http://web.eecs.utk.edu/~kurzak/tutorials/

PLASMA / QUARK

Parallel Linear Algebra Software for Multicore Architectures

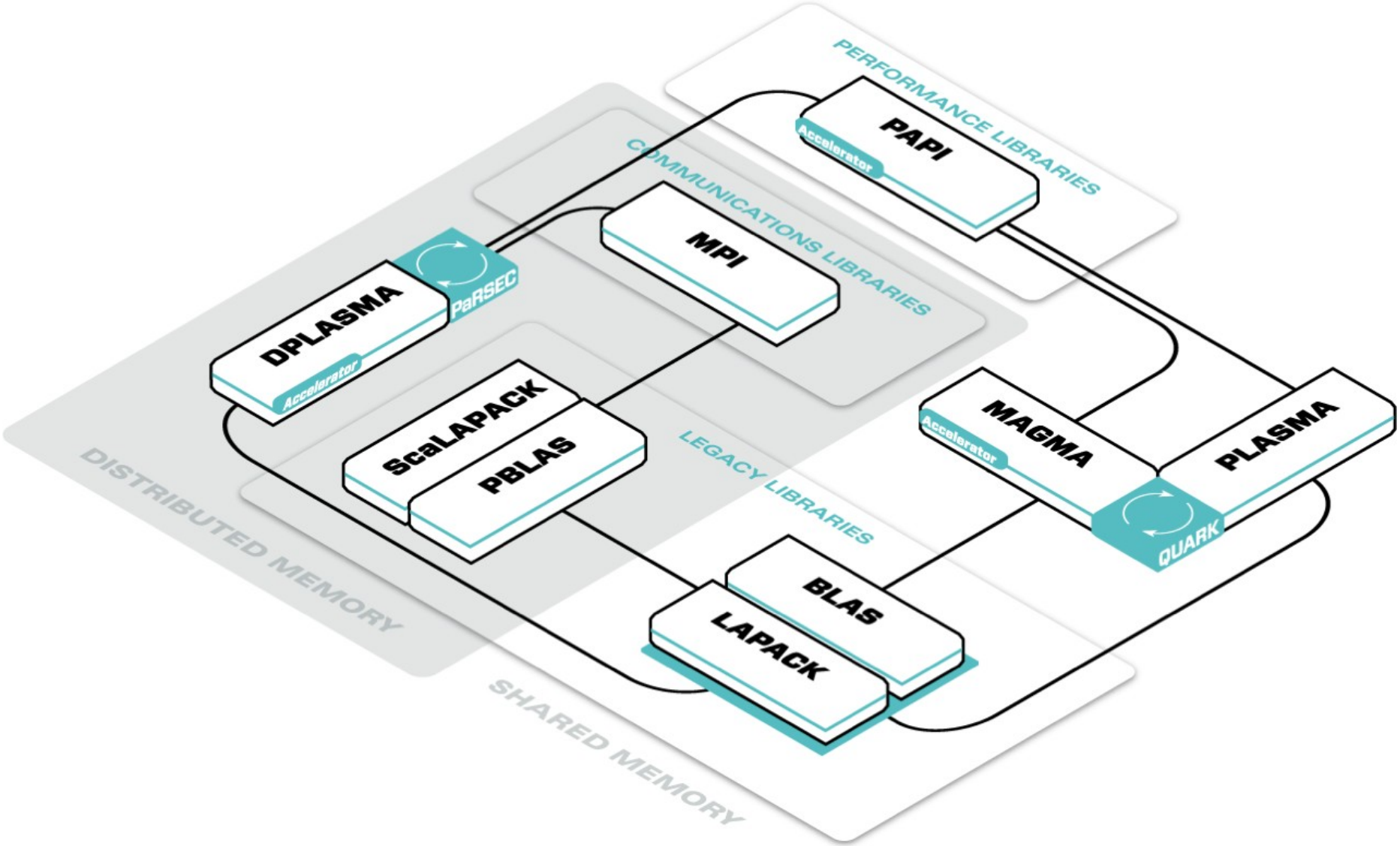
QUeuing And Runtime for Kernels

DPLASMA / PaRSEC

Distributed memory PLASMA

Parallel Runtime Scheduling and Execution Control

ICL Project Map



http://web.eecs.utk.edu/~kurzak/tutorials/

PLASMA

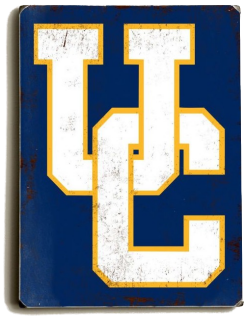
Parallel Linear Algebra Software for Multicore Architectures

TOC

- Functionality
- Performance
- Software Stack
- Matrix Layout
- Tile Algorithms
- Multithreading
- Naming conventions
- A word about the installer
- List of resources

Participants
and sponsors

Knoxville Tennessee



Berkeley California

Denver Colorado

Microsoft

FUJITSU

shaping tomorrow with you

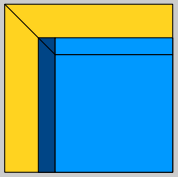
Developers
and contributors

- **Current Team**
  - Dulceneia Becker
  - Henricus Bouwmeester
  - Jack Dongarra
  - Mathieu Faverge
  - Mark Gates
  - Azzam Haidar
  - Blake Haugen
  - Jakub Kurzak
  - Julien Langou
  - Hatem Ltaief
  - Piotr Łuszczek
  - Ichitaro Yamazaki
  - Asim YarKhan
  - Vijay Joshi
- **Past Members**
  - Emmanuel Agullo
  - Wesley Alvaro
  - Alfredo Buttari
  - Bilel Hadri
- **Outside Contributors**
  - Fred Gustavson
  - Lars Karlsson
  - Bo Kågström

names listed alphabetically

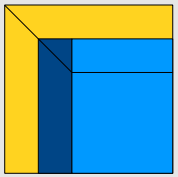
Dense Linear Algebra

software evolution



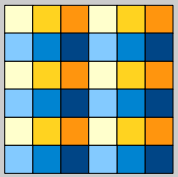
LINPACK (70's)
vector operations

- Level 1 BLAS



LAPACK (80's)
block operations

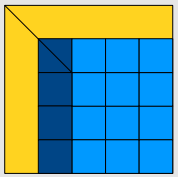
- Level 3 BLAS



ScaLAPACK (90's)
block cyclic
data distribution

- PBLAS
- BLACS

(message passing)



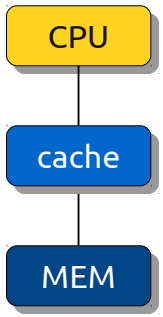
PLASMA (00's)
tile operations

- tile layout
- dataflow scheduling

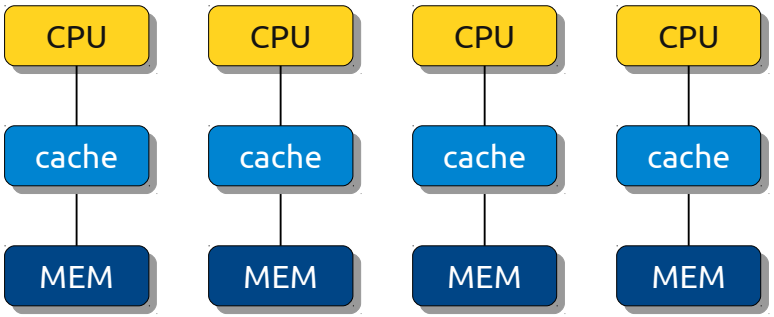
Dense Linear Algebra

architectural models

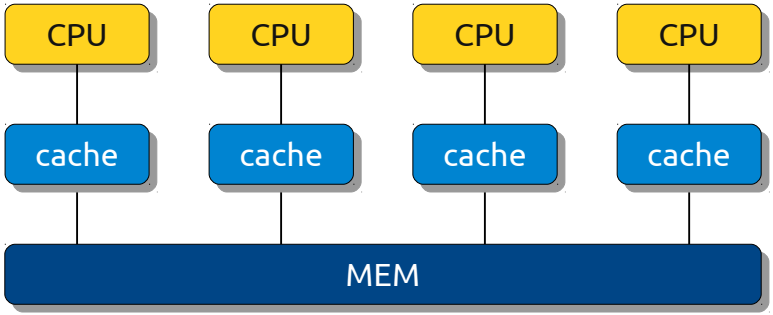
LAPACK



ScaLAPACK



PLASMA



PLASMA

in a nutshell

- **Software Library**
- **Dense Linear Algebra**
  - linear systems of equations
  - least square problems
  - singular value problems
  - eigenvalue problems
- **Multicore Systems**
  - multicore (e.g. 16-core AMD Bulldozer)
  - multi-socket (e.g. quad-socket Dell PowerEdge)
  - shared memory
  - possibly NUMA

hardware:
today – tens of cores
tomorrow – hundreds of cores

Linear Systems

AX = B

- **Fully Determined** **m = n**
  - PA = LU Gaussian elimination non-symmetric
  - A = LLT Cholesky factorization symmetric positive definite
  - A = LDLT LDLT decomposition non-symmetric positive definite
- **Overdetermined** **m > n** **(least squares)**
  - A = QR
- **Underdetermined** **m < n** **(minimum norm)**
  - A = LQ

Linear Systems

AX = B

- **Explicit Matrix Inversion**
  - non-symmetric
  - symmetric positive definite
  - explicitly calculating the inverse of a matrix
  - e.g., calculating the variance-covariance matrix in statistics
- **Tall and Skinny Factorizations** **m <<n** **m>>n**
  - fast (highly parallel) algorithms
  - based on tree reductions
- **Mixed Precision Iterative Refinement**
  - factorization in single precision
  - iterative refinement in double precision

Eigenvalues & SVD

$A = X\Lambda X^T$ $A = U\Sigma V$ $AX = \Lambda BX$

- **Symmetric Eigenvalue Problem**
  - reduction to “block tridiagonal” (band)
  - band reduction to tridiagonal (bulge chasing)
  - LAPACK eigensolver (QR/D&C/MRRR)

non-symmetric eigenvalue problem
work in progress

- **Singular Value Problem**
  - reduction to “block bidiagonal” (band)
  - band reduction to bidiagonal (bulge chasing)
  - QR algorithm (LAPACK)

extremely fast eigenvalues / singular values
fast eigenvectors / singular vectors
possibility of calculating a subset

- **Generalized Eigenvalue Problem**
  - Cholesky factorization of B and application to A
  - reduction to band & band reduction
  - LAPACK eigensolver (QR/D&C/MRRR)

symmetric positive definite matrices only

Additional Functionality

layout translation & tile level 3 BLAS

- **In-Place Matrix Layout Translation**
  - CM, RM (column-major / row-major)
  - CCRB, CRRB, RCRB, RRRB (column / row – rectangular block)
  - LAPACK ↔ PLASMA (CM ↔ CCRB)
  - any other combination
  - (fast transposition)
  - cache-efficient
  - parallel

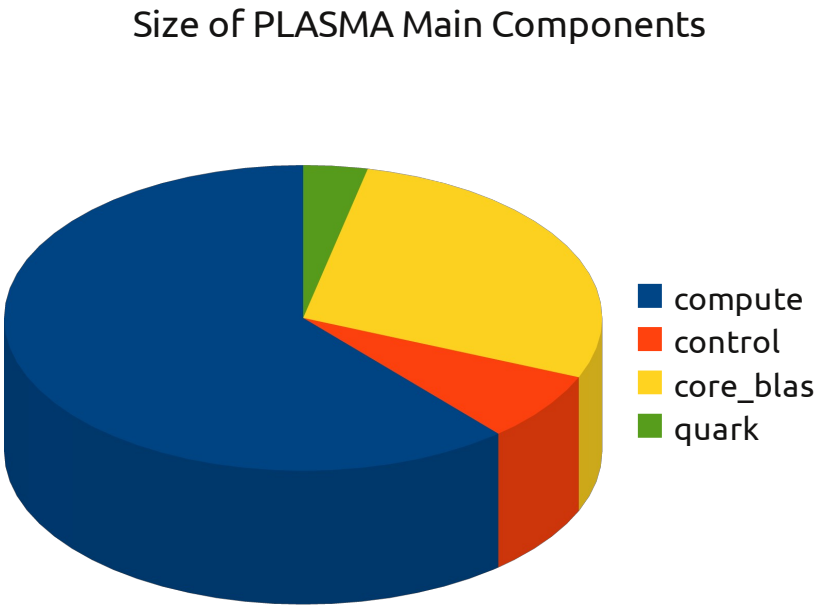
Thank you Fred, Lars, Bo!

There also is a fast out-of-place PLASMA ↔ LAPACK translation

- **Tile BLAS 3**
  - GEMM, HEMM, HER2K, HERK, SYMM, SYR2K, SYRK, TRMM, TRSM (complete set)

PLASMA Size

in numbers

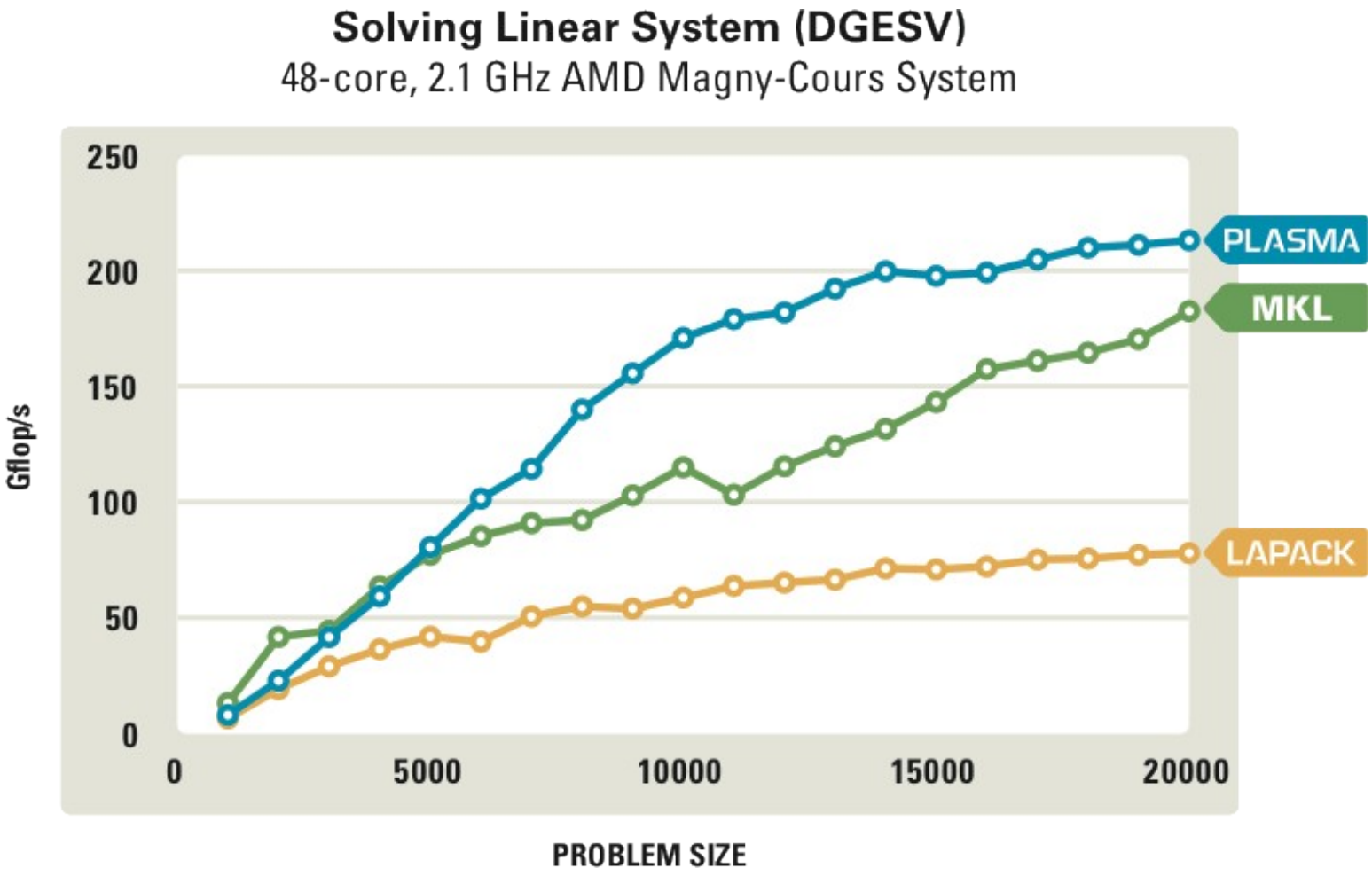


~200,000	lines of code
~1,000	source files
~600	API functions

Performance

solving a linear system using LU factorization

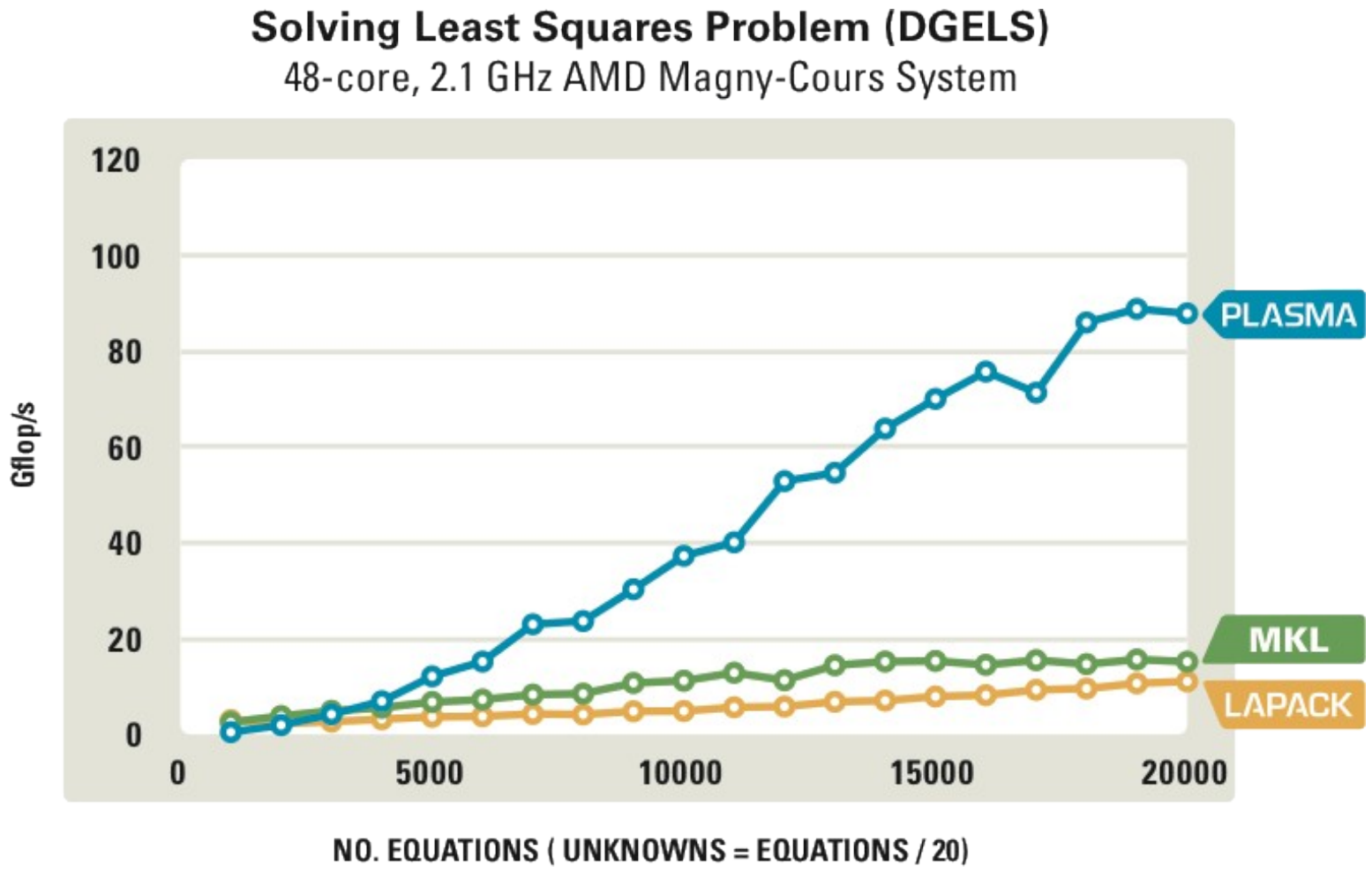
partial pivoting



Performance

solving a least squares problem using QR factorization

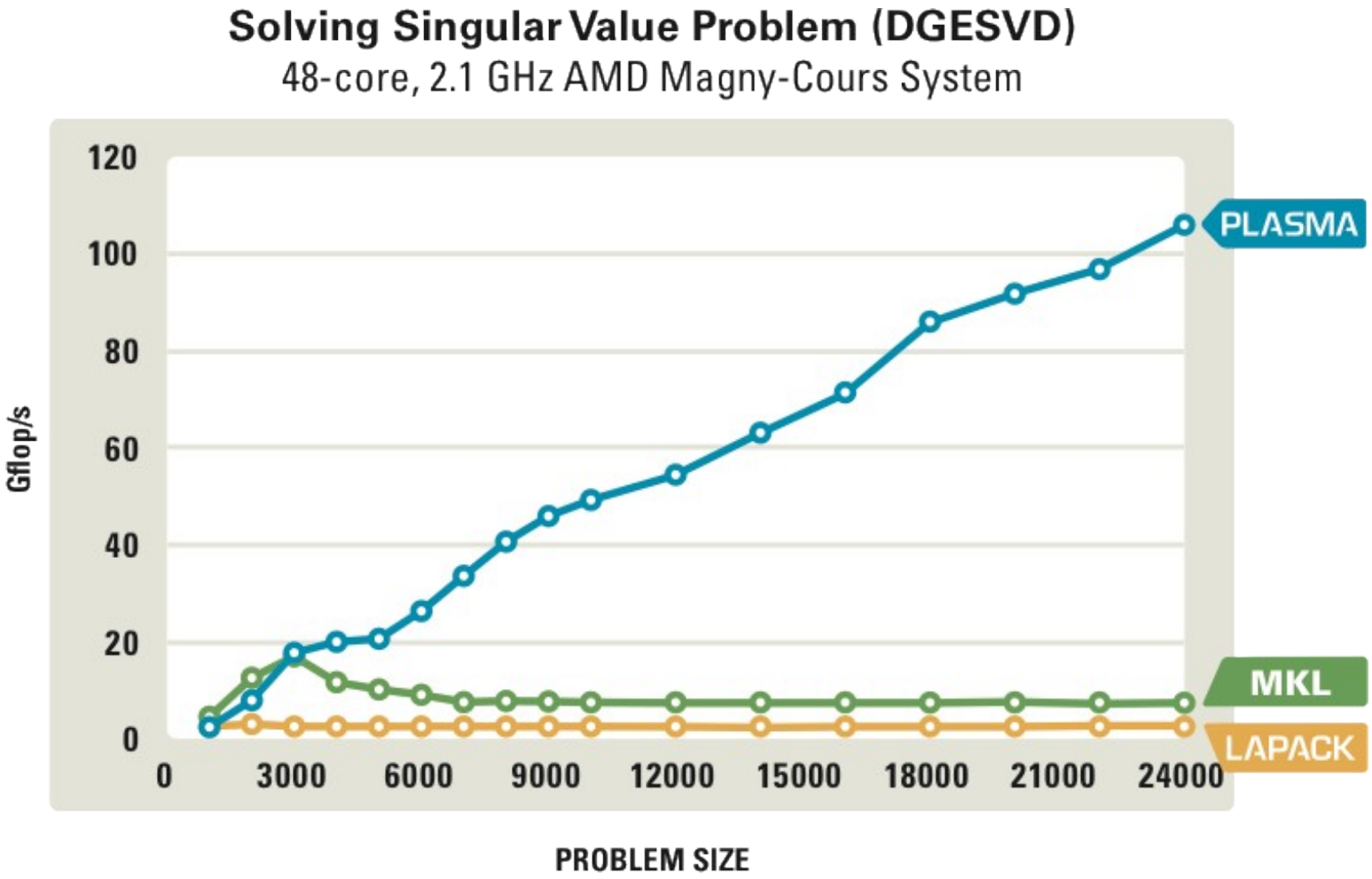
tree-based



Performance

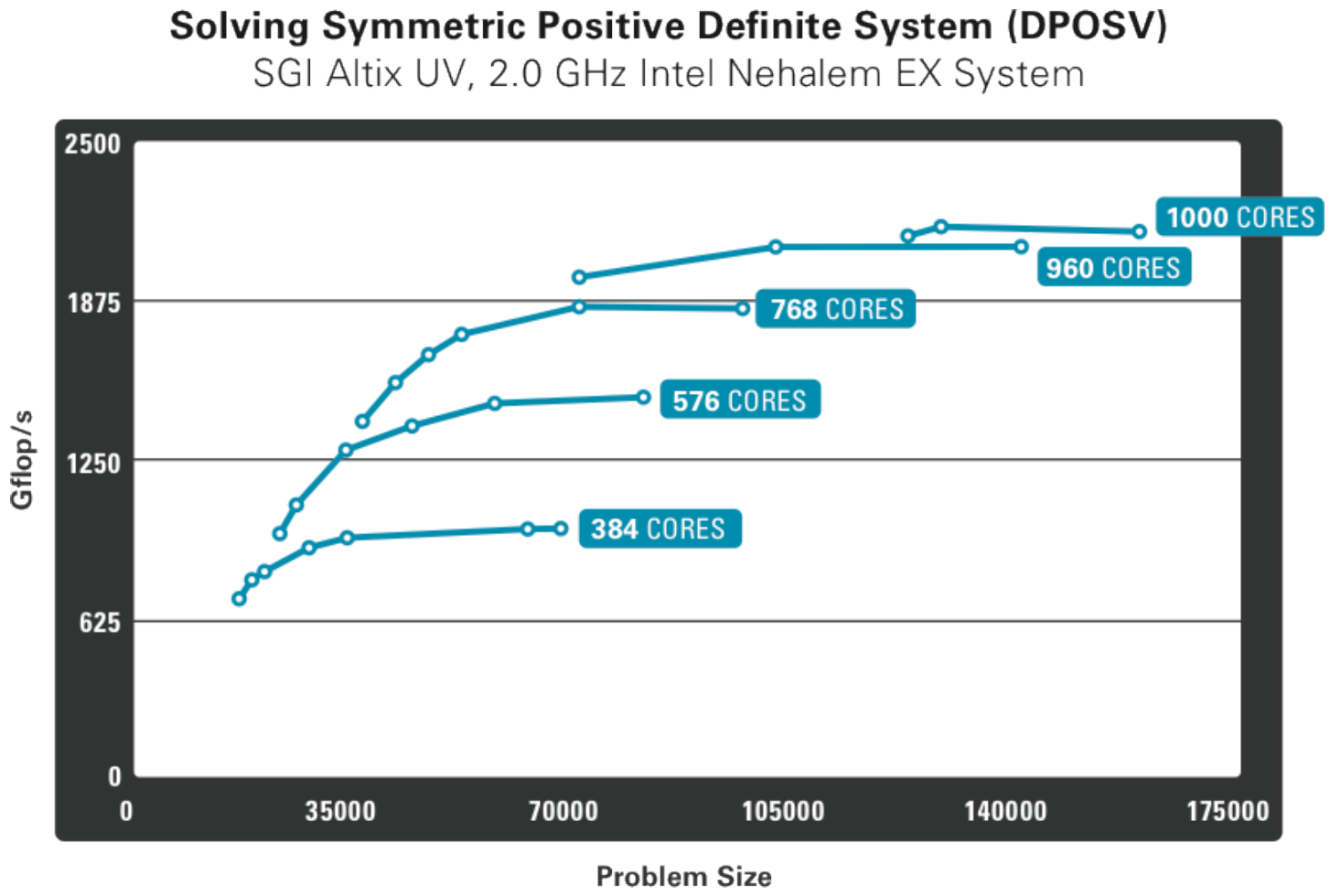
solving a singular value problem

no vectors



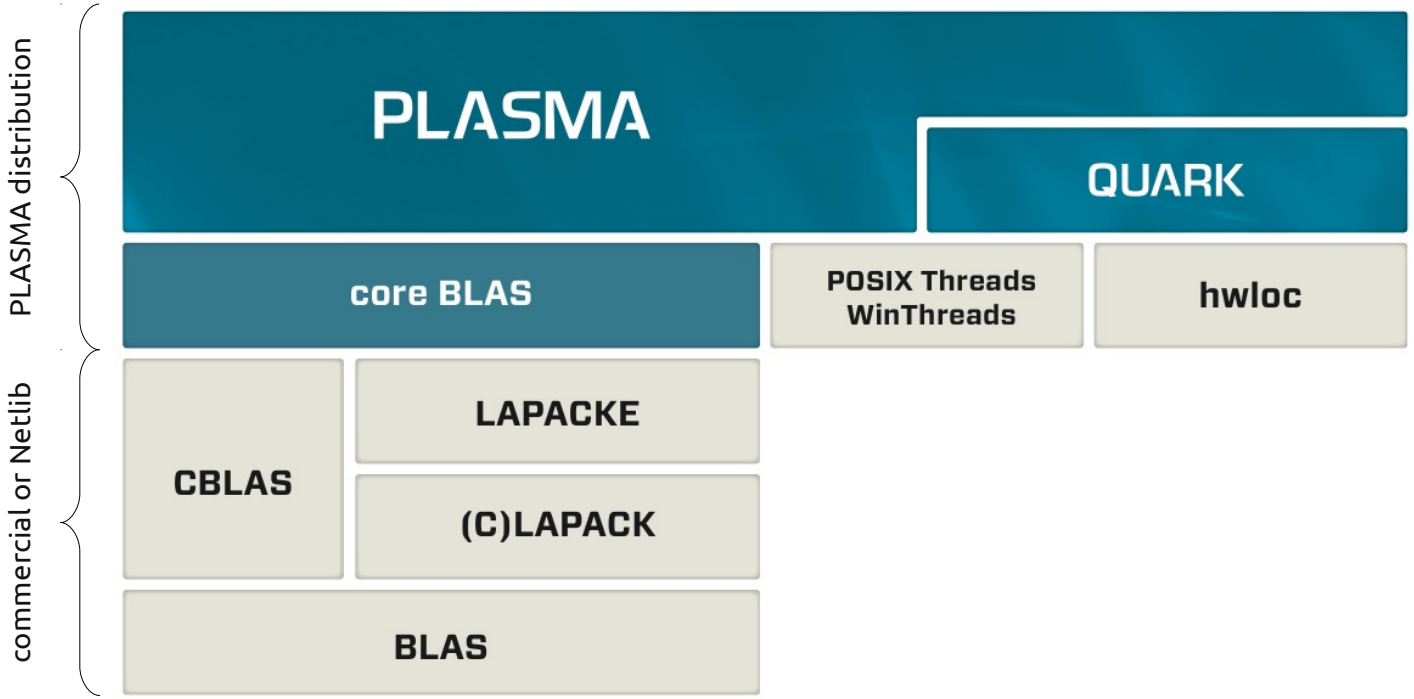
Performance

Cholesky factorization on 1000 cores



PLASMA

software stack



QUARK - **QU**euing **A**nd **R**untime for **K**ernels

LAPACK - **L**inear **A**lgebra **PACK**age

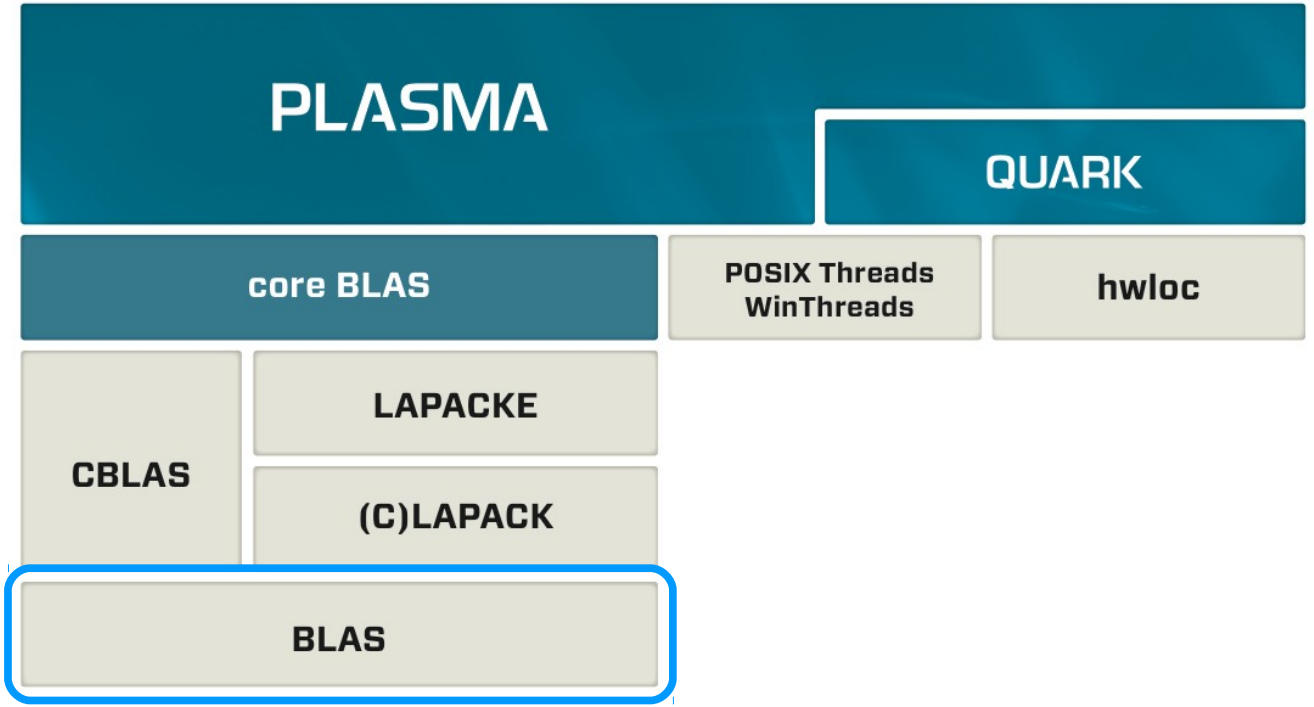
BLAS - **B**asic **L**inear **A**lgebra **S**ubroutines

hwloc - **h**ard**w**are **loc**ality

LAPACKE is now included in the Netlib LAPACK

PLASMA

software stack



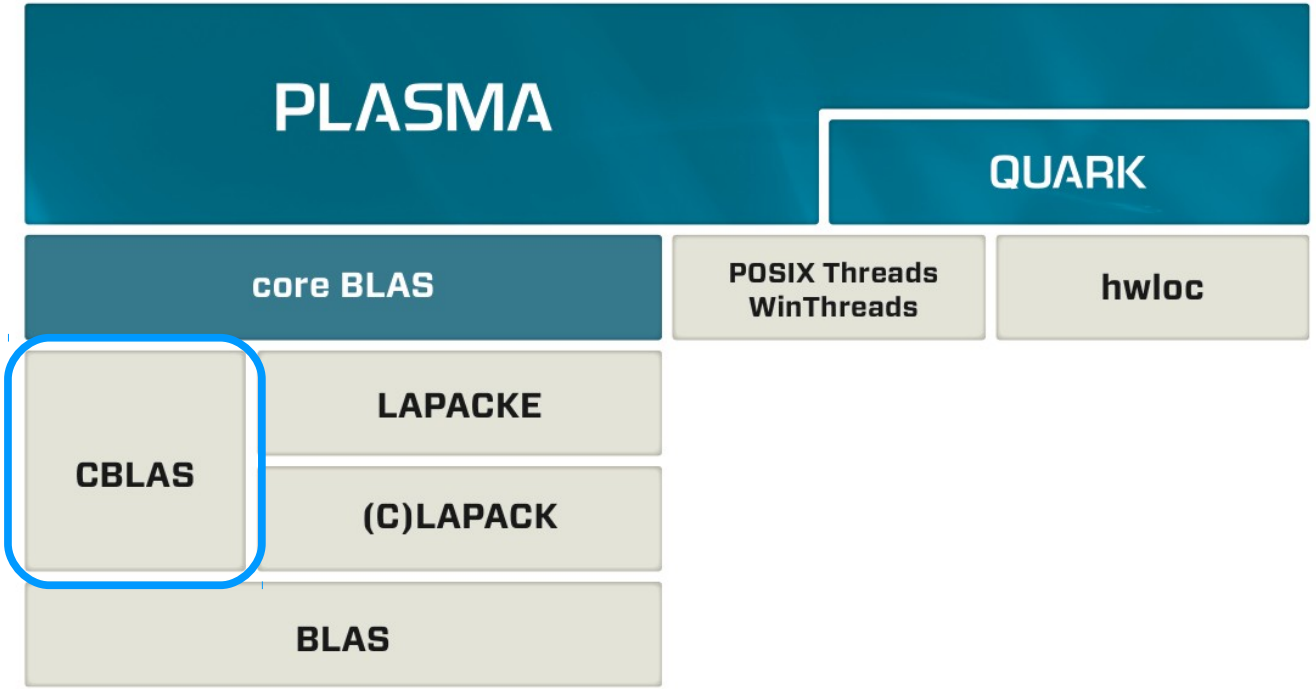
- **Basic Linea Algebra Subroutines**
  - critical for performance
  - vendor: Intel MKL, AMD ACML, IBM ESSL, Apple VecLib, Cray LibSci
  - academic: ATLAS, Goto BLAS
  - FORTRAN *definition* available from Netlib

DO NOT USE

www.netlib.org/blas/

PLASMA

software stack

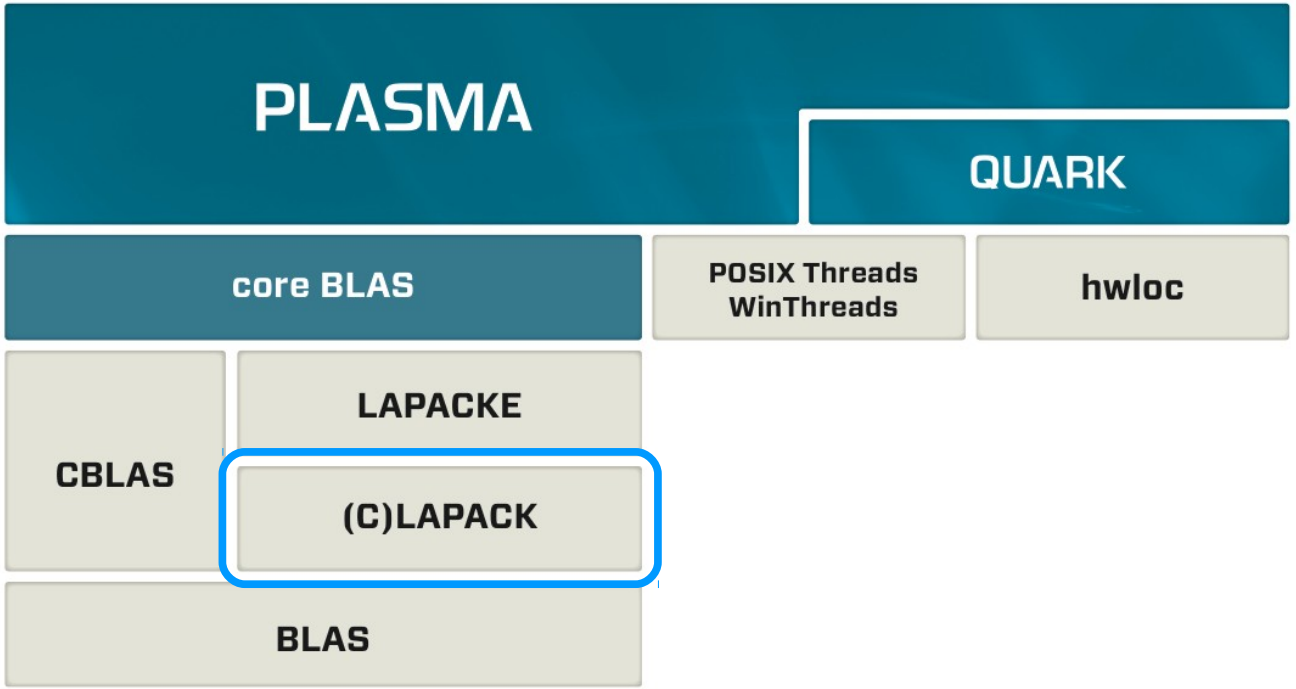


- **C API for BLAS**
  - a set of wrappers
  - not performance critical
  - commonly included in vendor and academic packages
  - available from Netlib

http://www.netlib.org/blas/blast-forum/cblas.tgz

PLASMA

software stack

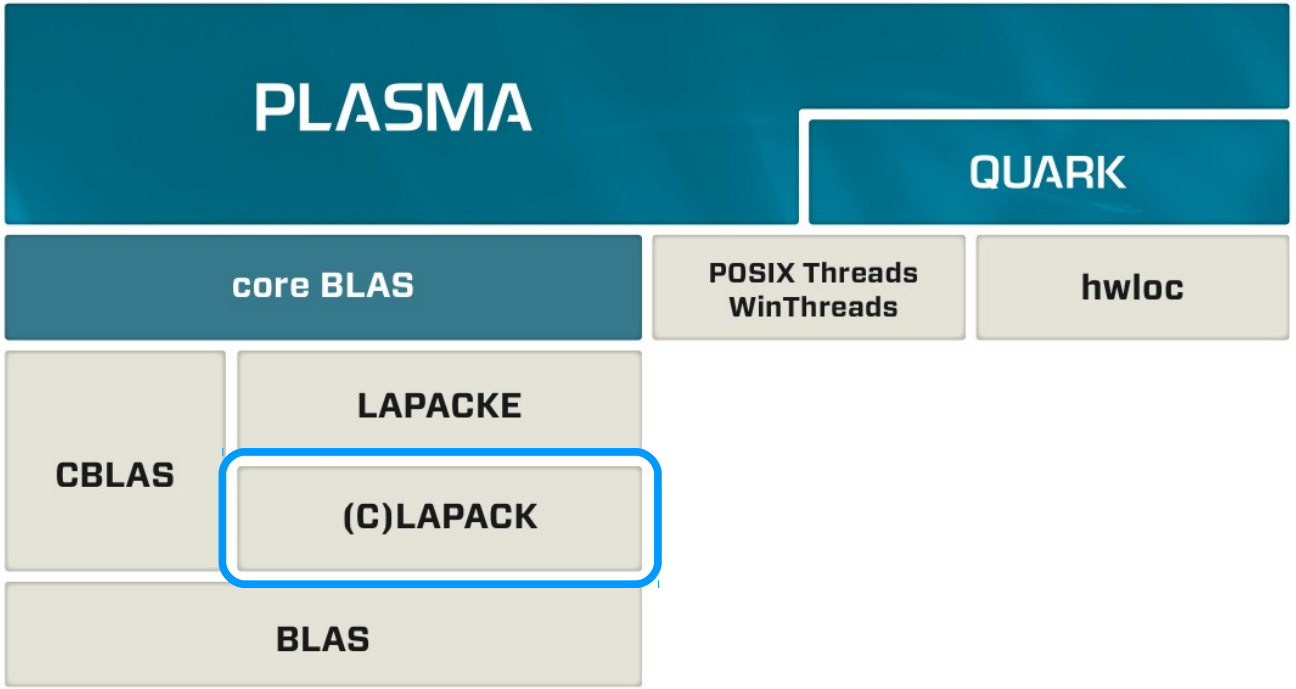


- **Linear Algebra PACKage**
  - dense linear algebra software library
  - FORTRAN implementation
  - no parallel constructs
  - available from Netlib

www.netlib.org/lapack/

PLASMA

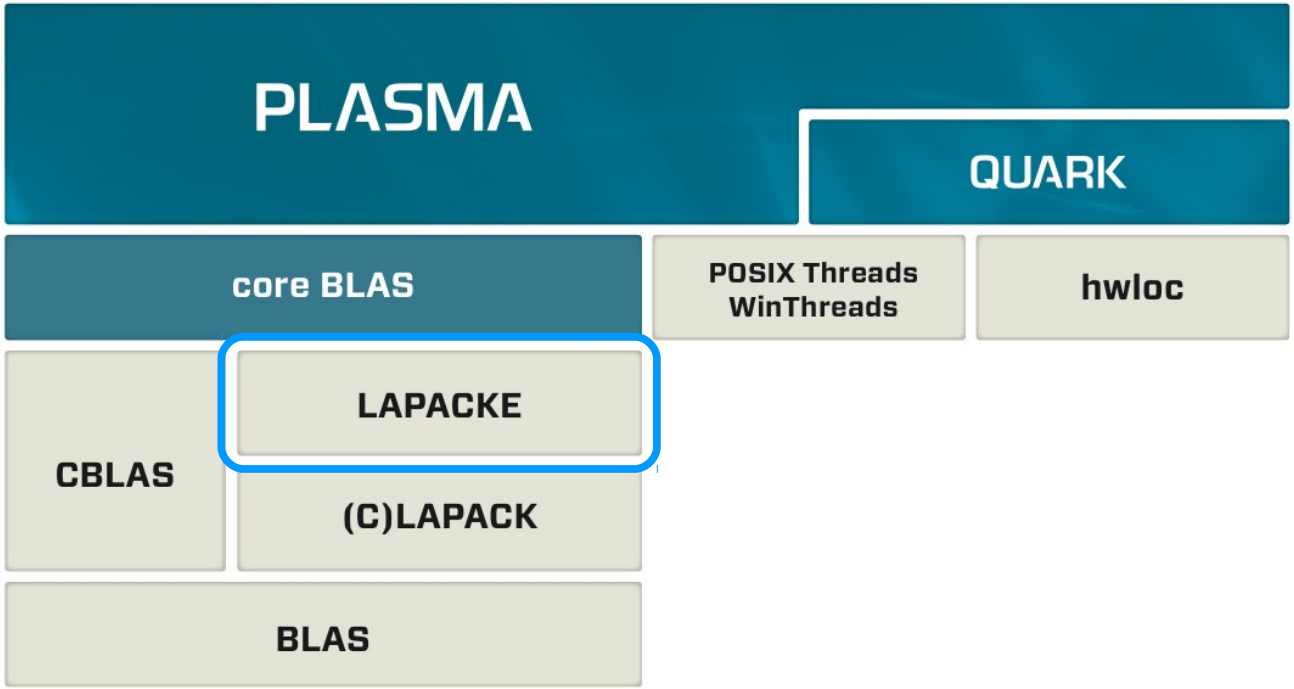
software stack



- **C Implementation of LAPACK**
  - automatically generated
  - C implementation
  - FORTRAN API
  - available from Netlib

DO NOT USE

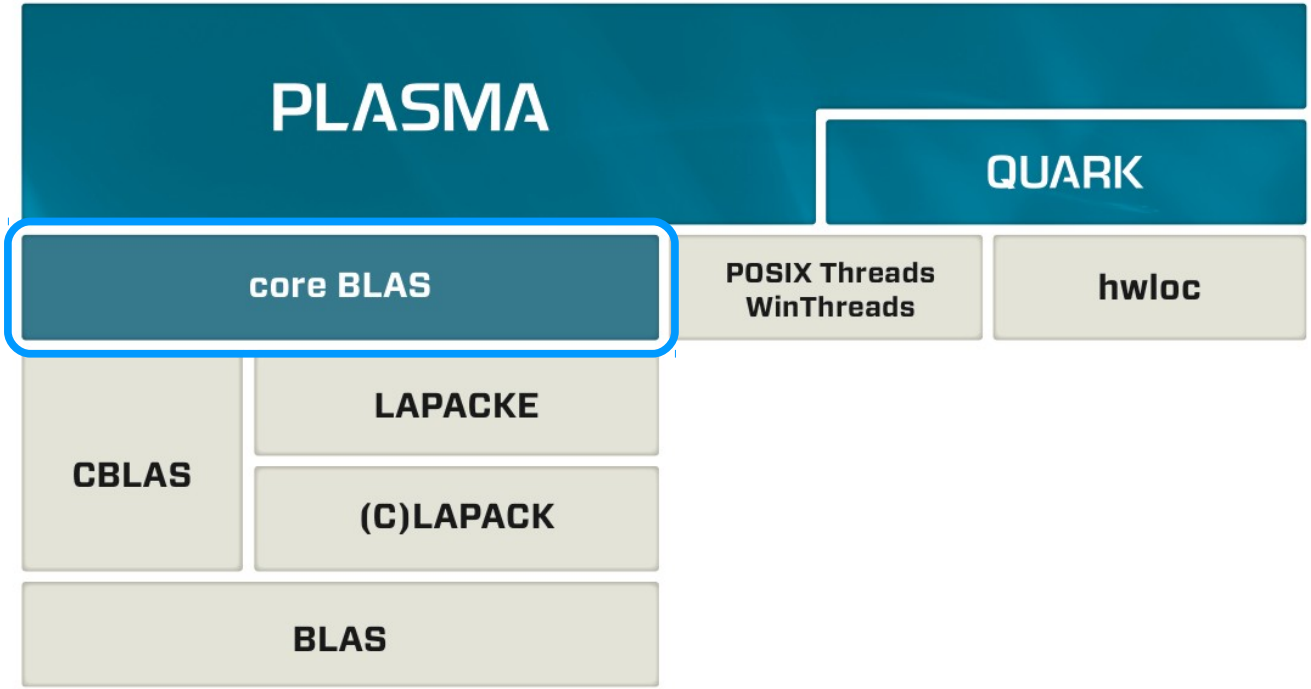
www.netlib.org/clapack/



- **C API for LAPACK**
  - developed by: LAPACK team & Intel
  - comes with LAPACK from Netlib
  - comes with MKL

PLASMA

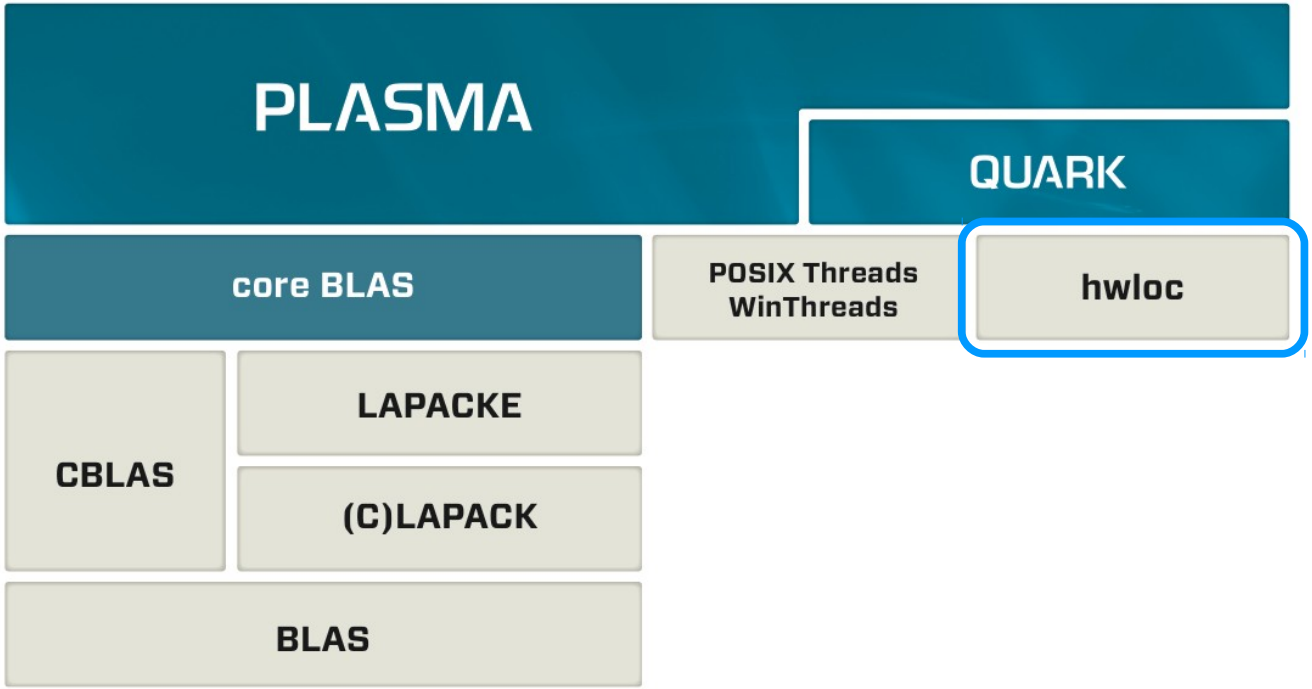
software stack



- **core BLAS**
  - serial kernels
  - PLASMA's building blocks
  - ideally monolithic & highly tuned
  - currently implemented with multiple calls to BLAS

PLASMA

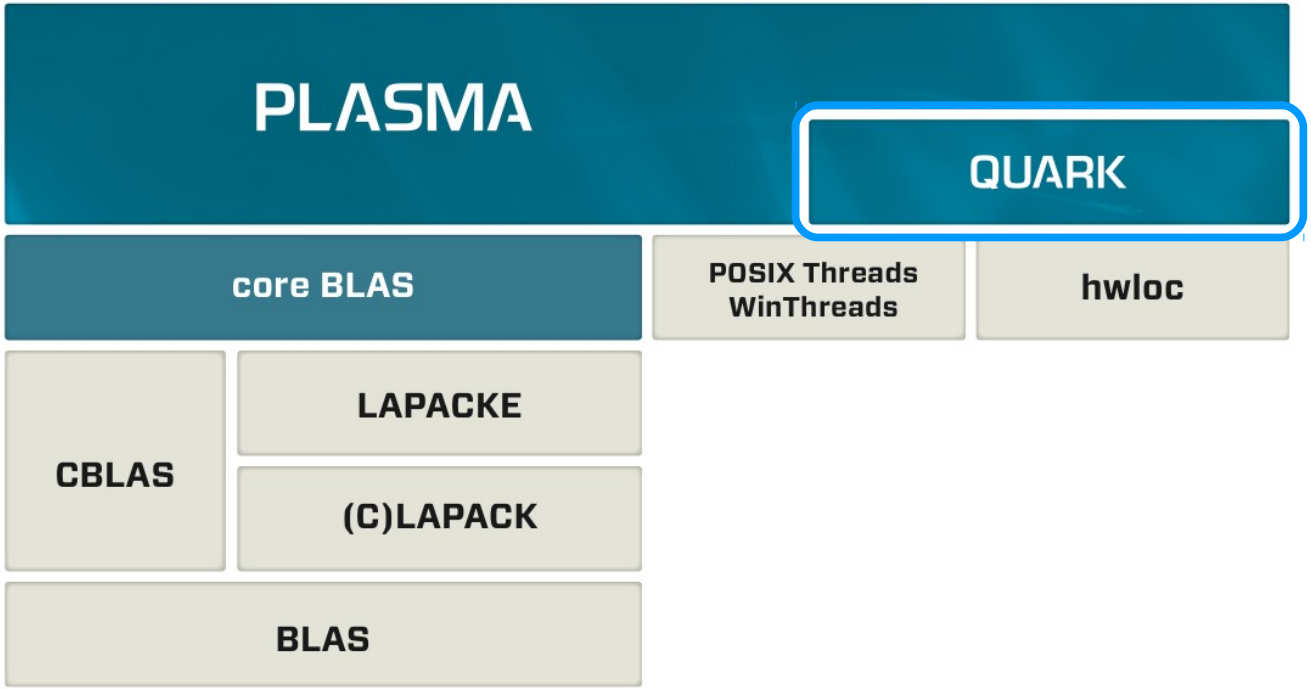
software stack



- **hwloc – Portable Hardware Locality**
  - topology abstraction
  - caches / NUMA nodes / sockets / cores / SMT
  - in PLASMA thread affinity control (thread placement)

PLASMA

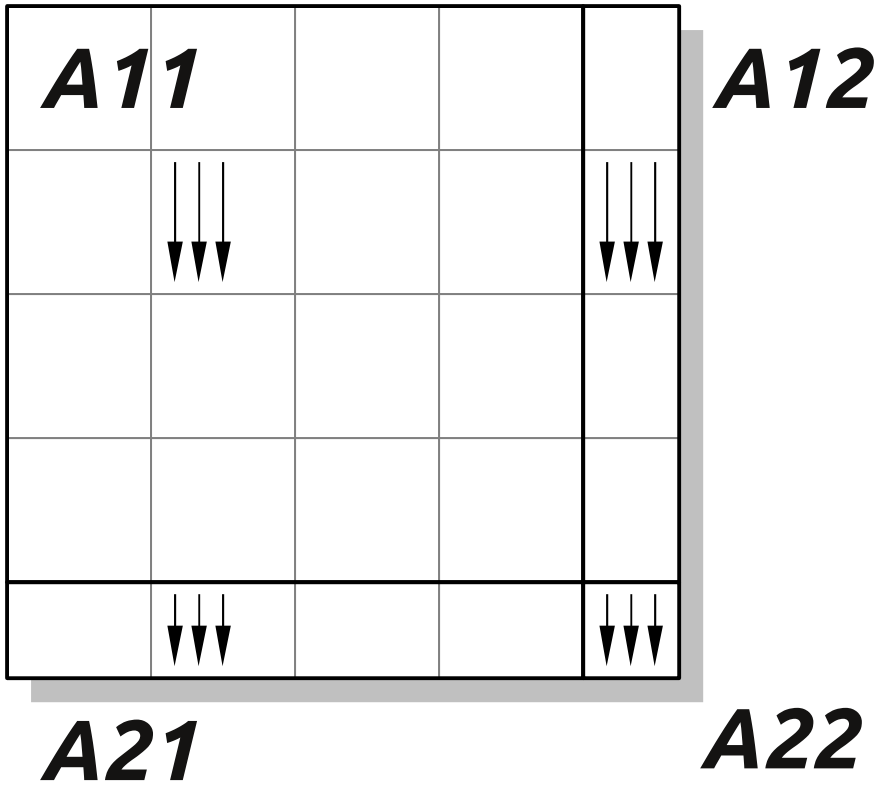
software stack



- **Queuing and Runtime for Kernels**
  - API-based
  - superscalar scheduler
  - similar to StarPU and SMPSs
  - suitable for implementing a numerical library

PLASMA

matrix layout

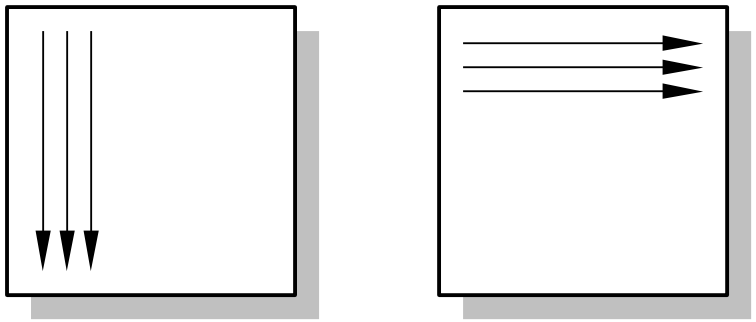


- One continuous chunk of memory containing (in order): A11, A21, A12, A22
- Each tile is continuous (column-major order)
- Tiles are stored in column-major order (A11)

Fred Gustavson

PLASMA

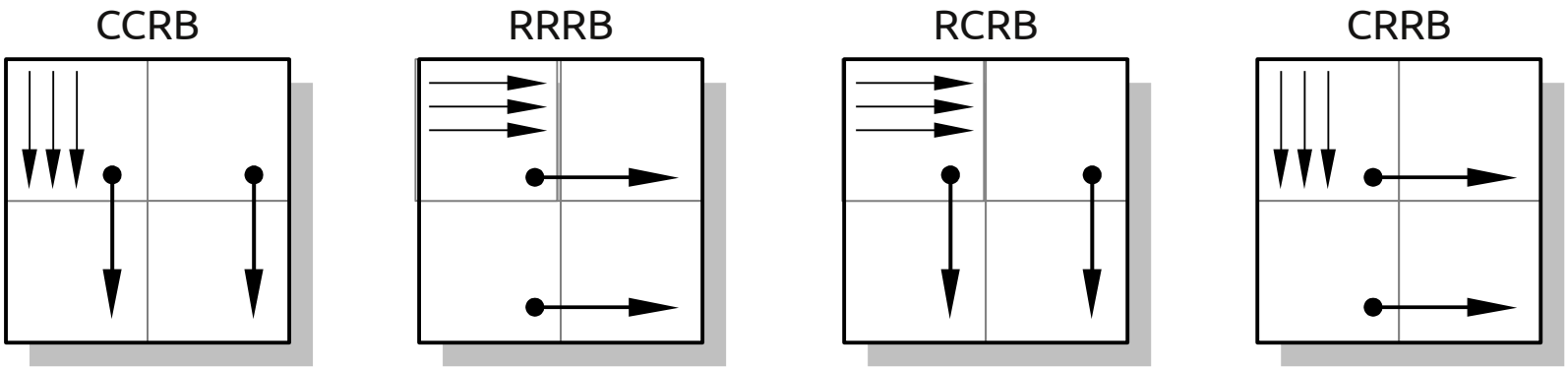
in-place translation routines



Column Major / Row Major

```
PLASMA_sgecfi()
PLASMA_cgecfi()
PLASMA_dgecfi()
PLASMA_zgecfi()
```

C|R C|R Rectangular Block

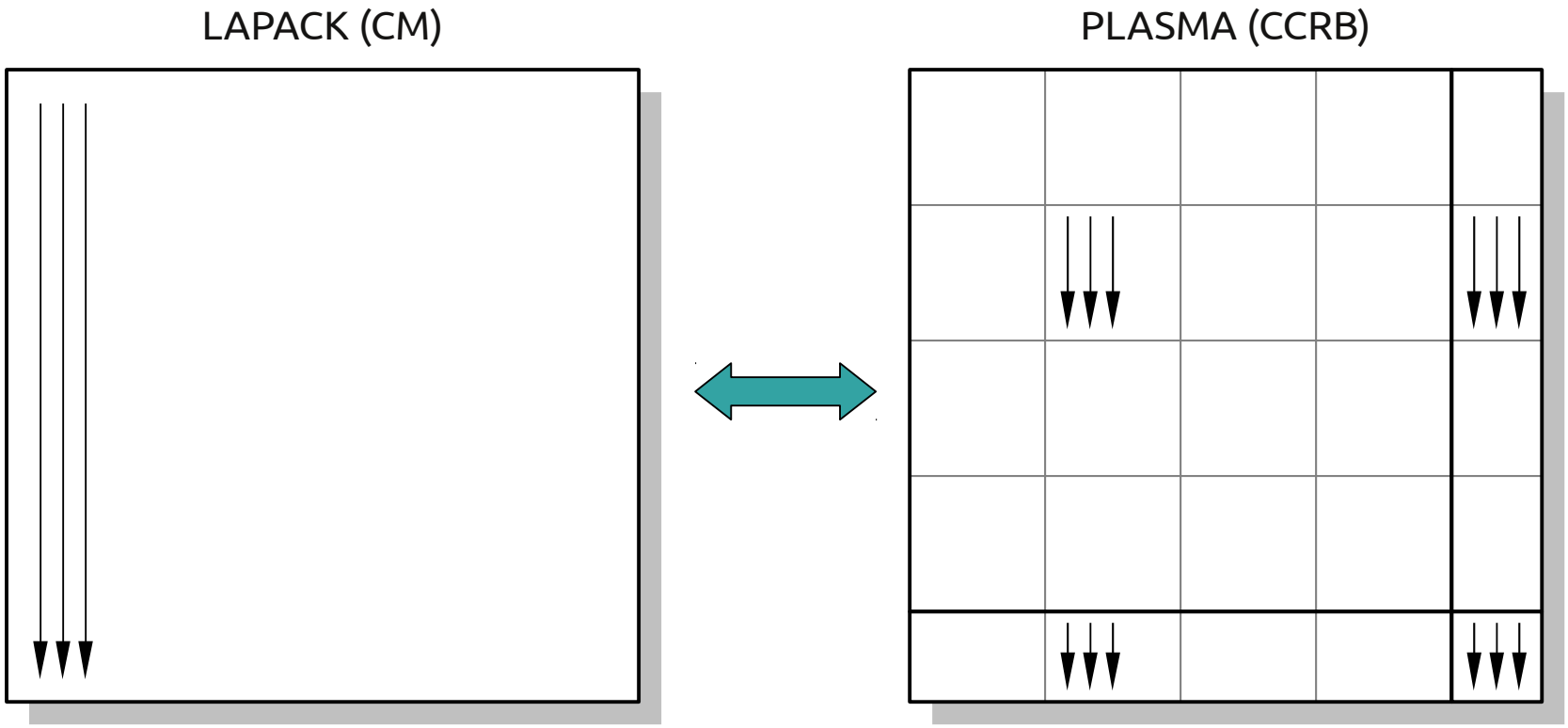


- PLASMA provides translation between each two layouts
- PLASMA only accepts input matrices in the CM and CCRB layouts (square tiles only)

Fred Gustavson

PLASMA

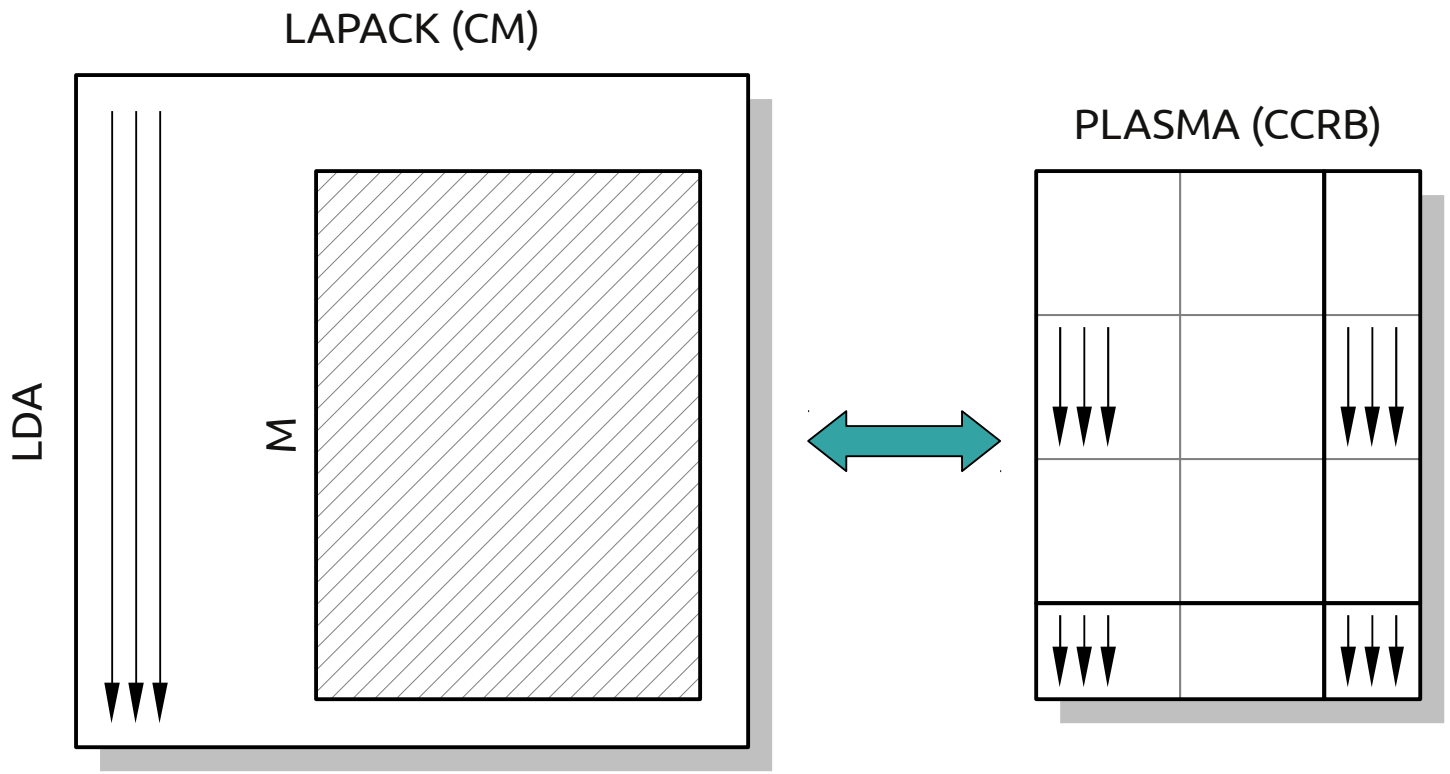
translation between LAPACK and PLASMA



- **Tile layout & LAPACK layout accepted**
- **LAPACK layout always converted to tile layout (CCRB / square tiles)**
  - in place translation
  - out of place translation

PLASMA

out-of-place translation

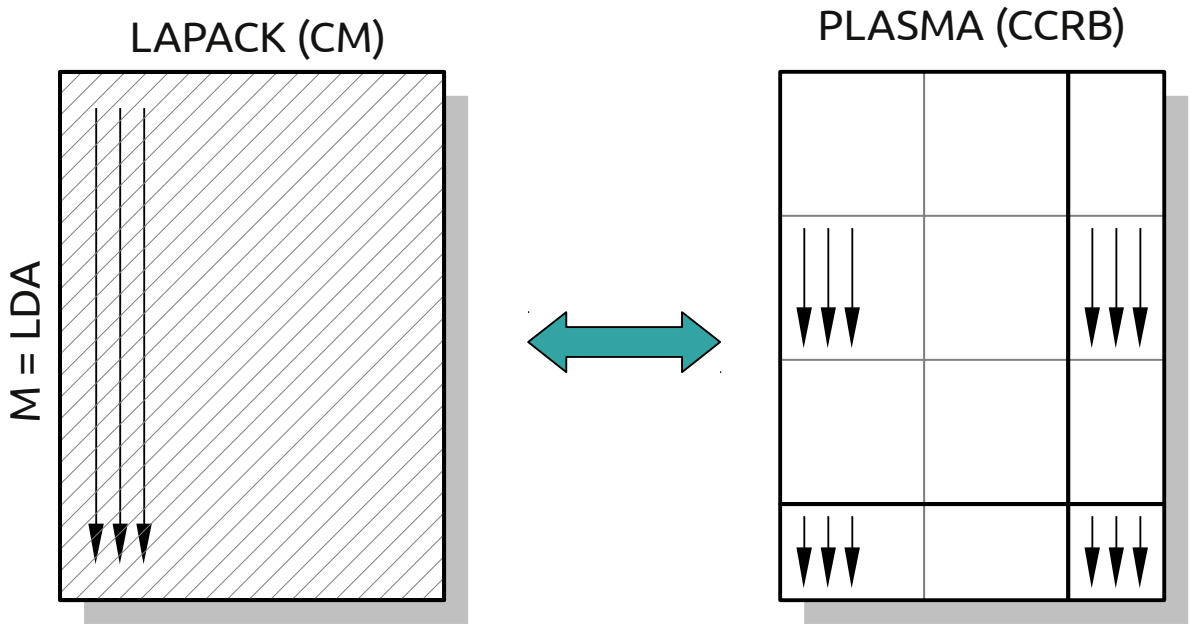


- **Out-of-Place Translation**
  - M <= LDA
  - CM ↔ CCRB only
  - storage overhead
  - fast (parallel & cache efficient)

```
PLASMA_[scdz]Lapack_to_Tile()
PLASMA_[scdz]Tile_to_Lapack()
```

PLASMA

in-place translation

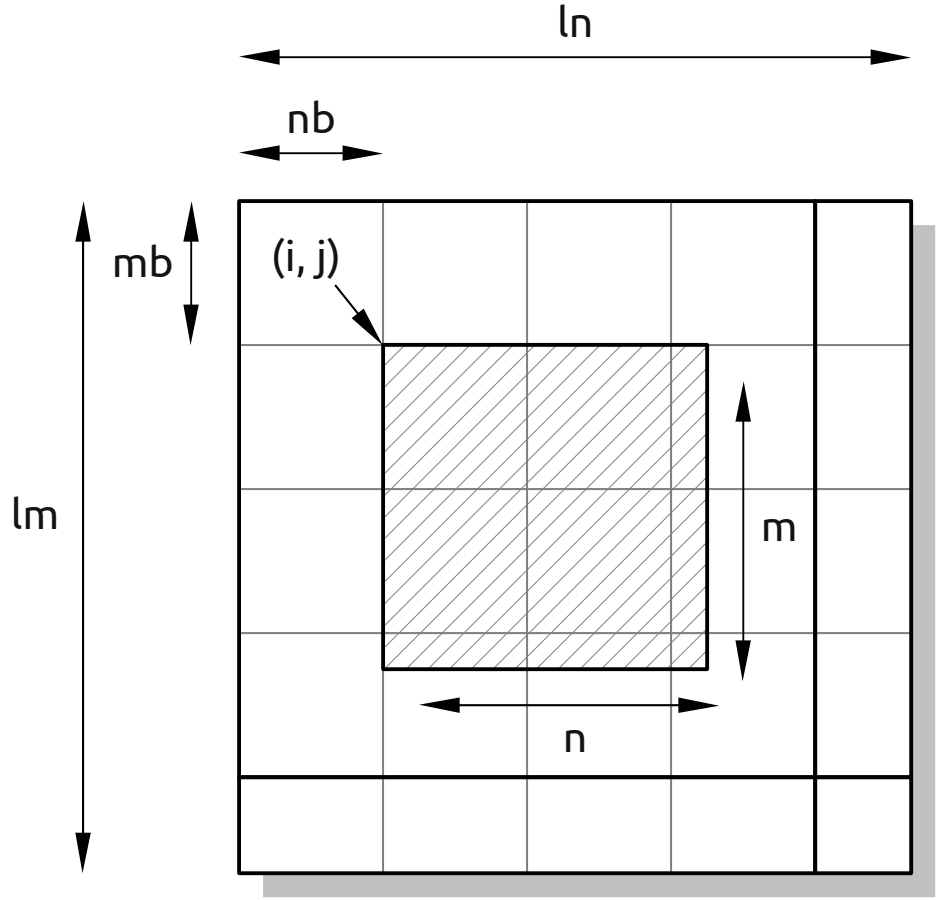


- **In-Place Translation**
  - M = LDA
  - any layout
  - no storage overhead
  - fast (parallel & cache efficient)

`PLASMA_[scdz]gecfi()`

PLASMA

matrix descriptor



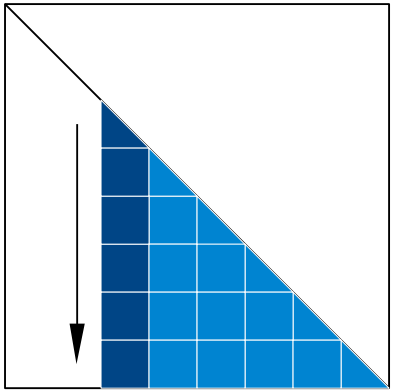

```
PLASMA_Desc_Create()
PLASMA_Desc_Destroy()
```

- **Tile Interface**
  - tiles have to be square (mb = nb)
  - submatrix has to be tile-aligned (i%mb = 0, j%nb = 0)

Algorithms

Cholesky

`PLASMA_[scdz]potrf[_Tile][_Async]()`

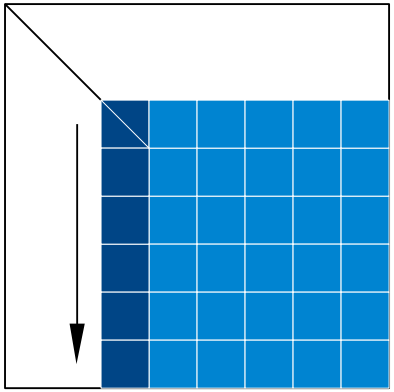


- **Algorithm**
  - equivalent to LAPACK
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

Algorithms

LU

`PLASMA_[scdz]getrf[_Tile][_Async]()`

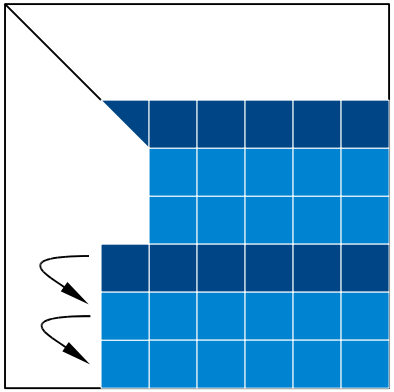


- **Algorithm**
  - equivalent to LAPACK
  - same pivot vector
  - same L and U factors
  - same forward substitution procedure
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

Algorithms

incremental QR

`PLASMA_[scdz]geqrt[_Tile][_Async]()`



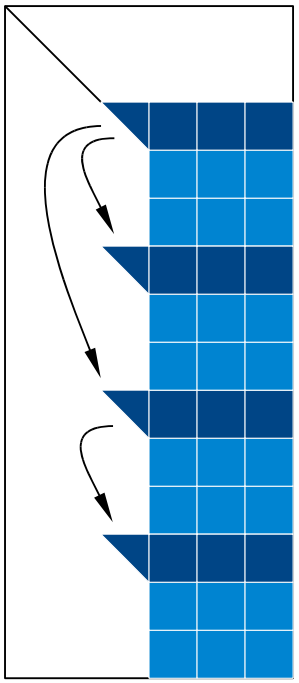
- **Algorithm**
  - the same R factor as LAPACK (absolute values)
  - different set of Householder reflectors
  - different Q matrix
  - different Q generation / application procedure
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

Algorithms

incremental QR

```
PLASMA_[scdz]geqrt[_Tile][_Async]()
```

```
PLASMA_Set(
    PLASMA_HOUSEHOLDER_MODE,
    PLASMA_TREE_HOUSEHOLDER);
```

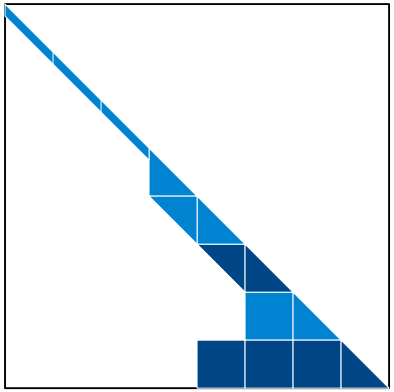


- **Algorithm**
  - the same R factor as LAPACK (absolute values)
  - different set of Householder reflectors
  - different Q matrix
  - different Q generation / application procedure
- **Numerics**
  - same as LAPACK
- **Performance**
  - absolutely superior for tall matrices

Algorithms

three-stage symmetric EVP

`PLASMA_[scdz]syev[_Tile][_Async]()`

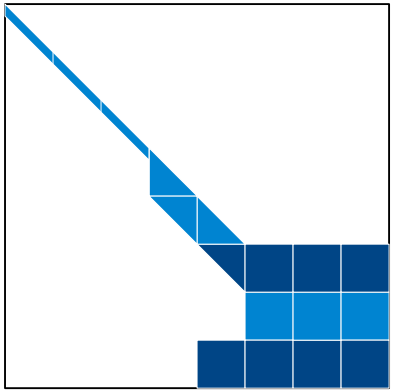


- **Algorithm**
  - two-stage tridiagonal reduction + QR iteration
  - fast eigenvalues, slower eigenvectors
    (possibility to calculate a subset)
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to MKL for very small problems
  - absolutely superior for larger problems

Algorithms

three-stage SVD

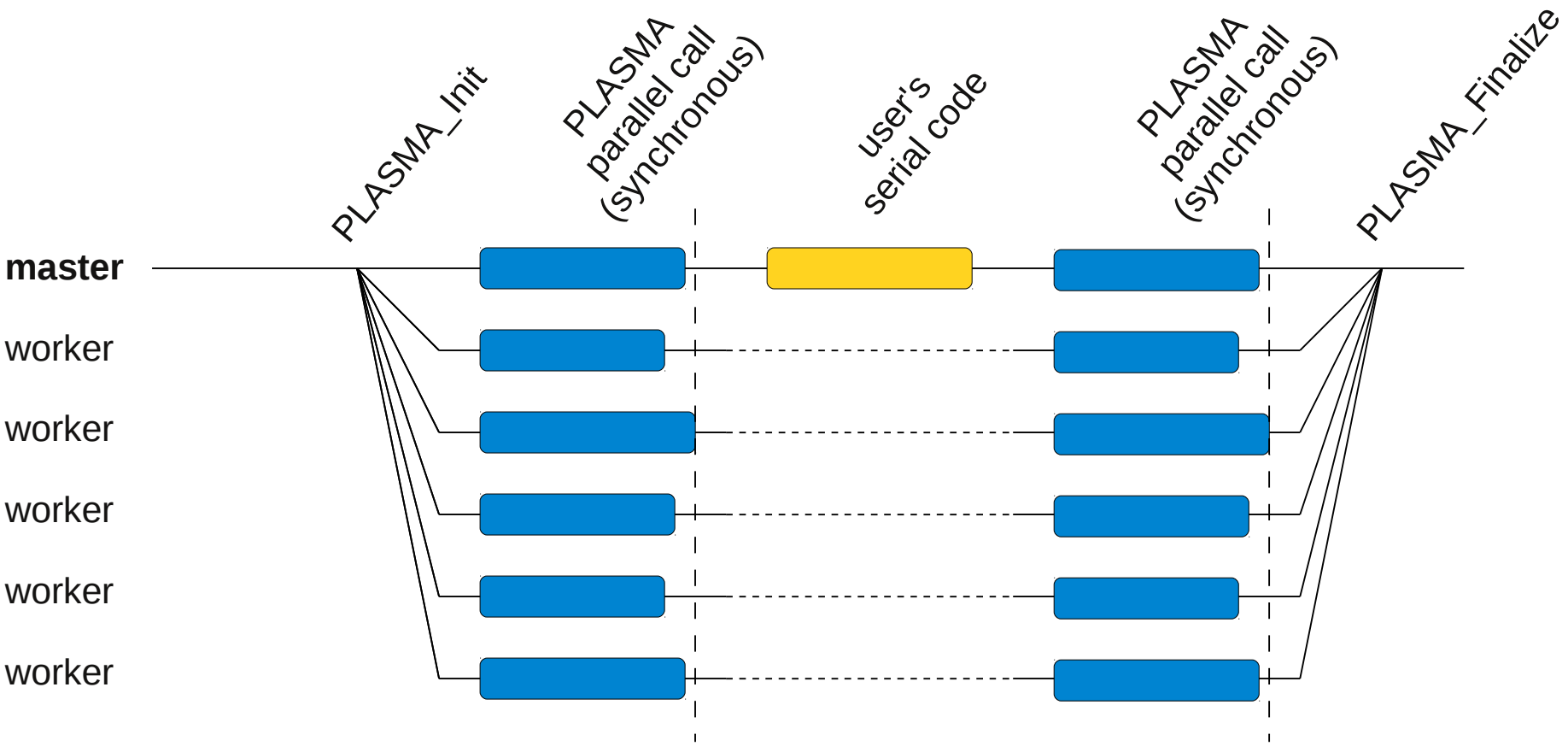
`PLASMA_[scdz]gesvd[_Tile][_Async]()`



- **Algorithm**
  - two-stage bidiagonal reduction + QR iteration
  - fast singular values, slower singular vectors (possibility of calculating a subset)
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable with MKL for very small problems
  - absolutely superior for larger problems

Multithreading

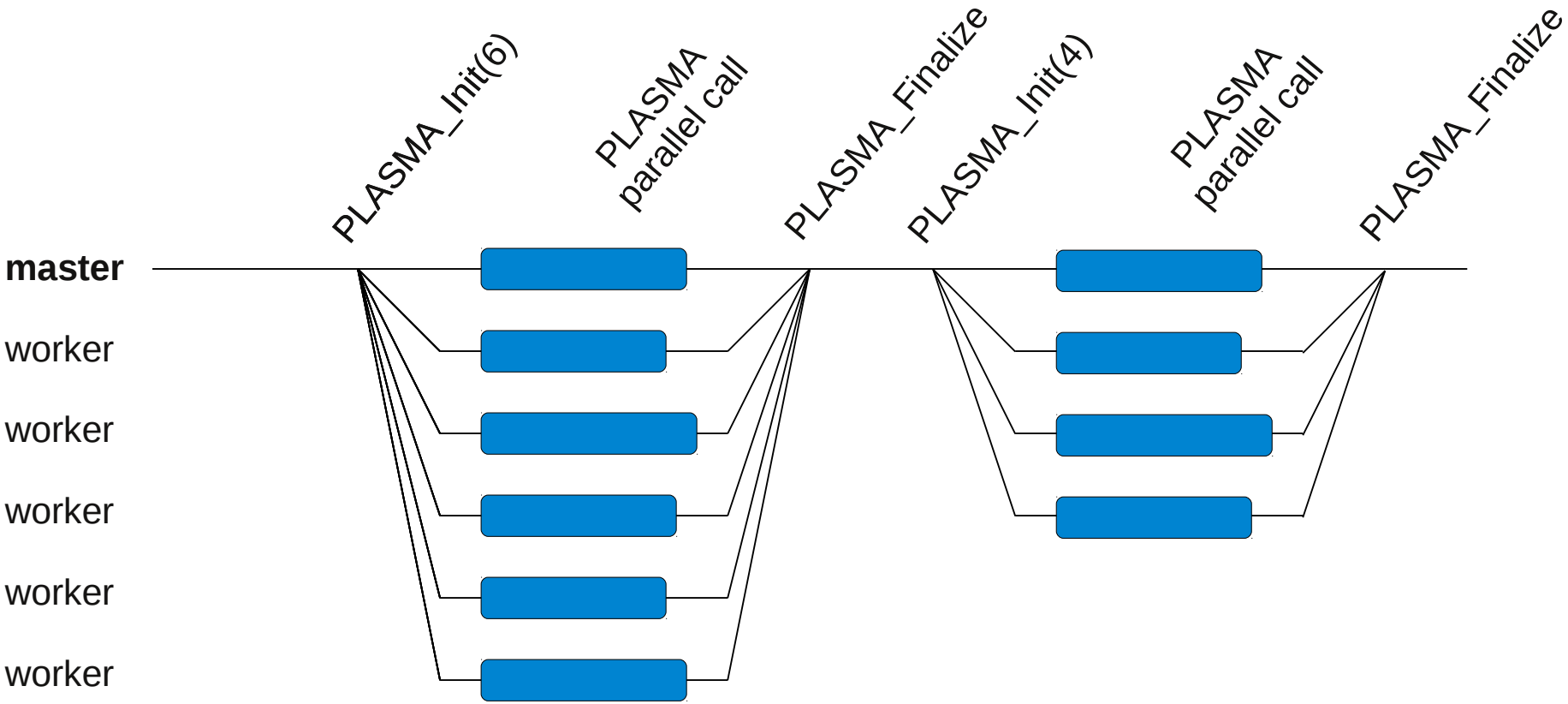
persistent threads



- PLASMA_Init launches N-1 persistent worker threads
- PLASMA_Finalize terminates worker threads
- Worker threads idle between PLASMA calls

Multithreading

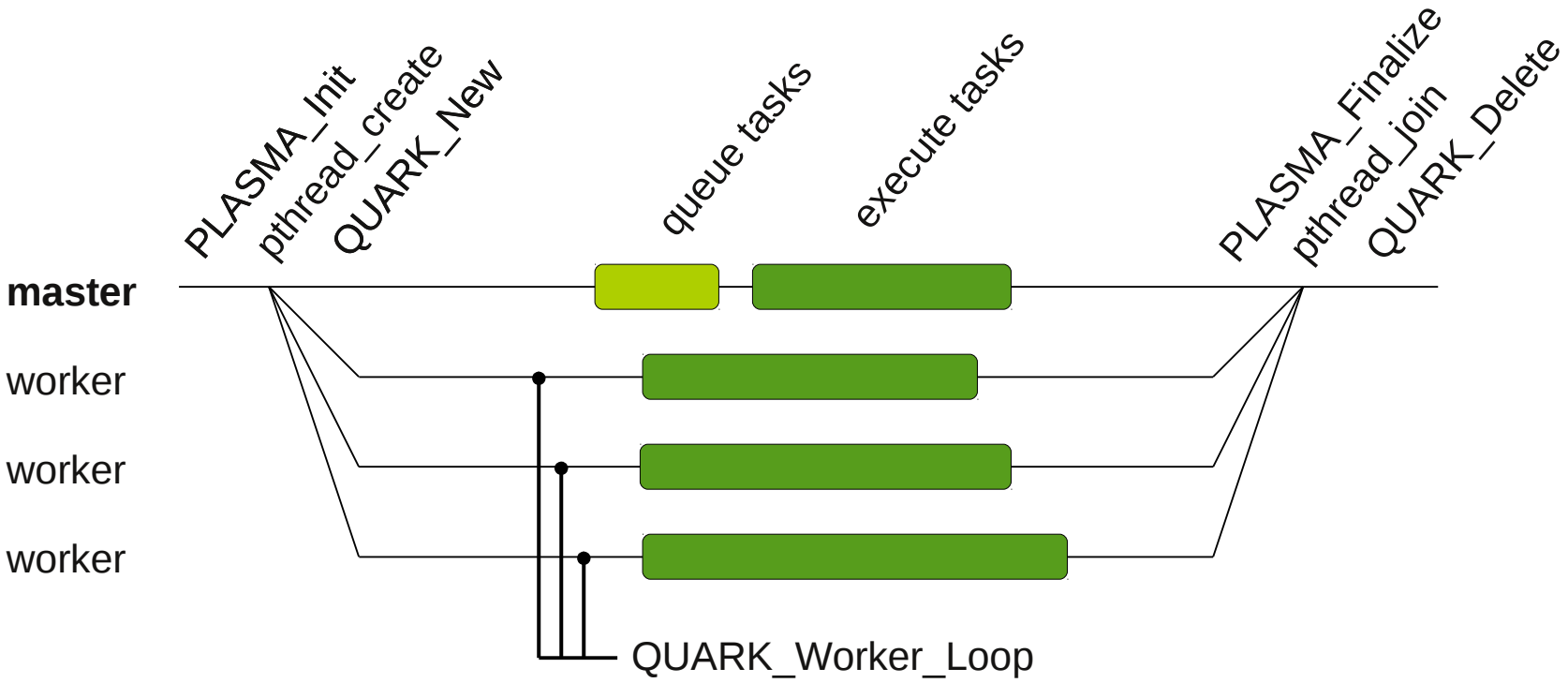
launching and terminating



- PLASMA_Init launches N-1 persistent worker threads
- PLASMA_Finalize terminates worker threads
- You can change the number of cores by terminating and re-initializing

Multithreading

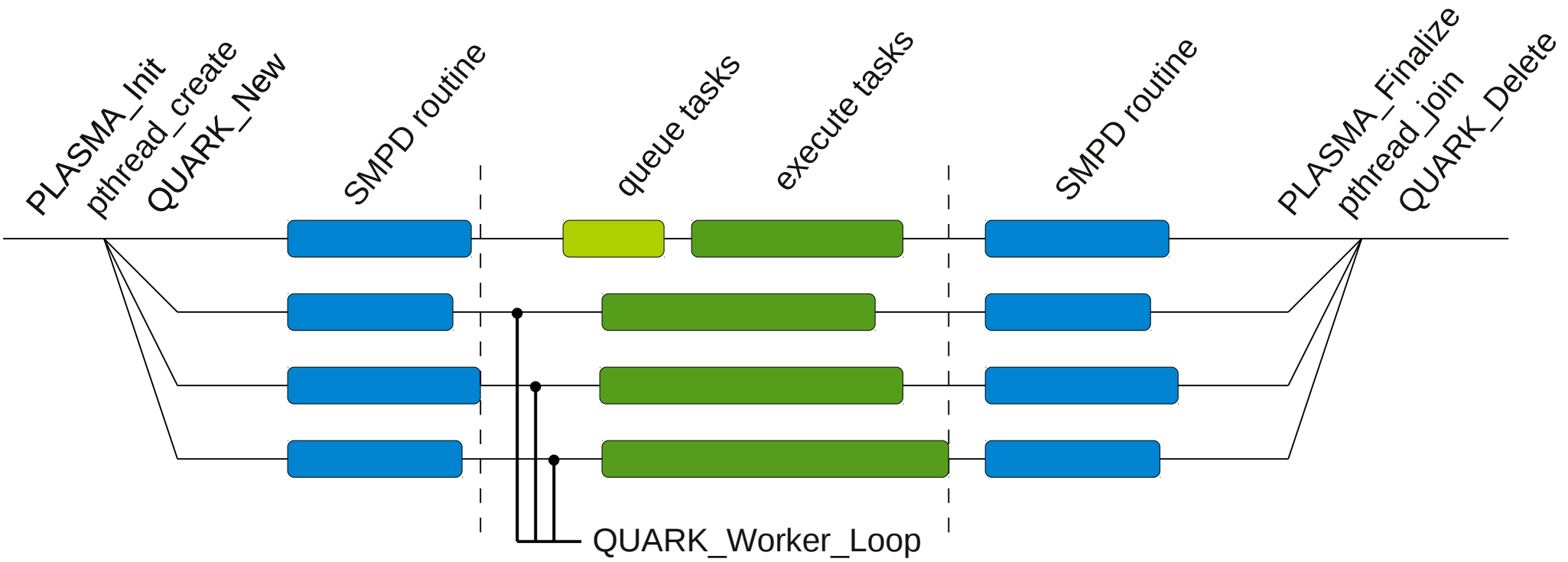
PLASMA with QUARK



- PLASMA instance creates one QUARK instance
- PLASMA master invokes QUARK master
- PLASMA workers become QUARK workers

Multithreading

mixing SPMD and dynamic scheduling



- PLASMA routine can provide an SPMD or a superscalar implementation (QUARK) or both
- PLASMA switches at tuntime depending on which implementation is available
- User can choose the preference at runtime if both implementations are available

Routines

initialize, finalize, configure

`PLASMA_Init()`	– initialize PLASMA
`PLASMA_Finalize()`	– finalize PLASMA
`PLASMA_Set()`	– set a parameter
`PLASMA_Get()`	– get the value of a parameter
`PLASMA_Enable()`	– enable a feature
`PLASMA_Disable()`	– disable a feature
`PLASMA_Version()`	– get PLASMA version number

Routines

linear systems

`PLASMA_dgesv()`	– solve a linear system using the LU fact.
`PLASMA_dgetrf()`	– LU factorization
`PLASMA_dgetrs()`	– forward and backward substitution
`PLASMA_dtrsm()`	– triangular solve
`PLASMA_dposv()`	– solve a linear system using the Cholesky fact.
`PLASMA_dpotrf()`	– Cholesky factorization
`PLASMA_dpotrs()`	– forward and backward substitution
`PLASMA_dtrsm()`	– triangular solve
`PLASMA_dtrtri()`	– compute an inverse of an SPD matrix

Routines

least squares

`PLASMA_dgels()`	– solve the problem using the QR or LQ fact.
`PLASMA_dgeqrf()`	– QR factorization
`PLASMA_dgelqf()`	– LQ factorization
`PLASMA_dormqr()`	– multiply by the Q matrix (QR)
`PLASMA_dormlq()`	– multiply by the Q matrix (LQ)
`PLASMA_dorgqr()`	– generate the Q matrix (QR)
`PLASMA_dorglq()`	– generate the Q matrix (LQ)
`PLASMA_dtrsm()`	– triangular solve

Installation

Python installer

- **Downloads and installs PLASMA**

  No need to download the PLASMA tarball, just the installer script.

- **Downloads and installs all dependencies**
  - CBLAS
  - LAPACK

The installer can also download and install Netlib BLAS.

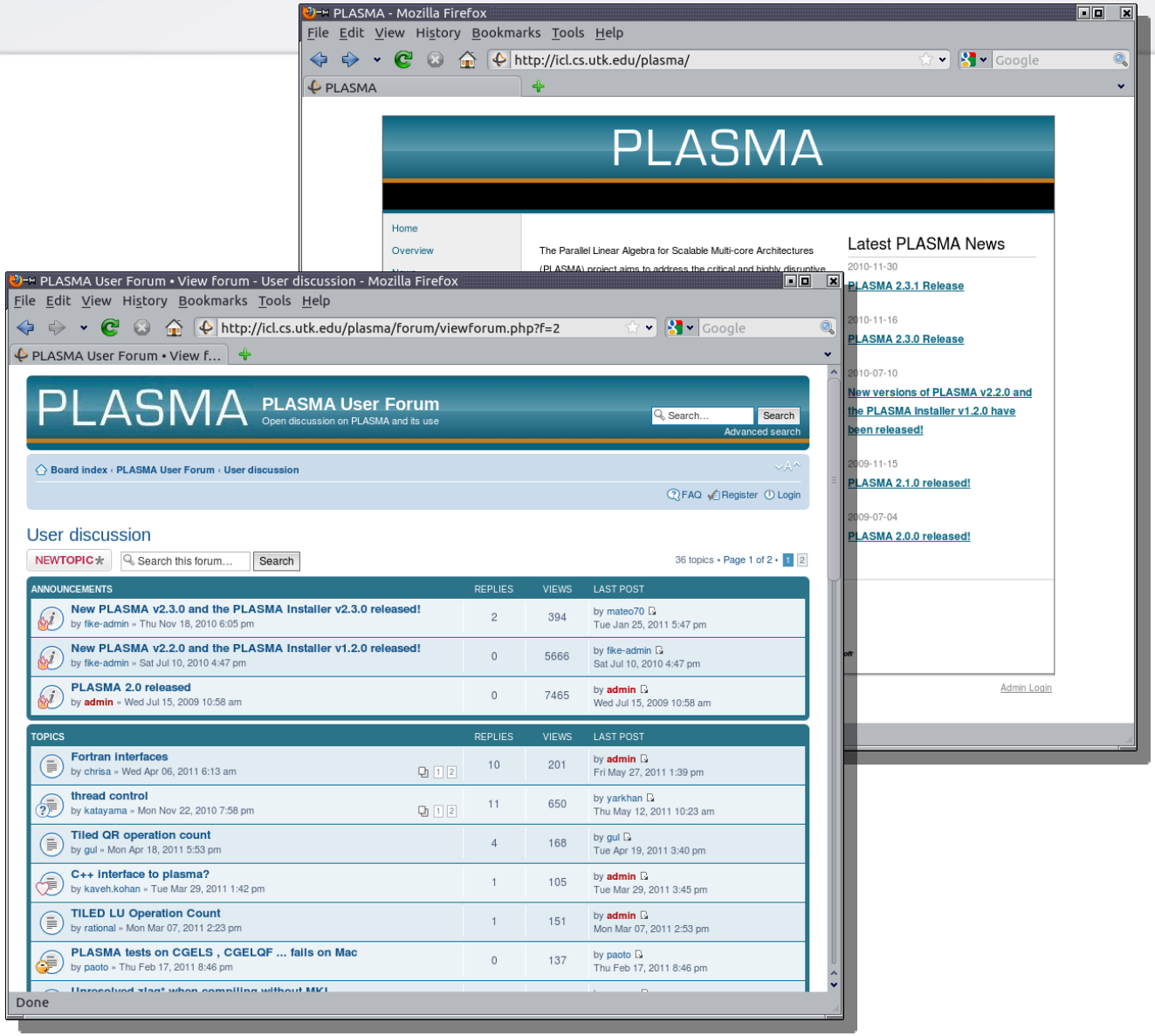
However, Netlib BLAS is unoptimized (reference implementation),

and will deliver very low performance.

PLASMA Documentation

website, online forum

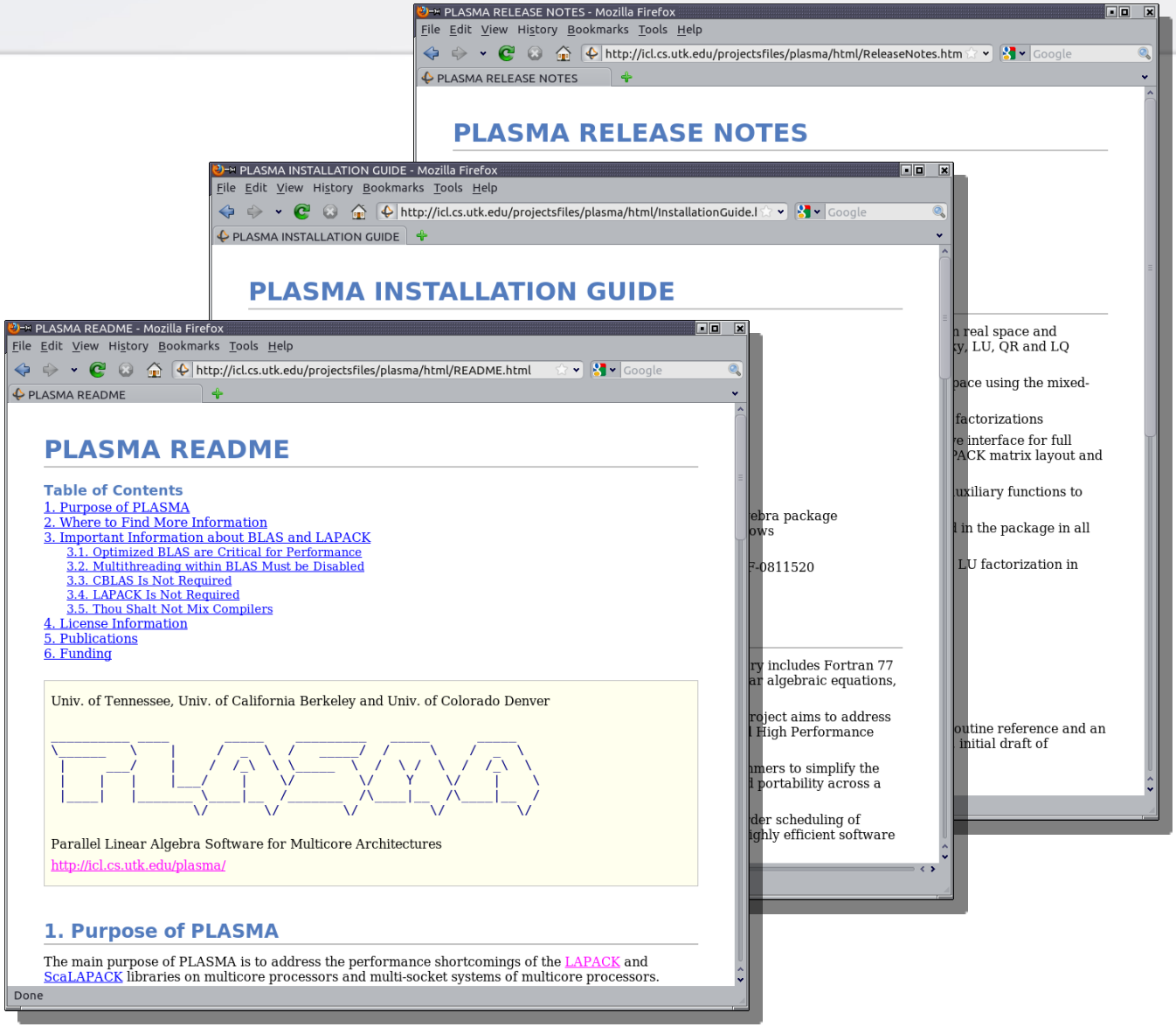
- Home
- Overview
- News
- **Software**
- Publications
- Links
- People
- **Documentation**
- **User Forum**



PLASMA Documentation

AsciiDoc

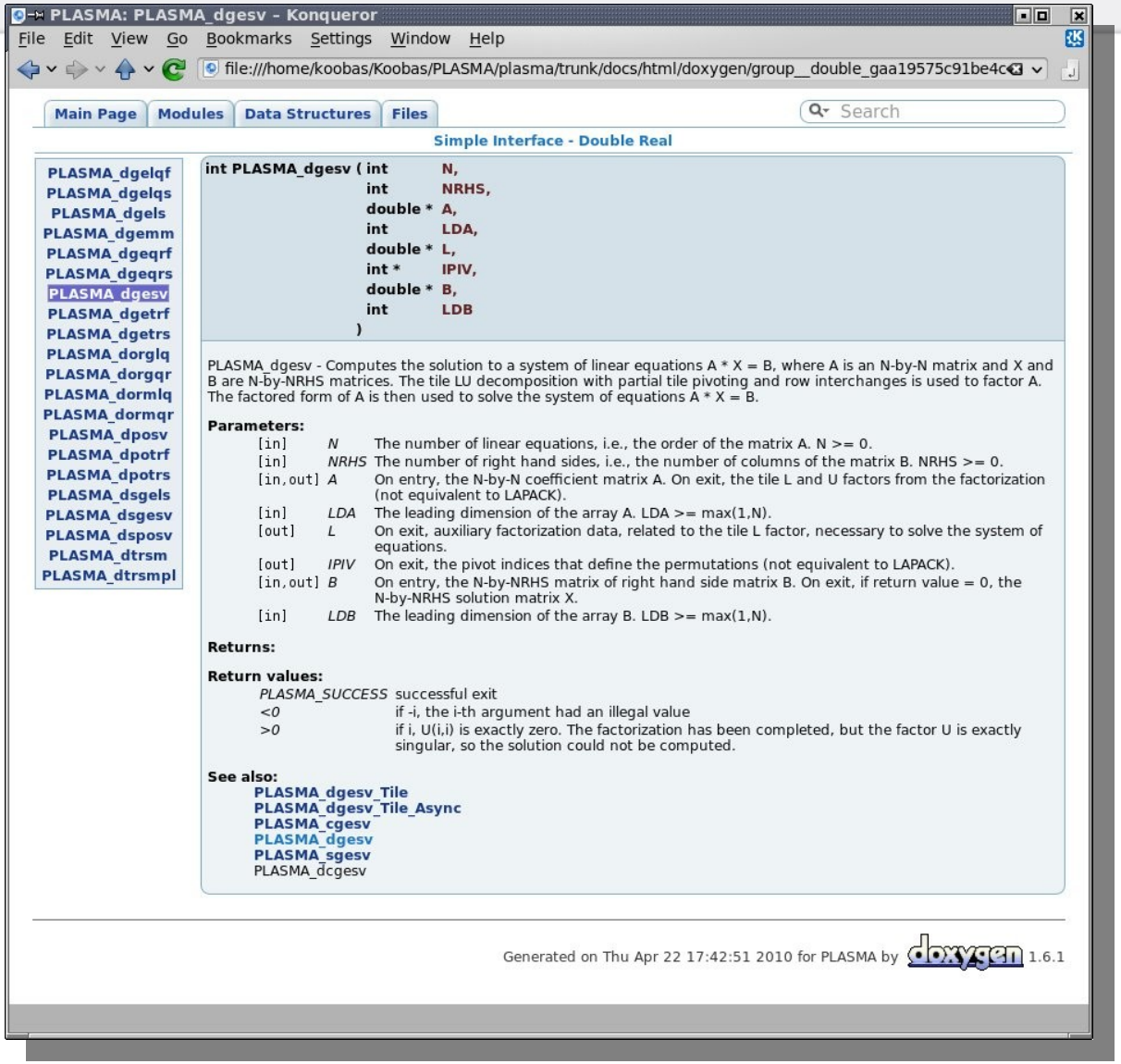
- README
- LICENSE
- Release Notes
- Installation Guide
- Cmake Build Notes



PLASMA Documentation

HTML function reference

- **Grouping by precision**
  - Z, C, D, C
  - ZC, DS
- **Grouping by interface**
  - standard
  - tile
  - tile Async
- **“see also” section**



http://web.eecs.utk.edu/~kurzak/tutorials/

QUARK

Queuing And Runtime for Kernels

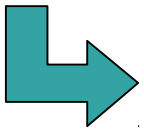
- Design Principles
- Usage Examples
- Advanced Features
- Distributed Memory Prototype

Manual Multithreading

SPMD code with spinlock synchronization

```
FOR k = 0..TILES-1
    FOR n = 0..k-1
        A[k][k] ← DSYRK(A[k][n], A[k][k])
    A[k][k] ← DPOTRF(A[k][k])
    FOR m = k+1..TILES-1
       FOR n = 0..k-1
            A[m][k] ← DGEMM(A[k][n], A[m][n], A[m][k])
        A[m][k] ← DTRSM(A[k][k], A[m][k])
```

definition (Cholesky)



```
k = 0; m = my_core_id;
while (m >= TILES) {
    k++; m = m-TILES+k;
} n = 0;

while (k < TILES && m < TILES) {
    next_n = n; next_m = m; next_k = k;

    next_n++;
    if (next_n > next_k) {
        next_m += cores_num;
        while (next_m >= TILES && next_k < TILES) {
            next_k++; next_m = next_m-TILES+next_k;
        } next_n = 0;
    }

    if (m == k) {
        if (n == k) {
            dpotrf(A[k][k]);
            core_progress[k][k] = 1;
        }
        else {
            while(core_progress[k][n] != 1);
            dsyrk(A[k][n], A[k][k]);
        }
    }
    else {
        if (n == k) {
            while(core_progress[k][k] != 1);
            dtrsm(A[k][k], A[m][k]);
            core_progress[m][k] = 1;
        }
        else {
            while(core_progress[k][n] != 1);
            while(core_progress[m][n] != 1);
            dgemm(A[k][n], A[m][n], A[m][k]);
        }
    }
    n = next_n; m = next_m; k = next_k;
}
```

code

SPMD code with spinlocks

QUARK

Shared Memory Superscalar Scheduling

```
FOR k = 0..TILES-1
    A[k][k] ← DPOTRF(A[k][k])
    FOR m = k+1..TILES-1
        A[m][k] ← DTRSM(A[k][k], A[m][k])
    FOR m = k+1..TILES-1
        A[m][m] ← DSYRK(A[m][k], A[m][m])
        FOR n = k+1..m-1
            A[m][n] ← DGEMM(A[m][k], A[n][k], A[m][n])
```

definition
(pseudocode)

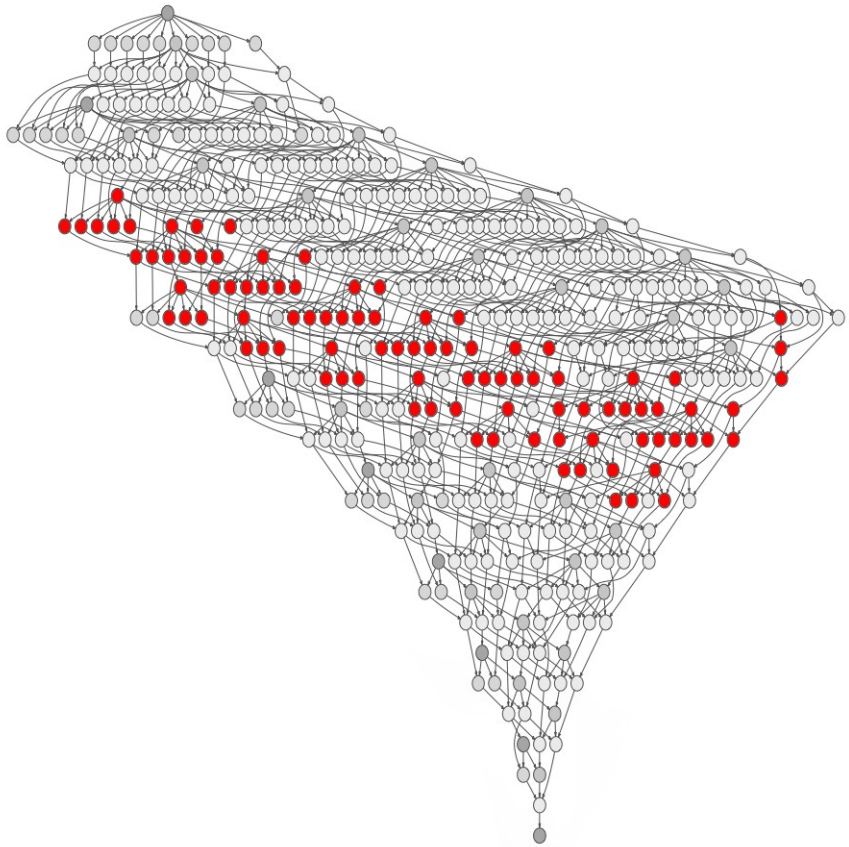
```
for (k = 0; k < A.mt; k++) {
    QUARK_CORE_dpotrf(...);
    for (m = k+1; m < A.mt; m++) {
        QUARK_CORE_dtrsm(...);
    }
    for (m = k+1; m < A.mt; m++) {
        QUARK_CORE_dsyrk(...);
        for (n = k+1; n < m; n++) {
            QUARK_CORE_dgemm(...)
        }
    }
}
```

implementation
(actual QUARK code in PLASMA)

QUARK Basics

A shared memory superscalar scheduler

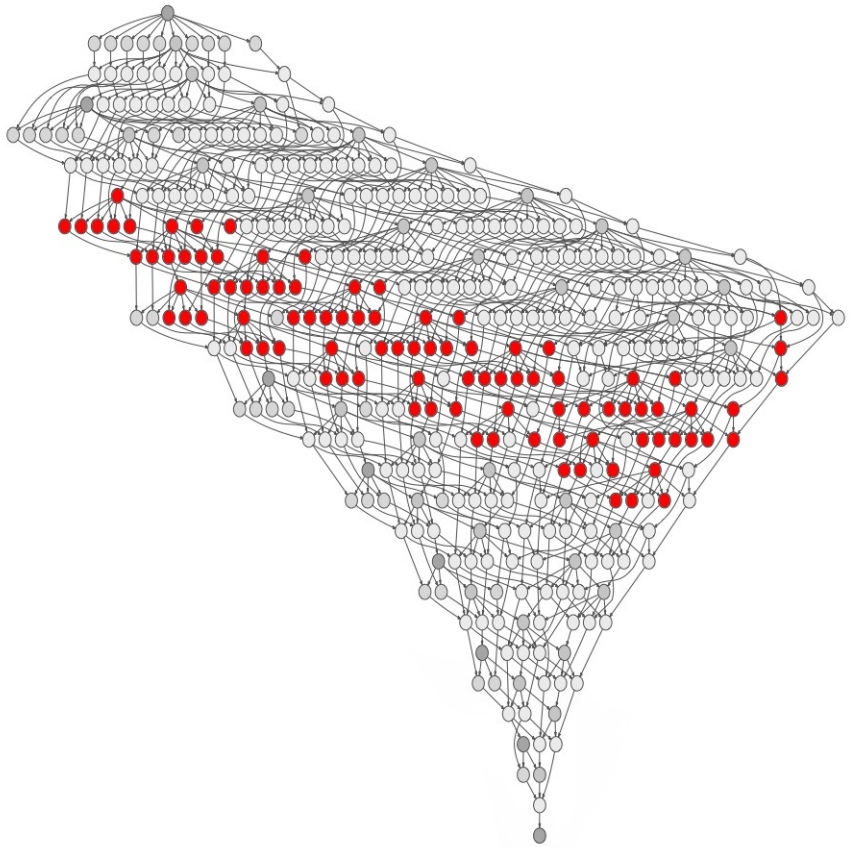
- **Superscalar Scheduling**
  - serial code
  - side-effect-free tasks
  - dependency resolution
- **Resolving Data Hazards**
  - Read After Write
  - Write after Read
  - Write after Write
- **Similar Projects**
  - SMPSs from Barcelona SC
  - StarPU from INRIA Bordeaus
  - Jade from Stanford (historical)



QUARK

Shared Memory Superscalar Scheduling

- **Simple API**
  - quick adoption
  - fast prototyping
- **Powerful extensions**
  - fine tuning of scheduling
  - fine tuning of affinity
- **Invaluable feedback**
  - automatic DAG (task graph) plotting
  - automatic tracing
- **Ambitious Roadmap**
  - distributed systems
  - hardware accelerators



QUARK Tasks

defining a task

```
void CORE_dtrsm(int side, int uplo,
                int trans, int diag,
                int m, int n,
                double alpha, double *A, int lda,
                double *B, int ldb)
{
    ...
}
```

```
void CORE_dtrsm_quark(Quark *quark)
{
    int side, uplo;
    int trans, diag;
    int m, n;
    double alpha, double *A;
    int lda;
    double *B;
    int ldb;

    quark_unpack_args_11(quark,
        side, uplo,
        trans, diag,
        m, n,
        alpha, A, lda,
        B, ldb);
    ...
}
```

side-effect free function

arguments fetched through a macro

QUARK Tasks

queuing a task

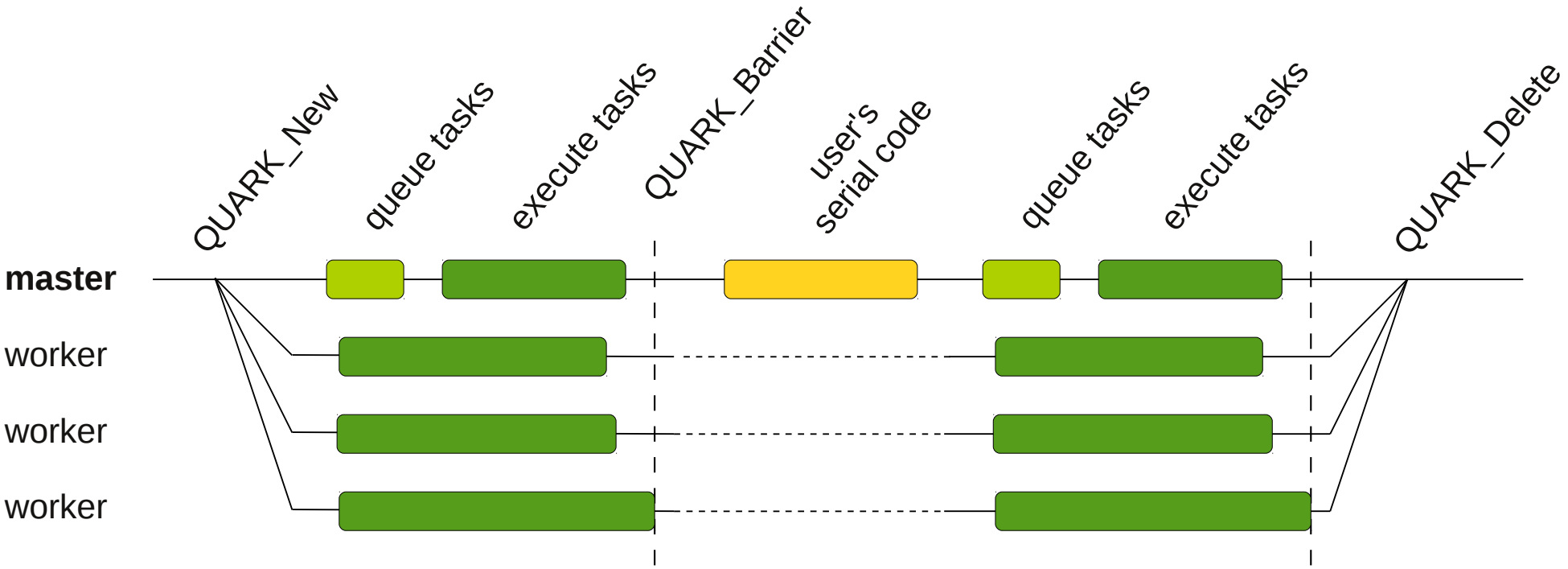
```
CORE_dtrsm(
    PlasmaRight, PlasmaLower,
    PlasmaTrans, PlasmaNonUnit,
    m, n,
    zone, A(k, k), ldak,
          A(m, k), ldam);
```

Scalars (VALUE) – pass by value semantics

```
QUARK_Insert_Task(quark, CORE_dtrsm_quark, task_flags,
    sizeof(PLASMA_enum),    &side,      VALUE,
    sizeof(PLASMA_enum),    &uplo,      VALUE,
    sizeof(PLASMA_enum),    &trans,     VALUE,
    sizeof(PLASMA_enum),    &diag,      VALUE,
    sizeof(int),            &m,         VALUE,
    sizeof(int),            &n,         VALUE,
    sizeof(double),         &alpha,     VALUE,
    sizeof(double)*nb*nb,     A,                  INPUT,
    sizeof(int),            &lda,       VALUE,
    sizeof(double)*nb*nb,     B,                  INOUT | LOCALITY,
    sizeof(int),            &ldb,       VALUE,
    0);
```

Multithreading

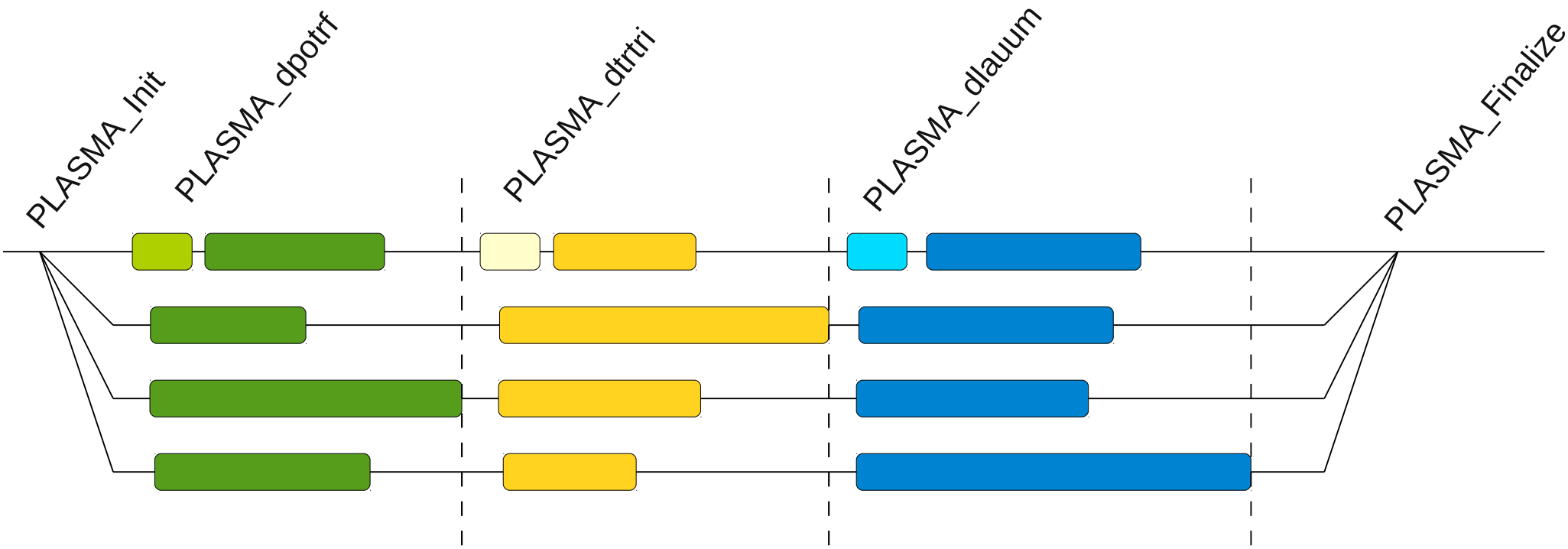
persistent threads



- QUARK_New launches N-1 persistent worker threads
- QUARK_Delete terminates worker threads
- QUARK is thread safe, i.e., allows for multiple instances to coexist
- QUARK_New returns a handle (explicit management of instances)

Multithreadin

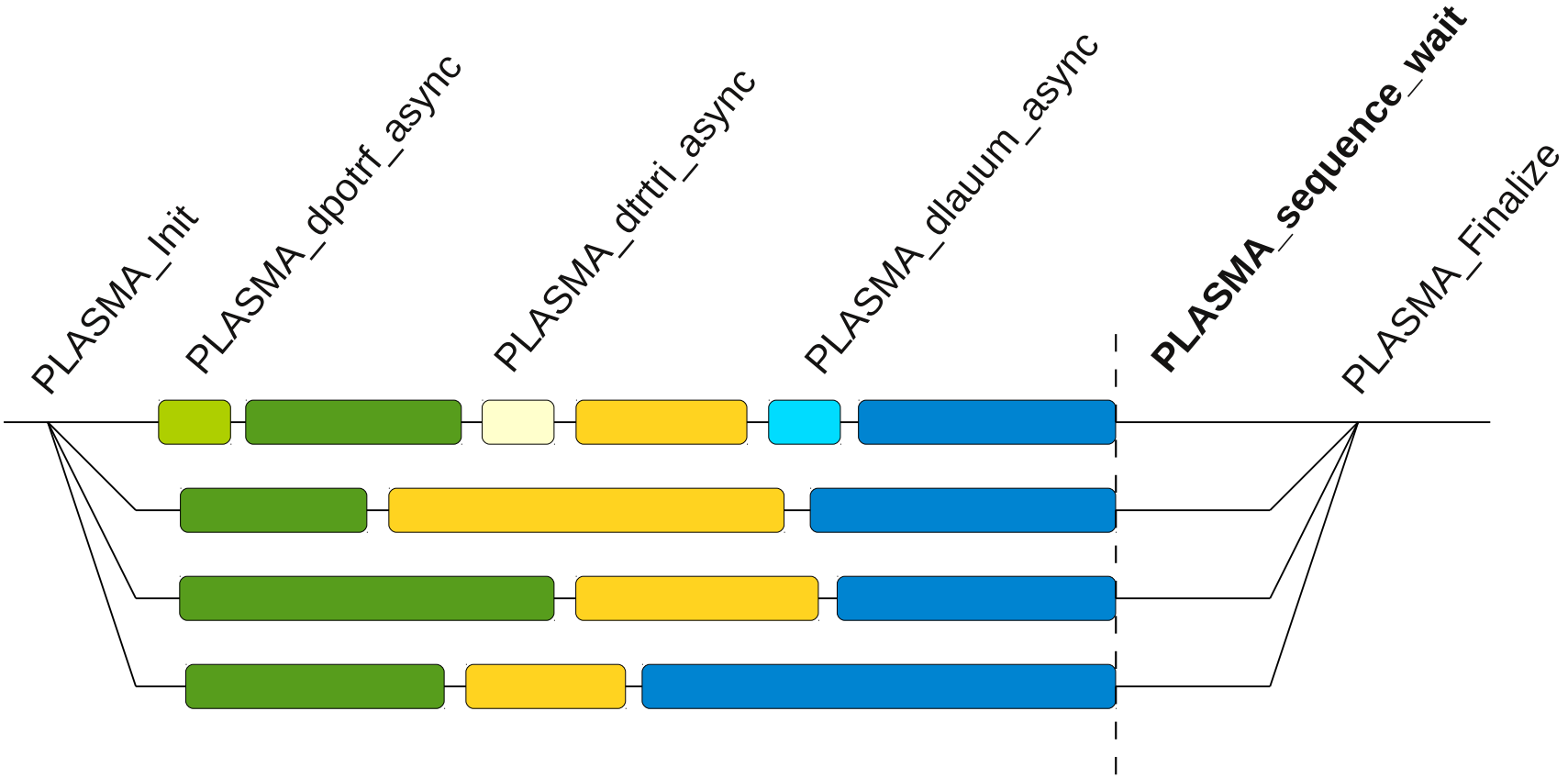
PLASMA synchronous interface



- PLASMA's synchronouse interface (default) includes a barrier after each user's call
- Dynamic scheduling within each user-level call

Multithreading

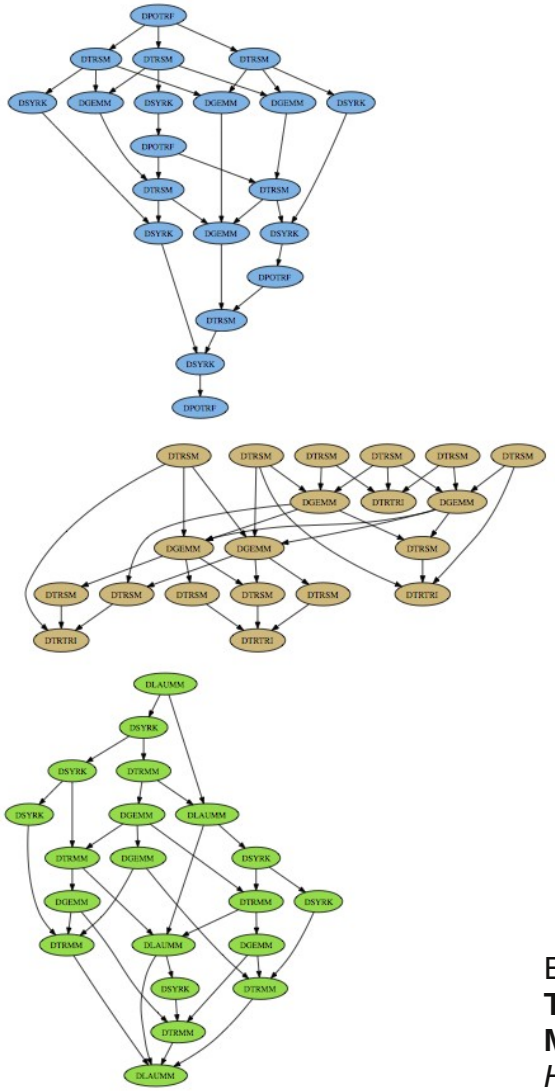
PLASMA asynchronous interface

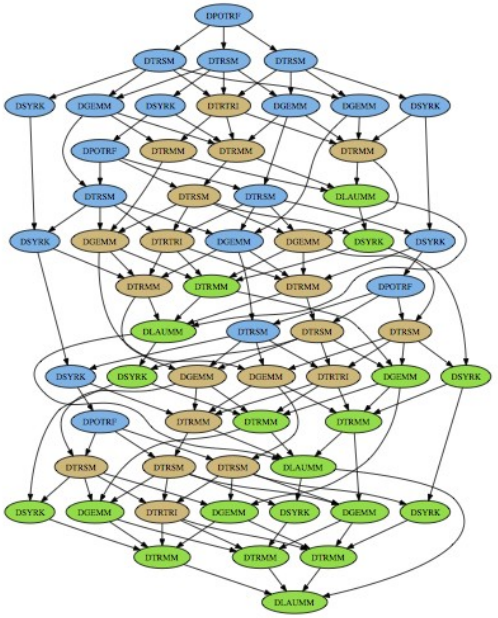


- PLASMA's asynchronous interface does not include the barrier after each user-level call
- QUARK will interleave the execution of a sequence of dynamically scheduled routines
- The user needs to put an explicit WAIT at the end fo a sequence of async calls

QUARK

parallel composition

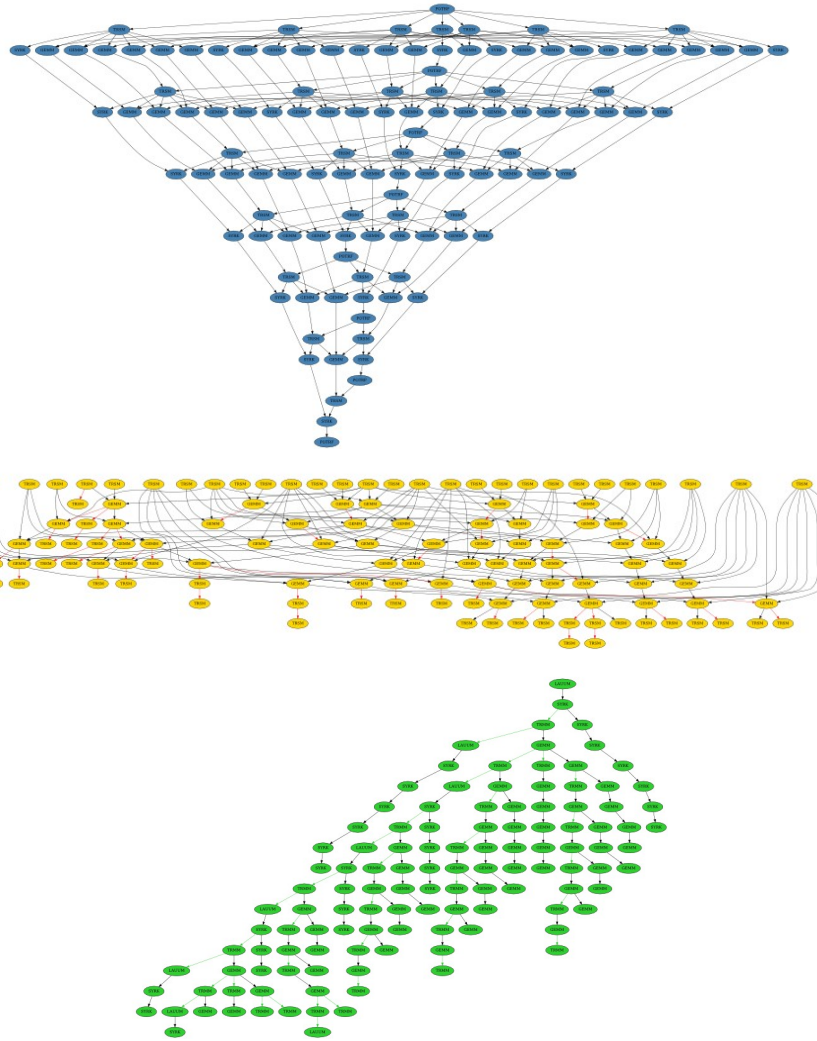


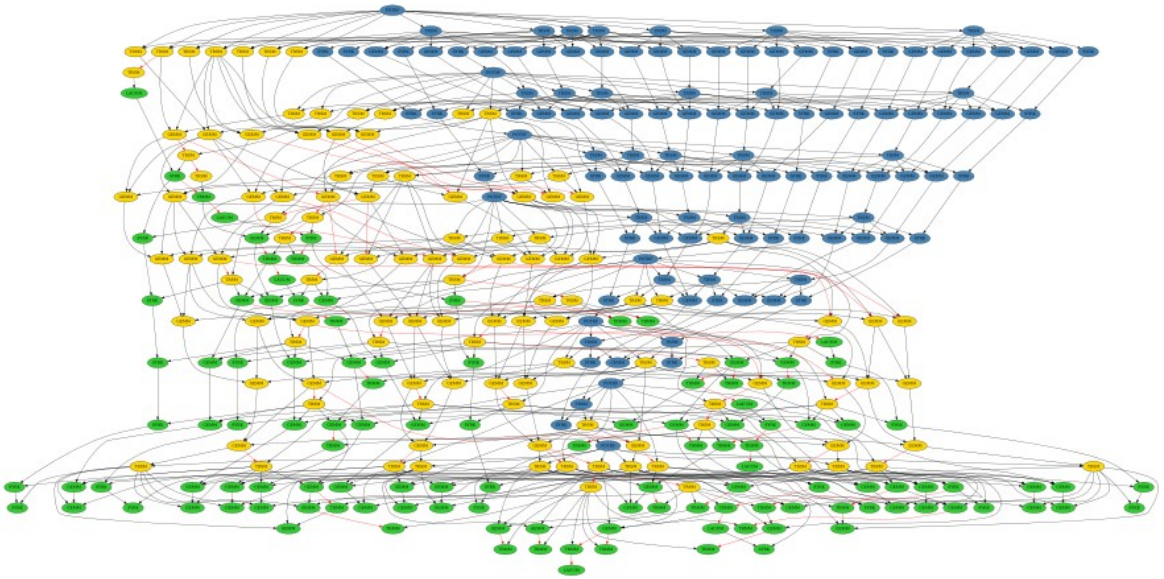


E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, L. Rosenberg
Towards an Efficient Tile Matrix Inversion of Symmetric Positive Definite Matrices on Multicore Architectures
High Performance Computing for Computational Science – VECPAR 2010

QUARK

parallel composition

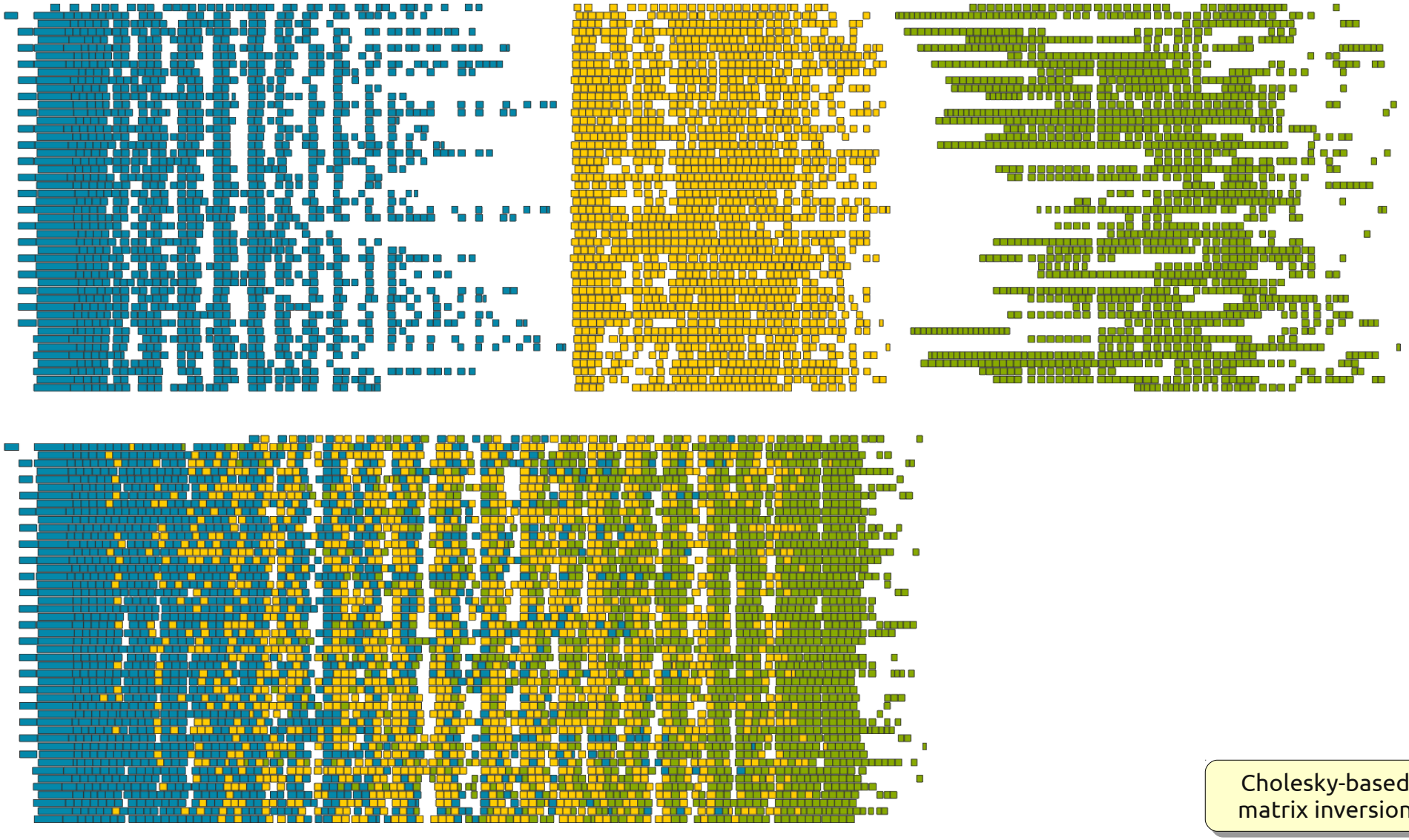




Cholesky-based matrix inversion

QUARK

parallel composition



Cholesky-based
matrix inversion

QUARK Features

task cancellation

- **Cancellation of a task**

  Task ID is returned when queuing a task.

  The task ID can be used to cancel a task that has been queued,

  but has not been executed yet.

- **Cancellation of a sequence of tasks**

  Tasks can be grouped in sequences.

  Entire sequence of tasks can be canceled.

  Many sequences can be in flight at the same time.

  One sequence can be canceled without affecting other sequences.

QUARK Features

hinting priority and data locality

- **Priority hinting**

  Priorities can be assigned to tasks.

  If tasks with different priorities are ready for execution at the same point in time, the task with the highest priority executes first.

  Priorities provide a way of hinting the critical path.

- **Locality (data reuse) hinting**

  Locality flag can be assigned to a data item.

  QUARK will try to keep that item on one core.

  If possible, consecutive tasks using that data item will be scheduled to the same core.

QUARK Features

relaxing dependencies

- **“Accumulator” tasks**

  Data item can be flagged with the “accumulator” flag.

  This means that the operation performed on that item is a reduction and QUARK is free to reorder the tasks to improve scheduling.

- **“Gatherv” tasks**

  Data item can be flagged with the “gatherv” flag.

  This means that the tasks operate on disjoint parts of the data and can execute simultaneously without causing race conditions.

QUARK Features

locking tasks to threads

- **Locking to a thread**

  A task can be locked to a particular thread.

  Other threads will not be allowed to steal that task through work stealing.

- **Locking to a thread mask**

  A task can be confined to a subset of threads by using a bit mask.

  QUARK will schedule the task to one of the treads in the bit mask.

  Outside threads will not be allowed to steal that task through work stealing.

QUARK Features

controlling work granularity

- **Nested-parallel tasks**

  The user can have a piece of code that is already multithreaded

  (using mutexes / conditional variables / busy waiting / etc.)

  QUARK can schedule such code to a subset of cores

  and track the dependencies as if it was a sequential task.

- **Incremental lists of dependencies**

  Complete list of dependencies for a task may not be know at compile time.

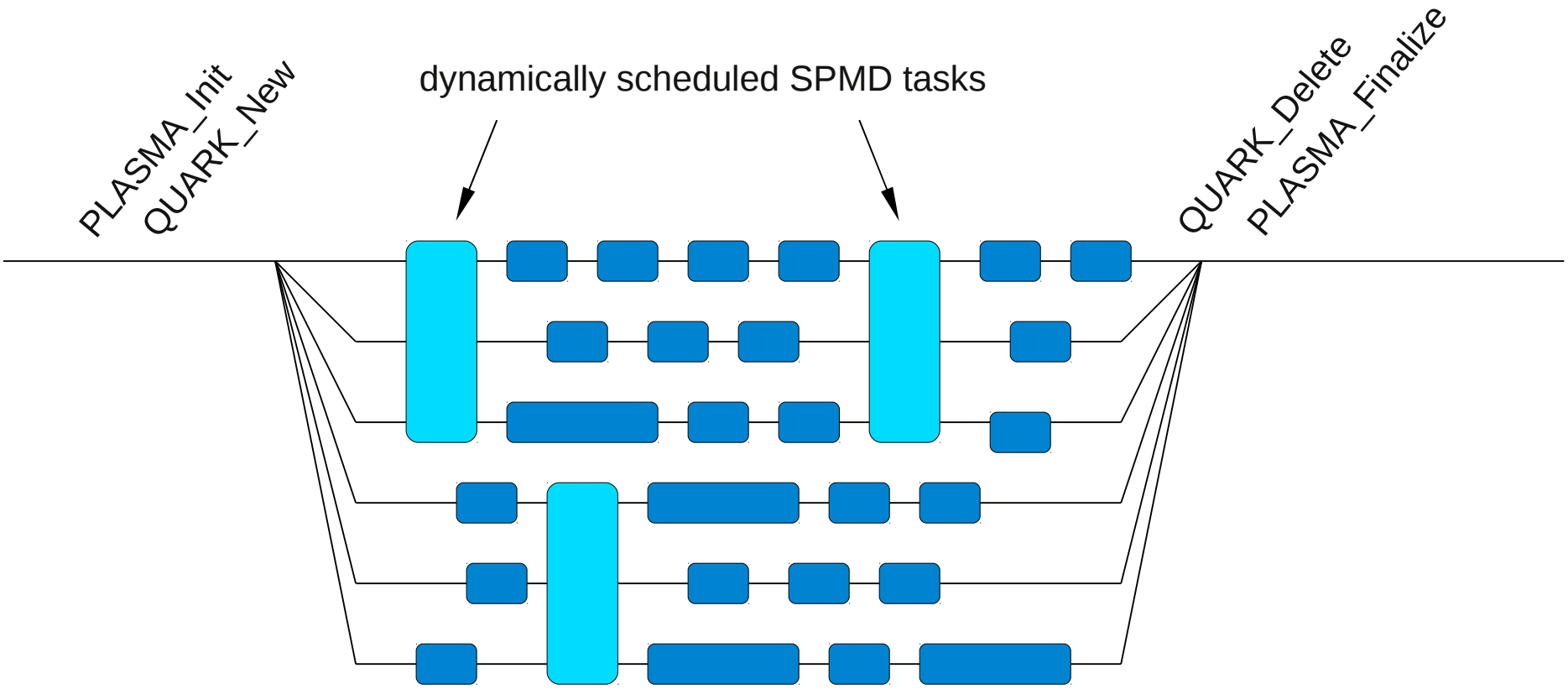
  In such a case, the list of dependencies can be created at runtime.

  First, a task is created with an empty list of dependencies.

  Then dependencies are added (incrementally), e.g. in a loop.

Multithreading

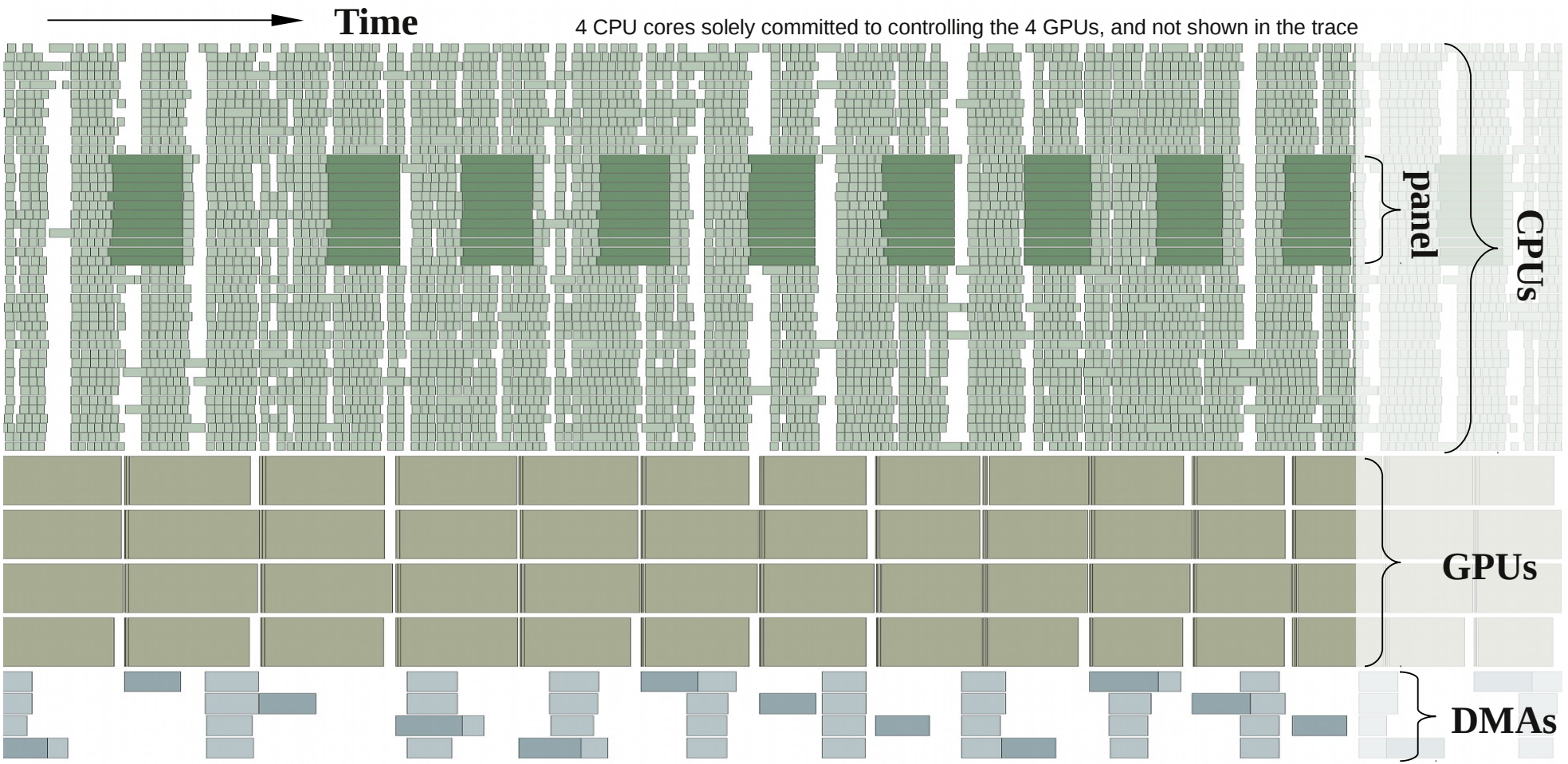
nested parallelism



- QUARK can dynamically schedule SPMD tasks
- SPMD tasks use a subset of QUARK threads / PLASMA threads
- No obvious way to use OpenMP, multithreaded BLAS, or another multithreaded library

QUARK on Accelerators

prototype implementation of the LU factorization using 48 cores and 4 GPUs

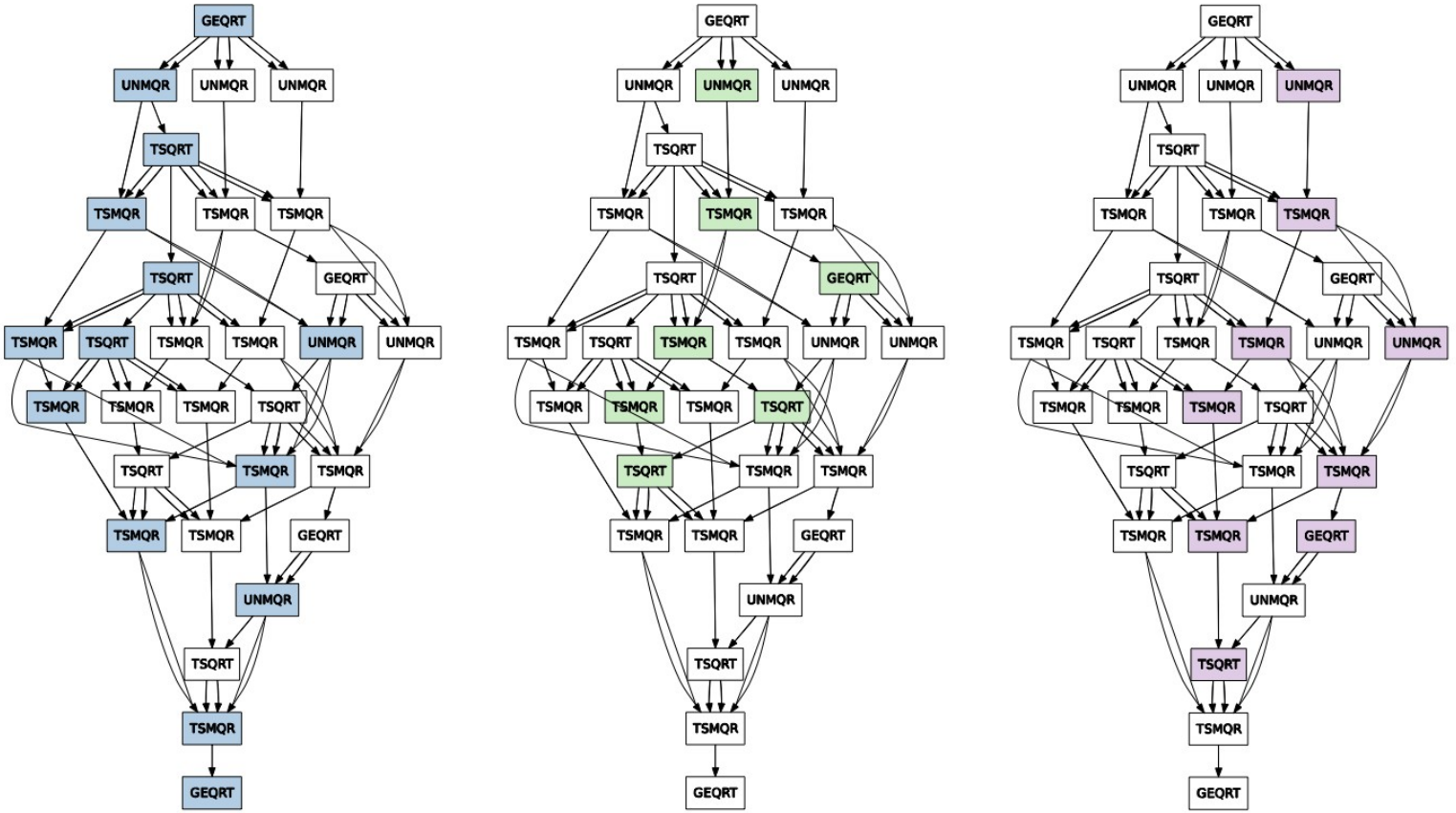


J. Kurzak, P. Luszczek, M. Faverge, J. Dongarra
Programming the LU Factorization for a Multicore System with Accelerators
High Performance Computing for Computational Science – VECPAR 2012

QUARK-D

distribured memory superscalar scheduling

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

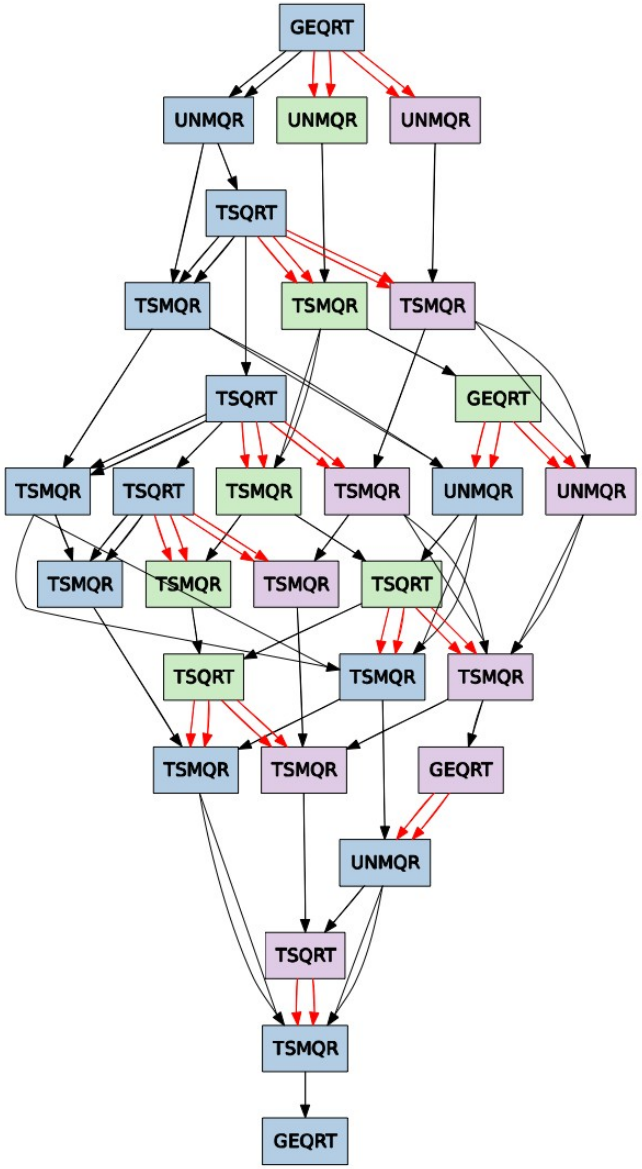


process 1 process 2 process 3

QUARK-D

distribured memory superscalar scheduling

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

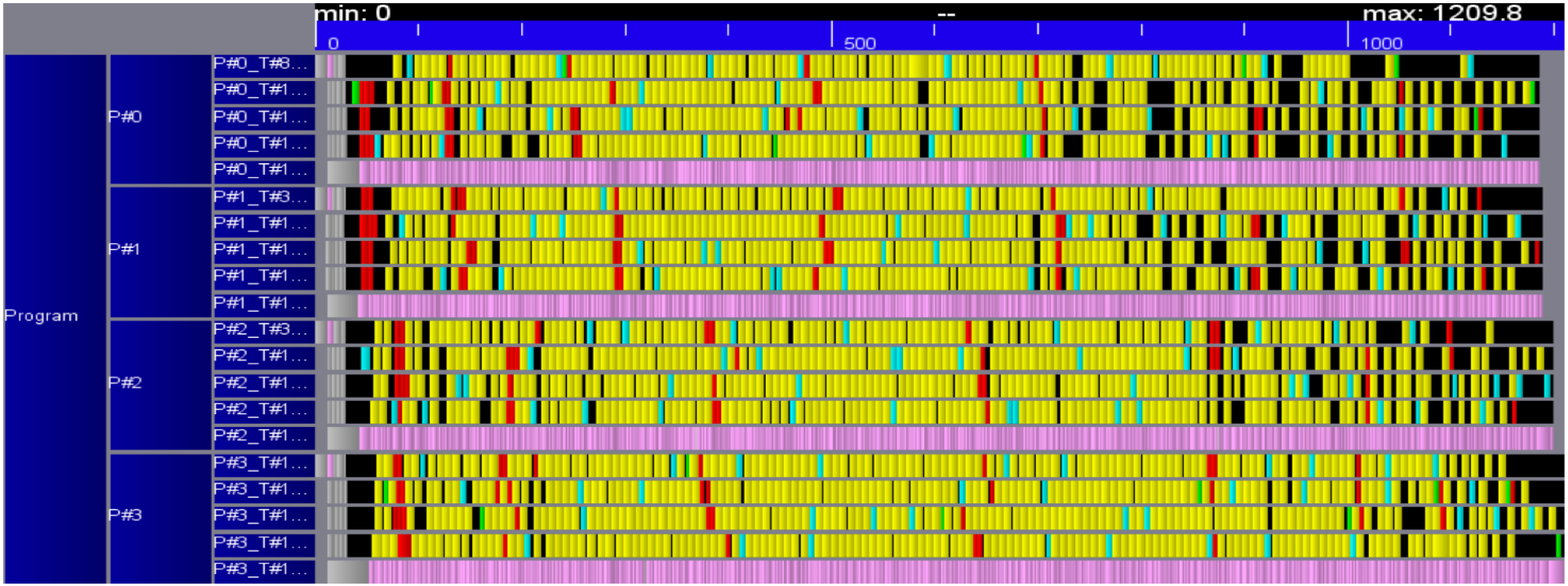


local depenency

communication

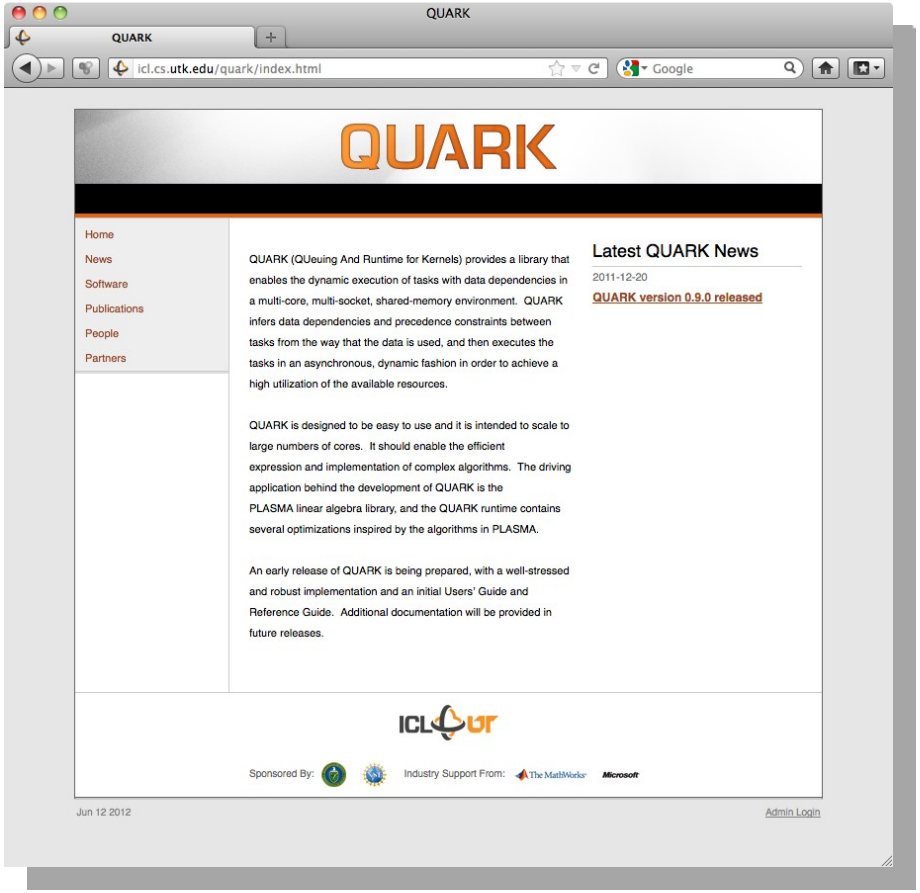
QUARK-D

distribured memory superscalar scheduling



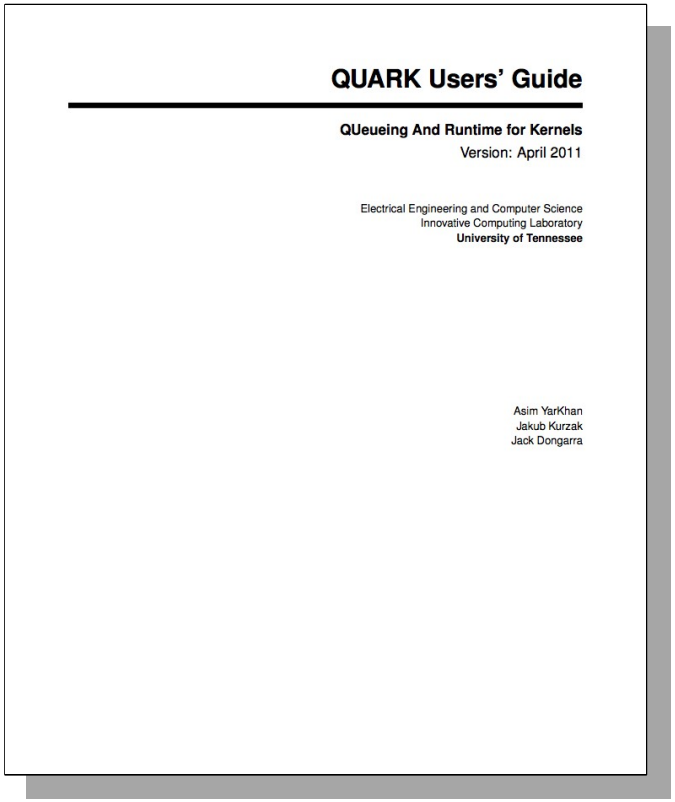
QUARK Resources

http://icl.utk.edu/quark/



copy & paste "hello world" examples
in /examples/ after installation

Users' Guide
in /docs/pdf/ after installation



http://web.eecs.utk.edu/~kurzak/tutorials/

DPLASMA / PaRSEC

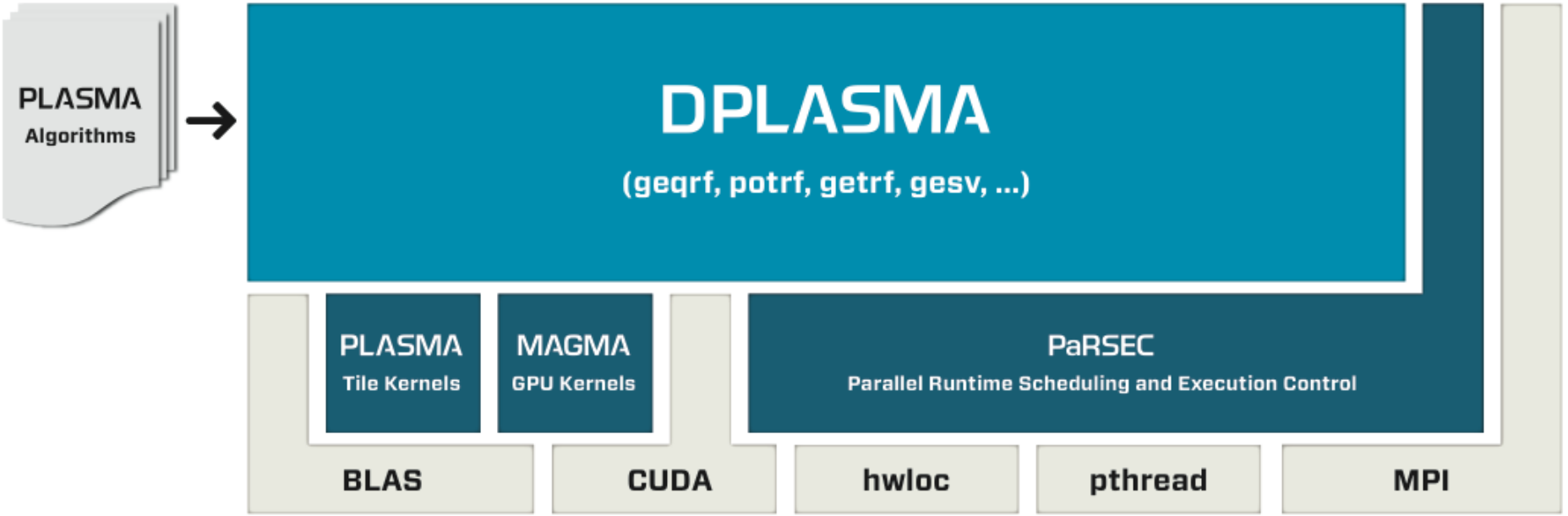
Distributed memory PLASMA
/
Parallel Runtime Scheduling and Execution Control

TOC

- Software Stack
- Functionality
- Design Principles
- Performance

DPLASMA

Distributed memory PLASMA



A. Bouteiller et al.
Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA
Parallel and Distributed Processing Workshops and Phd Forum - IPDPSW 2011

DPLASMA

Functionality

FUNCTIONALITY	COVERAGE
Linear Systems of Equations	Cholesky, LU (inc. pivoting, PP), LDL (prototype)
Least Squares	QR & LQ
Symmetric Eigenvalue Problem	Reduction to Band (prototype)
Level 3 Tile BLAS	GEMM, TRSM, TRMM, HEMM/SYMM, HERK/SYRK, HER2K/SYR2K

FEATURES

- Covering four precisions: double real, double complex, single real, single complex (D, Z, S, C)
- Providing ScaLAPACK-compatible interface for matrices in F77 column-major layout
- Supporting: Linux, Windows, Mac OS X, UN*X (depends on MPI, hwloc)

PaRSEC

Parallel Runtime Scheduling ane Execution Control

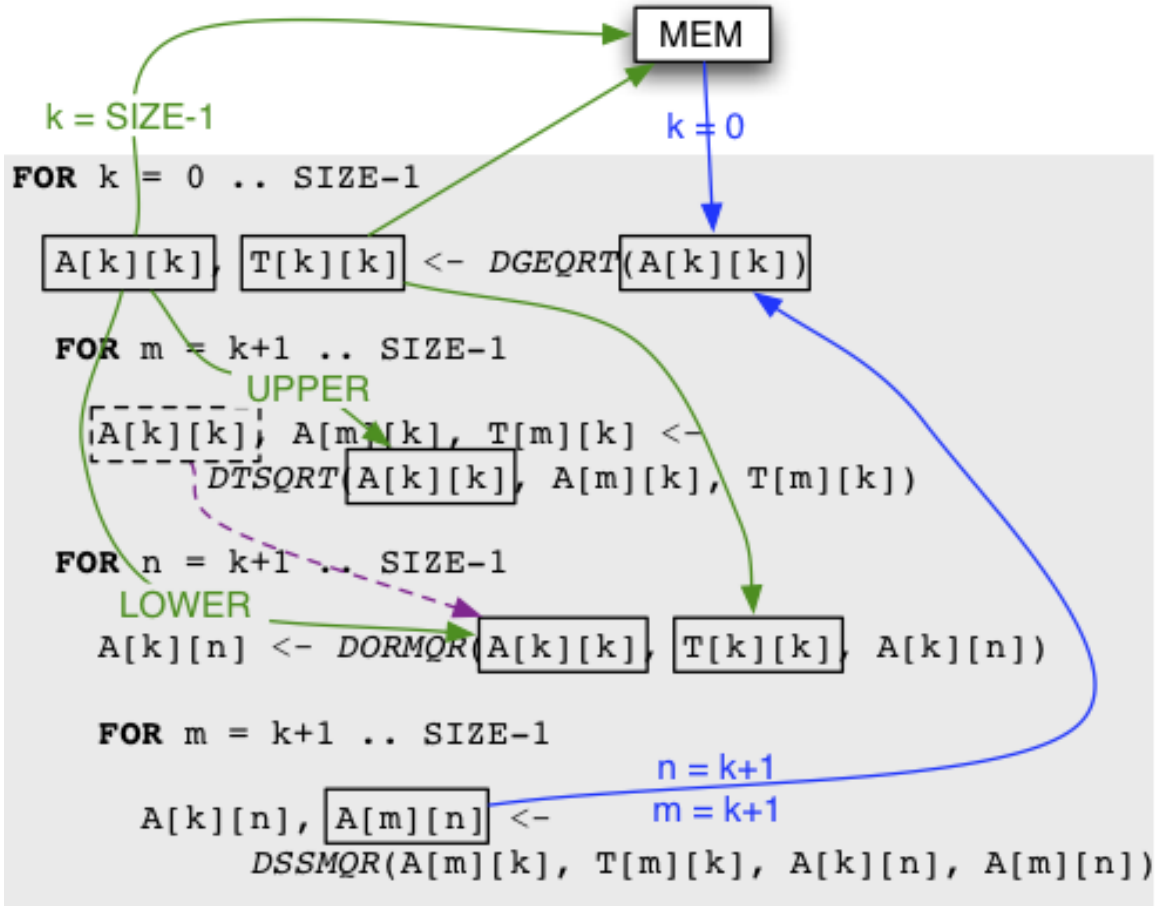
- **Serial definition as the starting poing**

```
FOR k = 0 .. SIZE-1
  A[k][k], T[k][k] <- DGEQRT(A[k][k])
  FOR m = k+1 .. SIZE-1
    A[k][k], A[m][k], T[m][k] <-
         DTSQRT(A[k][k], A[m][k], T[m][k])
  FOR n = k+1 .. SIZE-1
    A[k][n] <- DORMQR(A[k][k], T[k][k], A[k][n])
    FOR m = k+1 .. SIZE-1
      A[k][n], A[m][n] <-
           DSSMQR(A[m][k], T[m][k], A[k][n], A[m][n])
```

PaRSEC

Parallel Runtime Scheduling ane Execution Control

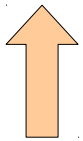
- **Translation to PTG through symbolic analysis**



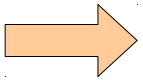
PaRSEC

Parallel Runtime Scheduling ane Execution Control

```
FOR k=0 TO N-1
  DGEQRT( inout A_kk )
  FOR n=k+1 to N
    DORMQR( in A◹_kk, inout A_kn )
  FOR m=k+1 to N
    DTSQRT( inout A◹_kk, inout A_mk )
    FOR n=k+1 to N
      DTSMQR( in A_mk, inout A_kn, inout A_mn )
```



serial

PTG 

a.k.a Job Dependency Format (JDF)

$DGEQRT_{kkk}$

$1_{ARG} \leftarrow A_{k,k} \mid DTSMQR_{k,k,k-1}$

$1_{ARG} \Rightarrow DORMQR_{k,k+1..N,k}(◹)$

$1_{ARG} \Rightarrow DTSQRT_{k+1,k,k}(◹)$

$1_{ARG} \Rightarrow A_{k,k}(◹)$

$DORMQR_{knk}$

$1_{ARG} \leftarrow DGEQRT_{k,k,k}(◹)$

$2_{ARG} \leftarrow A_{k,n} \mid DTSMQR_{k,n,k-1}$

$2_{ARG} \Rightarrow DTSMQR_{k+1,n,k}$

$2_{ARG} \Rightarrow A_{k,n}$

$DTSQRT_{mkk}$

$1_{ARG} \leftarrow DGEQRT_{m-1,k,k}(◹) \mid DTSQRT_{m-1,k,k}(◹)$

$1_{ARG} \Rightarrow DTSQRT_{m+1,k,k}(◹) \mid A_{k,k}(◹)$

$2_{ARG} \leftarrow A_{m,k} \mid DTSMQR_{m,k,k-1}$

$2_{ARG} \Rightarrow DTSMQR_{m,k+1..N,k}$

$2_{ARG} \Rightarrow A_{m,k}$

$DTSMQR_{mnk}$

$1_{ARG} \leftarrow DTSQRT_{m,k,k}$

$2_{ARG} \leftarrow DORMQR_{m-1,n,k} \mid DTSMQR_{m-1,n,k}$

$2_{ARG} \Rightarrow DTSMQR_{m+1,n,k} \mid A_{n,k}$

$3_{ARG} \leftarrow A_{m,n} \mid DTSMQR_{m,n,k-1}$

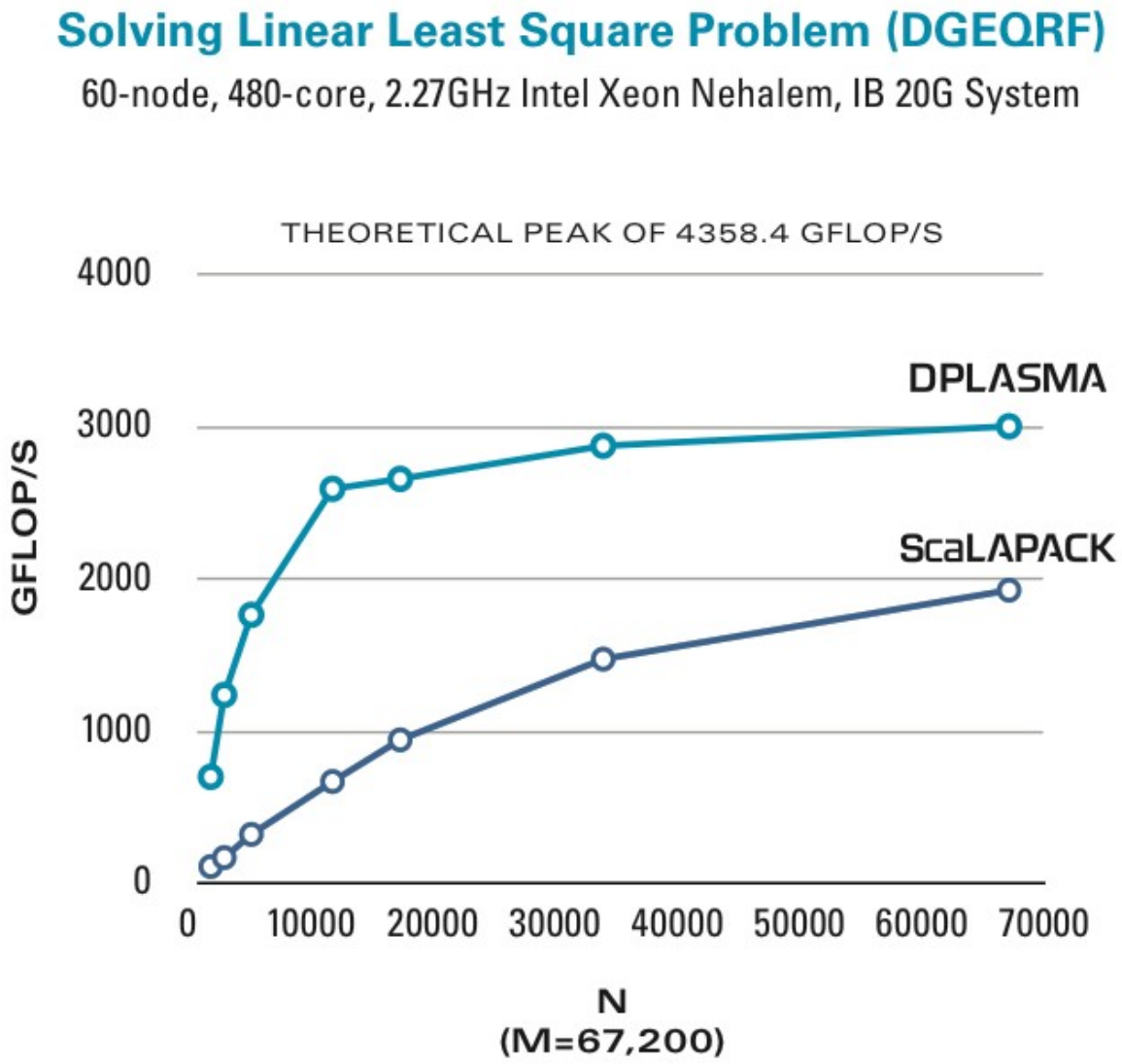
$3_{ARG} \Rightarrow DGEQRT_{m,n,k+1} \mid DORMQR_{m,n,k+1} \mid$

$\Rightarrow DTSQRT_{m,n,k+1} \mid DTSMQR_{m,n,k+1} \mid$

$\Rightarrow A_{m,n}$

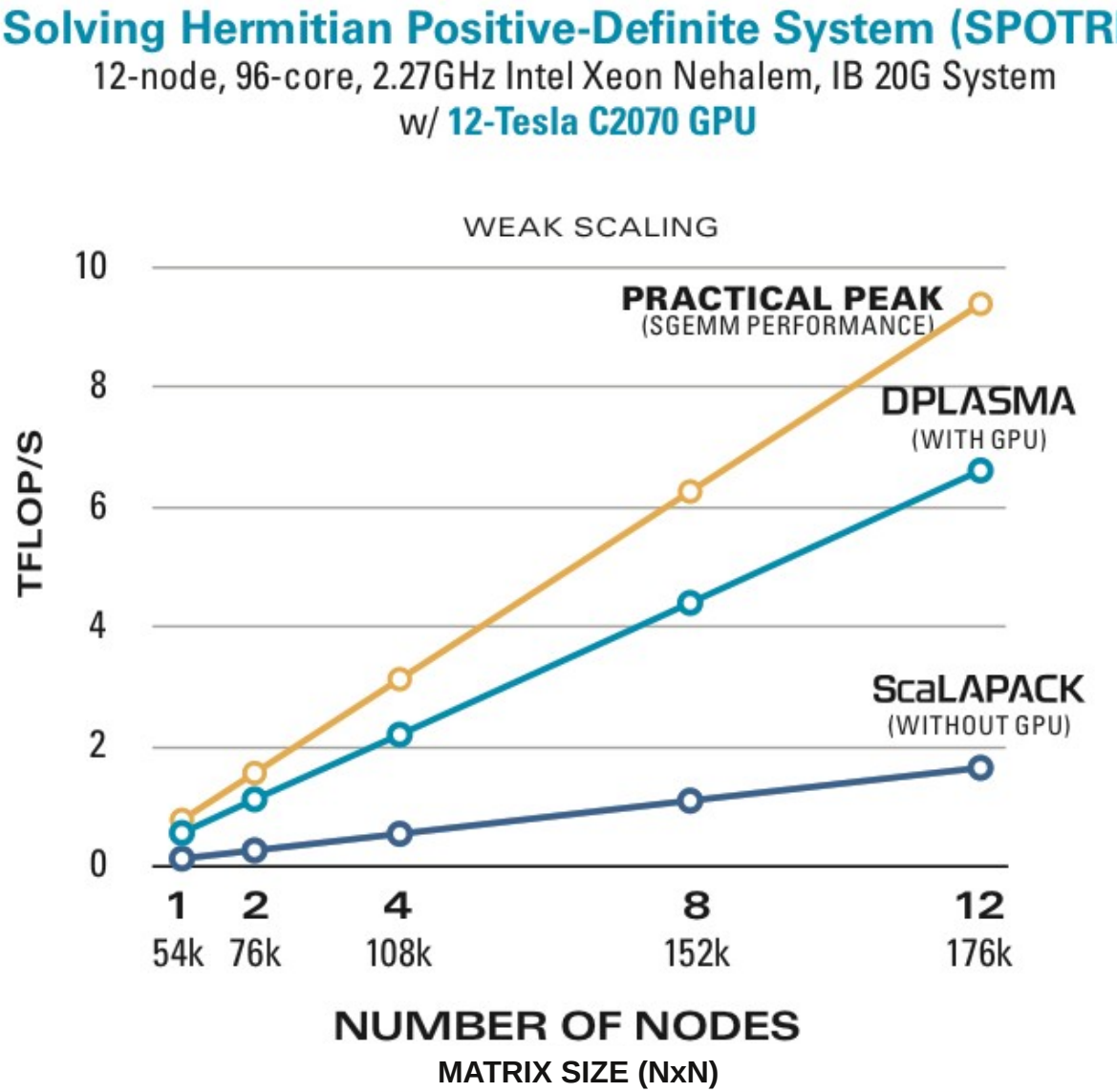
DPLASMA / PaRSEC

performance



DPLASMA / PaRSEC

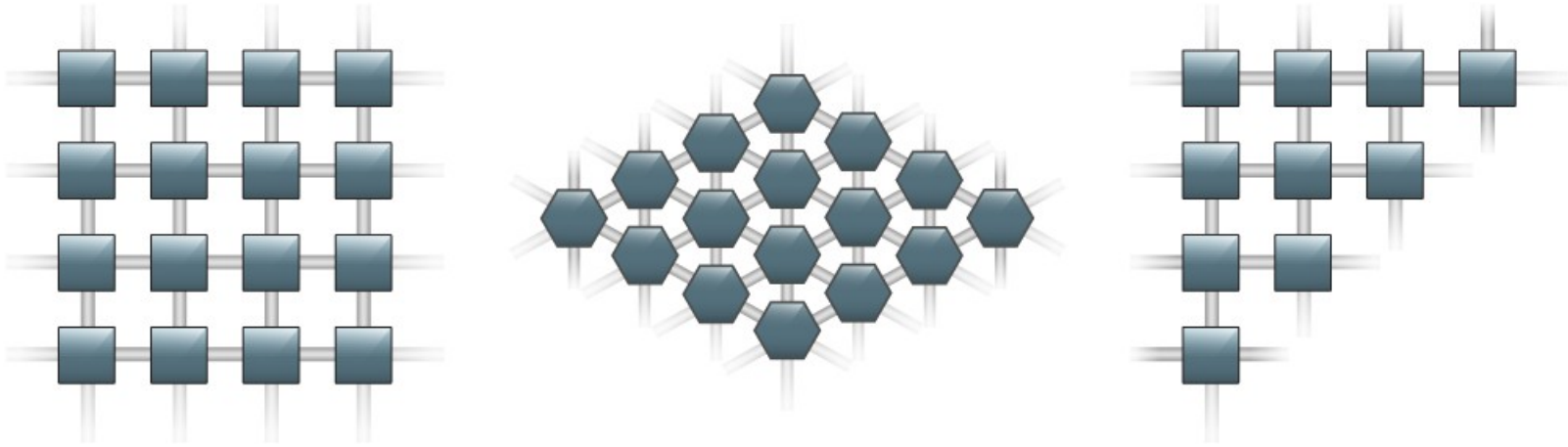
performance



http://web.eecs.utk.edu/~kurzak/tutorials/

PULSAR

Parallel Unified Linear algebra with Systolic ARrays



http://web.eecs.utk.edu/~kurzak/tutorials/

Questions?