QUARK

Queuing And Runtime for Kernels

Innovative Computing Laboratory
Electrical Engineering and Computer Science
University of Tennessee

Piotr Luszczek (presenter)

ICL UT INNOVATIVE COMPUTING LABORATORY THE UNIVERSITY of TENNESSEE

web.eecs.utk.edu/~luszczek/conf/

TOC: Table of Contents

Serial Program: Linear Algebra Loop Nests

```
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])
```

tile Cholesky

serial definitions

```
FOR k = 0..TILES-1
    A[k][k], T[k][k] ← DGRQRT(A[k][k])
    FOR m = k+1..TILES-1
        A[k][k], A[m][k], T[m][k] ← DTSQRT(A[k][k], A[m][k], T[m][k])
    FOR n = k+1..TILES-1
        A[k][n] ← DLARFB(A[k][k], T[k][k], A[k][n])
        FOR m = k+1..TILES-1
            A[k][n], A[m][n] ← DSSRFB(A[m][k], T[m][k], A[k][n], A[m][n])
```

tile QR

Parallel Program: Manual Multithreading

definition

```
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])
```

code

```
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;
}
```

fixed task assignment
progress table synchronization

QUARK Basics

- Superscalar Scheduling
  - serial code
  - side-effect-free tasks
  - dependency resolution
- Resolving Data Hazards
  - Read After Write (RAW)
  - Write after Read (WAR)
  - Write after Write (WAW)

- Similar Projects
  - SMPSs from Barcelona SC
  - StarPU from INRIA Bordeaus
  - Jade from Stanford (historical)
- Deceptively similar projects
  - Cilk (++)
  - Intel Thread Building Blocks
  - Apple Grand Central Dispatch
  - OpenMP Tasks

QUARK: 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

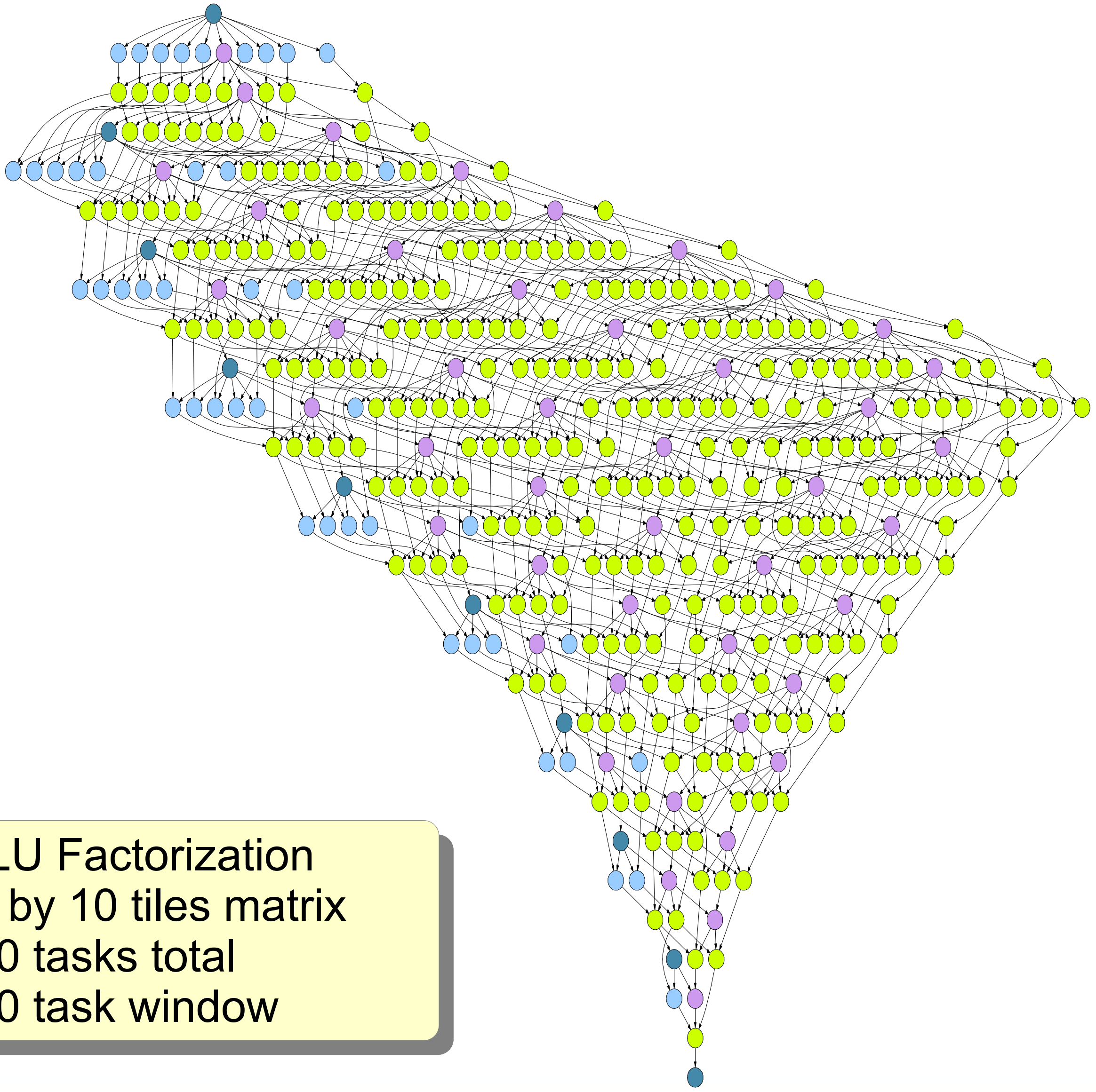
QUARK: Queuing a Task

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

Scalars (VALUE) – pass by value semantics

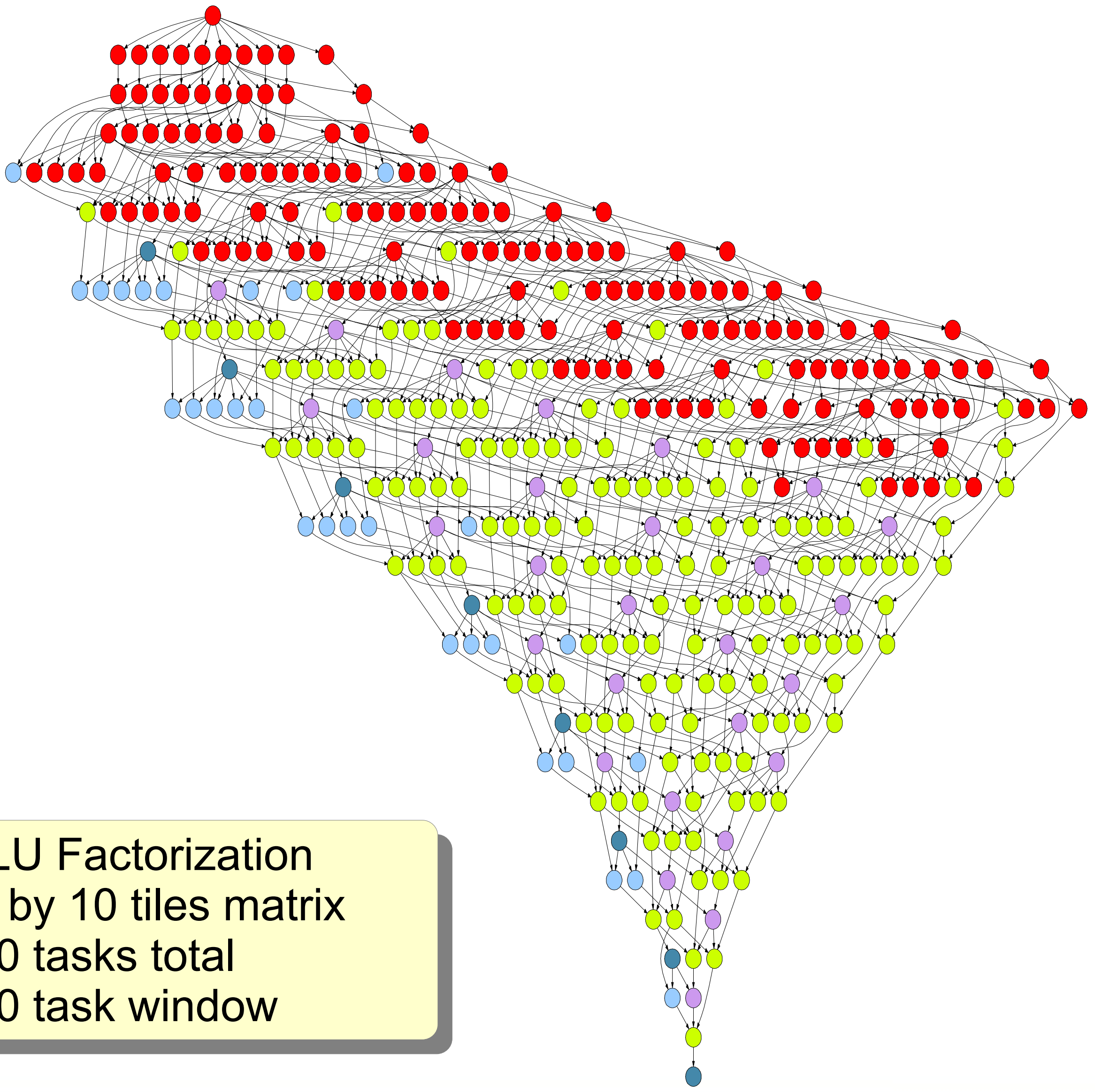
```
QUARK_Insert_Task(quark, CORE_dtrsm_quark, task_flags,
    sizeof(PLASMA_enum),     &side,       VALUE,
    sizeof(PLASMA_enum),     &uplo,       VALUE,
    sizeof(PLASMA_enum),     &trans,      VALUE,
    sizeof(PLASMA_enum),     &diag,       VALUE,
    sizeof(int),             &m,          VALUE,
    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);
```

DAG Exploration: Sliding Window Step 0



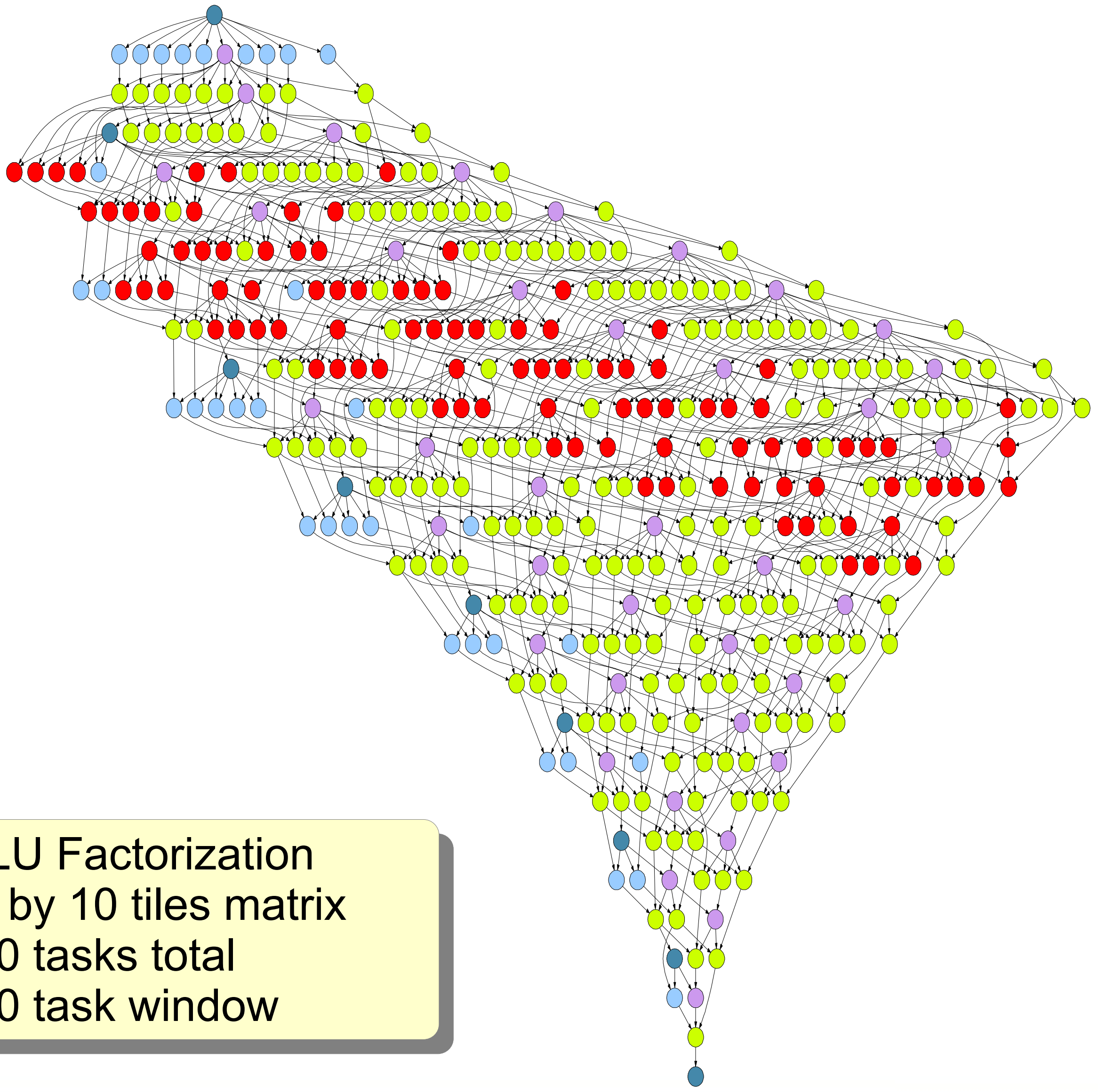
- Tile LU Factorization
  - 10 by 10 tiles matrix
  - 300 tasks total
  - 100 task window

DAG Exploration: Sliding Window Step 1



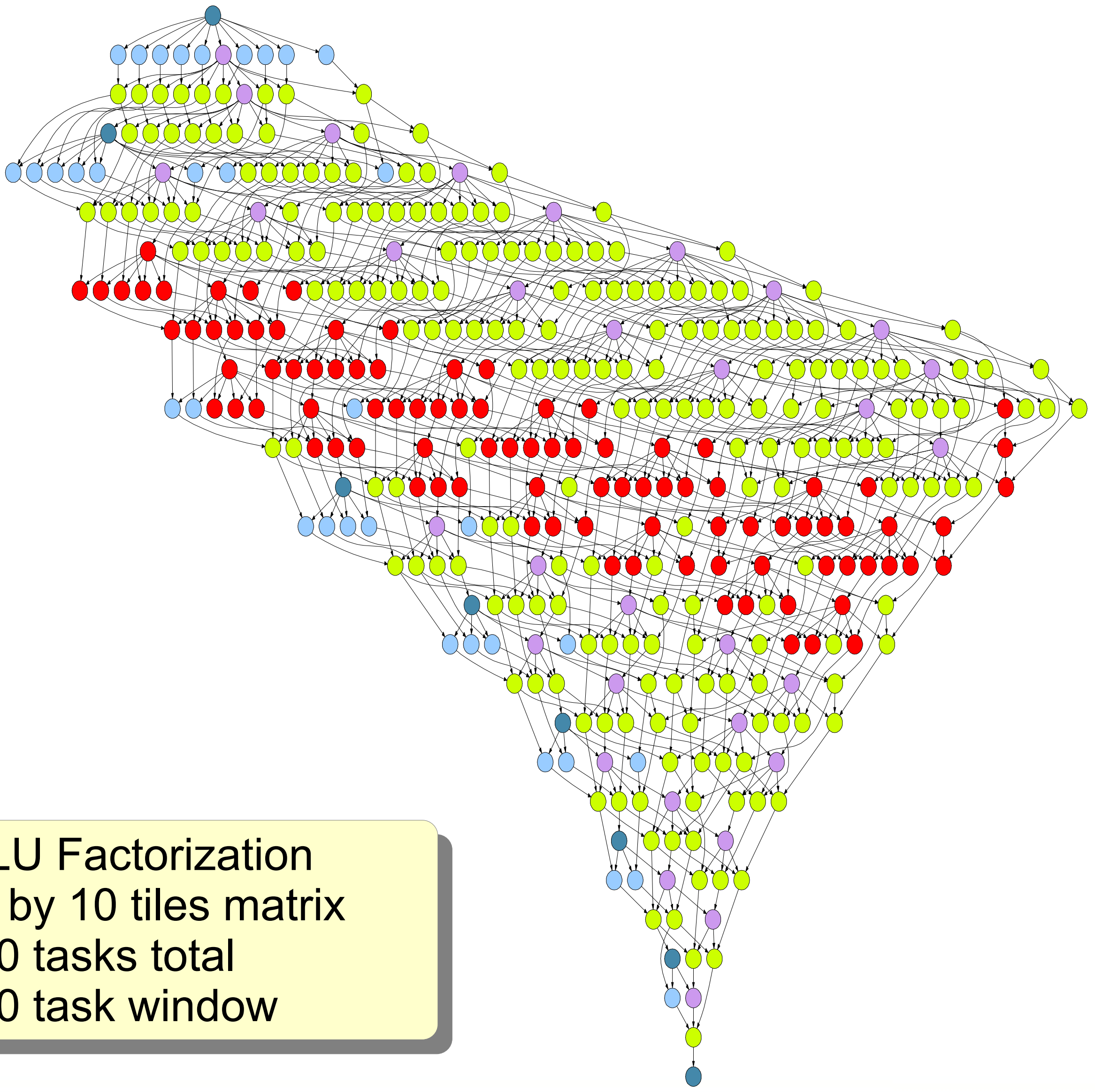
- Tile LU Factorization
  - 10 by 10 tiles matrix
  - 300 tasks total
  - 100 task window

DAG Exploration: Sliding Window Step 2



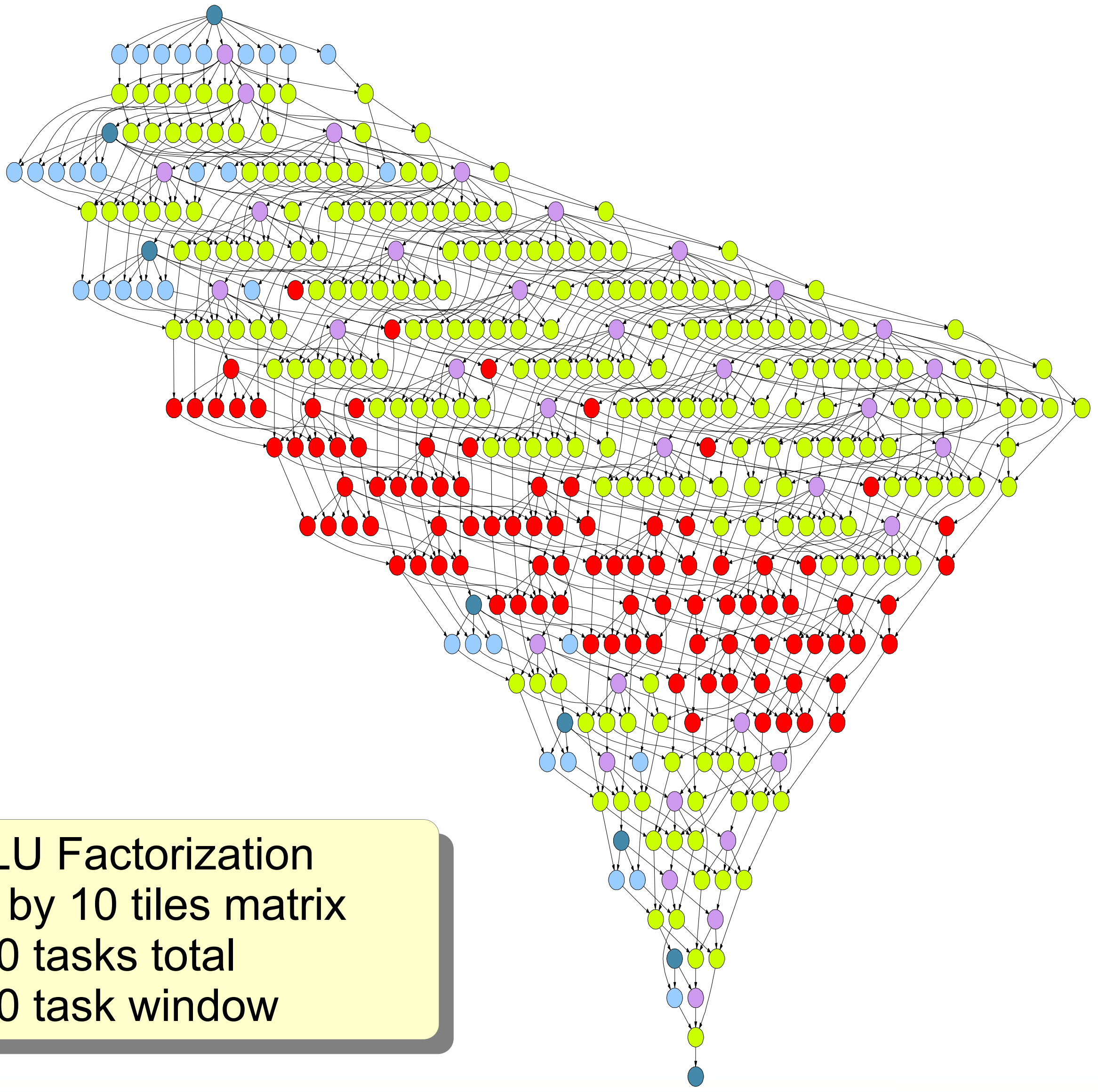
- Tile LU Factorization
  - 10 by 10 tiles matrix
  - 300 tasks total
  - 100 task window

DAG Exploration: Sliding Window Step 3



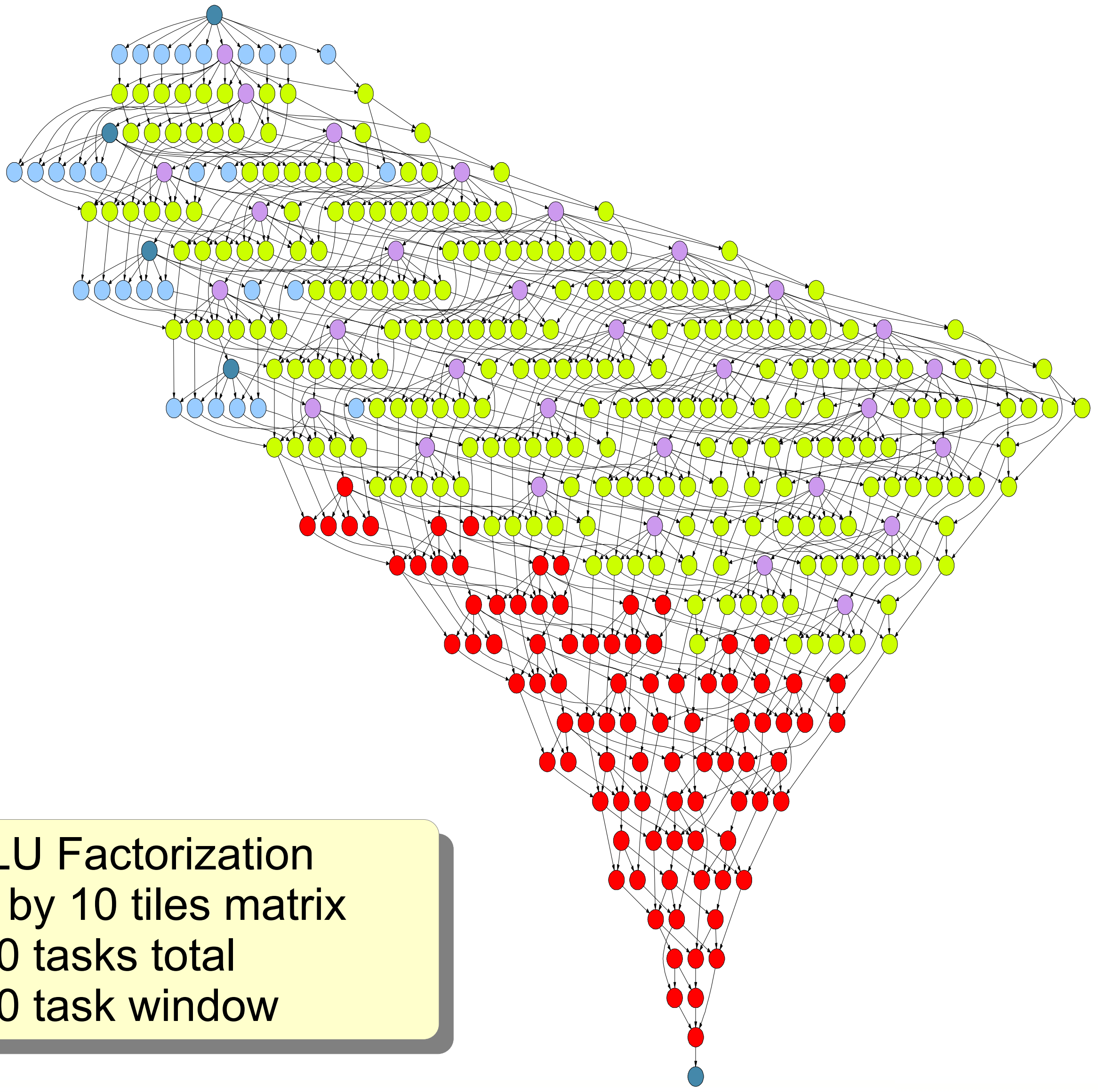
- Tile LU Factorization
  - 10 by 10 tiles matrix
  - 300 tasks total
  - 100 task window

DAG Exploration: Sliding Window Step 4



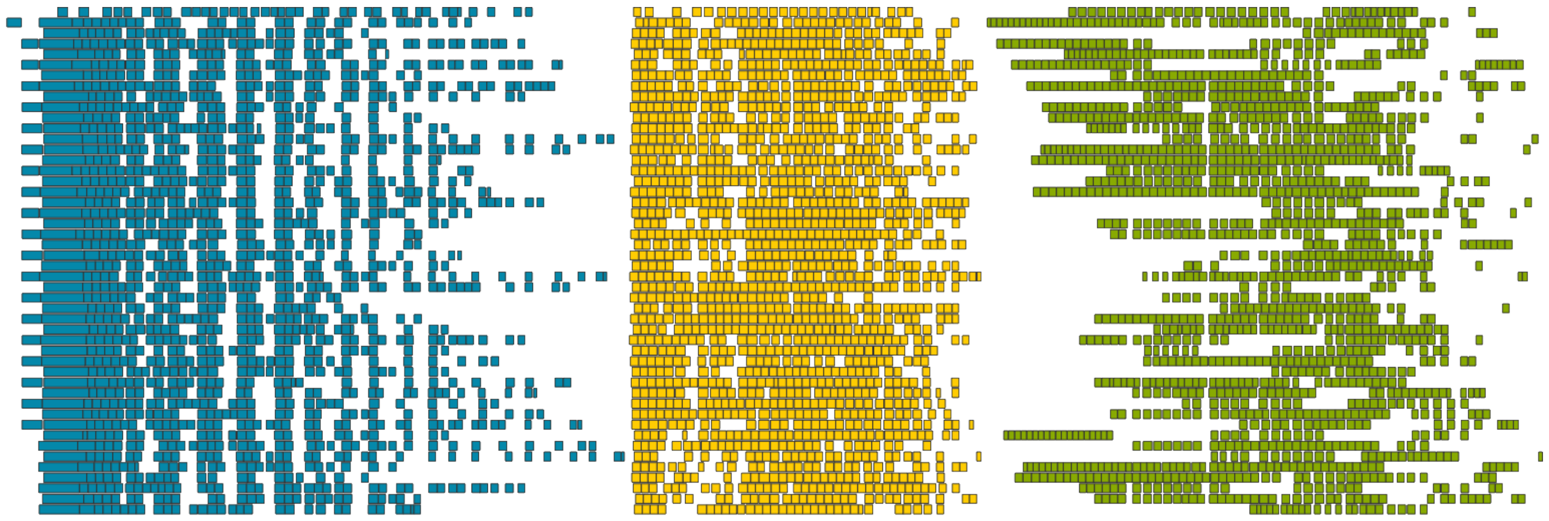
- Tile LU Factorization
  - 10 by 10 tiles matrix
  - 300 tasks total
  - 100 task window

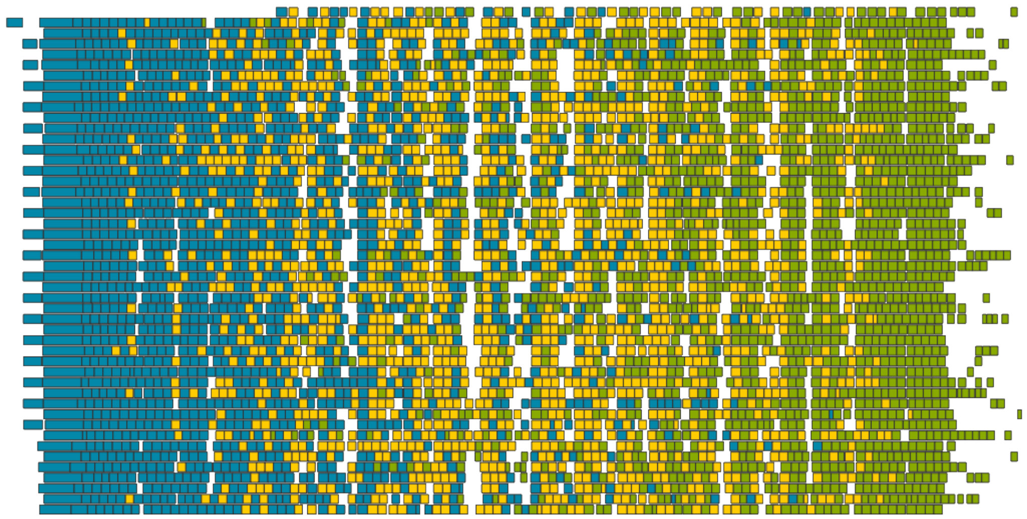
DAG Exploration: Sliding Window Step 5



- Tile LU Factorization
  - 10 by 10 tiles matrix
  - 300 tasks total
  - 100 task window

QUARK Parallel Composition





<u>POTRI:</u>
POTRF
TRTRI
LAUUM

QUARK Features

- Cancellation of a task
- Cancellation of a sequence of tasks
- Priority hinting
- Locality (data reuse) hinting
- “Accumulator” tasks (enables reordering)
- “Gatherv” tasks (allows simultaneous writes)
- Nested-parallel tasks
- Locking to a thread
- Locking to a thread mask
- Incremental lists of dependencies
- DAG plotting (custom colors, custom labels)

QUARK 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 Hinting

- 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: Relaxing Dependencies

- “Accumulator” tasks
  - Data item can be flagged with the “accumulator” flag.
  - 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.
  - The tasks operate on disjoint parts of the data and can execute simultaneously without causing race conditions.

QUARK Locking to Threads

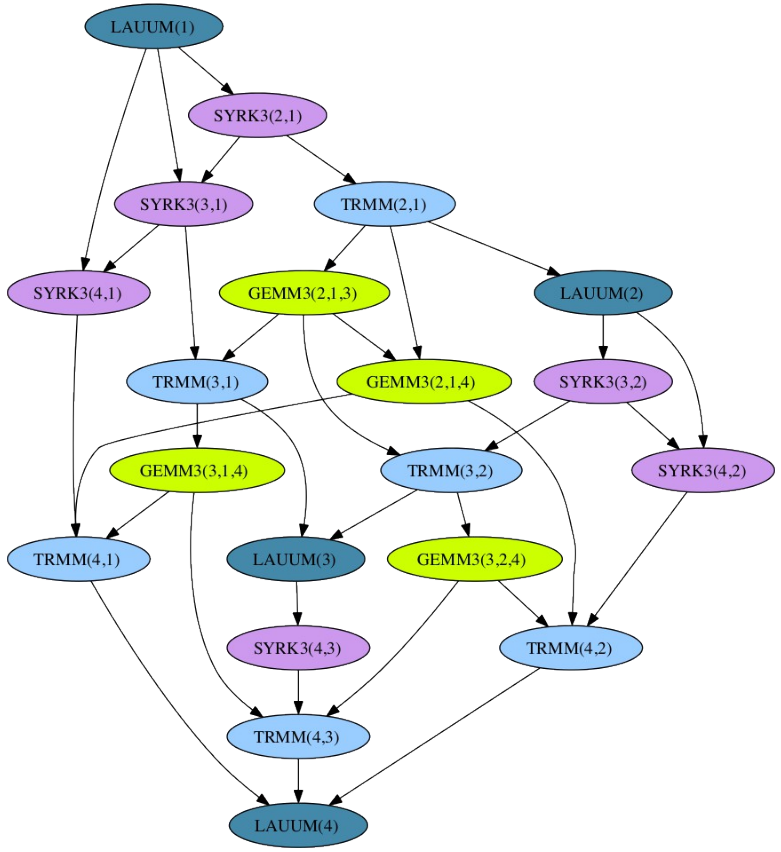
- Locking to a thread
  - A task can be locked to a particular thread.
  - Other threads will not be allowed to steal that task through work stealing.
- Locking to a thread mask
  - A task can be confined to a subset of threads by using a bit mask.
  - QUARK will schedule the task to one of the treads in the bit mask.
  - Outside threads will not be allowed to steal that task through work stealing.

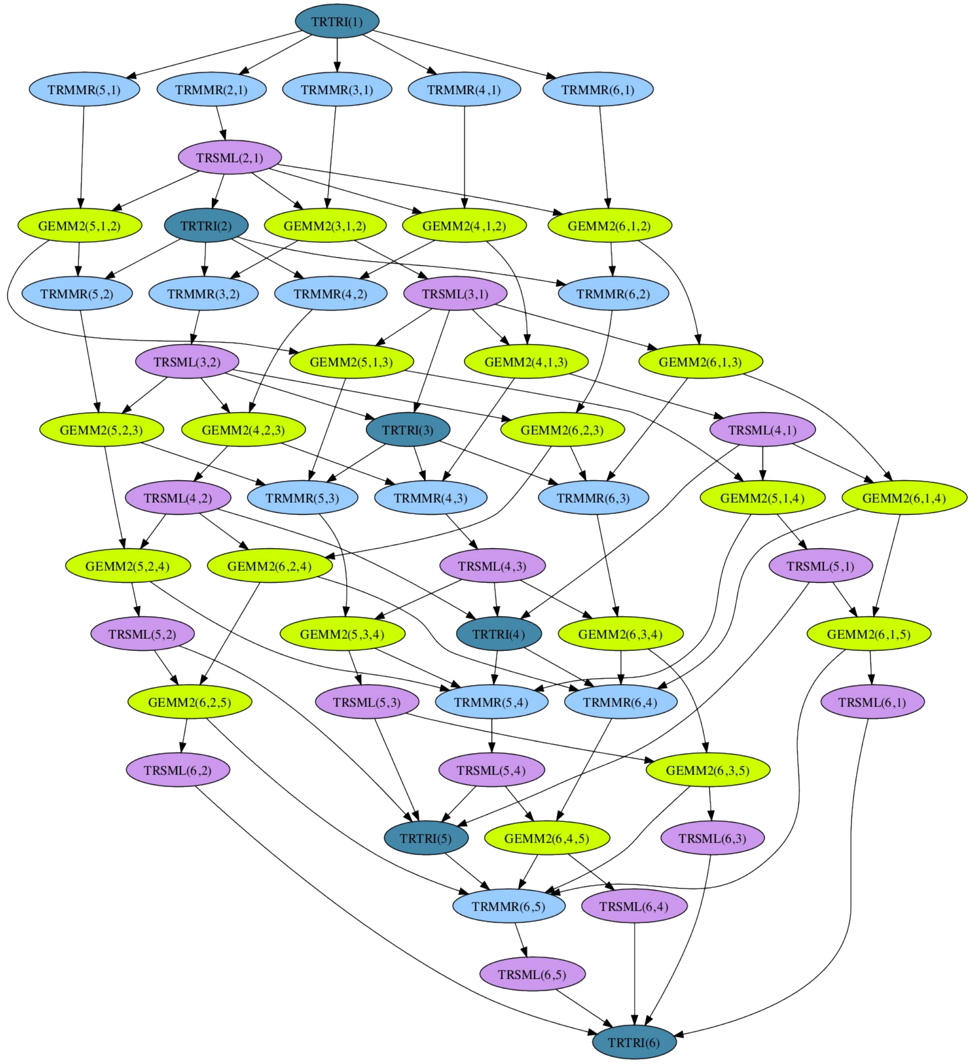
QUARK Controlling Granularity

- 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.

QUARK DAG Plotting

- Custom colors
- Custom labels



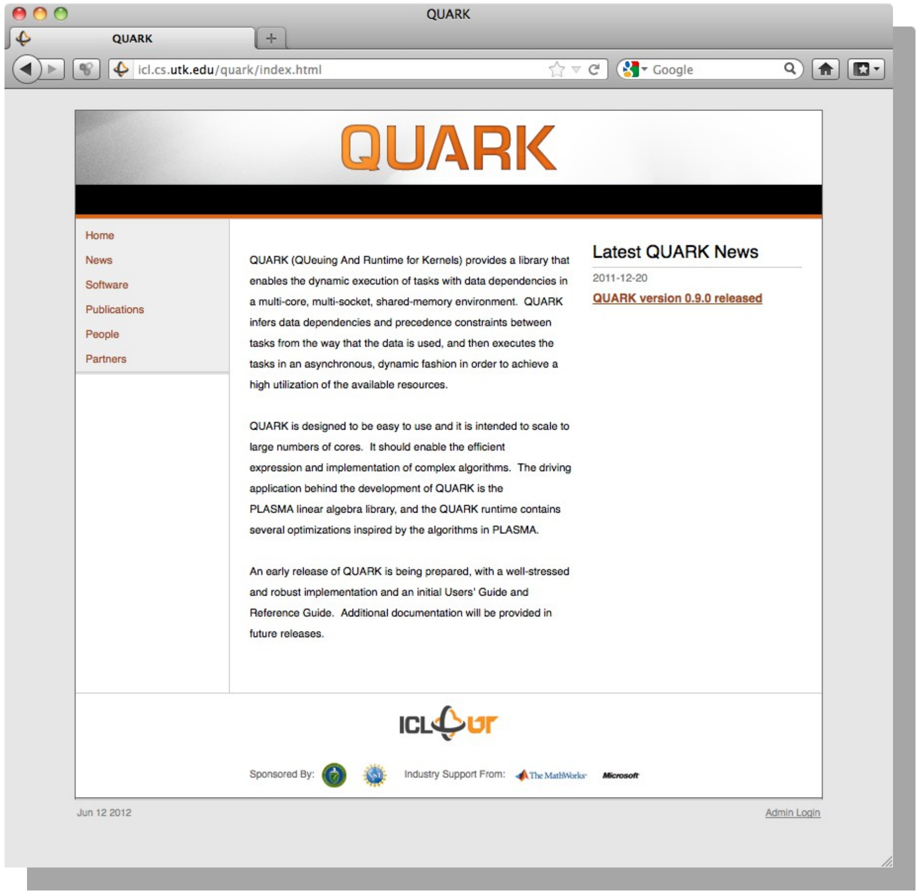


QUARK Resources

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

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

Users' Guide
in /docs/pdf/ after installation


QUARK

icl.cs.utk.edu/quark/index.html

QUARK

Home
News
Software
Publications
People
Partners

QUARK (QUeuing And Runtime for Kernels) provides a library that enables the dynamic execution of tasks with data dependencies in a multi-core, multi-socket, shared-memory environment. QUARK infers data dependencies and precedence constraints between tasks from the way that the data is used, and then executes the tasks in an asynchronous, dynamic fashion in order to achieve a high utilization of the available resources.

QUARK is designed to be easy to use and it is intended to scale to large numbers of cores. It should enable the efficient expression and implementation of complex algorithms. The driving application behind the development of QUARK is the PLASMA linear algebra library, and the QUARK runtime contains several optimizations inspired by the algorithms in PLASMA.

An early release of QUARK is being prepared, with a well-stressed and robust implementation and an initial Users' Guide and Reference Guide. Additional documentation will be provided in future releases.

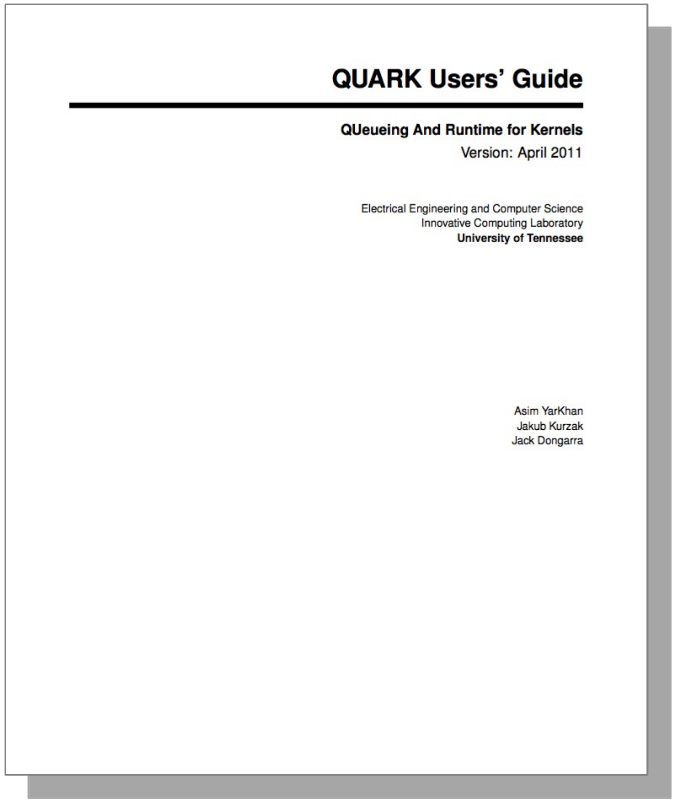
Latest QUARK News

2011-12-20

QUARK version 0.9.0 released

Sponsored By: Industry Support From: The MathWorks Microsoft

Jun 12 2012 Admin Login


QUARK Users' Guide

QUeueing And Runtime for Kernels

Version: April 2011

Electrical Engineering and Computer Science
Innovative Computing Laboratory
University of Tennessee

Asim YarKhan
Jakub Kurzak
Jack Dongarra