13
4	Use of PLASMA and Examples	15
4.1	Examples	15
4.2	PLASMA_dgesv example	15
5	Performance of PLASMA	19
5.1	A Library for Multicore Architectures	19
5.2	Comparison to other libraries	20
5.3	Tuning - Howto	20
6	Accuracy and Stability	23
6.1	Notations	23
6.2	Peculiarity of the Error Analysis of the Tile Algorithms	24
6.3	Tile Cholesky Factorization	24
6.4	Tile Householder QR Factorization	25
6.5	Tile LU Factorization	25
7	Troubleshooting	27
7.1	Wrong Results	27
7.1.1	Linux machine: Intel x86-64	28
7.1.2	Linux machine: Intel 32	28
7.1.3	Linux machine: AMD Opteron	29
7.1.4	Linux machine: IBM Power6	29
7.1.5	Non-Linux machine	29

Preface

PLASMA version 1.0 was released in November 2008 as a prototype software providing *proof-of-concept* implementation of a linear equations solver based on LU factorization, SPD linear equations solver based on Cholesky factorization and least squares problem solver based on QR and LQ factorizations, with support for real arithmetic in double precision only. The publication of this users's guide coincides with the release of version 2.0 of the PLASMA software, which extends the functionality and robustness of the software by introducing the following features:

Multiple Precision Support: Real arithmetic and complex arithmetic are supported in both single precision and double precision. Only complex-double code is developed. complex-single, real-double and real-single code is automatically generated from the complex-double *reference* implementation.

LAPACK Interface and Native Interface: All computational routines are available in two versions. One accepts input matrices in LAPACK column-major layout, second one accepts input matrices in tile layout, which is the native layout for PLASMA. The first option provides for convenience at the expense of translation overhead, the second one provides increased performance at the cost of ease of use. Conversion routines between the LAPACK layout and the tile layout are provided.

LAPACK-Compliant Error Handling: The error codes returned by the computational routines with LAPACK interface follow LAPACK convention for both numerical errors, related to the numerical properties of the input matrices, as well as errors caused by illegal values of input parameters.

LAPACK-Derived Testing Suite: The software is accompanied by a test suite derived from the one of LAPACK, which in an automated fashion generates thousands of calls, testing the software’s behavior under normal conditions, as well as in the presence of illegal arguments and numerically deficient matrices.

Convenient Workspace Allocation: Whenever possible, PLASMA allocates workspaces internally. In situations, where workspaces serve the purpose of passing data from one routine to another, e.g., from a factorization routine to a solve routine, PLASMA provides straightforward helper routines to take care of the allocations.

Thread Safety: PLASMA is thread safe, which means that multiple instances of PLASMA can co-exist within the address space of a single process. In other words, a single process can create multiple threads and each of these threads can use an independent instance and change its settings without conflicts with other threads.

Mixed-Precision Solver Prototype: A prototype of a mixed-precision solver based on the LU factorization is included. Currently, the implementation is not optimized for performance and does not provide performance benefits over the fixed-precision algorithm. It is only included to demonstrate the technique.

The PLASMA project is funded in part by the National Science Foundation, Department of Energy, Microsoft, and the MathWorks.

A major effort is underway to implement PLASMA-type algorithms for hybrid systems equipped with accelerator devices, GPUs and alike.

The PLASMA team would like to thank the following people, who have significant contributions to the PLASMA project:

Alfredo Buttari.

CHAPTER 1

Essentials

1.1 PLASMA

PLASMA is a software library, currently implemented using the FORTRAN and C programming languages, and providing interfaces for FORTRAN and C. It has been designed to be efficient on *homogeneous* multicore processors and multi-socket systems of multicore processors. The name PLASMA is an acronym for *Parallel Linear Algebra Software for Multi-core Architectures*.

PLASMA project website is located at:

<http://icl.cs.utk.edu/plasma>

PLASMA software can be downloaded from:

<http://icl.cs.utk.edu/plasma/software/>

PLASMA users' forum is located at:

<http://icl.cs.utk.edu/plasma/forum/>

and can be used to post general questions and comments as well as to report technical problems.

1.2 Problems that PLASMA Can Solve

PLASMA can solve dense systems of linear equations and linear least squares problems and associated computations such as matrix factorizations. Unlike LAPACK, currently PLASMA does not solve eigenvalue or singular value problems and does not support band matrices. Similarly to LAPACK, PLASMA does not support general sparse matrices. For all supported types of computation the same functionality is provided for real and complex matrices in single precision and double precision.

1.3 Computers for which PLASMA is Suitable

PLASMA is designed to give high efficiency on homogeneous multicore processors and multi-socket systems of multicore processors. As of today, the majority of such systems are on-chip symmetric multiprocessors with classic *super-scalar* processors as their building blocks (x86 and alike) augmented with short-vector SIMD extensions (SSE and alike). A parallel software project MAGMA (Matrix Algebra on GPU and Multicore Architectures), is being developed to address the needs of heterogeneous (hybrid) systems, equipped with hardware accelerators, such as GPUs.

<http://icl.cs.utk.edu/magma>

The name MAGMA is an acronym for

1.4 PLASMA versus LAPACK and ScaLAPACK

PLASMA has been designed to supercede LAPACK (and eventually ScaLAPACK), principally by restructuring the software to achieve much greater efficiency, where possible, on modern computers based on multicore processors. PLASMA also relies on new or improved algorithms.

Currently, PLASMA does not serve as a complete replacement of LAPACK due to limited functionality. Specifically, PLASMA does not support band matrices and does not solve eigenvalue and singular value problems. At this point, PLASMA does not replace ScaLAPACK as software for distributed memory computers, since it only supports shared-memory machines.

1.5 PLASMA and the BLAS

LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subroutines (BLAS). Highly efficient machine-specific implementations of the BLAS are available for most modern processors, including multi-threaded implementations.

The parallel algorithms in PLASMA are built using a small set of sequential routines as building blocks. These routines are referred to as *core BLAS*. Ideally, these routines would be implemented through monolithic machine-specific code, utilizing to the maximum a single processing core (through the use of short-vector SIMD extensions and appropriate cache and register blocking).

However, such machine-specific implementations are extremely labor-intensive and covering the entire spectrum of available architectures is not feasible. Instead, the core BLAS routines are built in a somewhat suboptimal fashion, by using the “standard” BLAS routines as building blocks. For that reason, just like LAPACK, PLASMA requires a highly optimized implementation of the BLAS in order to deliver good performance.

Although the BLAS are not part of either PLASMA or LAPACK, FORTRAN code for the BLAS is distributed with LAPACK, or can be obtained separately from Netlib:

<http://www.netlib.org/blas/blas.tgz>

However, it has to be emphasized that this code is only the “reference implementation” (the definition of the BLAS) and cannot be expected to deliver good performance. On most of today’s machines it will deliver performance an order of magnitude lower than that of optimized BLAS.

For information on available optimized BLAS libraries, as well as other BLAS-related questions, please refer to the BLAS FAQ:

<http://www.netlib.org/blas/faq.html>

1.6 Availability of PLASMA

PLASMA is distributed in source code and is, for the most part, meant to be compiled from source on the host system. In certain cases, a pre-built binary may be provided along with the source code. Such packages, built by the PLASMA developers, will be provided as separate archives on the PLASMA download page:

<http://icl.cs.utk.edu/plasma/software/>

The PLASMA team does not reserve exclusive right to provide such packages. They can be provided by other individuals or institutions. However, in case of problems with binary distributions acquired from other places, the provider needs to be asked for support rather than PLASMA developers.

1.7 Commercial Use of PLASMA

PLASMA is a freely available software package. Thus it can be included in commercial packages. The PLASMA team asks only that proper credit be given by citing this users' guide as the official reference for PLASMA.

Like all software, this package is copyrighted. It is not trademarked. However, if modifications are made that affect the interface, functionality, or accuracy of the resulting software, the name of the routine should be changed and the modifications to the software should be noted in the modifier's documentation.

The PLASMA team will gladly answer questions regarding this software. If modifications are made to the software, however, it is the responsibility of the individual or institution who modified the routine to provide support.

1.8 Installation of PLASMA

A PLASMA installer is available at:

<http://icl.cs.utk.edu/plasma/software/>

Further details are provided in the chapter 2 Installing PLASMA.

1.9 Documentation of PLASMA

PLASMA package comes with a variety of pdf and html documentation.

- The PLASMA Users Guide (this document)
- The PLASMA README
- The PLASMA Installation Guide
- The PLASMA Routine Description

- The PLASMA and Tau Guide
- The PLASMA Routine browsing

You will find all of these in the documentation section on the PLASMA website <http://icl.cs.utk.edu/plasma>.

1.10 Support for PLASMA

PLASMA has been thoroughly tested before release, using multiple combinations of machine architectures, compilers and BLAS libraries. The PLASMA project supports the package in the sense that reports of errors or poor performance will gain immediate attention from the developers. Such reports – and also descriptions of interesting applications and other comments – should be posted to the PLASMA users' forum:

<http://icl.cs.utk.edu/plasma/forum/>

CHAPTER 2

Installing PLASMA

The minimum requirements for installing PLASMA on a UNIXTM system are a C compiler, a Fortran compiler, a BLAS library and the availability of the pthread library. For requirement and instructions on Microsoft WindowsTM see Section 2.6. Before PLASMA can be built or tested, you must define all machine-specific parameters for the architecture on which you are installing PLASMA. All machine-specific parameters are contained in the file `make.inc`. To ease the installation process, we provide an installer which can generate the required machine-specific `make.inc` file. Users are strongly encouraged to use it.

2.1 Getting the PLASMA Installer

The PLASMA installer is a set of python scripts developed to ease the installation of the PLASMA library. It can automatically download, configure and compile the PLASMA library including a BLAS dependency.

It is available on PLASMA download page:

<http://icl.cs.utk.edu/plasma/software/>

2.2 PLASMA Installer flags

Here's a list of the flags that can be used to provide the installer with information about the system, for example, the C and Fortran compilers, the location of a local BLAS library, or whether the reference BLAS needs to be downloaded.

`./setup.py`

<code>-h</code> or <code>--help</code>	: prints this message
<code>--prefix</code>	: path where to create the libraries, build and log of the installer
<code>--cc=[CMD]</code>	: the C compiler.
<code>--fc=[CMD]</code>	: the Fortran compiler.
<code>--ccflags=[FLAGS]</code>	: the flags for the C compiler (default <code>-O2</code>)
<code>--fcflags=[FLAGS]</code>	: the flags for the Fortran compiler (default <code>-O2</code>)
<code>--ldflags_c=[flags]</code>	: loader flags when main program is in C. Some compilers (e.g. PGI) require different options when linking C main programs to Fortran subroutines and vice-versa
<code>--ldflags_fc=[flags]</code>	: loader flags when main program is in Fortran. Some compilers (e.g. PGI) require different options when linking Fortran main programs to C subroutines and vice-versa
<code>--makecmd=[CMD]</code>	: the make command (make by default)
<code>--blaslib=[LIB]</code>	: a BLAS library (path should be absolute if <code>--prefix</code> is used)
<code>--downblas</code>	: if you do not want to provide a BLAS we can download and install the reference BLAS from Netlib
<code>--nbcores</code>	: The number of cores to be used by the testing. (2 by default)

2.3. PLASMA INSTALLER USAGE

`--notesting` : disables the PLASMA testing. The BLAS library is not required in this case.

`--clean` : cleans up the installer directory.

The installer will set the following environment variables

```
OMP_NUM_THREADS=1
GOTO_NUM_THREADS=1
MKL_NUM_THREADS=1
```

in order to disable multithreading within BLAS. **IMPORTANT** Do not forget to set those environment variables for any further usage of the PLASMA libraries.

2.3 PLASMA Installer Usage

For an installation with **gcc, gfortran and reference BLAS**

```
./setup.py --cc gcc --fc gfortran --downblas
```

For an installation with **ifort, icc and MKL** (em64t architecture)

```
./setup.py --cc icc --fc ifort --blaslib="-lmkl_em64t -lguide"
```

For an installation with **gcc, gfortran and Veclib** (Mac OS/X)

```
./setup.py --cc gcc --fc gfortran --blaslib="-framework veclib"
```

For an installation with **gcc, gfortran, ATLAS**

```
./setup.py --cc gcc --fc gfortran --blaslib="-lf77blas -lcblas -latlas"
```

For an installation with **gcc, gfortran, goto BLAS and 4 cores**

```
./setup.py --cc gcc --fc gfortran --blaslib="-lgoto" --nbcores=4
```

For an installation with **xlc, xlf, essl and 8 cores**

```
./setup.py --cc xlc --fc xlf --blaslib="-lessl" --nbcores=8
```

2.4 PLASMA Installer Support

Please note that this is an alpha version of the installer and, even though it has been tested on a wide set of systems, it may not work. If you encounter a problem, your feedback would be greatly appreciated and would be very useful for improving the quality of this installer. Please submit your complaints and suggestions to the PLASMA forum:

<http://icl.cs.utk.edu/plasma/forum/>

2.5 Tips and Tricks

2.5.1 Tests are slow

If the installer is asked to automatically download and install a BLAS library (using the `--downblas` flag), the reference BLAS from Netlib is installed and very low performance is to be expected. It is strongly recommended that you use an optimized BLAS library (such as ATLAS, Cray Scientific Library, HP MLIB, Intel MKL, GotoBLAS, IBM ESSL, VecLib etc.) and provide its location through the `--blaslib` flag.

2.5.2 Installing BLAS on a Mac

Optimized BLAS are available using VecLib if you install the Xcode developer package provided with your Mac OS installation CD. On MAC OS/X you may be required to add the following flag.

```
--ccflags="-I/usr/include/sys/"
```

2.5.3 Processors with Hyper-threading

The PLASMA installer cannot detect if you have hyper-threading enabled on your machine. We advise you to limit the number of cores for the initial testing of the PLASMA library to half the numbers of cores available if you do not know your exact architecture. Using hyper-threading will cause the PLASMA testing to hang. When using PLASMA, the number of cores should not exceed the number of actual compute cores (ignore cores appearing due to hyper-threading).

2.5.4 Problems with Downloading

If the required packages cannot be automatically downloaded (for example, because no network connection is available on the installation system), you can obtain them from the following URLs and place them in the build subdirectory of the installer (if the directory does not exist, create it):

BLAS Reference BLAS written in Fortran can be obtained from

<http://netlib.org/blas/blas.tgz>

PLASMA The PLASMA source code is available via a link on the download page:

<http://icl.cs.utk.edu/plasma/software/>

2.6 PLASMA under Windows

We provide installer packages for 32-bit and 64-bit binaries on Windows available from the PLASMA web site.

<http://icl.cs.utk.edu/plasma/software/>

These Windows packages are created using Intel C and Fortran compilers using multi-threaded static linkage, and the packages should include all required redistributable libraries. The binaries included in this distribution link the PLASMA libraries with Intel MKL BLAS in order to provide basic functionality tests. However, any BLAS library should be usable with the PLASMA libraries.

2.6.1 Using the Windows PLASMA binary package

Using the PLASMA libraries requires at least a C99 or C++ compiler and a BLAS implementation. The `examples` directory contains several examples of using PLASMA's linear algebra functions and a simplified makefile (`Makefile.nmake`). This file provides guidance on how to compile and link C or Fortran code using several different compilers (Microsoft Visual C, Intel C, Intel Fortran) and different BLAS libraries (Intel MKL, AMD ACML) in 32- or 64-bit floating-point precisions. The examples in `Makefile.nmake` will have to be adjusted to the appropriate locations of the compilers and libraries.

You **need** to make sure that the BLAS libraries are being used in a sequential mode, either by linking with a sequential version of the BLAS libraries, or by setting the appropriate environment variable for that library. PLASMA manages the parallelism on a machine and

it will cause substantial performance degradation and a possible hang of the code if the BLAS calls are also running in parallel.

2.6.2 Building PLASMA libraries

Rebuilding the PLASMA libraries should not be required, but if you wish to do so, the tested build environment uses Intel C and Fortran compilers, Intel MKL BLAS, and the CMake build system. This build enforces an out-of-source build to avoid clobbering pre-existing files. The following example commands show how the build might be done from a command shell window where the path has been setup for the appropriate (either 32-bit or 64-bit) Intel environment.

```
mkdir BUILD
cd BUILD
cmake -DCMAKE_Fortran_COMPILER=ifort -DCMAKE_C_COMPILER=icl
      -DCMAKE_CXX_COMPILER=icl -G "NMake Makefiles" ..
nmake
```

Native threading API (as supplied with modern Windows systems starting with Windows 2000) is used to provide parallelism within the PLASMA code. Wrapper functions translate any PLASMA threading calls into native Windows thread calls. For example, the `_beginthreadex()` function is called to spawn PLASMA threads.

CHAPTER 3

PLASMA Testing Suite

There are two distinct sets of test programs for PLASMA routines, simple test programs written in C and advanced test programs written in Fortran. These programs test the linear equation routines (eigensystem routines are not yet available) in each data type single, double, complex, and double complex (sdcz).

3.1 Simple Test Programs

In the `testing` directory, you will find simple C codes which test the `PLASMA_[sdcz]gels`, `PLASMA_[sdcz]gesv`, `PLASMA_[sdcz]posv` routines as well as routines using iterative refinement for LU factorization using random matrices. Each solving routine is also tested using different scenarios. For instance, testing complex least square problems on tall matrices is done by a single call to `PLASMA_zgels` as well as successive calls to `PLASMA_zgeqrf`, `PLASMA_zunmqr` and `PLASMA_ztrsm`.

A python script `plasma_testing.py` runs all testing routines in all precisions. This script can take the number of cores to run the testing as a parameter, the default being half of the cores available.

A brief summary is printed out on the screen as the testing procedure runs. A detailed summary is written in `testing_results.txt` at the end of the testing phase and can be

3.2. ADVANCED TEST PROGRAMS

sent the PLASMA development group if any failures are encountered (see Section 3.3).

Each test can also be run individually. The usage of each test is shown by typing the desired testing program without any arguments. For example:

```
> ./testing_cgesv
Proper Usage is : ./testing_cgesv ncores N LDA NRHS LDB with
- ncores : number of cores
- N : the size of the matrix
- LDA : leading dimension of the matrix A
- NRHS : number of RHS
- LDB : leading dimension of the matrix B
```

3.2 Advanced Test Programs

In the `testing/lin` directory, you will find Fortran codes which check the `PLASMA_[sdcz]gels`, `PLASMA_[sdcz]gesv` and `PLASMA_[sdcz]posv` routines. This allows us to check not only the correctness of the routines but also the PLASMA Fortran interface. This test suite has been taken from LAPACK 3.2 with necessary modifications to safely integrate PLASMA routine calls.

There is also a python script called `lapack_testing.py` which tests all routines in all precisions. A brief summary is printed out on the screen as the testing procedure runs. A detailed summary is written in `testing_results.txt` at the end of the testing procedure and can be sent to us if any failures are encountered (see Section 3.3).

Each test can also be run individually. For single, double, complex, and double complex precision tests, the calls are respectively:

```
xlintsts < stest.in > stest.out
xlintstd < dtest.in > dtest.out
xlintstc < ctest.in > ctest.out
xlintstz < ztest.in > ztest.out
```

For more information on the test programs and how to modify the input files, please refer to LAPACK Working Note 41 [10].

3.3 Send the Results to Tennessee

You made it! You have now finished installing and testing PLASMA. If you encountered failures in any phase of the installing or testing process, please consult Chapter 7 as well as

3.3. *SEND THE RESULTS TO TENNESSEE*

the README file located in the root directory of your PLASMA installation. If the suggestions do not fix your problem, please feel free to send a post in the PLASMA users' forum <http://icl.cs.utk.edu/plasma/forum/>.

Tell us the type of machine on which the tests were run, the version of the operating system, the compiler and compiler options that were used, and details of the BLAS library or libraries that you used. You should also include a copy of the output file in which the failure occurs. We encourage you to make the PLASMA library available to your users and provide us with feedback from their experiences.

CHAPTER 4

Use of PLASMA and Examples

4.1 Examples

In the `./examples` directory, you will find simple example codes to call the `PLASMA_[sdcz]gels`, `PLASMA_[sdcz]gesv` and `PLASMA_[sdcz]posv` routines from a C program or from a Fortran program. In this section we further explain the necessary steps for a correct use of these PLASMA routines.

4.2 `PLASMA_dgesv` example

Before calling the PLASMA routine `PLASMA_dgesv`, some initialization steps are required.

Firstly, we set the dimension of the problem $Ax = B$. In our example, the coefficient matrix A is 10-by-10, and the right-hand side matrix B 10-by-5. We also allocate the memory space required for these two matrices.

```
in C:
    int N = 10;
    int NRHS = 5;
    int NxN = N*N;
```

4.2. PLASMA_DGESV EXAMPLE

```
int NxNRHS = N*NRHS;
double *A = (double *)malloc(NxN*sizeof(double));
double *B = (double *)malloc(NxNRHS*sizeof(double));
in Fortran:
      INTEGER          N, NRHS
      PARAMETER        ( N = 10 )
      PARAMETER        ( NRHS = 5 )
      DOUBLE PRECISION A( N, N ), B( N, NRHS )
```

Secondly, we initialize the matrix *A* and *B* with random values. Since we cruelly lack imagination, we use the LAPACK function `dlarnv` for this task. For a starter, you are welcome to change the values in the matrix. Remember that the interface of `PLASMA_dgesv` uses column major format.

```
in C:
int IONE=1;
int ISEED[4] = {0,0,0,1}; /* initial seed for dlarnv() */
/* Initialize A */
dlarnv(&IONE, ISEED, &NxN, A);
/* Initialize B */
dlarnv(&IONE, ISEED, &NxNRHS, B);
in Fortran:
      INTEGER          I
      INTEGER          ISEED( 4 )
      EXTERNAL         DLARNV
      DO I = 1, 4
         ISEED( I ) = 1
      ENDDO
!-- Initialization of the matrix
      CALL DLARNV( 1, ISEED, N*N, A )

!-- Initialization of the RHS
      CALL DLARNV( 1, ISEED, N*NRHS, B )
```

Thirdly, we initialize PLASMA by calling the `PLASMA_Init` routine. The variable `cores` specifies the number of cores PLASMA will use.

```
in C:
int cores = 2;
/* Plasma Initialize */
INFO = PLASMA_Init(cores);
in Fortran:
```

4.2. PLASMA_DGESV EXAMPLE

```
INTEGER      CORES
PARAMETER    ( CORES = 2 )
INTEGER      INFO
EXTERNAL      PLASMA_INIT
! -- Initialize Plasma
CALL PLASMA_INIT( CORES, INFO )
```

Before we can call `PLASMA_dgesv`, we need to allocate some workspace necessary for the `PLASMA_dgesv` routine to operate. In `PLASMA`, each routine has its own routine to allocate its specific workspace arrays. Those routines are all defined in the `workspace.c` file. Their names are of the kind `PLASMA_Alloc_Workspace_` with the name of the associated routine in lower case for C and `PLASMA_ALLOC_WORKSPACE_` with the name of the associated routine in upper case for Fortran. You will also need to check the interface since they are different from one routine to the other. For the `PLASMA_dgesv` routine, two arrays need to be initialized, the workspace (here it is called `L` in the C code and `HL` in the Fortran Code) and the pivots (here it is called `IPIV` in the C code and `HIPIV` in the Fortran Code). Note that in C, you need to use standard pointers; while in Fortran, you need to use handle defined as an array of two elements of `INTEGER*4`.

```
in C:
double *L;
int *IPIV;
PLASMA_Alloc_Workspace_dgesv(N, &L, &IPIV);
in Fortran:
INTEGER*4      HL( 2 ), HIPIV( 2 )
EXTERNAL       PLASMA_ALLOC_WORKSPACE_DGESV
! -- Allocate L and IPIV
CALL PLASMA_ALLOC_WORKSPACE_DGESV( N, HL, HIPIV, INFO )
```

Finally, we can call the `PLASMA_dgesv` routine. You can report to the header of the routine for a complete description of the arguments. The description is also available online in the `PLASMA` routine description html page: <http://icl.cs.utk.edu/projectsfiles/plasma/html/routine.html>. As in LAPACK, `PLASMA` is returning a return variable, `INFO`, that indicates if the exit was successful or not. A successful exit is indicated by `INFO` equal to 0. In a case of `INFO` different from 0, the value of `INFO` should help you to understand the reason of the failure. (See the return value section in the above cited documentation.)

```
in C:
/* Solve the problem */
INFO = PLASMA_dgesv(N, NRHS, A, N, L, IPIV, B, N);
if (INFO < 0)
```

4.2. PLASMA_DGESV EXAMPLE

```
        printf("-- Error in DGESV example ! \n");
    else
        printf("-- Run successful ! \n");
in Fortran:
! -- Perform the LU solve
    CALL PLASMA_DGESV( N, NRHS, A, N, HL, HPIV, B, N, INFO )
    IF ( INFO < 0 ) THEN
        WRITE(*,*) " -- Error in DGESV example !"
    ELSE
        WRITE(*,*) " -- Run successful !"
    ENDIF
```

Before exiting the program, we need to free the resource used by our arrays and finalize PLASMA. To deallocate the C array, a simple call to `free` is needed whereas in Fortran, PLASMA provides the routine `PLASMA_DEALLOC_HANDLE` to do so. `PLASMA_Finalize` call will free all the internal allocated variables and finalize PLASMA.

```
in C:
    /* Deallocate L and IPIV */
    free(L); free(IPIV);
    /* Plasma Finalize */
    INFO = PLASMA_Finalize();
in Fortran:
! -- Deallocate L and IPIV
    CALL PLASMA_DEALLOC_HANDLE( HL, INFO )
    CALL PLASMA_DEALLOC_HANDLE( HPIV, INFO )
!-- Finalize Plasma
    CALL PLASMA_FINALIZE( INFO )
```

CHAPTER 5

Performance of PLASMA

5.1 A Library for Multicore Architectures

To achieve high performance on multicore architectures, PLASMA relies on tile algorithms, which provide fine granularity parallelism. The standard linear algebra algorithms can then be represented as Directed Acyclic Graphs (DAG) where nodes represent tasks and edges represent dependencies among them. Our programming model enforces asynchronous, out of order scheduling of operations. This concept is used as the basis for a scalable yet highly efficient software framework for computational linear algebra applications.

In LAPACK, parallelism is obtained through the use of multithreaded *Basic Linear Algebra Subprograms* (BLAS). In PLASMA, parallelism is no longer hidden inside the BLAS but is brought to the fore to yield much better performance.

PLASMA performance strongly depends on tunable execution parameters trading off utilization of different system resources. The outer block size (NB) trades off parallelization granularity and scheduling flexibility with single core utilization, while the inner block size (IB) trades off memory load with extra-flops.

PLASMA is currently scheduled statically with a trade off between load balancing and data reuse.

5.2 Comparison to other libraries

We present here the performance of the three following one sided factorizations: Cholesky, QR, and LU. We compare PLASMA against the two established linear algebra packages LAPACK and ScaLAPACK. The experiments were conducted on two different multicore architectures based on Intel Xeon EMT64 and IBM Power6.

PLASMA, LAPACK and ScaLAPACK are all linked with the optimized vendor BLAS available on the system provided within Intel MKL 10.1 and IBM ESSL 4.3 on the Intel64 and Power6 architectures, respectively. The first architecture is a quad-socket quad-core machine based on an Intel Xeon EMT64 E7340 processor operating at 2.39 GHz. Its theoretical peak is equal to 9.6 Gflop/s/ per core or 153.2 Gflop/s for the whole node (16 cores). The second architecture is a SMP node composed of 16 dual-core Power6 processors. Each dual-core Power6 processor runs at 4.7 GHz, leading to a theoretical peak of 18.8 Gflop/s per core and 601.6 Gflop/s per node (32 cores).

We only report double precision performance numbers for simplicity purposes. PLASMA is tuned with the pruned search method as described in [1]. For ScaLAPACK, we have tuned the data distribution parameters (p,q,nb) as functions of the number of cores and the matrix size through an exhaustive search. For reference LAPACK, we have been using the default block size (no tuning).

ScaLAPACK and PLASMA interfaces allow the user to provide data distributed on the cores. In our shared-memory multicore environment, because we do not flush the caches, these libraries have thus the advantage to start the factorization with part of the data distributed on the caches. This is not negligible. For instance, a 8000×8000 double precision matrix can be held distributed on the L3 caches of the 32 cores of a Power6 node.

Figures 5.1, 5.2, 5.3 present performance for the Cholesky, QR and LU factorizations, respectively.

5.3 Tuning - Howto

Users willing to obtain good performance from PLASMA need to tune PLASMA parameters. The example code below illustrates how to change the NB and IB parameters before calling the appropriate PLASMA routine to factorize or solve a linear system.

We recall that QR and LU algorithms requires both NB and IB parameters while Cholesky needs only NB.

```
...
    /* Plasma Tune */
    PLASMA_Disable(PLASMA_AUTOTUNING);
```

5.3. TUNING - HOWTO

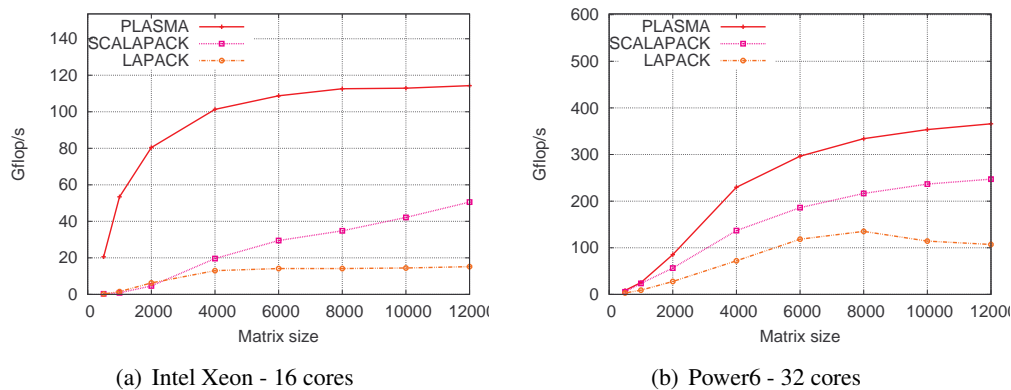


Figure 5.1: Performance of the Cholesky factorization (Gflop/s).

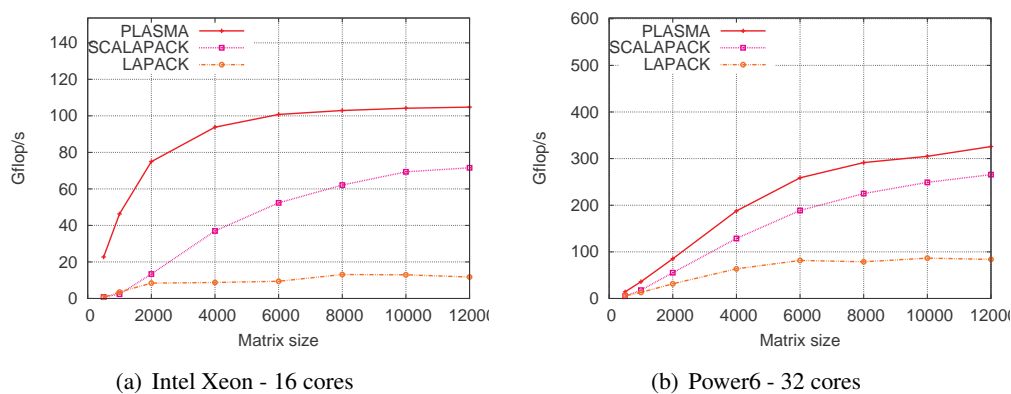


Figure 5.2: Performance of the QR factorization (Gflop/s).

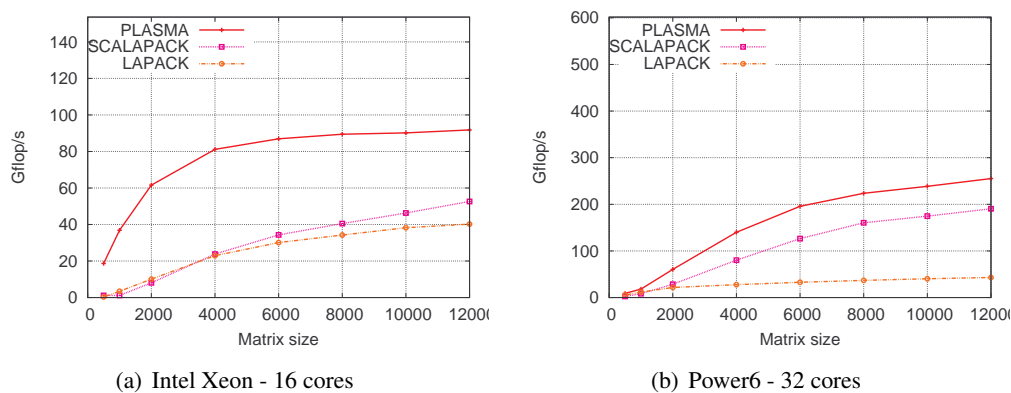


Figure 5.3: Performance of the LU factorization (Gflop/s).

5.3. TUNING - HOWTO

```
PLASMA_Set(PLASMA_TILE_SIZE, NB);  
PLASMA_Set(PLASMA_INNER_BLOCK_SIZE, IB);  
...
```

A pruned search method to obtain “good” parameters is described in [\[1\]](#). We note that autotuning is part of the PLASMA’s roadmap; unfortunately, as of 2.0.0, the PLASMA software does not have its autotuning component available for release.

CHAPTER 6

Accuracy and Stability

We present backward stability results of the algorithms used in PLASMA. Backward stability is a concept developed by Wilkinson in [8, 9]. For more general discussion on accuracy and stability of numerical algorithms, we refer the users to Higham [7]. We follow Higham's notation and methodology here; in fact, we mainly present his results.

6.1 Notations

- We assume the same floating-point system as Higham [7]. His assumptions are fulfilled by the IEEE standard. The unit round-off is u which is about 1.11×10^{-16} for double.
- The absolute value notation $|\cdot|$ is extended from scalar to matrices where matrix $|A|$ denotes the matrix with (i, j) entry $|a_{ij}|$.
- Inequalities between matrices hold componentwise. If we write $A < B$, this means that for any i and any j , $a_{ij} < b_{ij}$ (A and B are of the same size).
- If $nu < 1$, we define $\gamma_n \equiv \frac{nu}{1-nu}$.
- Computed quantities are represented with an overline or with the $fl(\cdot)$ notation.
- Inequality with $|\cdot|$ can be converted to norms easily. (See [7, Lem.6.6].)

6.2 Peculiarity of the Error Analysis of the Tile Algorithms

Numerical linear algebra algorithms rely at their roots on inner products. A widely used result of error analysis of the inner product is given in Theorem 6.2.1

Theorem 6.2.1 (Higham, [7, Eq.(3.5)]) *Given x and y , two vectors of size n , if $x^T y$ is evaluated in floating-point arithmetic, then, no matter what the order of evaluation, we have*

$$|x^T y - fl(x^T y)| \leq \gamma_n |x|^T |y|.$$

While there exists a variety of implementations and interesting research that aim to reduce errors in inner products (see for example [7, chapter 3 and 4]), we note that Theorem 6.2.1 is given independently of the order of evaluation in the inner products. The motivation for being independent of the order of evaluation is that inner products are performed by optimized libraries which use associativity of the addition for grouping the operations in order to obtain parallelism and data locality in matrix-matrix multiplications.

Theorem 6.2.2 presents a remark from Higham. Higham notes that one can significantly reduce the error bound of an inner product by accumulating it in pieces (which is indeed what an optimized BLAS library would do to obtain performance).

Theorem 6.2.2 (Higham, [7, §3.1]) *Given x and y , two vectors of size n , if $x^T y$ is evaluated in floating-point arithmetic by accumulating the inner product in k pieces of size n/k , then, we have*

$$|x^T y - fl(x^T y)| \leq \gamma_{n/k+k-1} |x|^T |y|.$$

Theorem 6.2.2 has been used by Castaldo, Whaley, and Chronopoulos [5] to improve the error bound in matrix-matrix multiplications.

The peculiarity of tile algorithms is that they explicitly work on small pieces of data and therefore, benefit in general from a *better* error bounds than their LAPACK counterparts.

6.3 Tile Cholesky Factorization

The Cholesky factorization algorithm in PLASMA performs the same operations than any version of the Cholesky. The organization of the operations in the inner products might be different from one algorithm to the other. The error analysis is however essentially the same for all the algorithms.

Theorem 6.3.1 presents Higham's result for the Cholesky factorization.

Theorem 6.3.1 (Higham, [7, Th.10.3]) *If Cholesky factorization applied to the symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$ runs to completion then the computed factor \bar{R} satisfies*

$$\bar{R}^T \bar{R} = A + \Delta A, \quad \text{where } |\Delta A| \leq \gamma_{n+1} |\bar{R}^T| |\bar{R}|.$$

Following Higham's proof, we note that Higham's Equation (10.4) can be tighten in our case since we know that the inner product are accumulated within tiles of size b . The γ_i term becomes $\gamma_{i/b+b-1}$ and we obtain the improved error bound given in Theorem 6.3.2.

Theorem 6.3.2 *If Tile Cholesky factorization applied to the symmetric positive definite matrix $A \in \mathbb{R}^{n \times n}$ runs to completion then the computed factors \bar{R} satisfies*

$$\bar{R}^T \bar{R} = A + \Delta A, \quad \text{where } |\Delta A| \leq \gamma_{n/b+b} |\bar{R}^T| |\bar{R}|.$$

We note that the error bound could even be made smaller by taking into account the inner blocking used in the Cholesky factorization of each tile.

Higham explains how to relate the backward error of the factorization with the backward error of the solution of a symmetric positive definite linear system of equations.

6.4 Tile Householder QR Factorization

The *Tile Householder QR* is backward stable since it is obtained through a product of backward stable transformations. One can obtain a tighter error bound with tile algorithms than with standard ones.

Higham explains how to relate the backward error of the factorization with the backward error of the solution of a linear least squares.

6.5 Tile LU Factorization

Theorem 6.5.1 (Bientinesi and van de Geijn, [2, Th.6.5]) *Given $A \in \mathbb{R}^{n \times n}$, assume that the blocked right-looking algorithm in Fig. 6.1 completes. Then the computed factors \bar{L} and \bar{U} are such that*

$$\bar{L}\bar{U} = A + \Delta A, \quad \text{where } |\Delta A| \leq \gamma_{n/b+b} (|A| + |\bar{L}||\bar{U}|).$$

We note that Bientinesi and van de Geijn do not consider permutations.

Higham explains how to relate the backward error of the factorization with the backward error of the solution of a linear system of equations.

Words of cautions: It is important to note that Theorem 6.5.1 does not state that the algorithm is stable. For the algorithm to be stable, we need to have

$$|\overline{L}||\overline{U}| \sim |A|.$$

Whether this is the case or not, this is still an ongoing research topic. Therefore, we recommend users to manipulate PLASMA_dgesv with care. In case of doubt, it is better to use PLASMA_dgels.

CHAPTER 7

Troubleshooting

7.1 Wrong Results

PLASMA is a software package distributed in source code, subject to different hardware / software configurations. PLASMA may deliver wrong numerical results due to a number of problems outside of PLASMA, such as:

- aggressive compiler optimizations violating code correctness,
- aggressive compiler optimizations violating IEEE floating point standard,
- hardware floating point arithmetic implementations violating IEEE standard,
- ABI differences between compilers, if mixing compilers,
- aggressive optimizations in BLAS implementations,
- bugs in BLAS implementations.

PLASMA is distributed with an installer with the intention to spare the user the process of setting up compilation and linking options. Nevertheless, it might become necessary for the user to do so. In such circumstances, the following recommendations should be followed.

7.1. WRONG RESULTS

When building PLASMA, it is recommended that dangerous compiler optimizations are avoided and instead flags enforcing IEEE compliance are used. It is generally recommended that “-O2” optimization level is used and not higher. Users are strongly cautioned against using different compilers for PLASMA and BLAS when building BLAS from source code. Users are also strongly advised to pay attention to the linking sequence and follow vendor recommendations, when vendor BLAS is used.

PLASMA software is tested daily by running a subset of LAPACK testing suite. Each pass involves hundreds of thousands of tests including both test for numerical results, as well as tests for detection of input errors, such as invalid input parameters. Currently the hardware / software configurations (in different combinations) known to pass all the tests include the following architectures:

7.1.1 Linux machine: Intel x86-64

C compiler	Fortran compiler	BLAS	testing	testing/lin
GNU gcc 4.1.2	GNU gfortran 4.1.2	ATLAS 3.8.1	PASSED	PASSED
GNU gcc 4.1.2	GNU gfortran 4.1.2	GotoBLAS 1.24	PASSED	info error: 160
GNU gcc 4.1.2	GNU gfortran 4.1.2	Intel MKL 11.0	PASSED	PASSED
GNU gcc 4.1.2	GNU gfortran 4.1.2	Reference BLAS	PASSED	PASSED
Intel icc 11.0	Intel ifort 11.0	GotoBLAS 1.24	PASSED PASSED PASSED	numerical failure: 87 (CPO, ZPO) illegal error: 675 info error: 188
Intel icc 11.0	Intel ifort 11.0	Intel MKL 11.0	PASSED	PASSED
Intel icc 11.0	Intel ifort 11.0	Reference BLAS	PASSED	PASSED

7.1.2 Linux machine: Intel 32

C compiler	Fortran compiler	BLAS	testing	testing/lin
GNU gcc 4.3.3	GNU gfortran 4.3.3	Reference BLAS	PASSED	PASSED
GNU gcc 4.3.3	GNU gfortran 4.3.3	Intel MKL 10.1	PASSED	PASSED
GNU gcc 4.3.3	GNU gfortran 4.3.3	ATLAS 3.8.1	PASSED	PASSED
GNU gcc 4.3.3	GNU gfortran 4.3.3	GotoBLAS 1.24	PASSED	ERROR
Intel icc 11.0	Intel ifort 11.0	Reference BLAS	PASSED	PASSED
Intel icc 11.0	Intel ifort 11.0	Intel MKL 10.1	PASSED	PASSED
Intel icc 11.0	Intel ifort 11.0	ATLAS 3.8.1	PASSED	PASSED
Intel icc 11.0	Intel ifort 11.0	GotoBLAS 1.24	PASSED	ERROR

7.1. WRONG RESULTS

7.1.3 Linux machine: AMD Opteron

C compiler	Fortran compiler	BLAS	testing	testing/lin
GNU gcc 4.1.2	GNU gfortran 4.1.2	Reference BLAS	PASSED	PASSED
GNU gcc 4.1.2	GNU gfortran 4.1.2	ACML 14.3.0	PASSED	PASSED
PATHSCALE pathcc 2.5	PATHSCALE pathf90 2.5	INTEL MKL 10.0.1	PASSED	PASSED
PORTLAND pgcc 8.0-6	PORTLAND pgf90 8.0-6	Reference BLAS	PASSED	PASSED

7.1.4 Linux machine: IBM Power6

C compiler	Fortran compiler	BLAS	testing	testing/lin
GNU gcc 4.3.1	GNU gfortran 4.3.2	Reference BLAS	PASSED	PASSED

7.1.5 Non-Linux machine

Machine	C compiler	Fortran compiler	BLAS	testing	testing/lin
MAC OS/X Leopard	GNU gcc 4.3.0	GNU gfortran 4.3.0	Reference BLAS	PASSED	PASSED
MAC OS/X Leopard	GNU gcc 4.3.0	GNU gfortran 4.3.0	Veclib framework	PASSED	PASSED
AIX 5.3	IBM XLC 10.1	IBM XLF 12.1	ESSL 4.3	PASSED	PASSED

Currently the hardware / software configurations known to fail PLASMA tests are:

- Intel x86-64, GCC, GFORTRAN, Goto BLAS,
- Intel x86-64, ICC, IFORT, Goto BLAS.

Bibliography

- [1] E. Agullo, B. Hadri, H. Ltaief and J. Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In the *Proceedings of SC'09: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2009
- [2] P. Bientinesi and R. A. van de Geijn. The science of deriving stability analyses. FLAME Working Note #33. 2009.
- [3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008. <http://dx.doi.org/10.1002/cpe.1301>.
- [4] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35:38–53, 2009. <http://dx.doi.org/10.1016/j.parco.2008.10.002>.
- [5] A. M. Castaldo, R. C. Whaley, and A. T. Chronopoulos. Reducing floating point error in dot product using the superblock family of algorithms. *SIAM J. Sci. Comput.*, 31(2):1156–1174, 2008. <http://dx.doi.org/10.1137/070679946>.
- [6] J. W. Demmel, N. J. Higham, and R. S. Schreiber. Stability of block *LU* factorization. *Numerical Linear Algebra with Applications*, 2(2):173–190, 1995. <http://dx.doi.org/10.1002/nla.1680020208>.
- [7] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*, Second Edition Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. xxx+680 pp. ISBN 0-89871-521-0 (hardback).

BIBLIOGRAPHY

- [8] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Notes on Applied Science No. 32. Her Majesty's Stationery Office, London, 1963. vi+161 pp. Also published by Prentice-Hall, Englewood Cliffs, NJ, USA. Reprinted by Dover, New York, 1994. ISBN 0-486-67999-3.
- [9] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965. xviii+662 pp. ISBN 0-19-853403-5 (hardback), ISBN 0-19-853418-3 (paperback).
- [10] S. Blackford and J. Dongarra. *Installation Guide for LAPACK*. LAPACK Working Note 41 (UT-CS-92-151). Department of Computer Science, University of Tennessee, Knoxville TN 37996, USA. March, 1992. <http://www.netlib.org/lapack/lawnspdf/lawn41.pdf>.