Targeting Multi-Core systems in Linear Algebra applications

Alfredo Buttari, Jack Dongarra, Jakub Kurzak and Julien Langou

presented by Dan Terpstra
terpstra@cs.utk.edu

CScADS Autotuning Workshop

Snowbird, Utah, July 9 - 12, 2007

ICL UT Innovative Computing Laboratory

The University of Tennessee Computer Science Department

The free lunch is over

Hardware

Problem	Solution
• power consumption • heat dissipation • pins	reduce clock and increase execution units = Multicore

Software

Consequence

Non-parallel software won't run any faster. A new approach to programming is required.

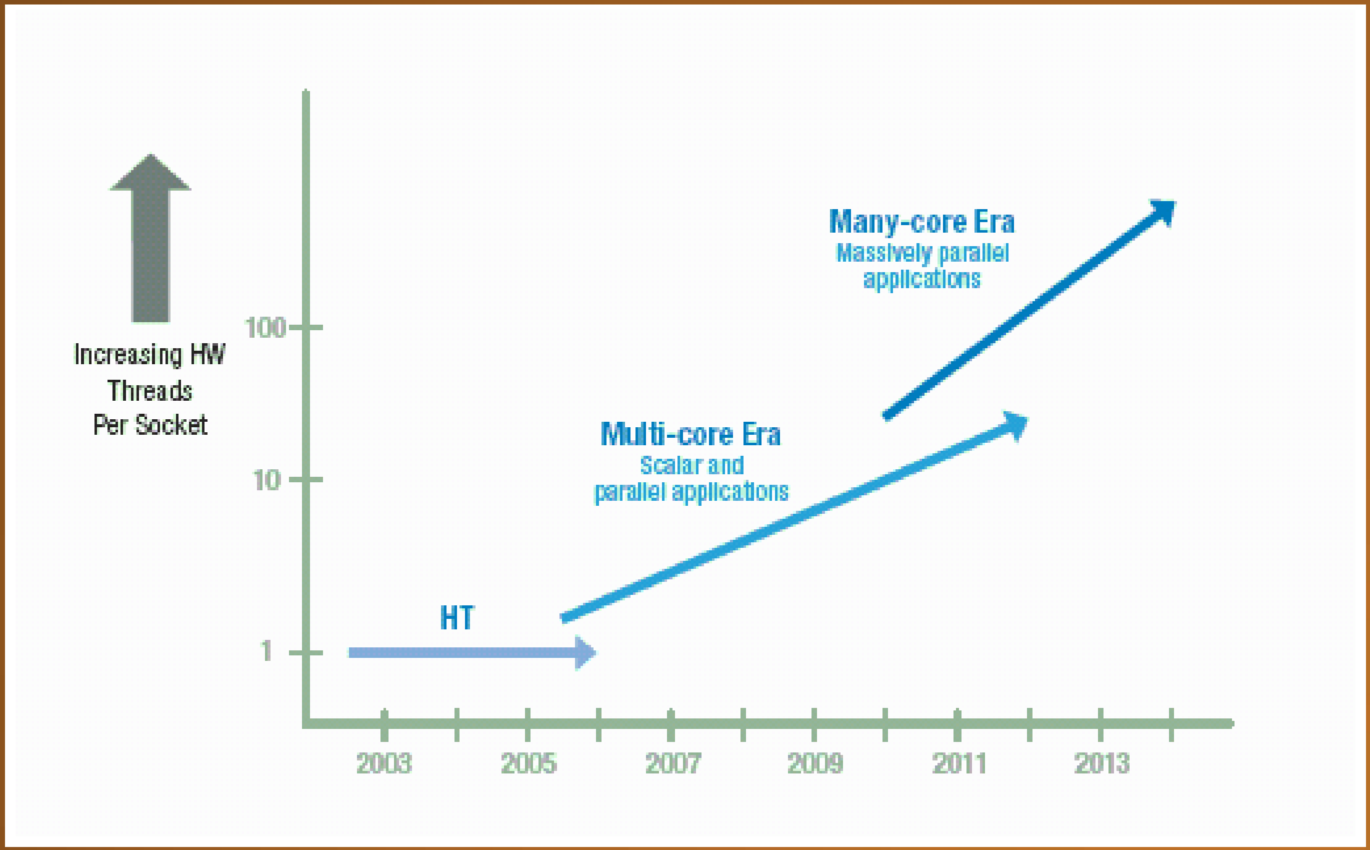
What is a Multicore processor, BTW?

"a processor that combines two or more independent processors into a single package" (wikipedia)

What about:

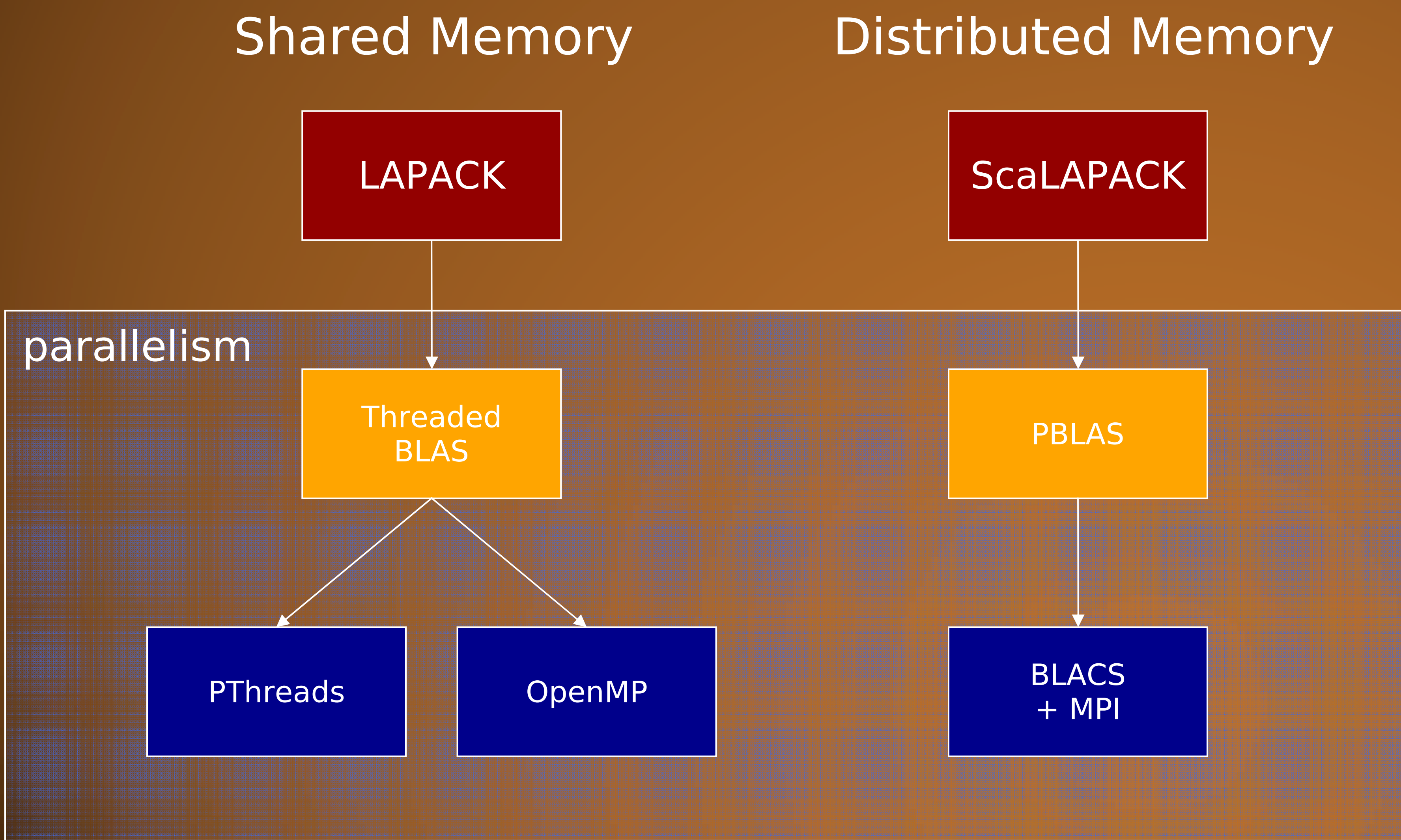
- types of core?
  - homogeneous (AMD Opteron, Intel Woodcrest...)
  - heterogeneous (STI Cell, Sun Niagara, NVIDIA...)
- memory?
  - how is it arranged?
- bus?
  - is it going to be fast enough?
- cache?
  - shared? (Intel/AMD)
  - not present at all? (STI Cell)
- communications?

What's the Multicore timeline?

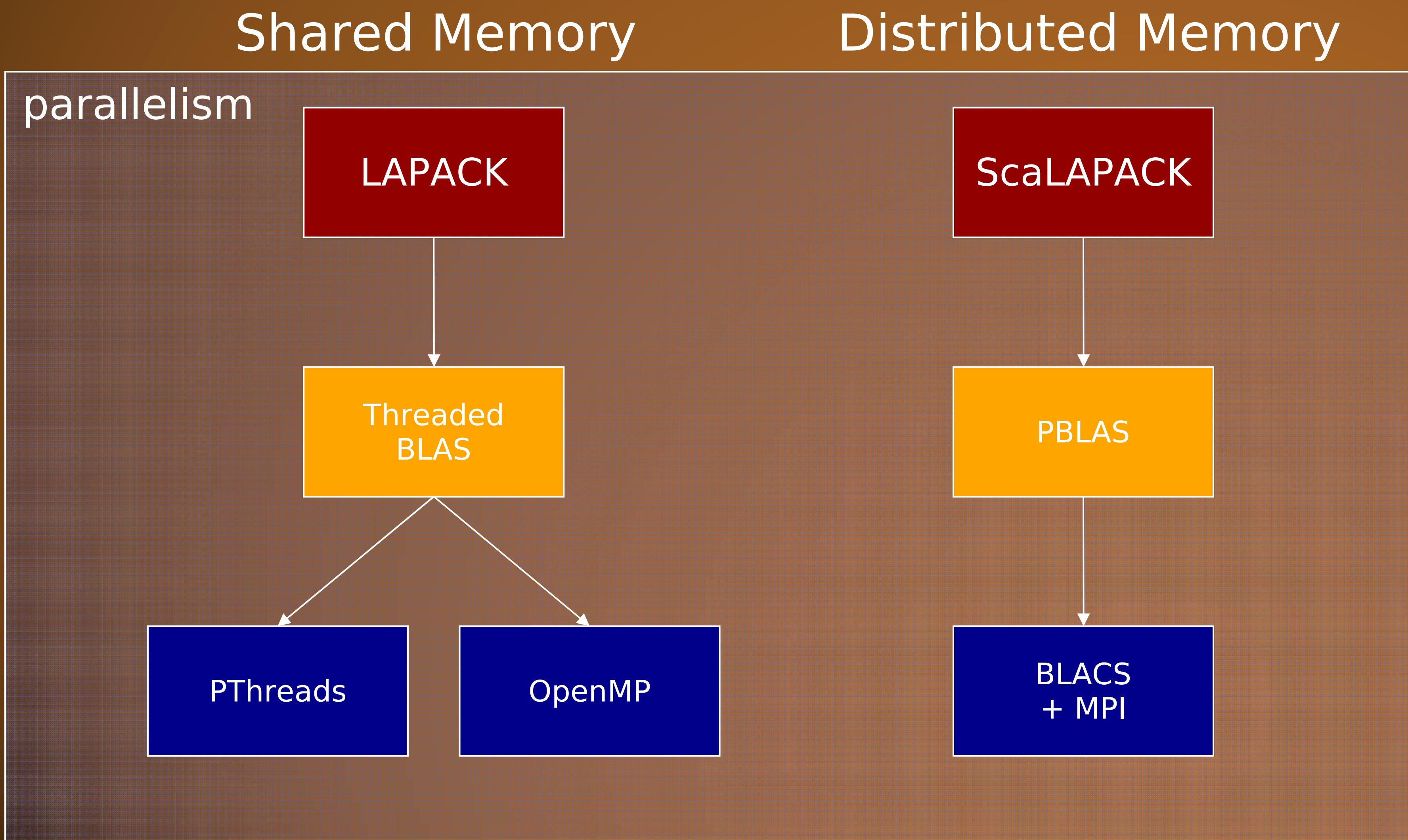


* Source: Platform 2015: Intel® Processor and Platform Evolution for the Next Decade, Intel White Paper (via LaBarta, et. al. SC06)

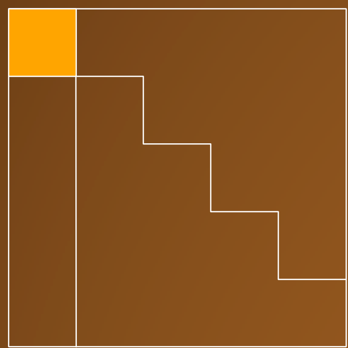
Parallelism in Linear Algebra software so far



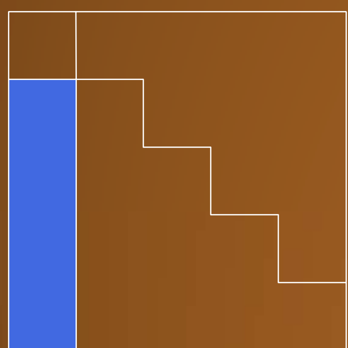
Parallelism in Linear Algebra software so far



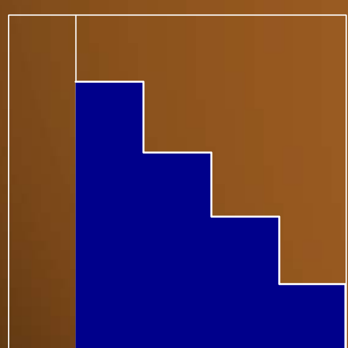
Parallelism in LAPACK: Cholesky factorization



DPOTF2: BLAS-2
non-blocked factorization of the panel

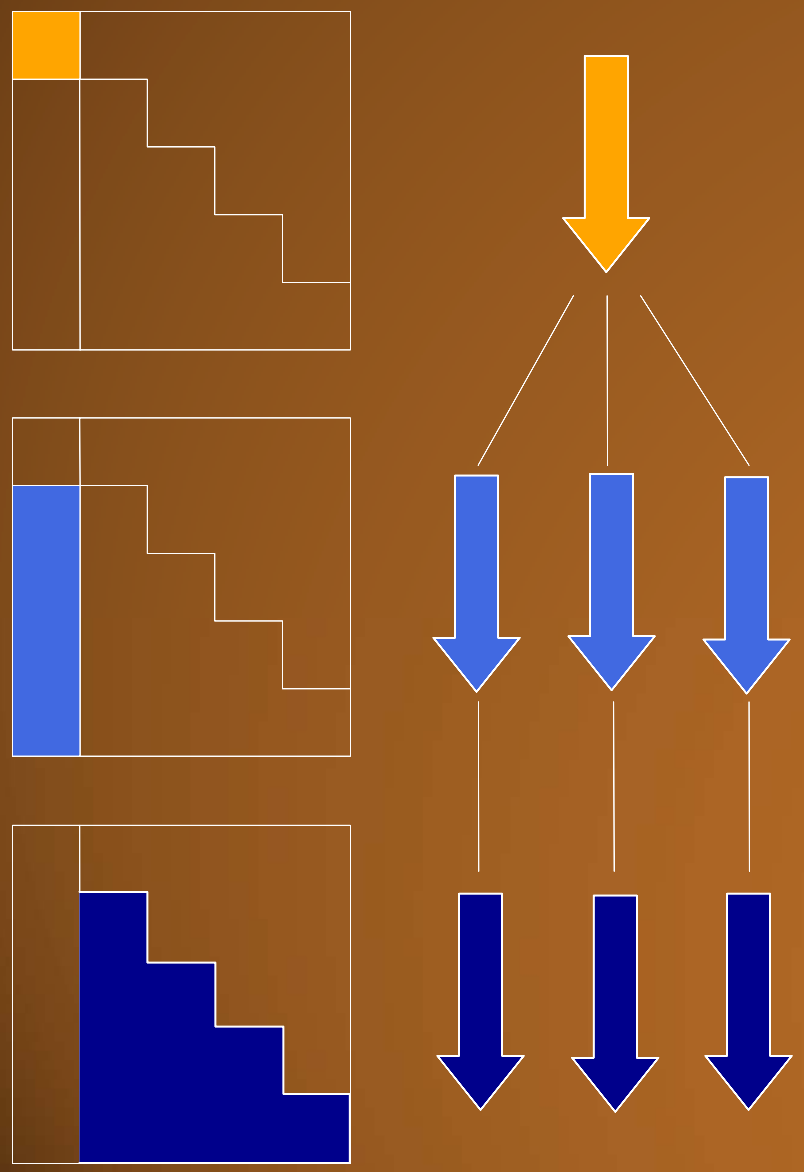


DTRSM: BLAS-3
updates by applying the $U = L^T$
transformation computed in DPOTF2



DGEMM (DSYRK): BLAS-3
updates trailing submatrix

Parallelism in LAPACK: Cholesky factorization

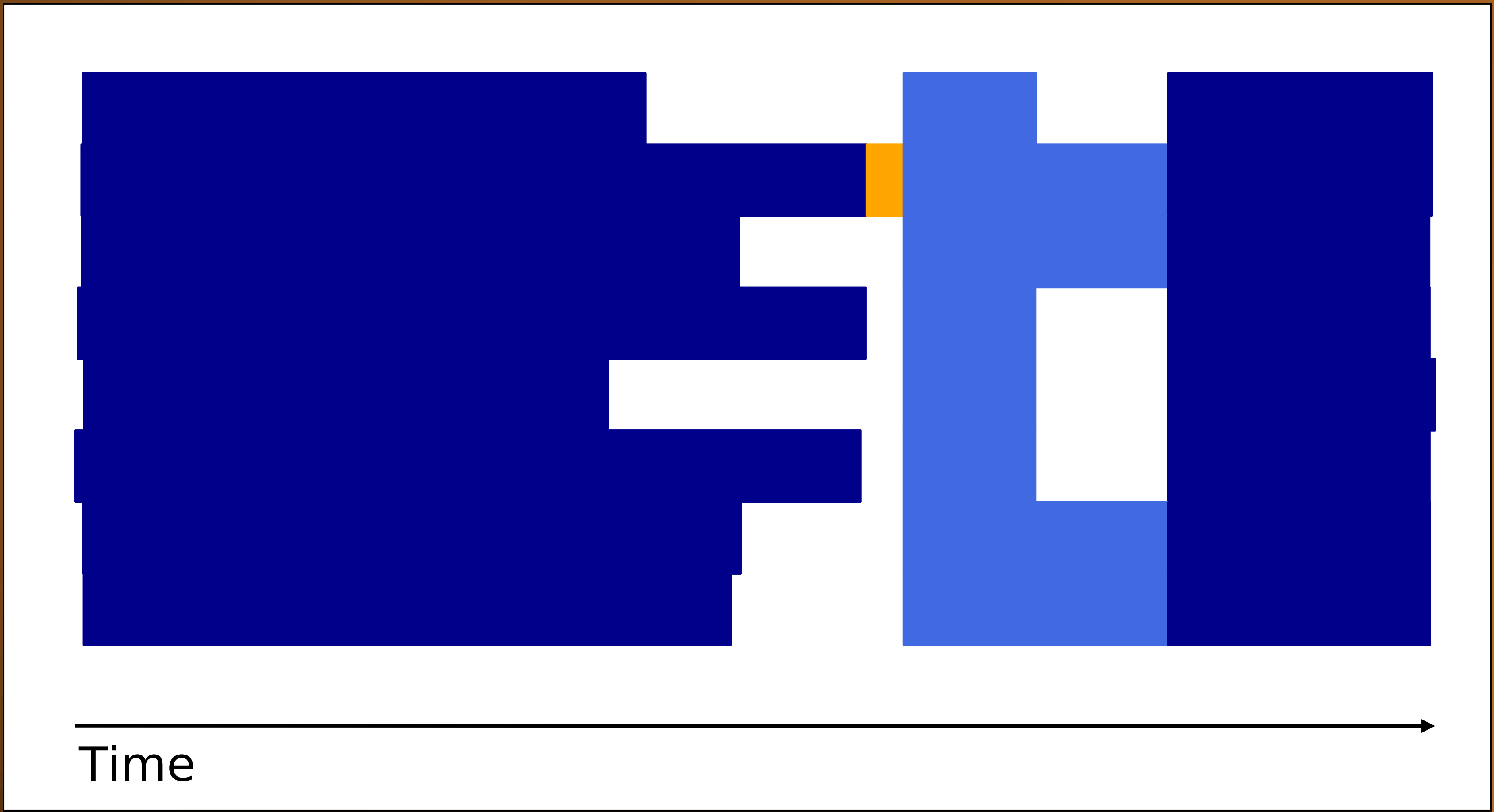


BLAS2 operations cannot be efficiently parallelized because they are bandwidth bound.

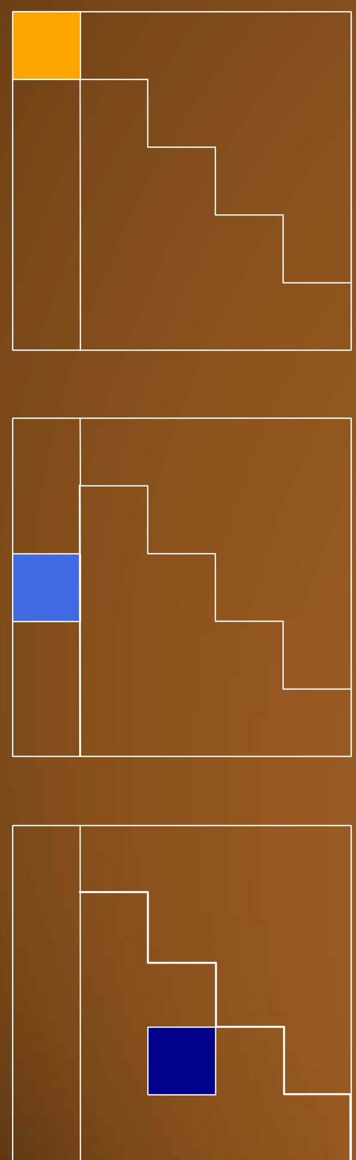
- strict synchronizations
- poor parallelism
- poor scalability

Parallelism in LAPACK: Cholesky factorization

The execution flow if filled with stalls due to synchronizations and sequential operations.



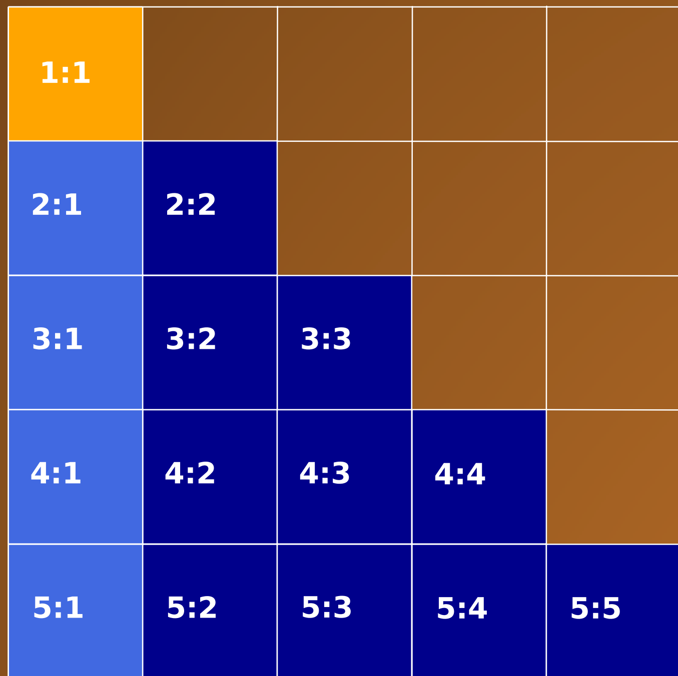
Parallelism in LAPACK: Cholesky factorization

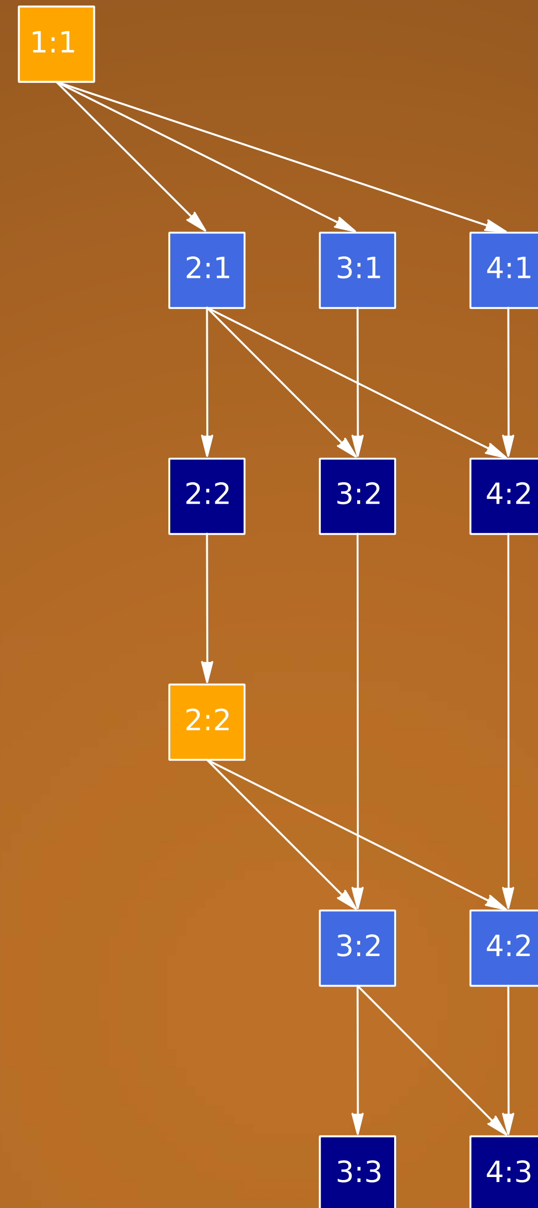


Tiling operations:

```
do DPOTF2 on

    for all
        do DTRSM on
    end

    for all
        do DGEMM on
    end

end
```

Parallelism in LAPACK: Cholesky factorization



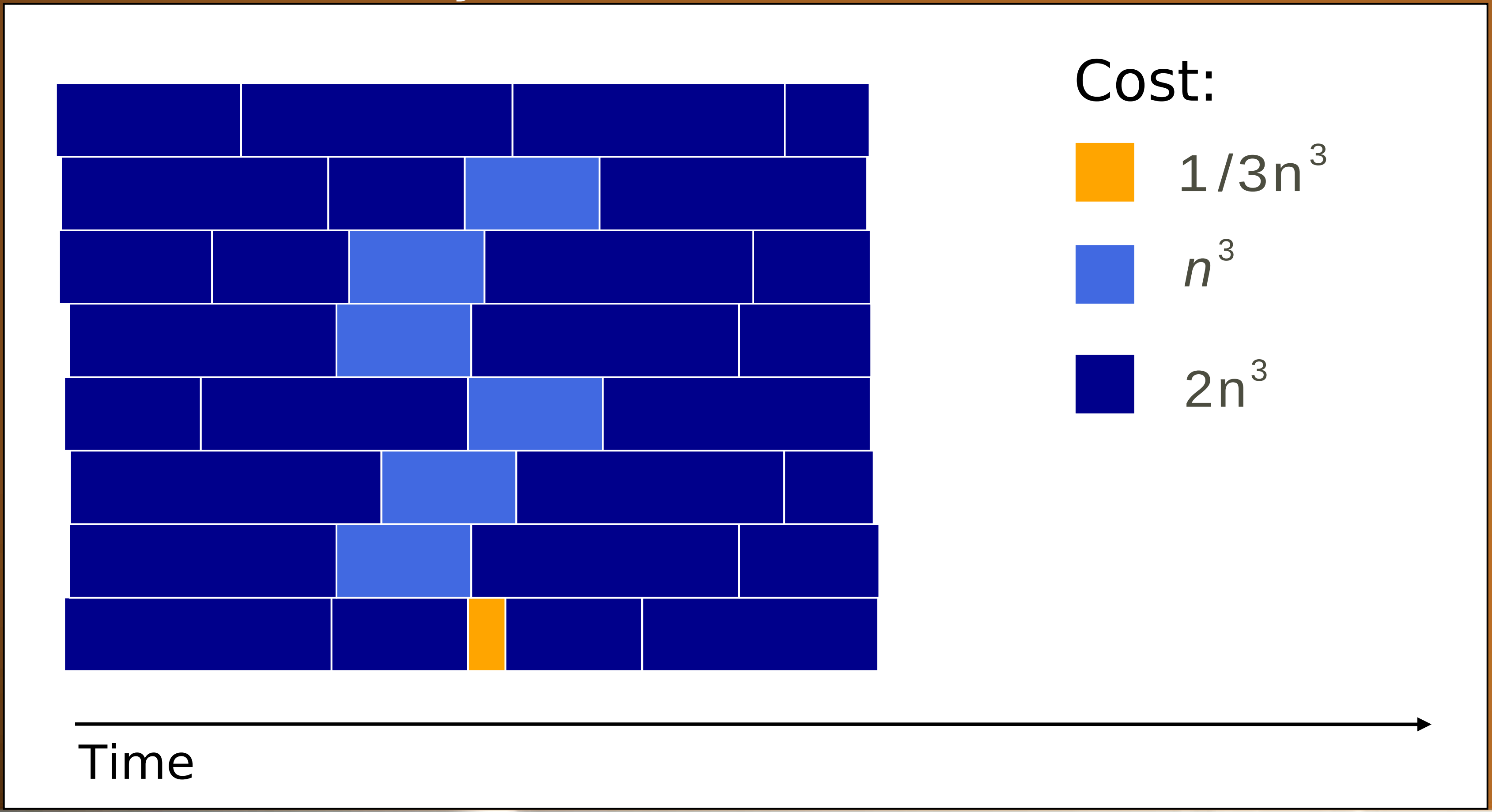


Cholesky can be represented as a Directed Acyclic Graph (DAG) where nodes are subtasks and edges are dependencies among them.

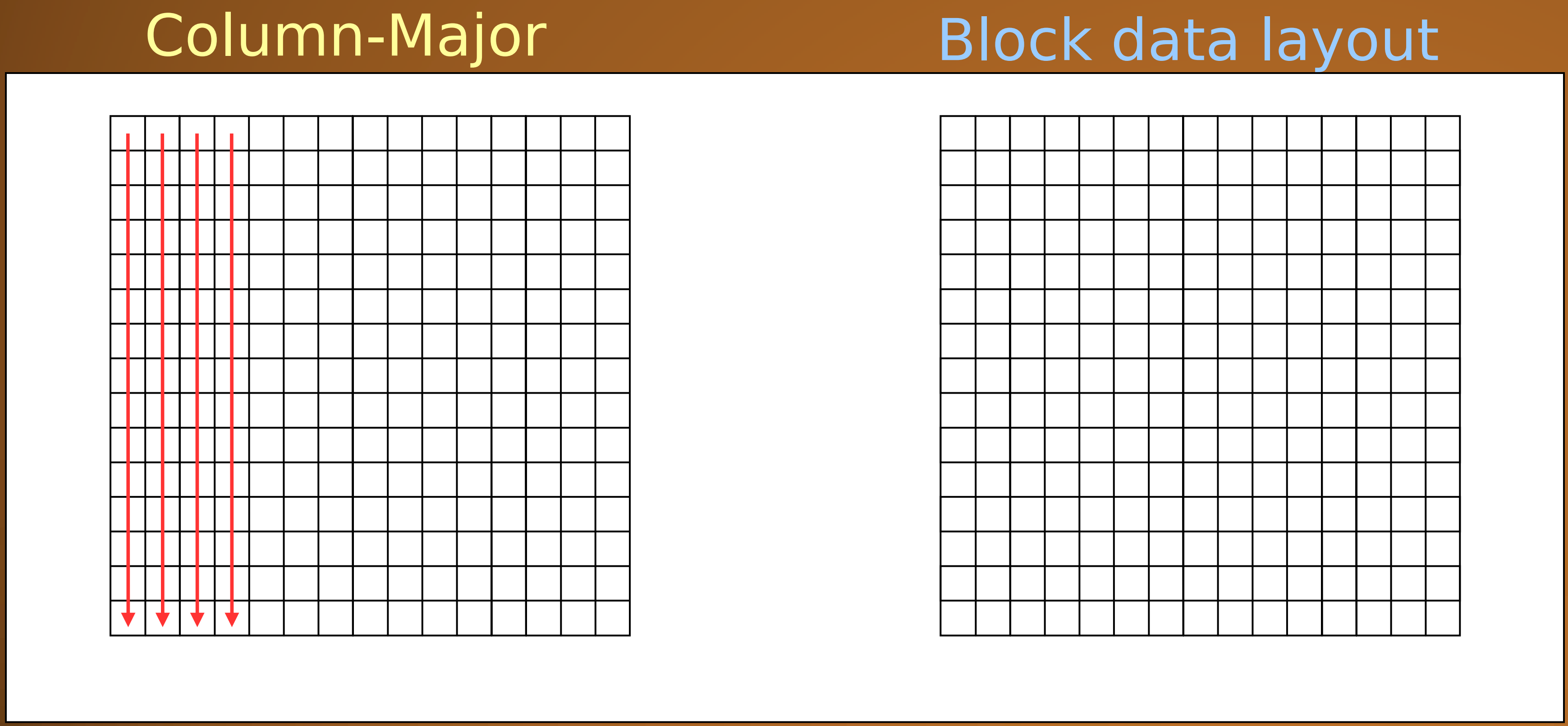
As long as dependencies are not violated, tasks can be scheduled in any order.

Parallelism in LAPACK: Cholesky factorization

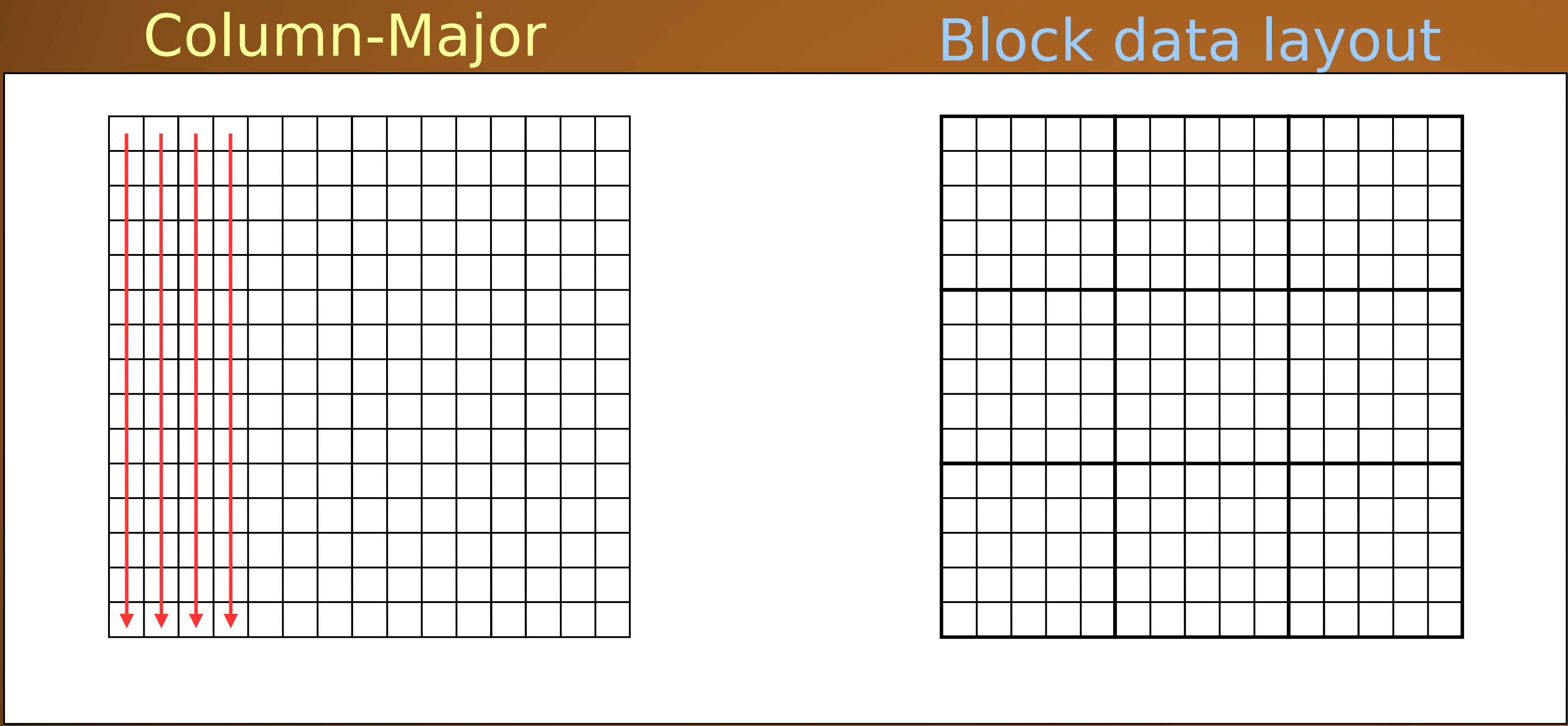
- higher flexibility
- some degree of adaptativity
- no idle time
- better scalability



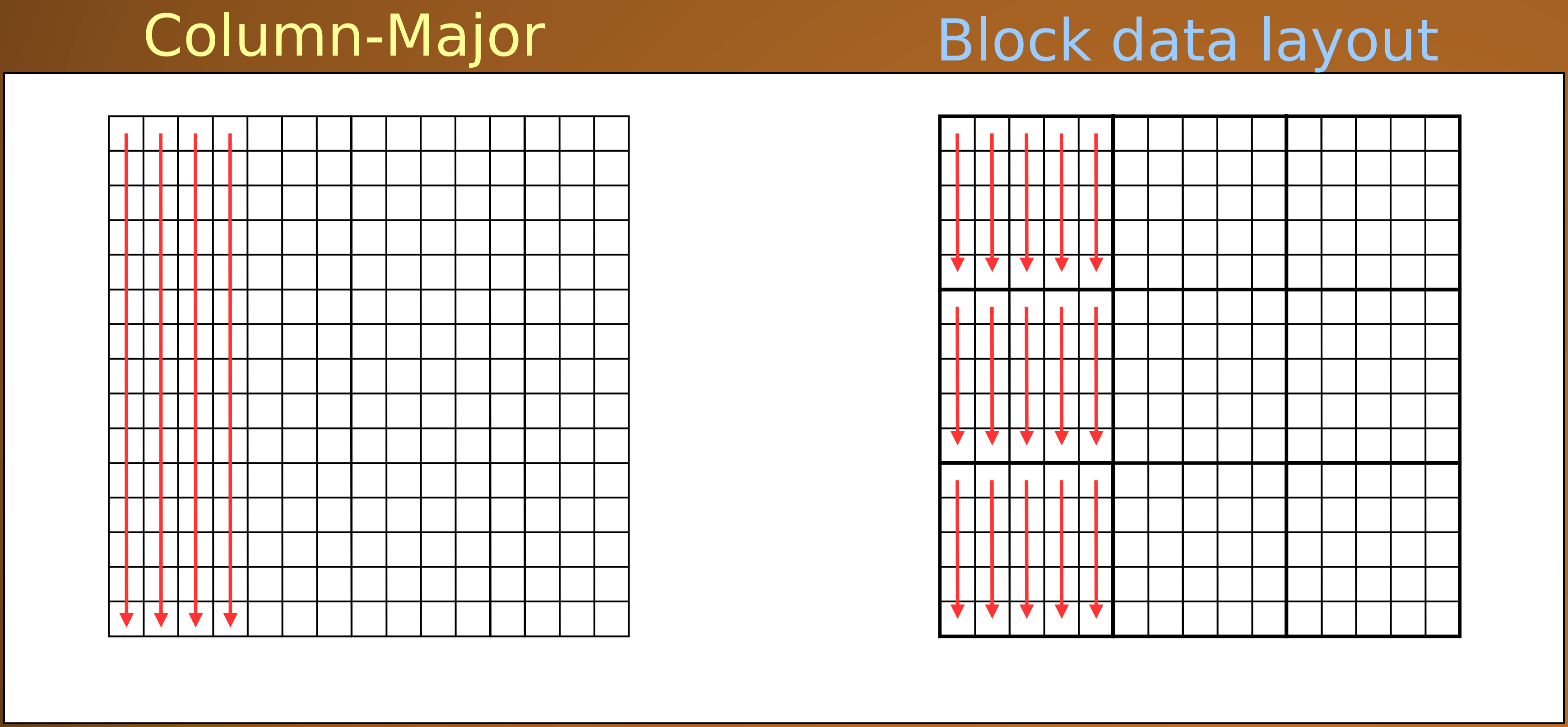
Parallelism in LAPACK: block data layout



Parallelism in LAPACK: block data layout

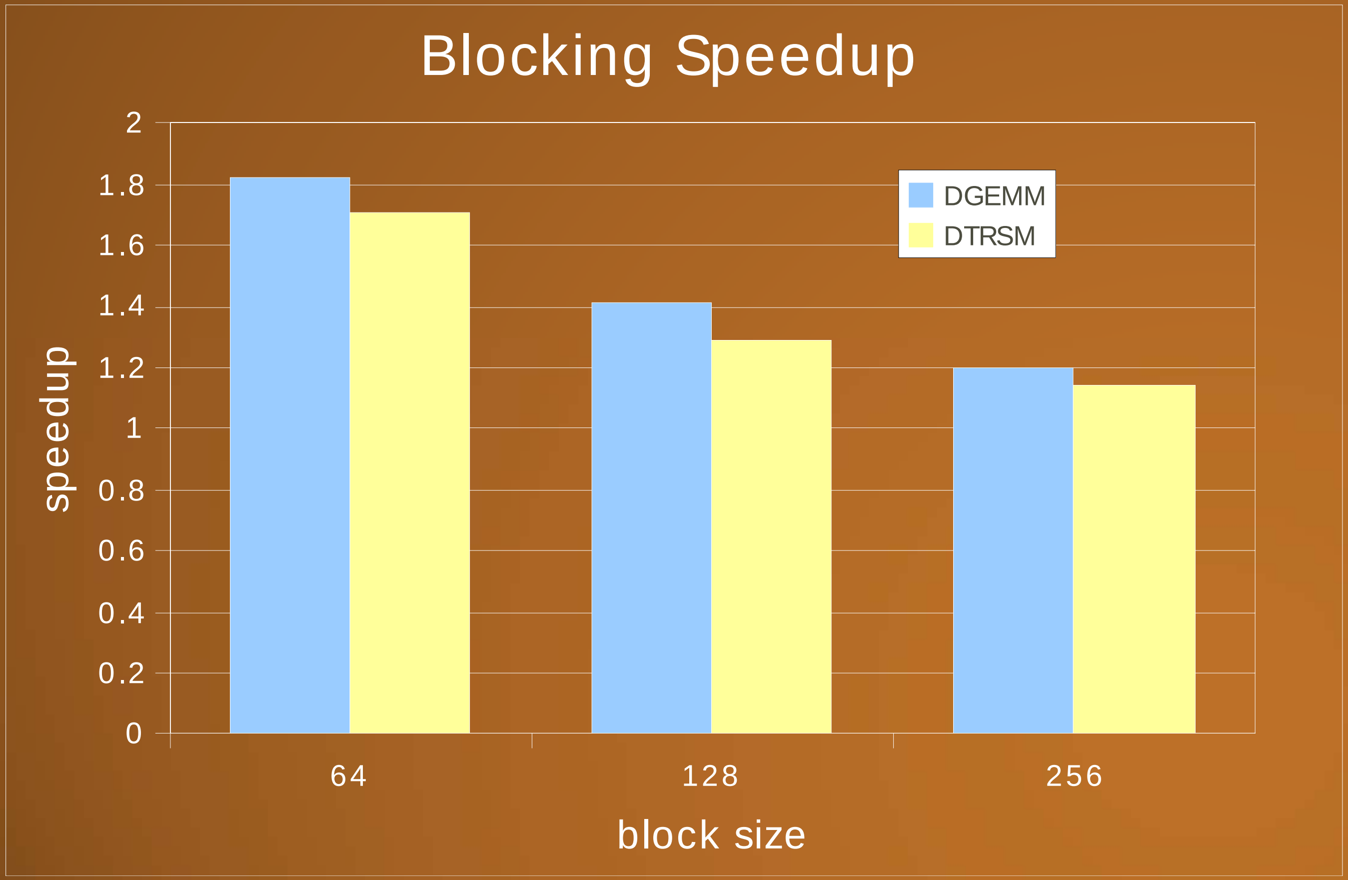


Parallelism in LAPACK: block data layout

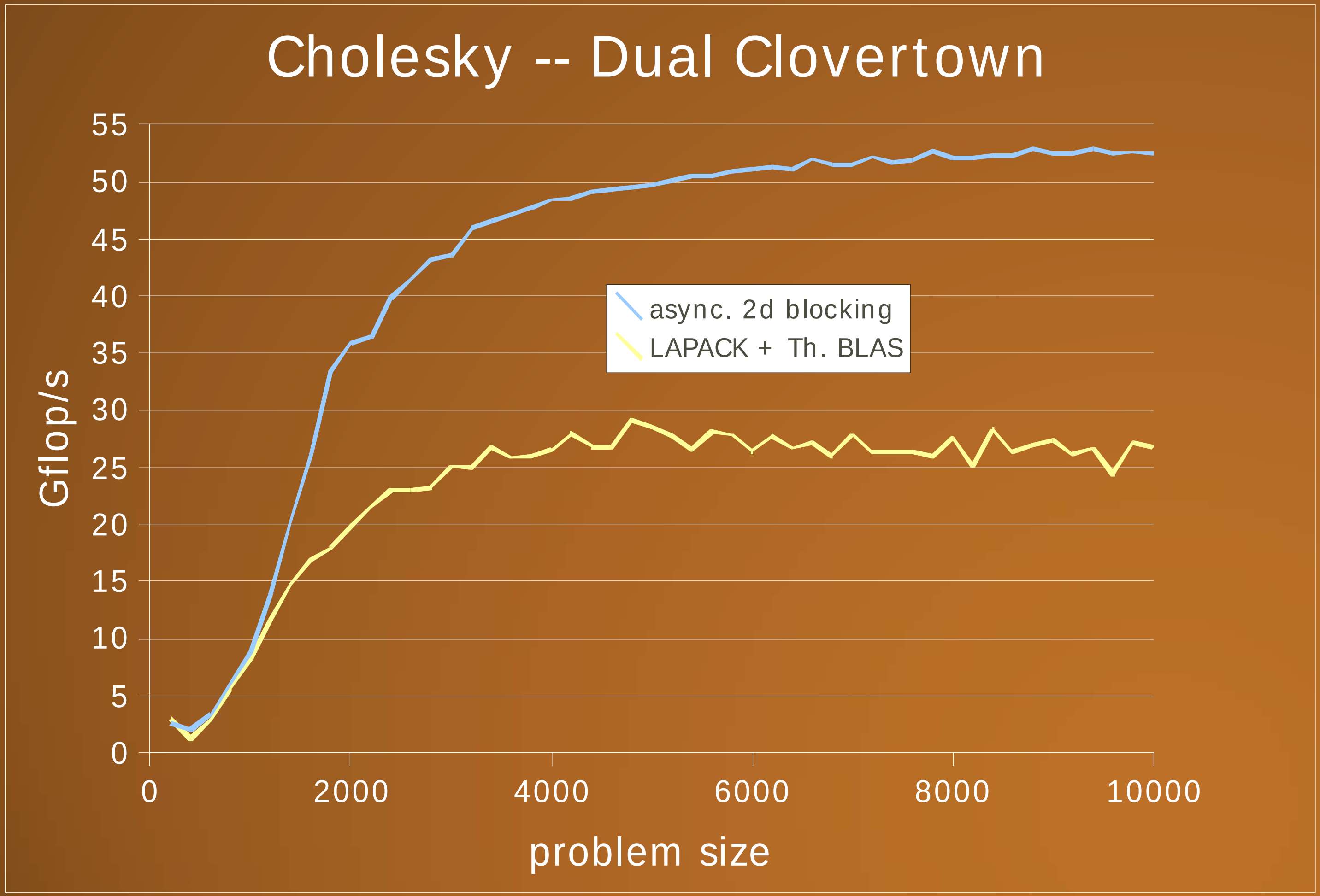


Parallelism in LAPACK: block data layout

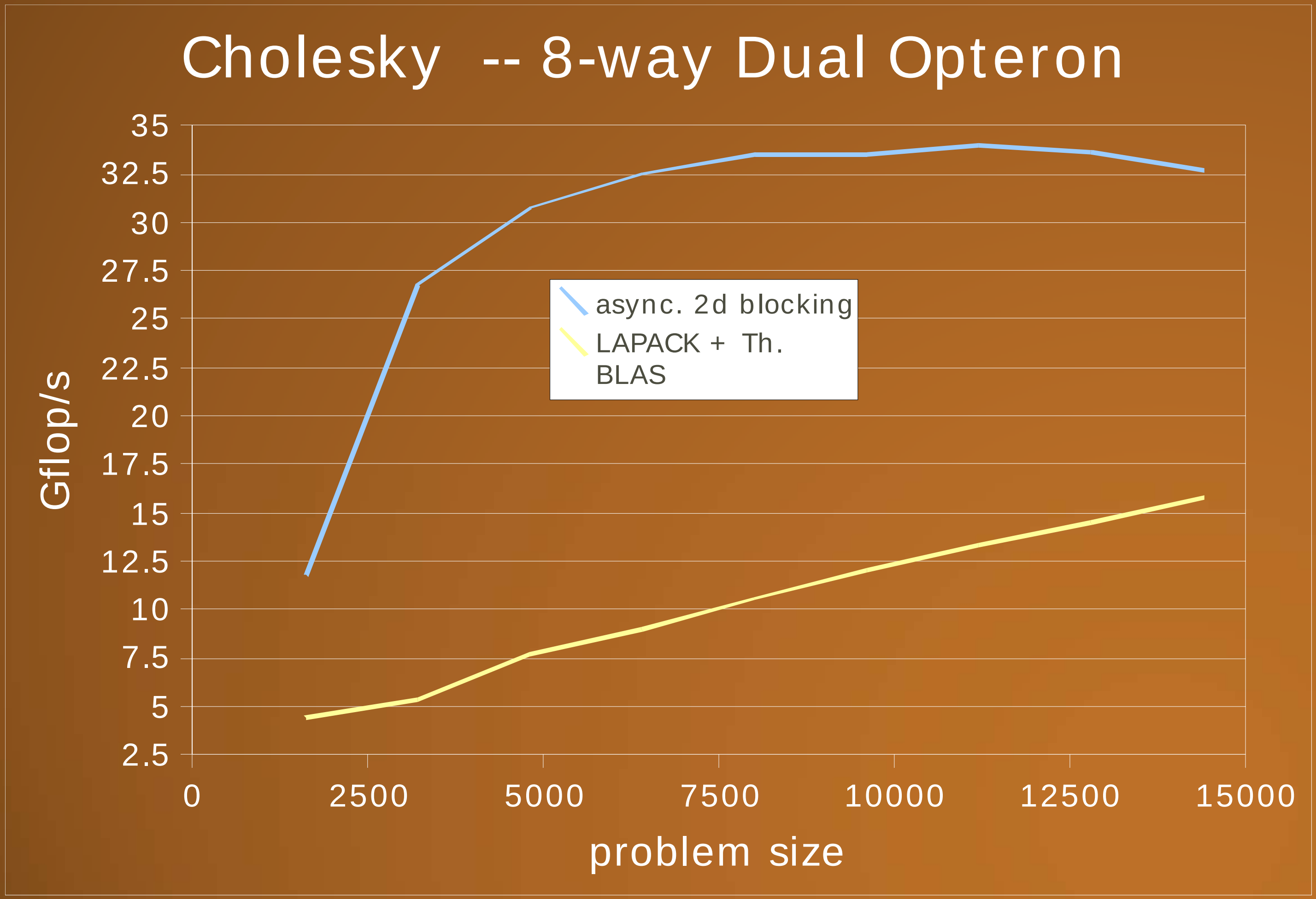
The use of block data layout storage can significantly improve performance



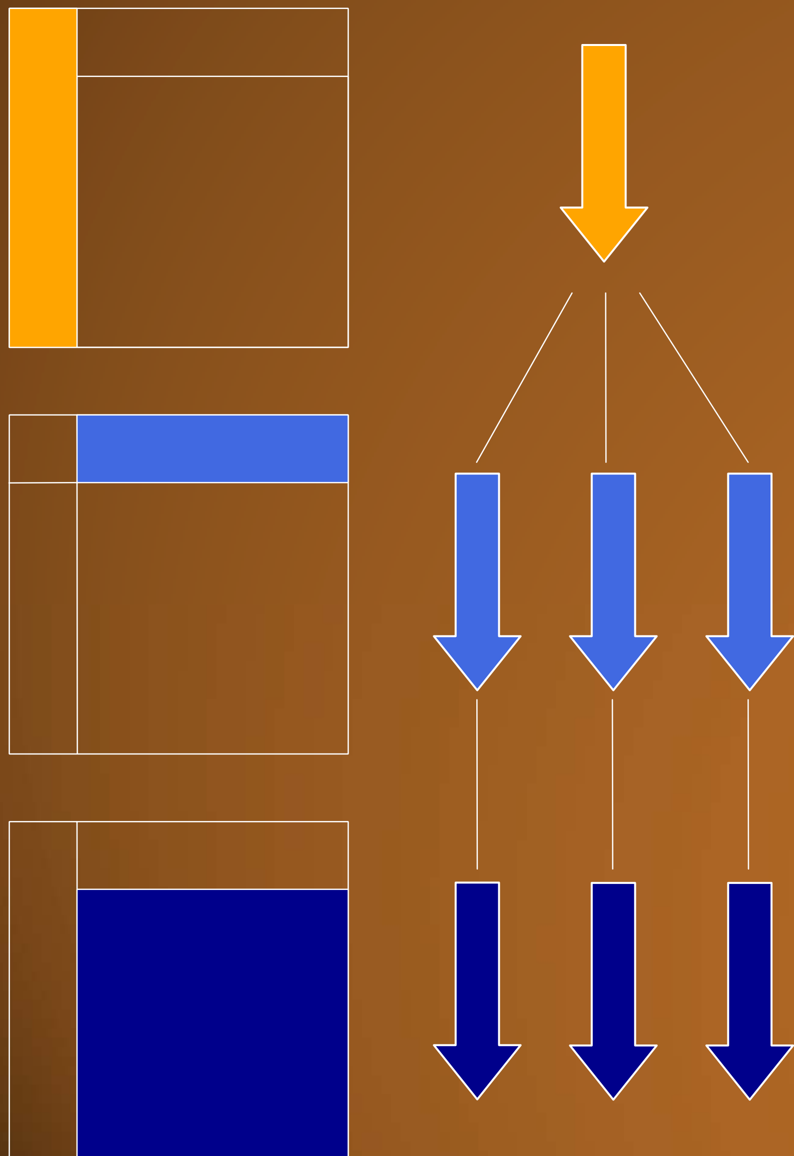
Cholesky: performance



Cholesky: performance



Parallelism in LAPACK: LU/QR factorizations

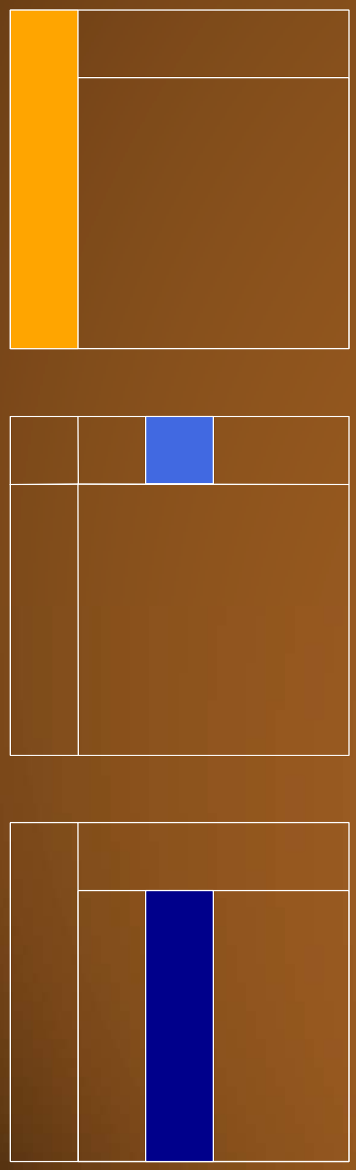


DGETF2: BLAS-2 non-blocked panel factorization

DTRSM: BLAS-3 updates U with transformation computed in DGETF2

DGEMM: BLAS-3 updates the trailing submatrix

Parallelism in LAPACK: LU/QR factorizations

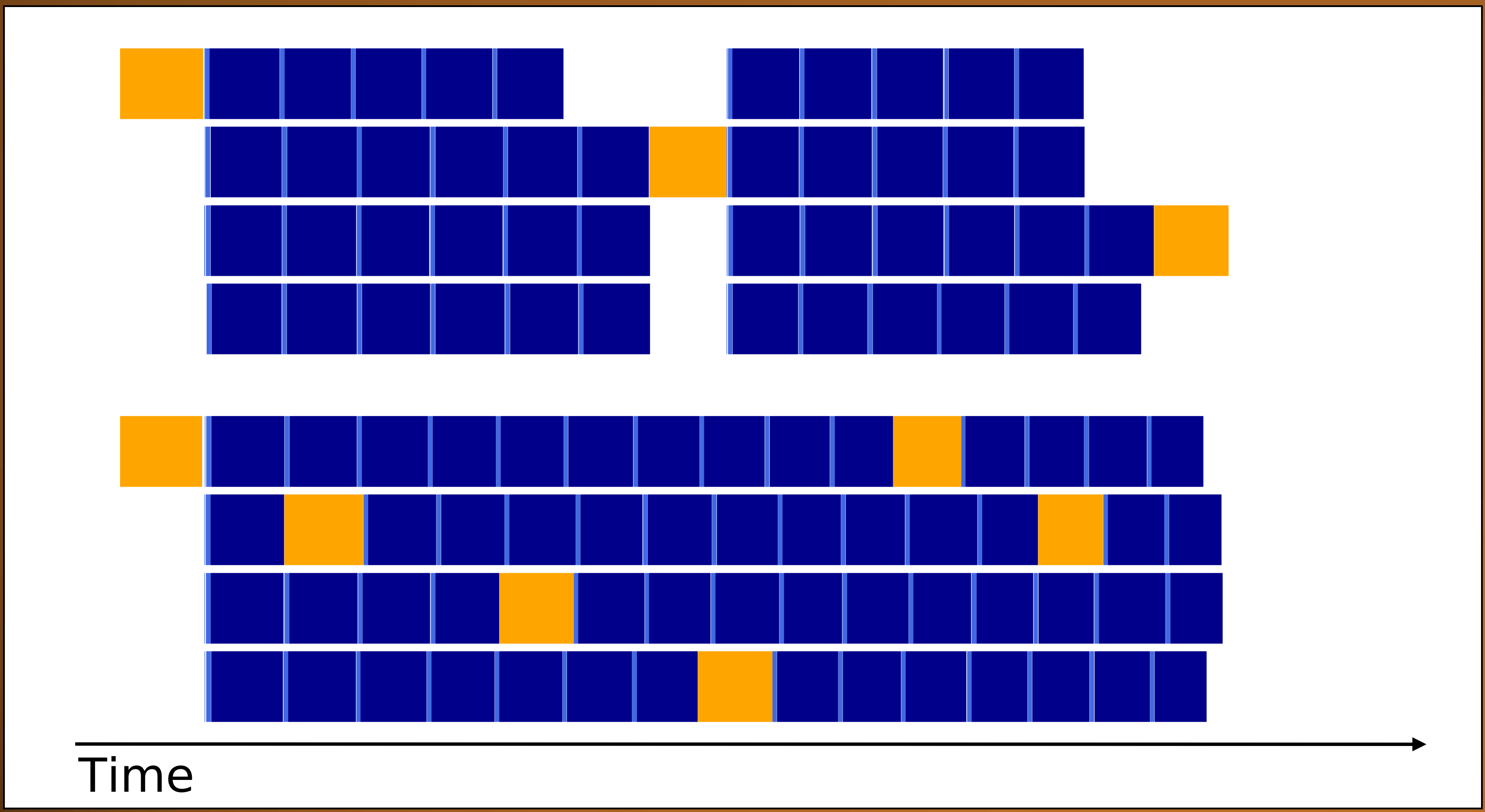


The LU and QR factorizations algorithms in LAPACK don't allow for 2D distribution and block storage format.

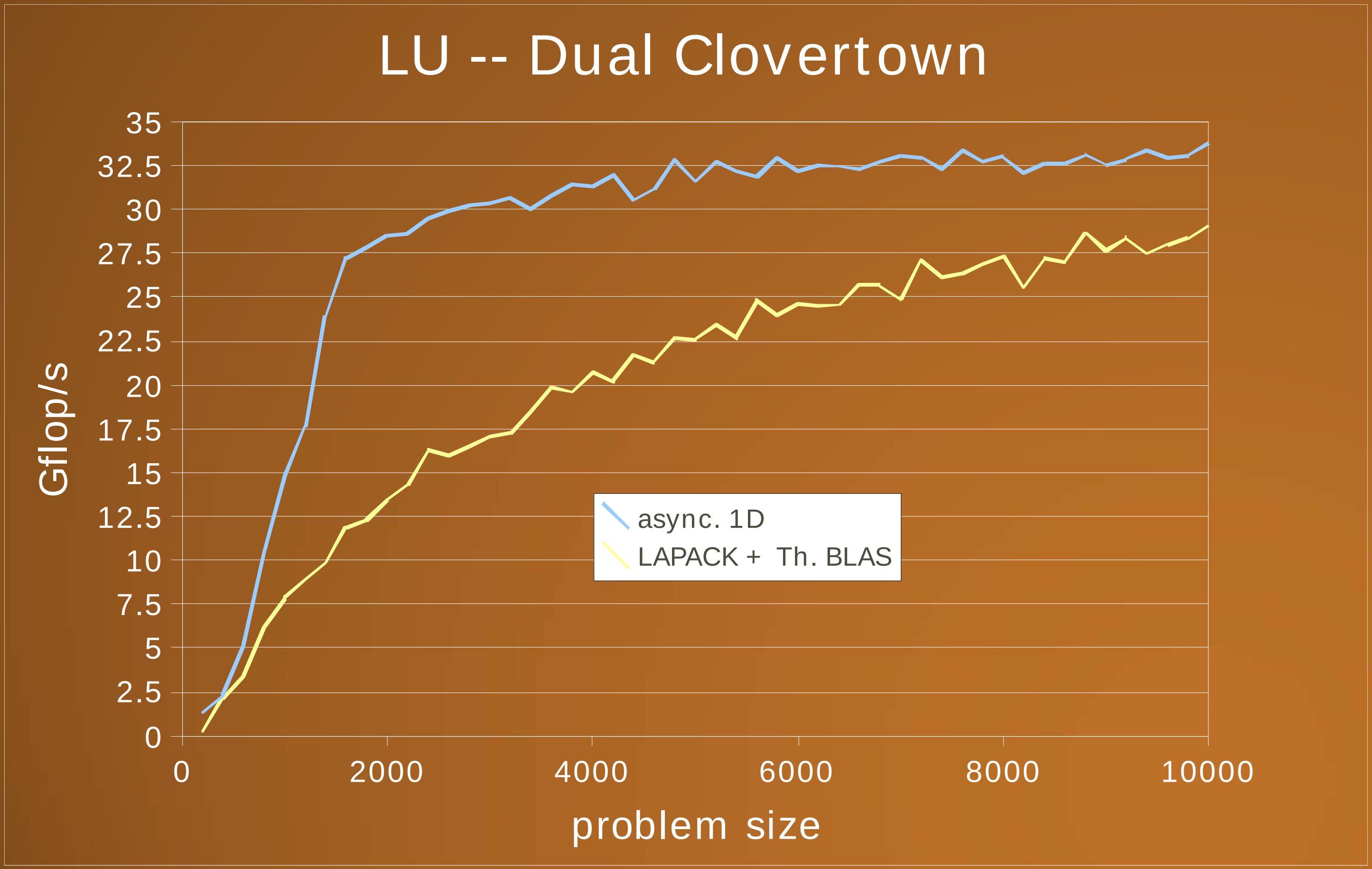
- LU: pivoting takes into account the whole panel and cannot be split in a block fashion.
- QR: the computation of Householder reflectors acts on the whole panel.
- The application of the transformation can only be sliced but not blocked.

Parallelism in LAPACK: LU/QR factorizations

LU



LU factorization: performance



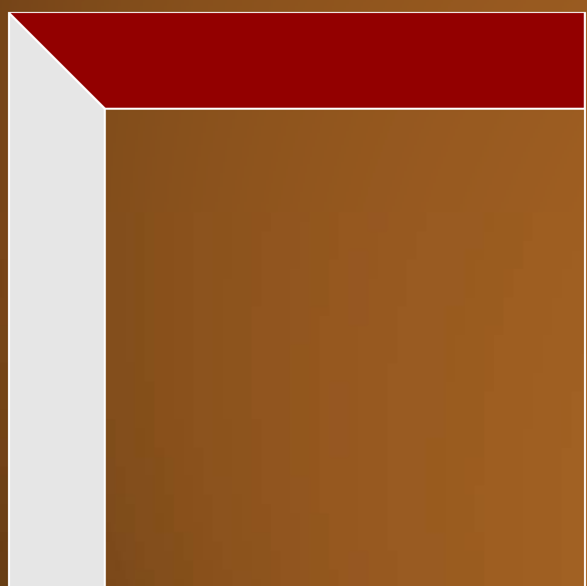
Multicore friendly, "*delightfully parallel*[*]", algorithms

Computer Science can't go any further on old algorithms.
We need some math...

* quote from Prof. S. Kale

The QR factorization in LAPACK

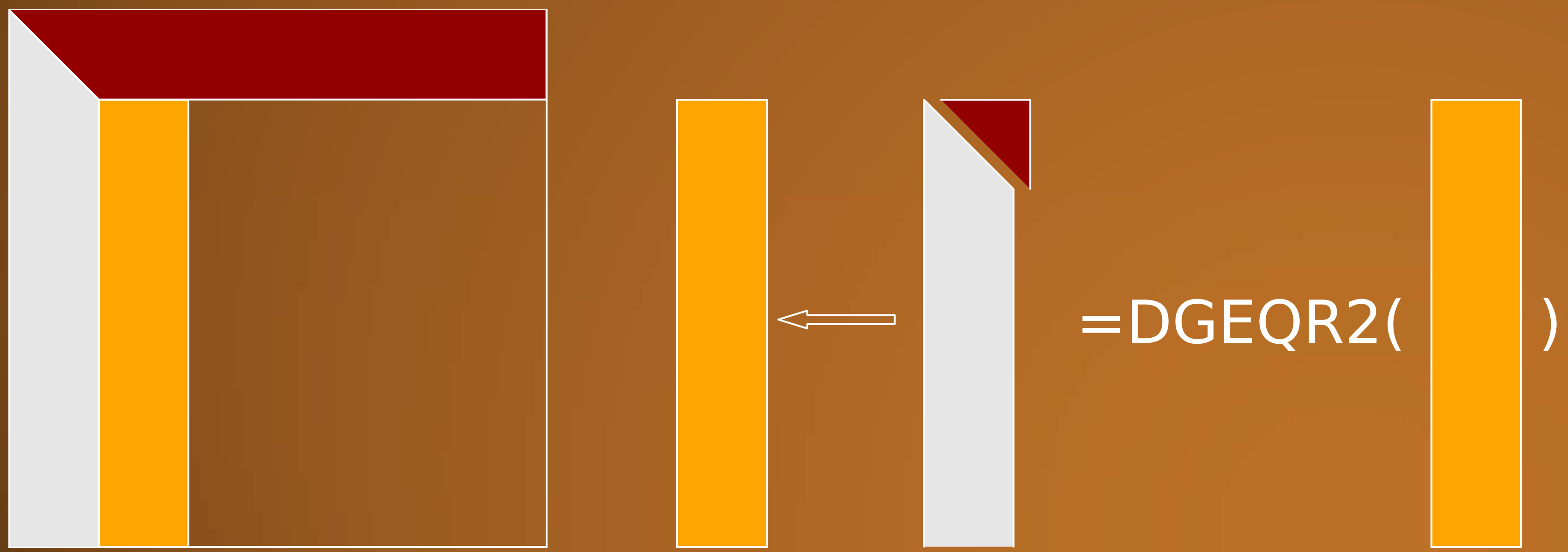
The QR transformation factorizes a matrix A into the factors Q and R where Q is unitary and R is upper triangular. It is based on Householder reflections.



Assume that ■ is the part of the matrix that has been already factorized and □ contains the Householder reflectors that determine the matrix Q.

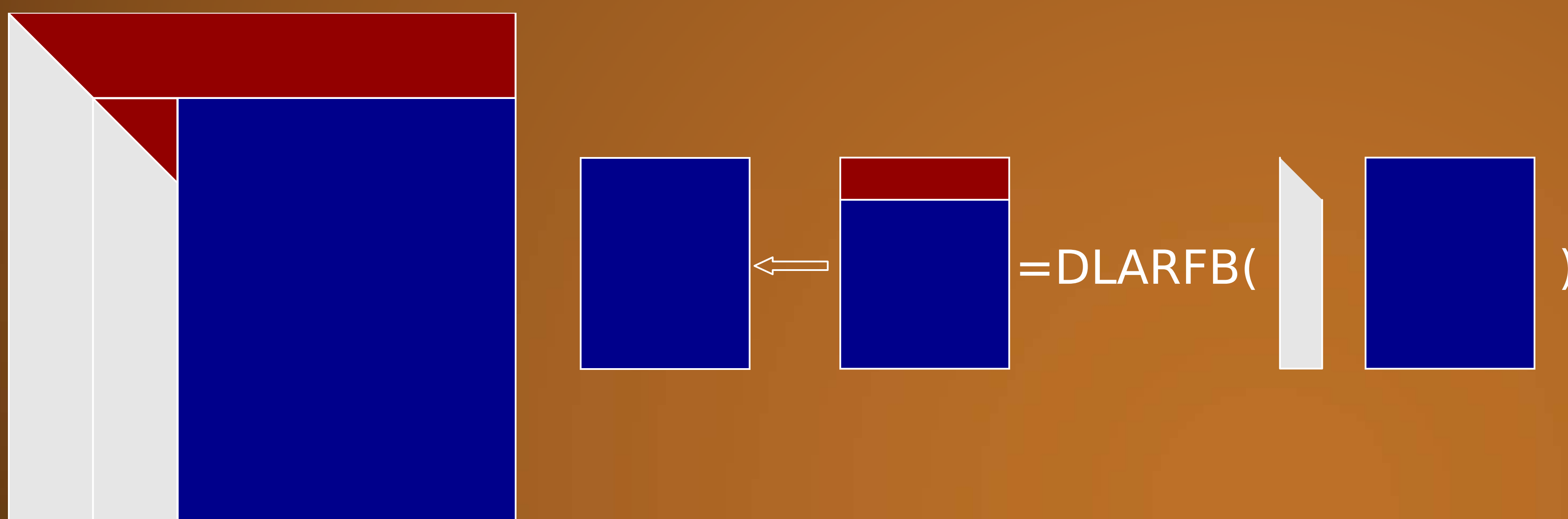
The QR factorization in LAPACK

The QR transformation factorizes a matrix A into the factors Q and R where Q is unitary and R is upper triangular. It is based on Householder reflections.



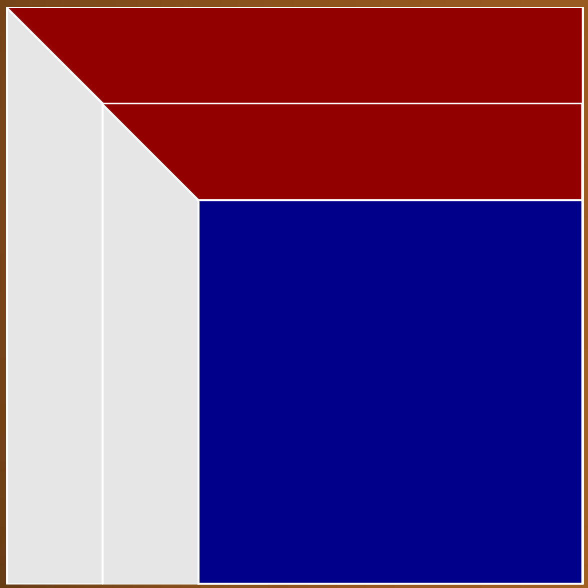
The QR factorization in LAPACK

The QR transformation factorizes a matrix A into the factors Q and R where Q is unitary and R is upper triangular. It is based on Householder reflections.



The QR factorization in LAPACK

The QR transformation factorizes a matrix A into the factors Q and R where Q is unitary and R is upper triangular. It is based on Householder reflections.

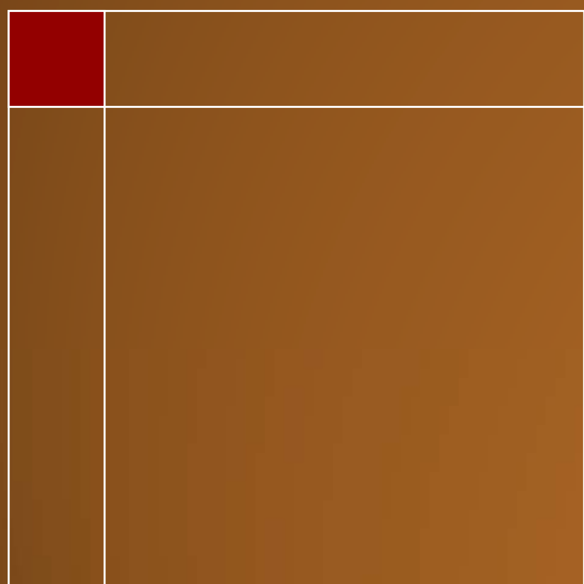


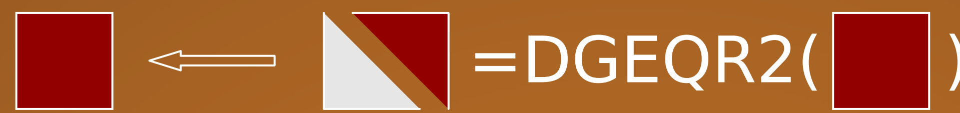
How does it compare to LU?

- It is stable because it uses Householder transformations that are orthogonal
- It is more expensive than LU because its operation count is $4/3\,n^3$ versus $2/3\,n^3$

Multicore friendly algorithms: QR

A different algorithm can be used where operations can be broken down into tiles.

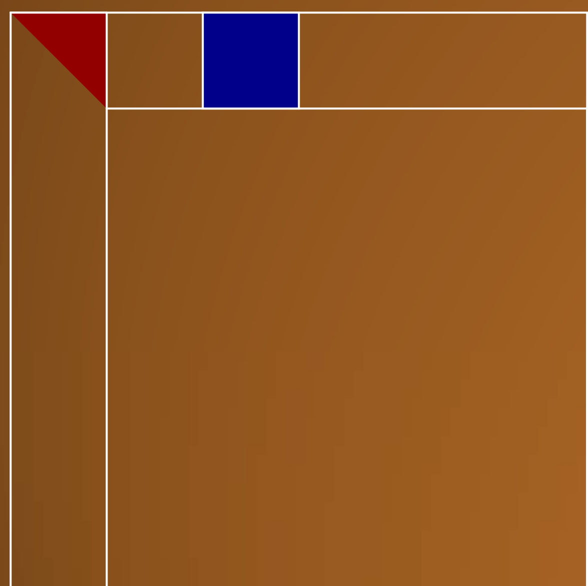




The QR factorization of the upper left tile is performed. This operation returns a small R factor: 
and the corresponding Householder reflectors: 

Multicore friendly algorithms: QR

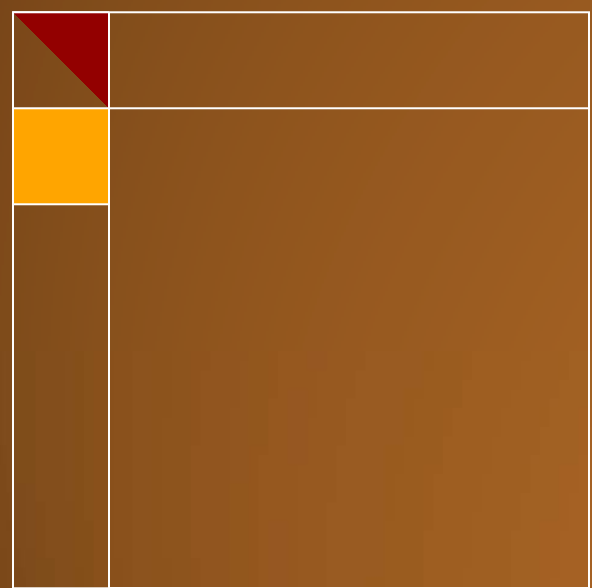
A different algorithm can be used where operations can be broken down into tiles.

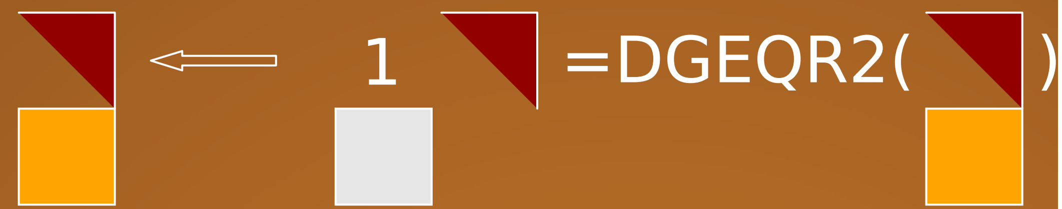


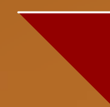
All the tiles in the first block-row are updated by applying the transformation  computed at the previous step.

Multicore friendly algorithms: QR

A different algorithm can be used where operations can be broken down into tiles.

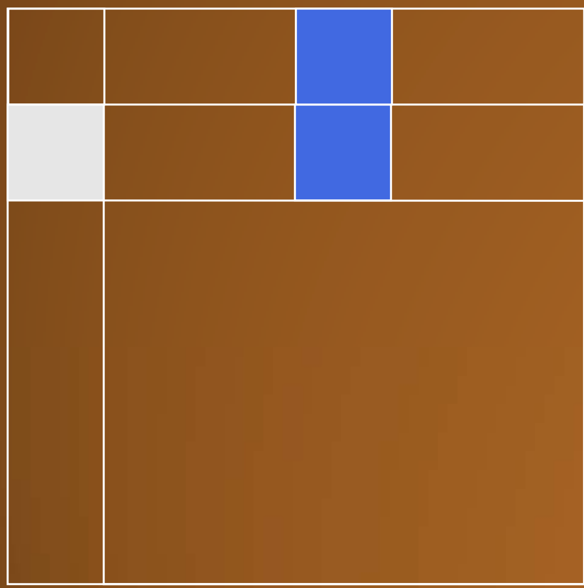




The R factor  computed at the first step is coupled with one tile in the block-column and a QR factorization is computed. Flops can be saved due to the shape of the matrix resulting from the coupling.

Multicore friendly algorithms: QR

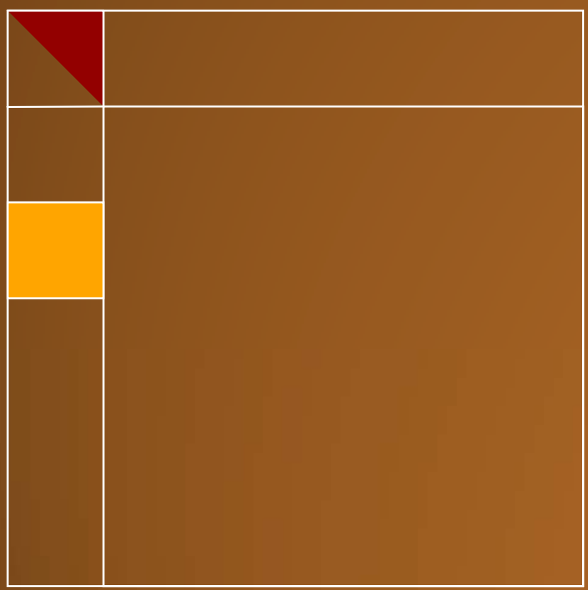
A different algorithm can be used where operations can be broken down into tiles.

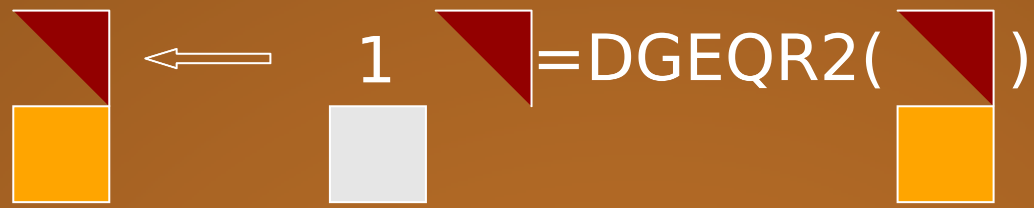


Each couple of tiles along the corresponding block rows is updated by applying the transformations computed in the previous step. Flops can be saved considering the shape of the Householder vectors.

Multicore friendly algorithms: QR

A different algorithm can be used where operations can be broken down into tiles.

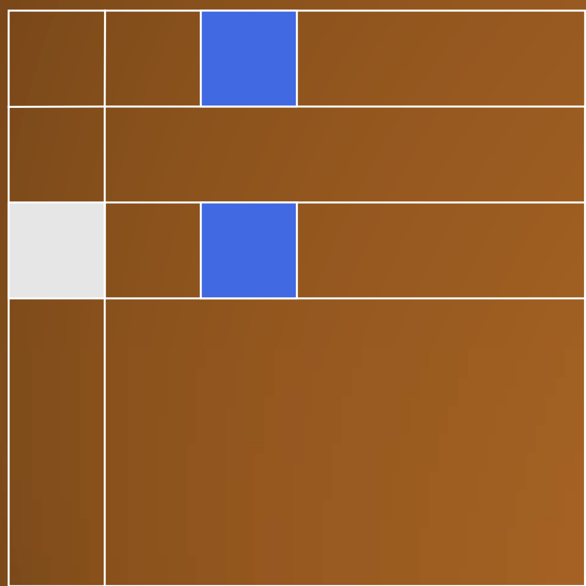


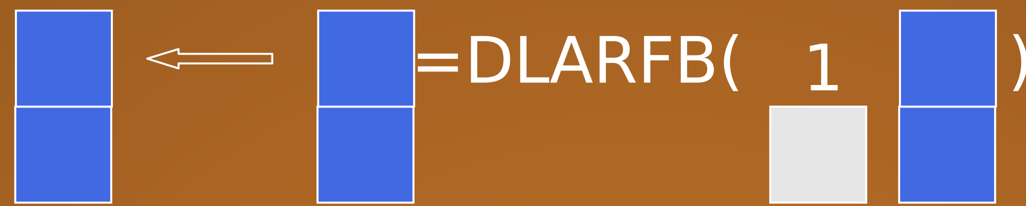


The last two steps are repeated for all the tiles in the first block-column.

Multicore friendly algorithms: QR

A different algorithm can be used where operations can be broken down into tiles.

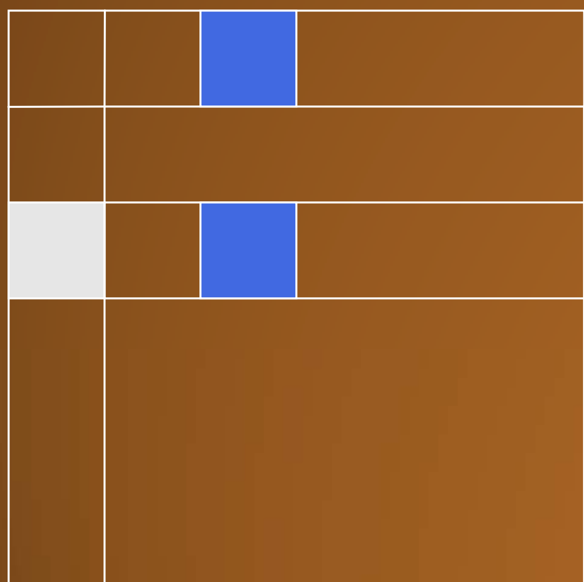


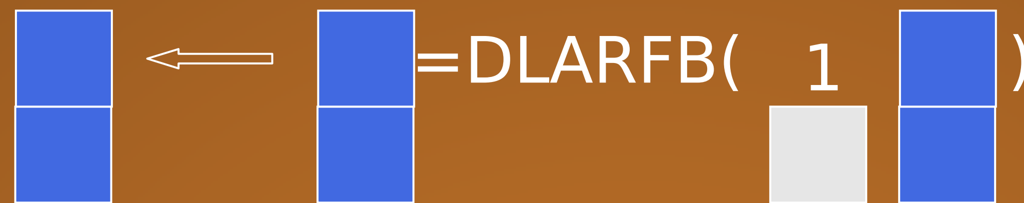


The last two steps are repeated for all the tiles in the first block-column.

Multicore friendly algorithms: QR

A different algorithm can be used where operations can be broken down into tiles.



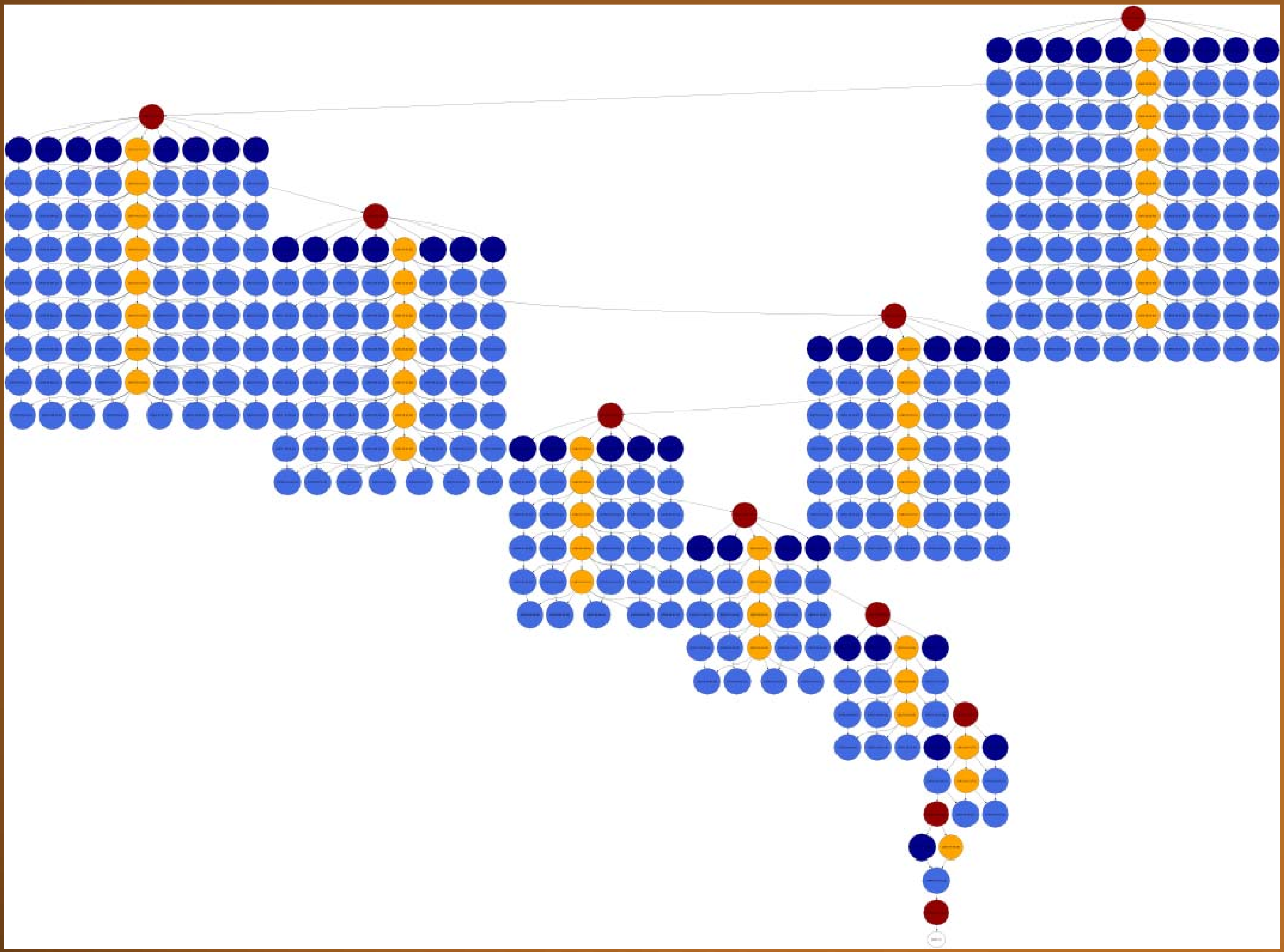


The last two steps are repeated for all the tiles in the first block-column.

25% more Flops than the LAPACK version!!!*

*we are working on a way to remove these extra flops.

Multicore friendly algorithms: QR

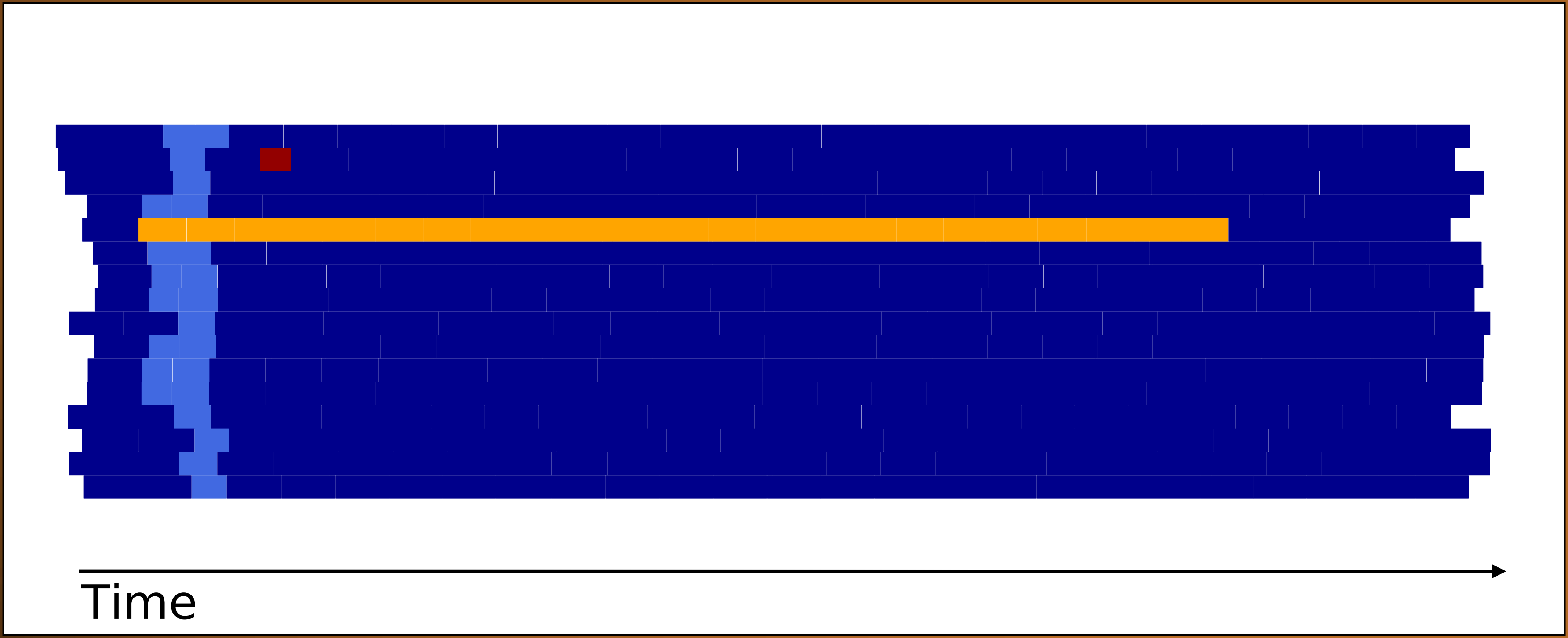


Multicore friendly algorithms: QR

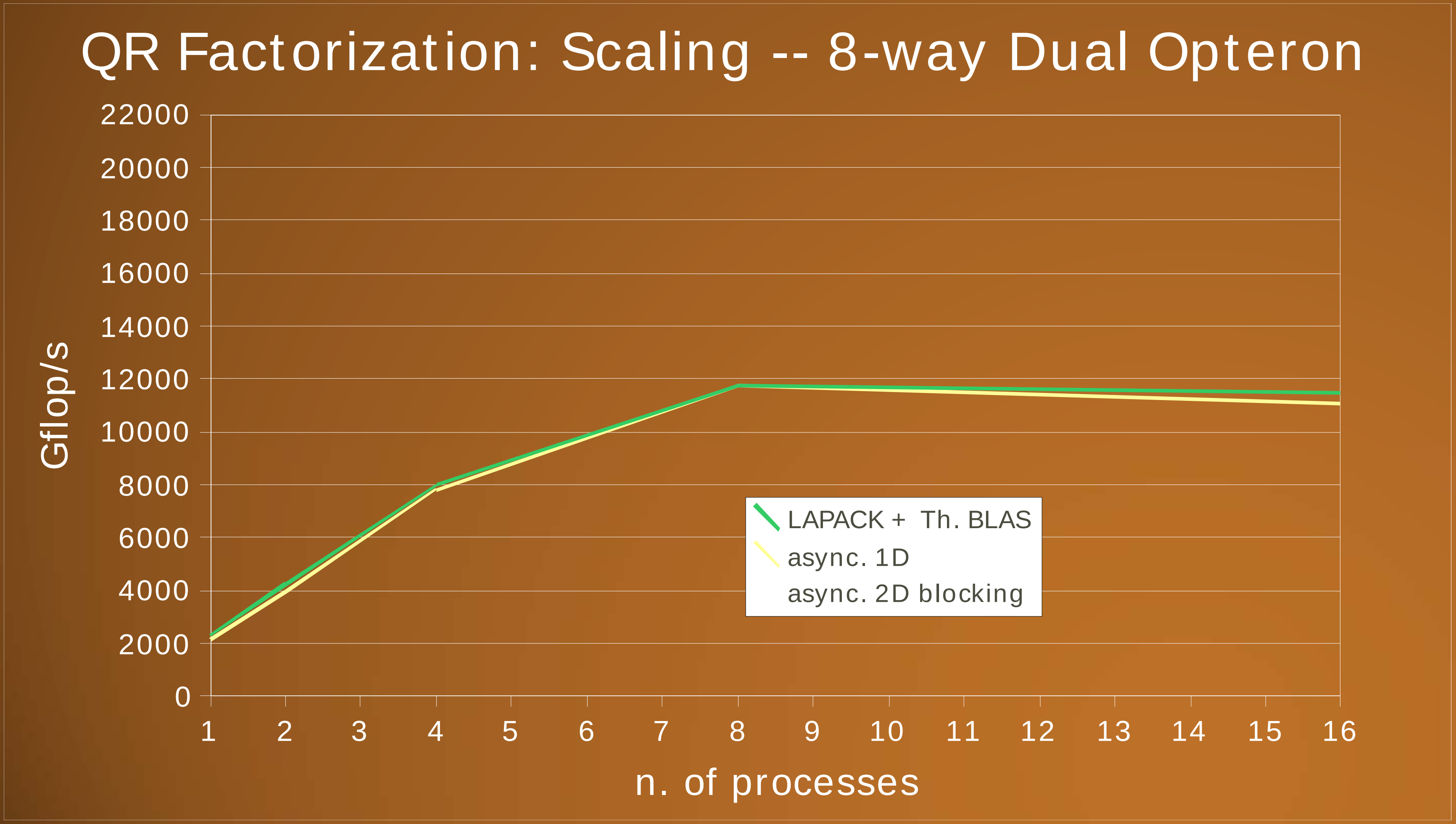
- Very fine granularity
- Few dependencies, i.e., high flexibility for the scheduling of tasks
- Block data layout is possible

Multicore friendly algorithms: QR

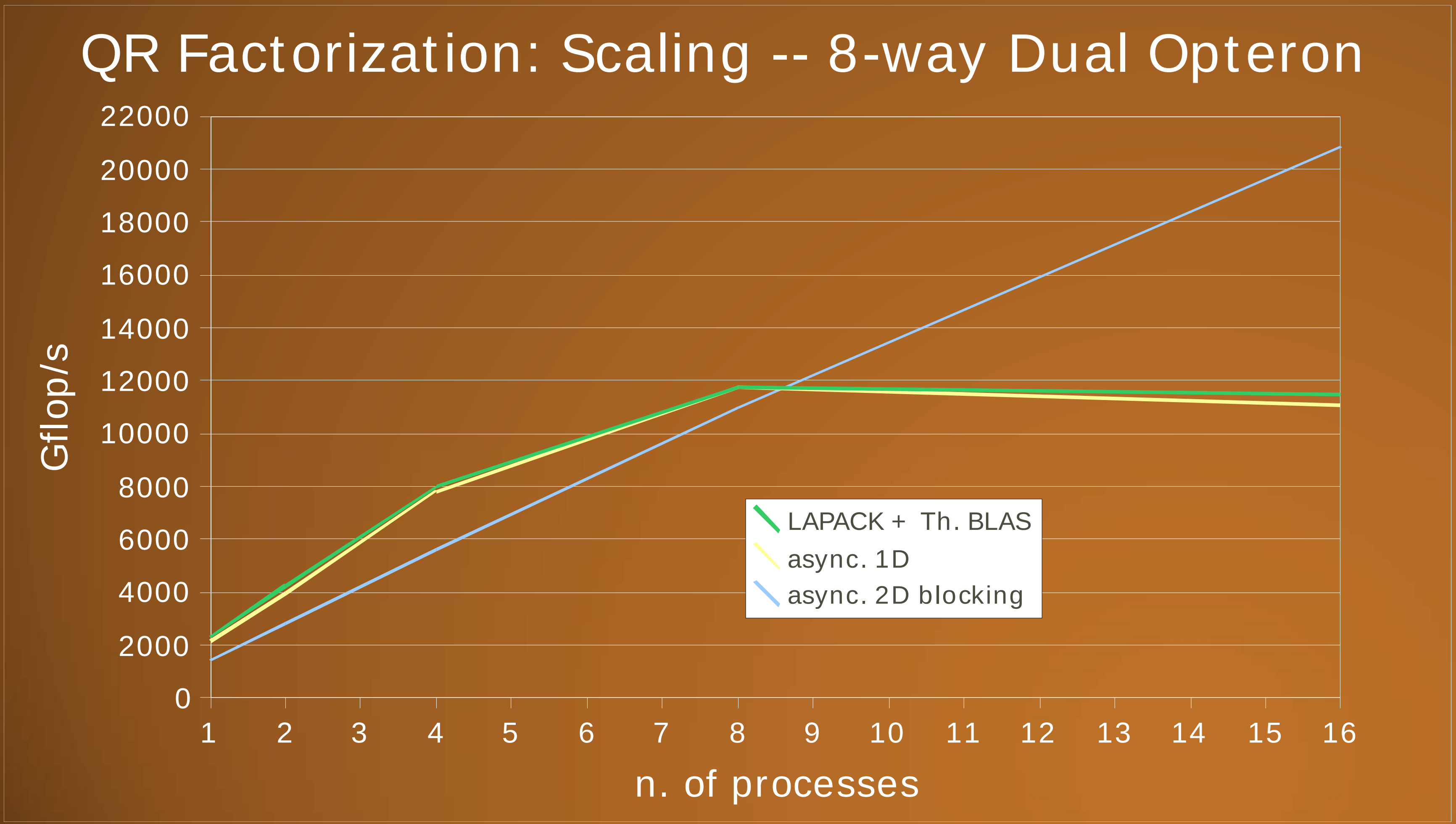
Execution flow on a 8-way dual core Opteron.



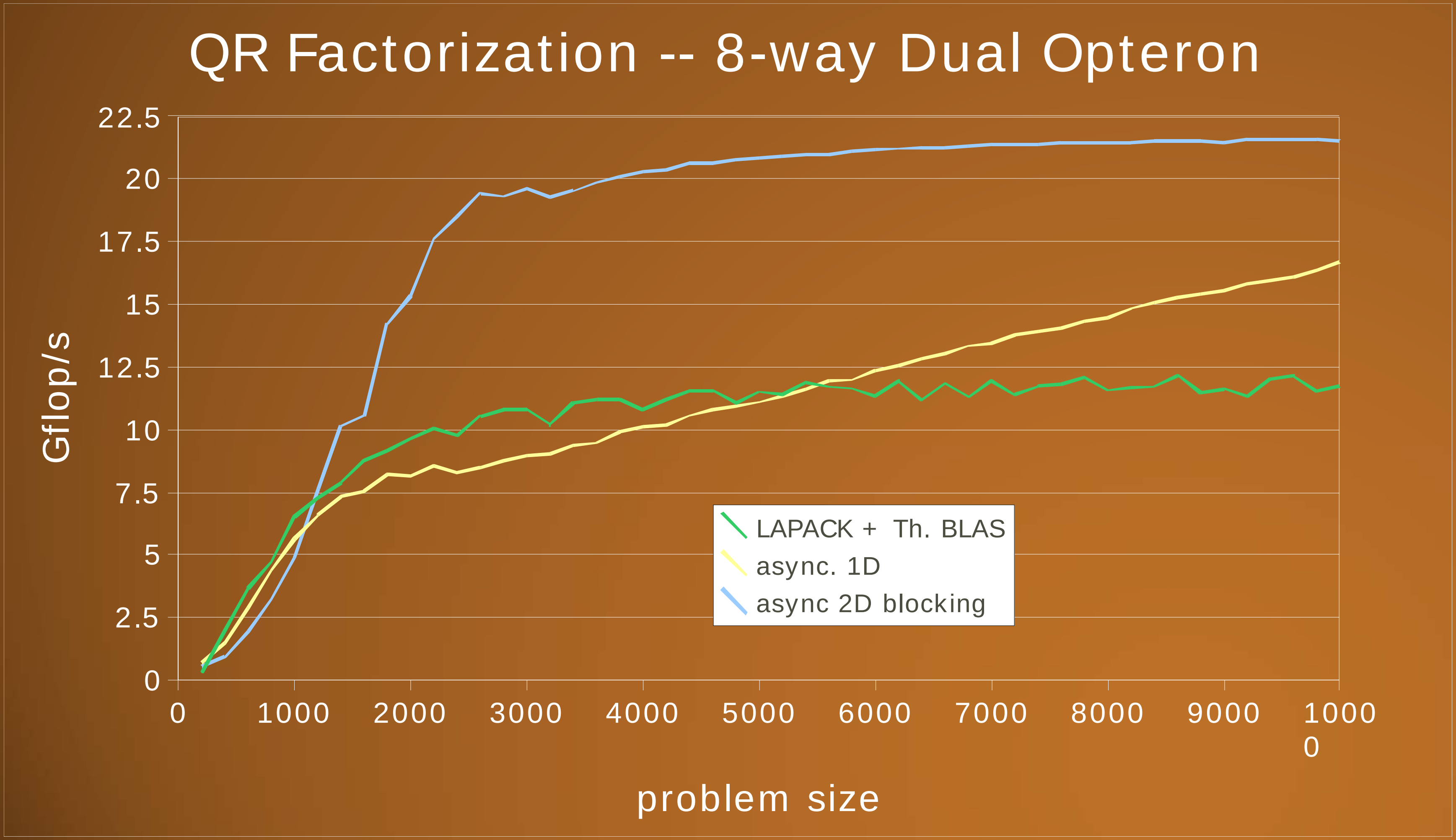
Multicore friendly algorithms: QR



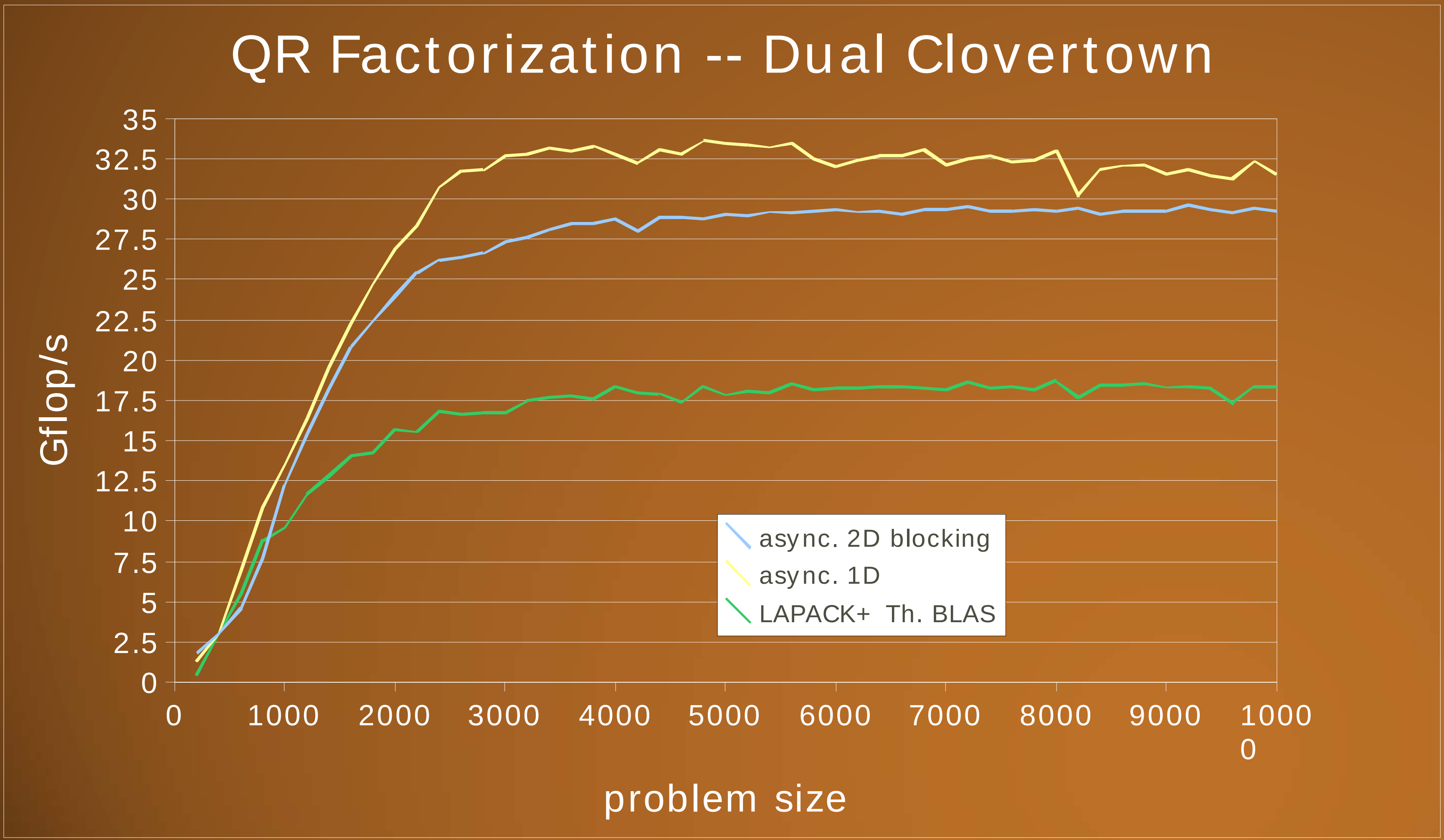
Multicore friendly algorithms: QR



Multicore friendly algorithms: QR



Multicore friendly algorithms: QR



Current work and future plans

Current work and future plans

- Implement LU factorization on multicores
- Is it possible to apply the same approach to two-sided transformations (Hessenberg, Bi-Diag, Tri-Diag)?
- Explore techniques to avoid extra flops
- Implement the new algorithms on distributed memory architectures (J. Langou and J. Demmel)
- Implement the new algorithms on the Cell processor
- Explore automatic exploitation of parallelism through graph driven programming environments

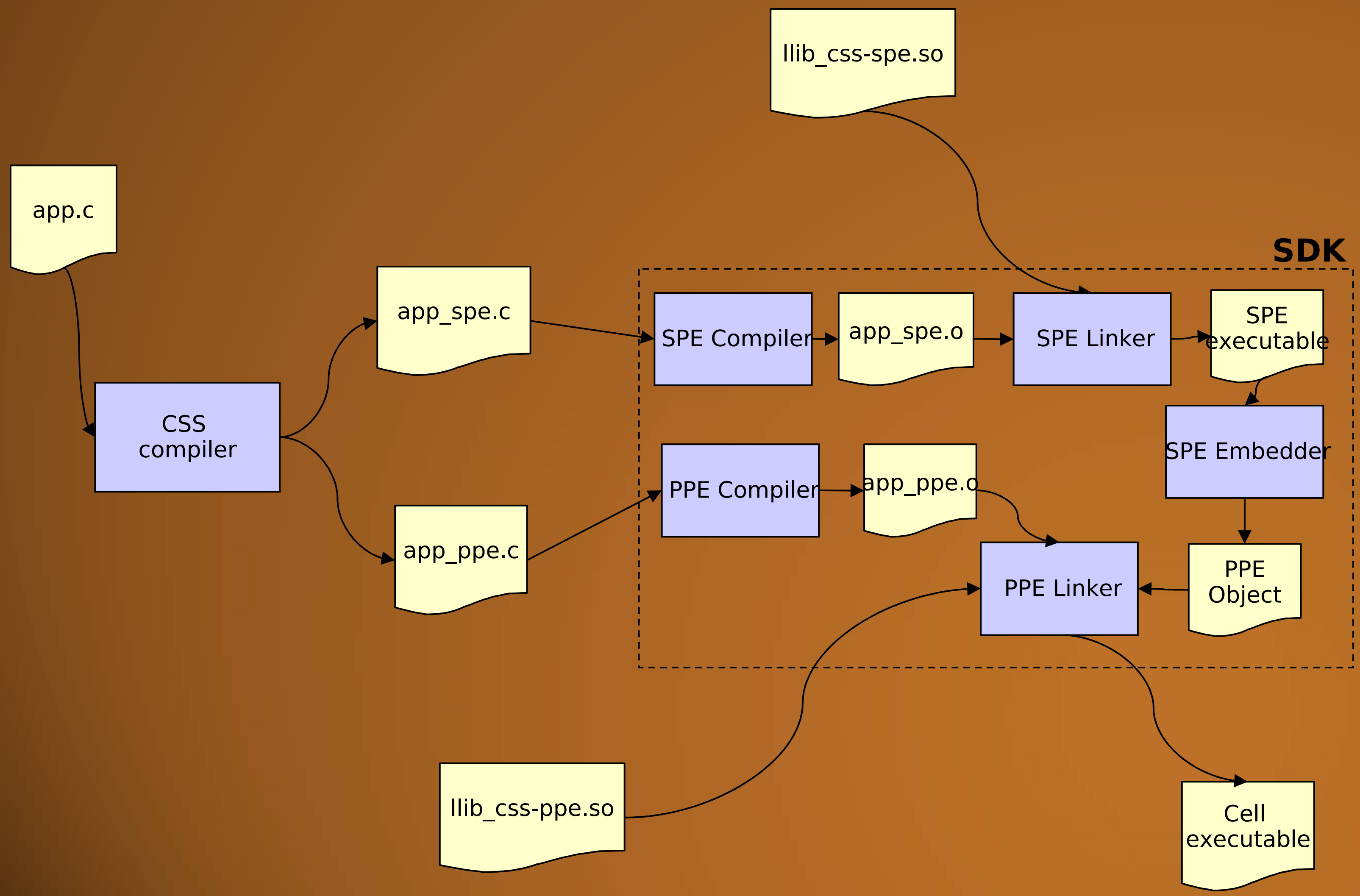
CellSuperScalar and SMPSuperScalar

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

http://www.bsc.es/cellsuperscalar

- uses source-to-source translation to determine dependencies among tasks
- scheduling of tasks is performed automatically by means of the features provided by a library
- it is easily possible to explore different scheduling policies
- all of this is obtained by decorating the code with pragmas and, thus, is transparent to other compilers

Compilation Environment



CellSuperScalar and SMPSuperScalar

```
for (i = 0; i < DIM; i++) {
  for (j= 0; j< i-1; j++){
     for (k = 0; k < j-1; k++) {
         sgemm_tile( A[i][k], A[j][k], A[i][j] );
     }
     strsm_tile( A[j][j], A[i][j] );
  }
  for (j = 0; j < i-1; j++) {
     ssyrk_tile( A[i][j], A[i][i] );
  }
  spotrf_tile( A[i][i] );
}


void sgemm_tile(float *A, float *B, float *C)


void strsm_tile(float *T, float *B)


void ssyrk_tile(float *A, float *C)
```

CellSuperScalar and SMPSuperScalar

```
for (i = 0; i < DIM; i++) {
  for (j= 0; j< i-1; j++){
     for (k = 0; k < j-1; k++) {
         sgemm_tile( A[i][k], A[j][k], A[i][j] );
     }
     strsm_tile( A[j][j], A[i][j] );
  }
  for (j = 0; j < i-1; j++) {
     ssyrk_tile( A[i][j], A[i][i] );
  }
  spotrf_tile( A[i][i] );
}

#pragma css task input(A[64][64], B[64][64]) inout(C[64][64])
void sgemm_tile(float *A, float *B, float *C)

#pragma css task input (T[64][64]) inout(B[64][64])
void strsm_tile(float *T, float *B)

#pragma css task input(A[64][64], B[64][64]) inout(C[64][64])
void ssyrk_tile(float *A, float *C)
```

Empirical Tuning of MADNESS

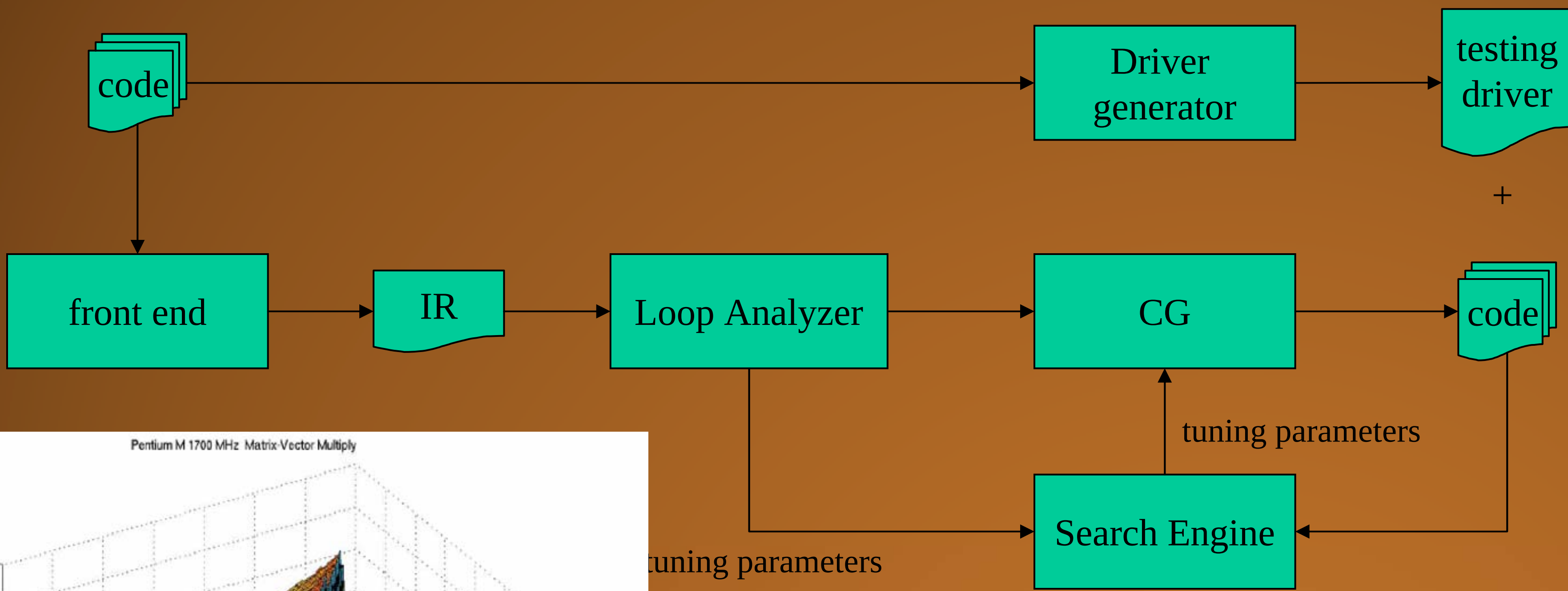
Haihang You and Keith Seymour

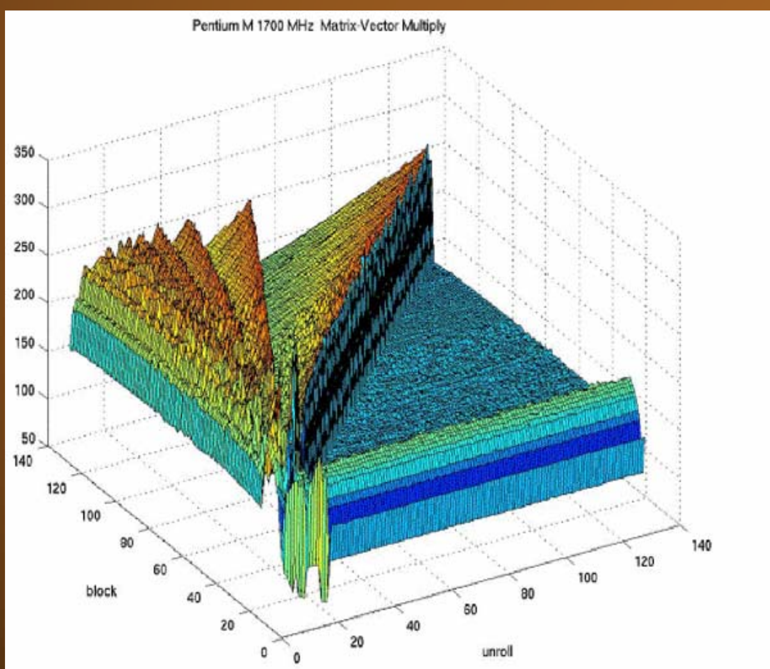
ICL UT
Innovative Computing Laboratory
The University of Tennessee
Computer Science Department

What's MADNESS?

- SciDAC code by Robert Harrison @ ORNL
- Framework for adaptive multiresolution methods in multiwavelet bases
- Collaborative optimization effort as part of UTK's participation in PERI, the Performance Engineering Research Institute

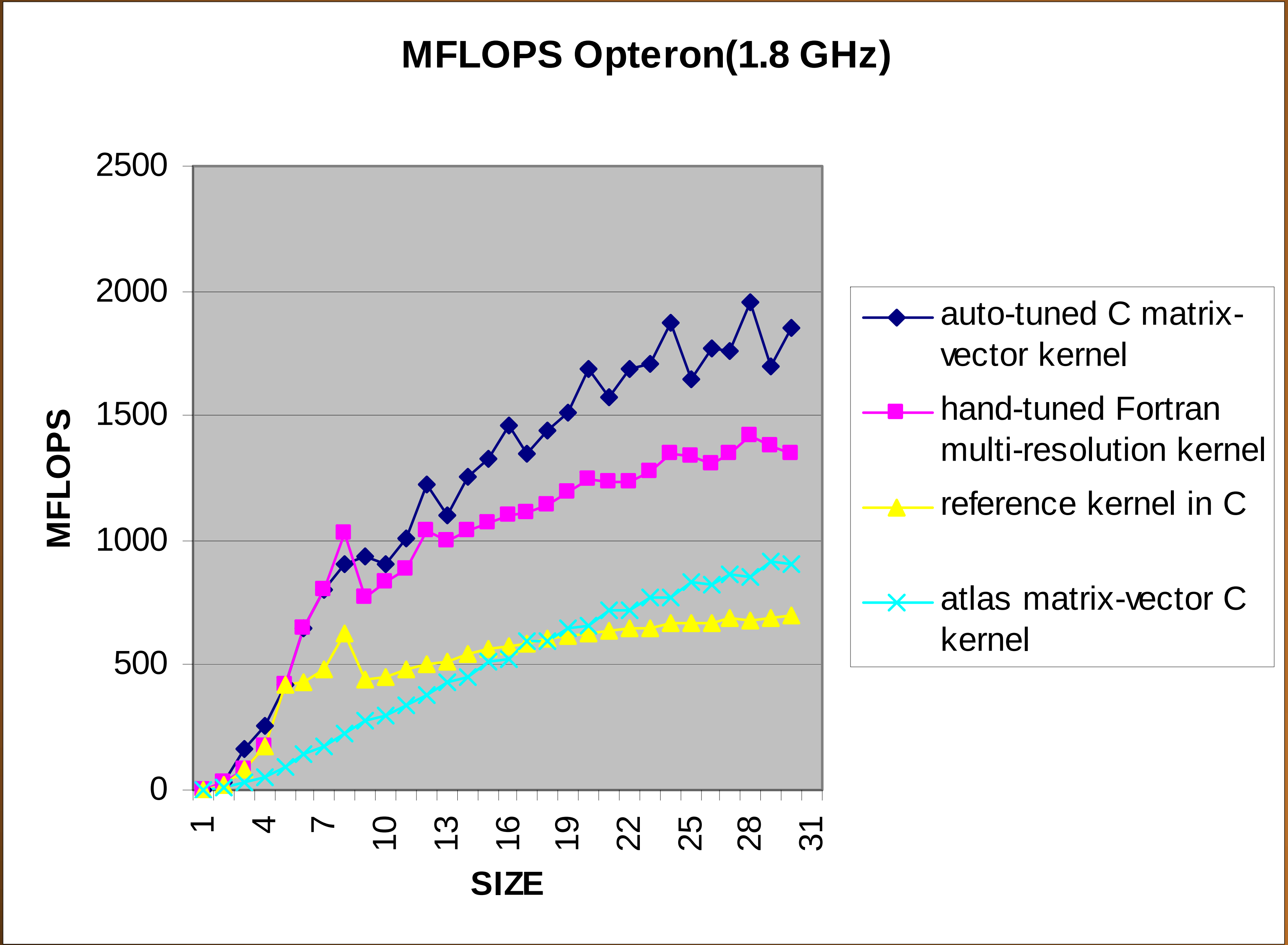
GCO Framework

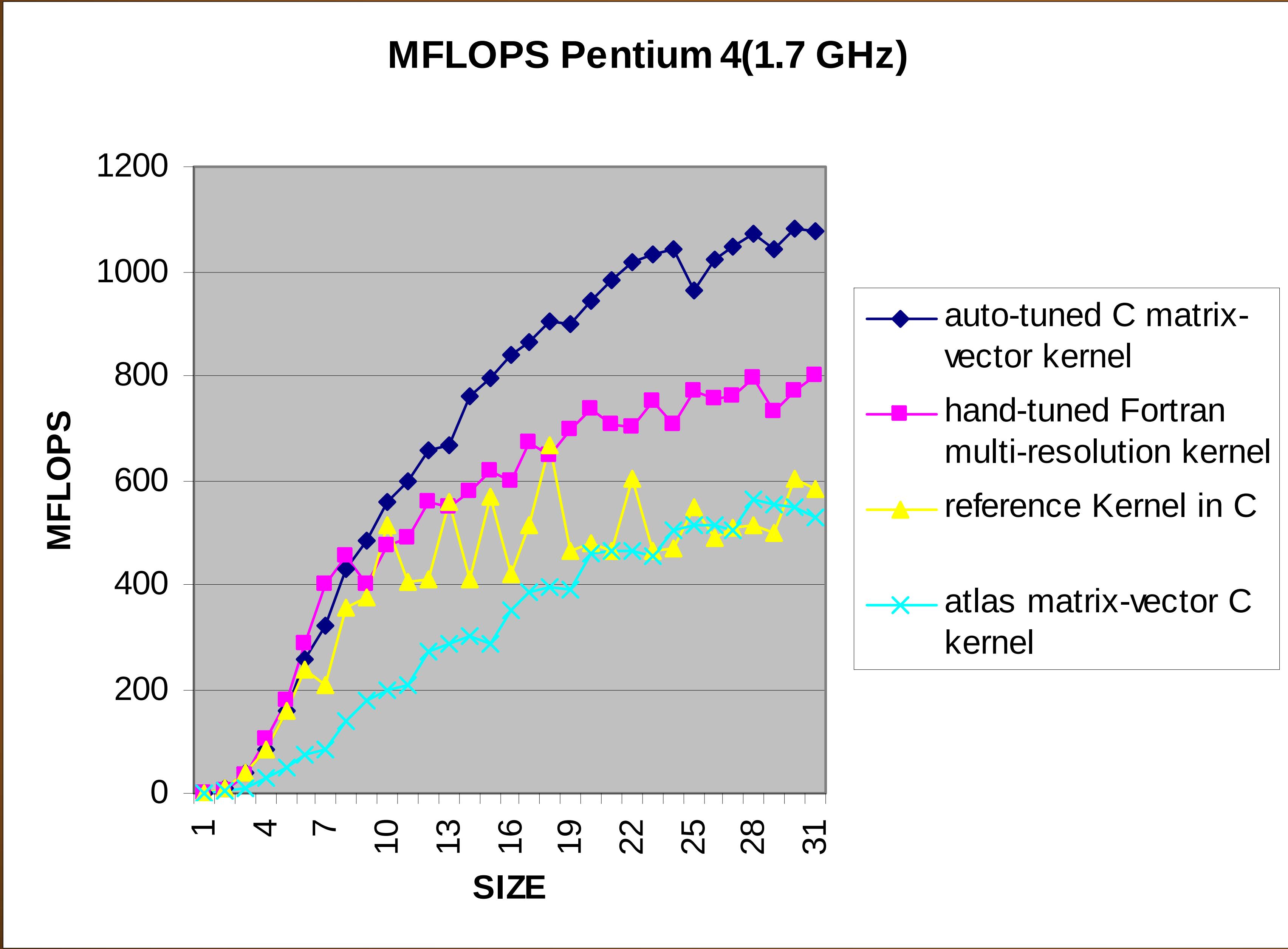


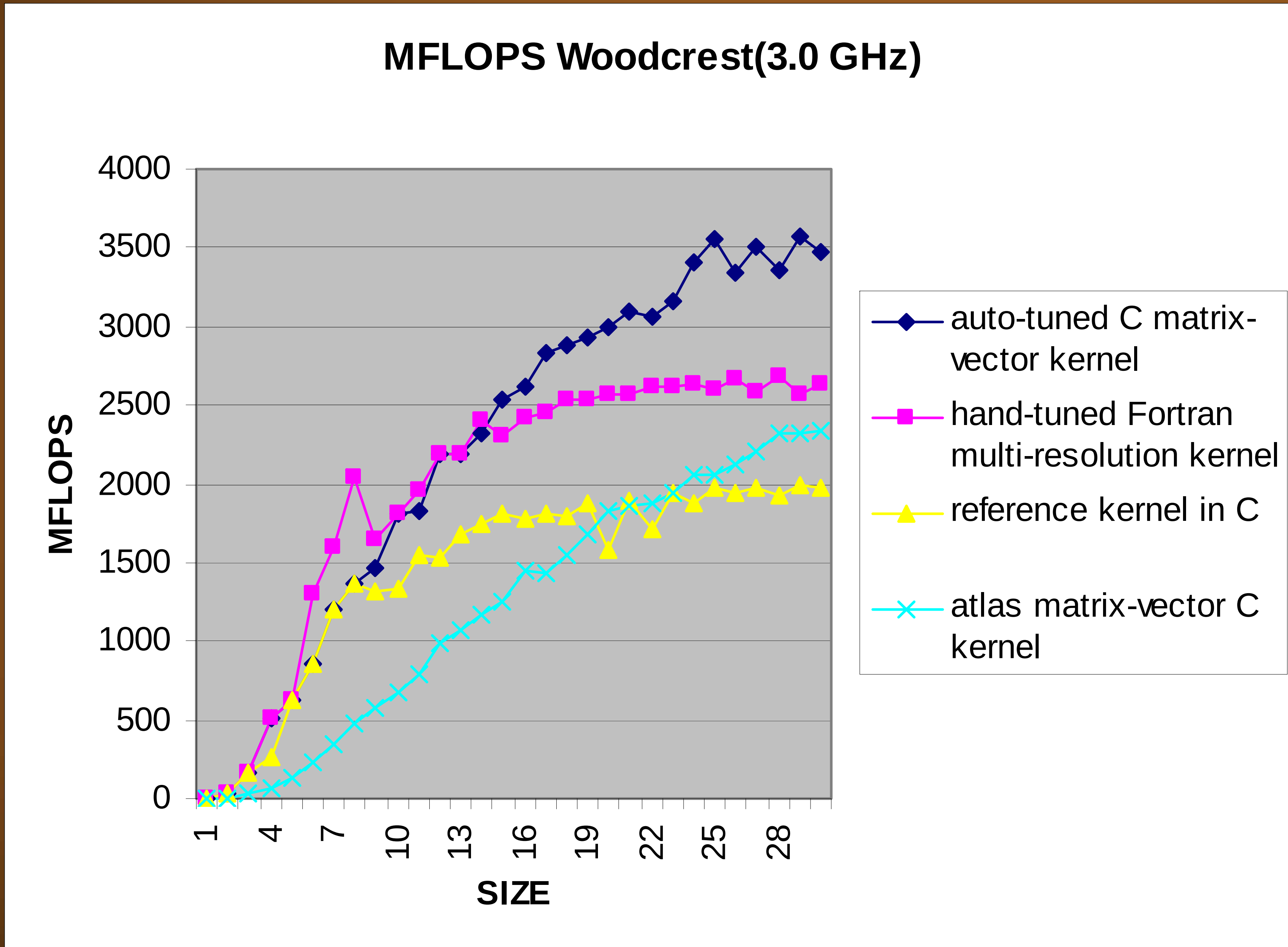


MADNESS Kernel Tuning

- GCO didn't work!
- Instead:
  - Extract matrix-vector multiplication kernel from **doitgen** routine
  - Design and hand-code a specific code generator for small size matrix-vector multiplication
  - Tune optimal block size and unrolling factor separately for each input size


MFLOPS Opteron(1.8 GHz)
2500
2000
1500
1000
500
0
MFLOPS
1
4
7
10
13
16
19
22
25
28
31
SIZE
auto-tuned C matrix-vector kernel
hand-tuned Fortran multi-resolution kernel
reference kernel in C
atlas matrix-vector C kernel


MFLOPS Pentium 4(1.7 GHz)
MFLOPS
SIZE
0
200
400
600
800
1000
1200
1
4
7
10
13
16
19
22
25
28
31
auto-tuned C matrix-vector kernel
hand-tuned Fortran multi-resolution kernel
reference Kernel in C
atlas matrix-vector C kernel



MFLOPS Woodcrest(3.0 GHz)
4000
3500
3000
2500
2000
1500
1000
500
0
MFLOPS
1
4
7
10
13
16
19
22
25
28
SIZE
auto-tuned C matrix-vector kernel
hand-tuned Fortran multi-resolution kernel
reference kernel in C
atlas matrix-vector C kernel

MADNESS Conclusions

- We have demonstrated an effective empirical tuning strategy for optimizing the **`doitgen`** computational kernel code
  - less effort than hand tuning
  - better performance than either:
    - hand-tuned or
    - general purpose optimization
- Future
  - Aggressive code generator for MV multiplication
  - Parallelize parameter search

Thank you

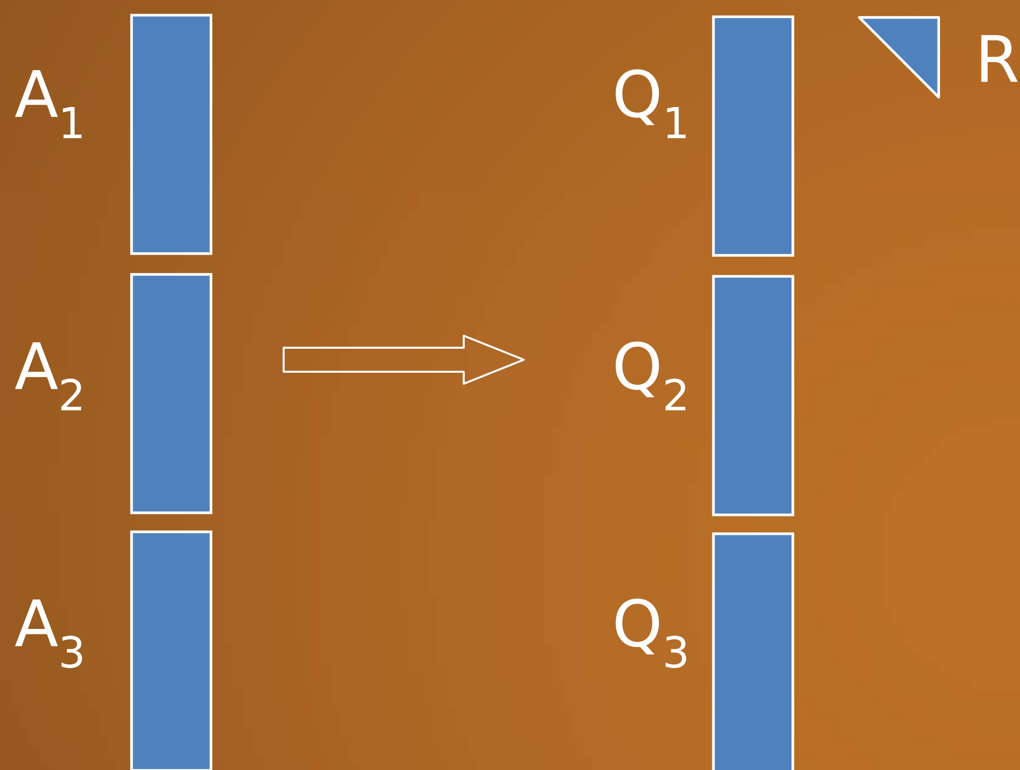
http://icl.cs.utk.edu

AllReduce algorithms

The QR factorization of a long and skinny matrix with its data partitioned vertically across several processors arises in a wide range of applications.

Input:
A is block distributed by rows

Output:
Q is block distributed by rows
R is global



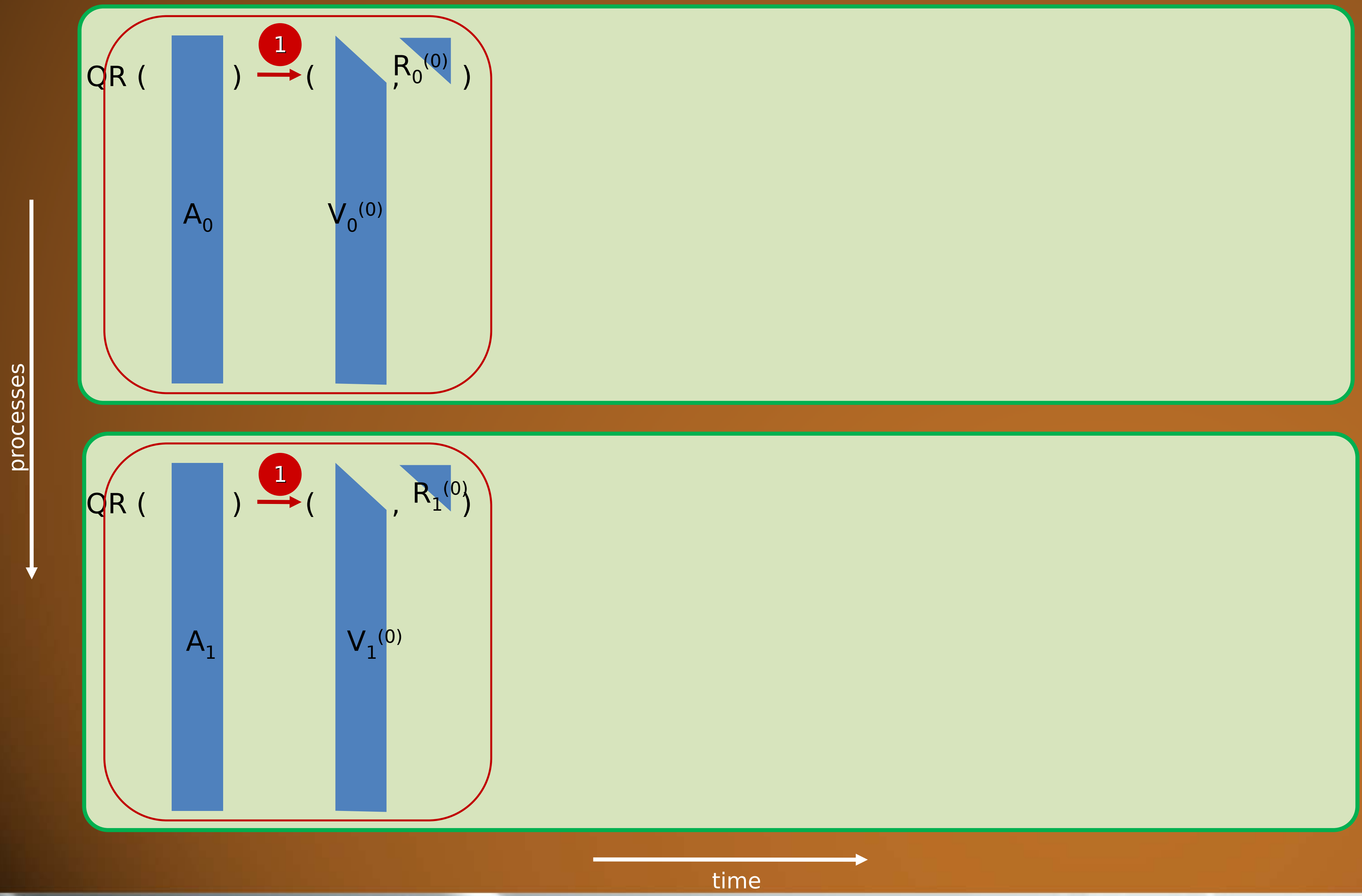
AllReduce algorithms

They are used in:

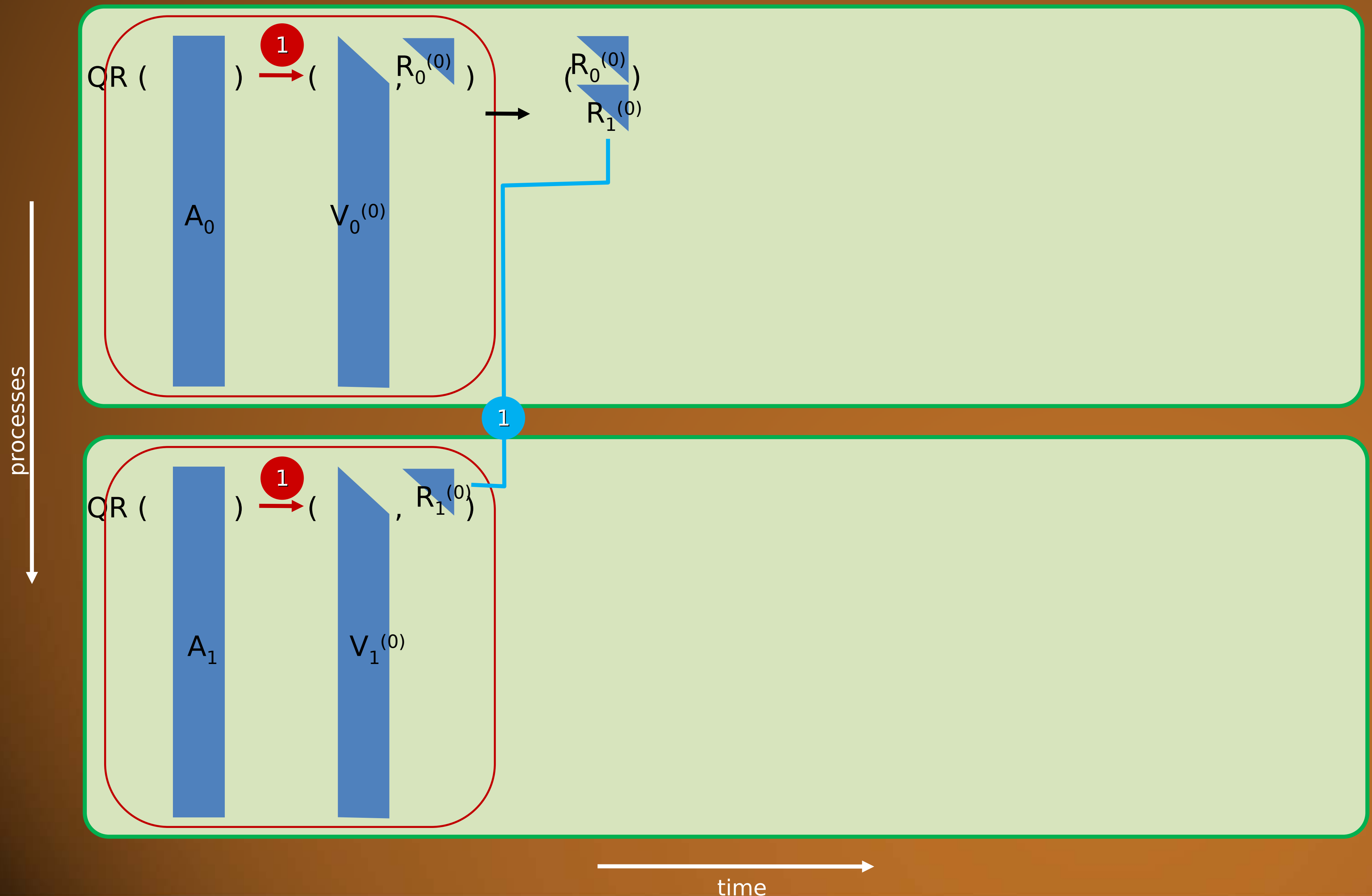
- **in iterative methods with multiple right-hand sides** (block iterative methods:)
  - Trilinos (Sandia National Lab.) through Belos (R. Lehoucq, H. Thornquist, U. Hetmaniuk).
  - BlockGMRES, BlockGCR, BlockCG, BlockQMR, …
- **in iterative methods with a single right-hand side**
  - s-step methods for linear systems of equations (e.g. A. Chronopoulos),
  - LGMRES (Jessup, Baker, Dennis, U. Colorado at Boulder) implemented in PETSc,
  - Recent work from M. Hoemmen and J. Demmel (U. California at Berkeley).
- **in iterative eigenvalue solvers,**
  - PETSc (Argonne National Lab.) through BLOPEX (A. Knyazev, UCDHSC),
  - HYPRE (Lawrence Livermore National Lab.) through BLOPEX,
  - Trilinos (Sandia National Lab.) through Anasazi (R. Lehoucq, H. Thornquist, U. Hetmaniuk),
  - PRIMME (A. Stathopoulos, Coll. William & Mary)

AllReduce algorithms

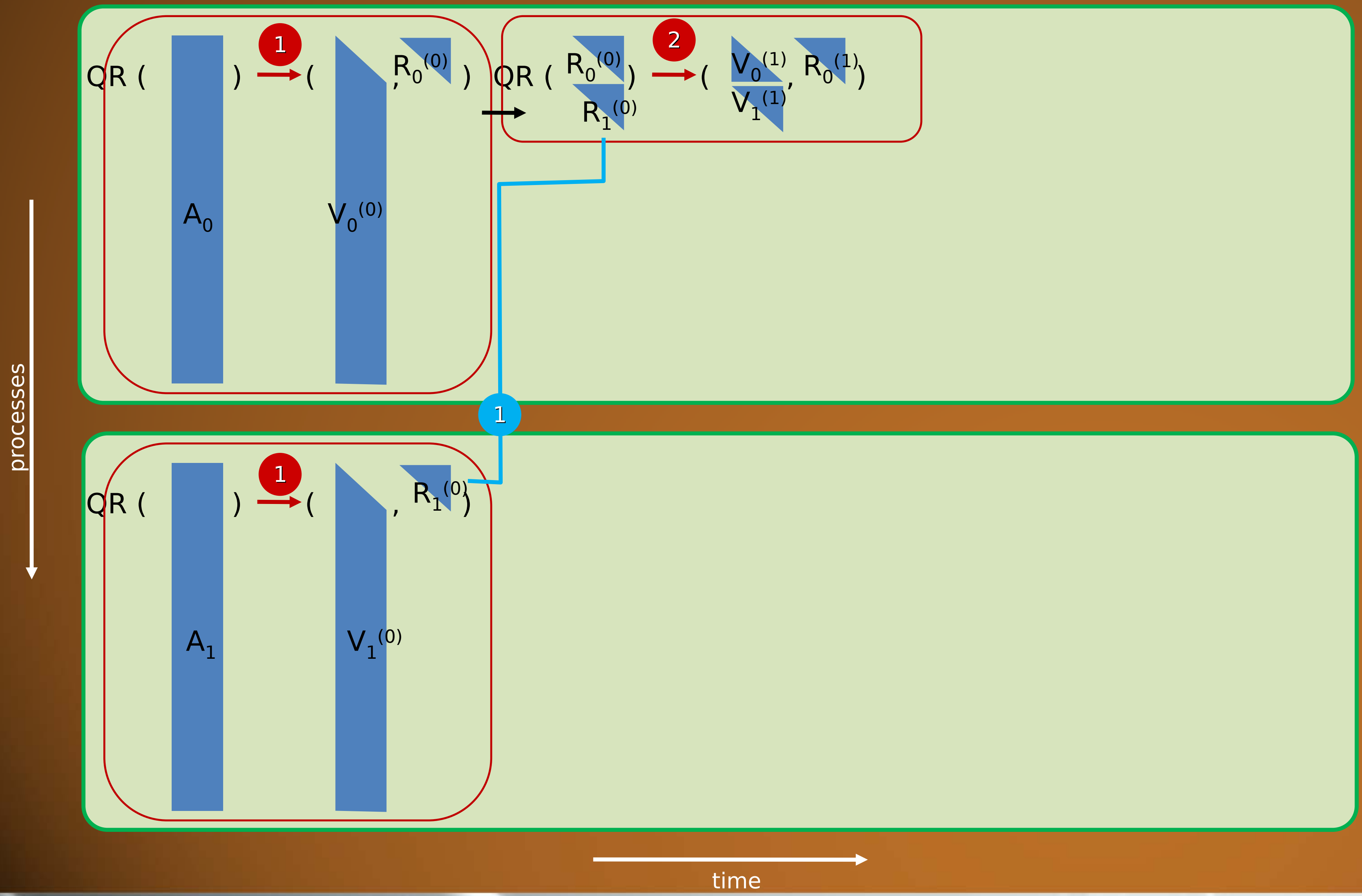
AllReduce algorithms



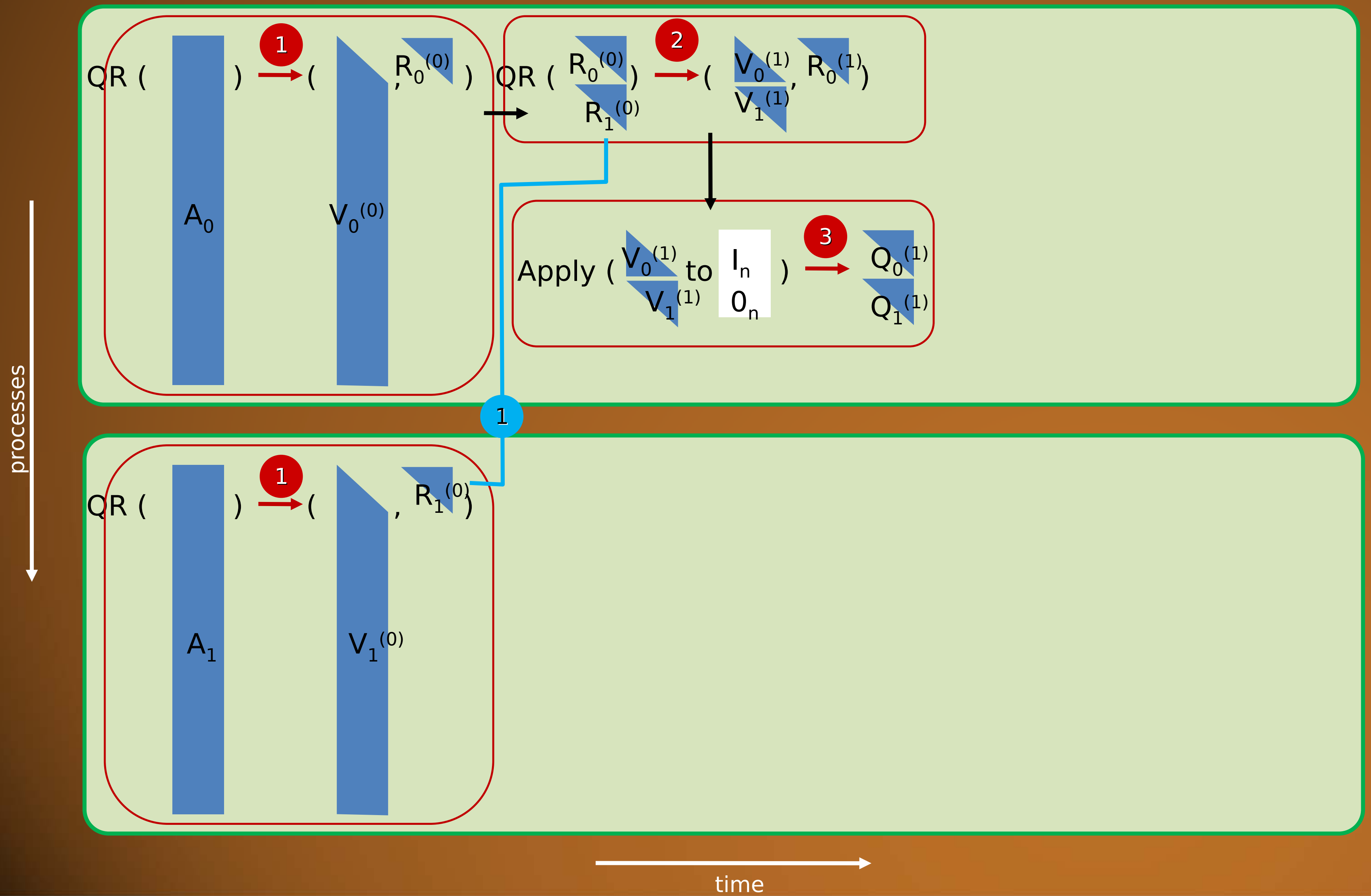
AllReduce algorithms



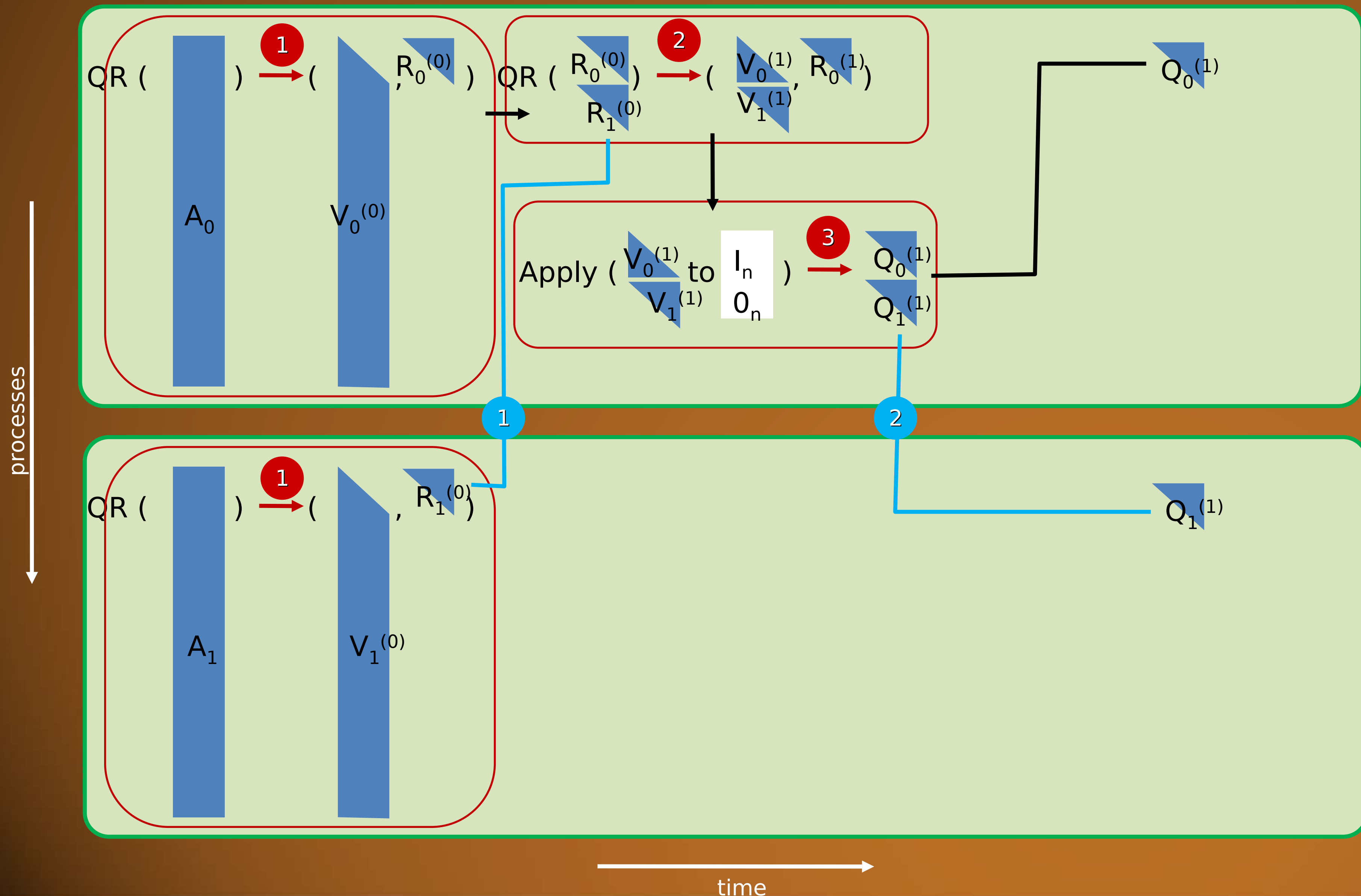
AllReduce algorithms



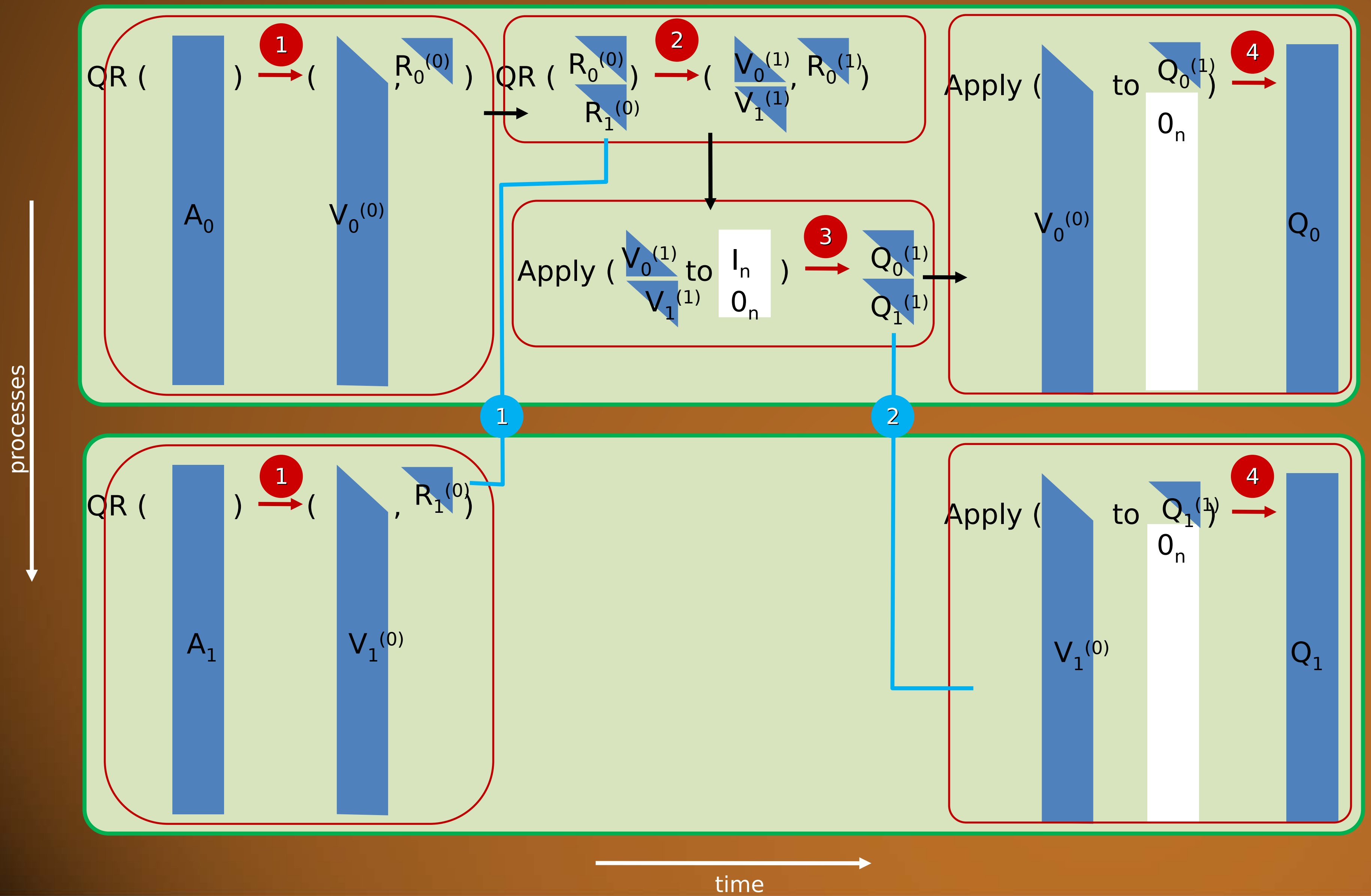
AllReduce algorithms



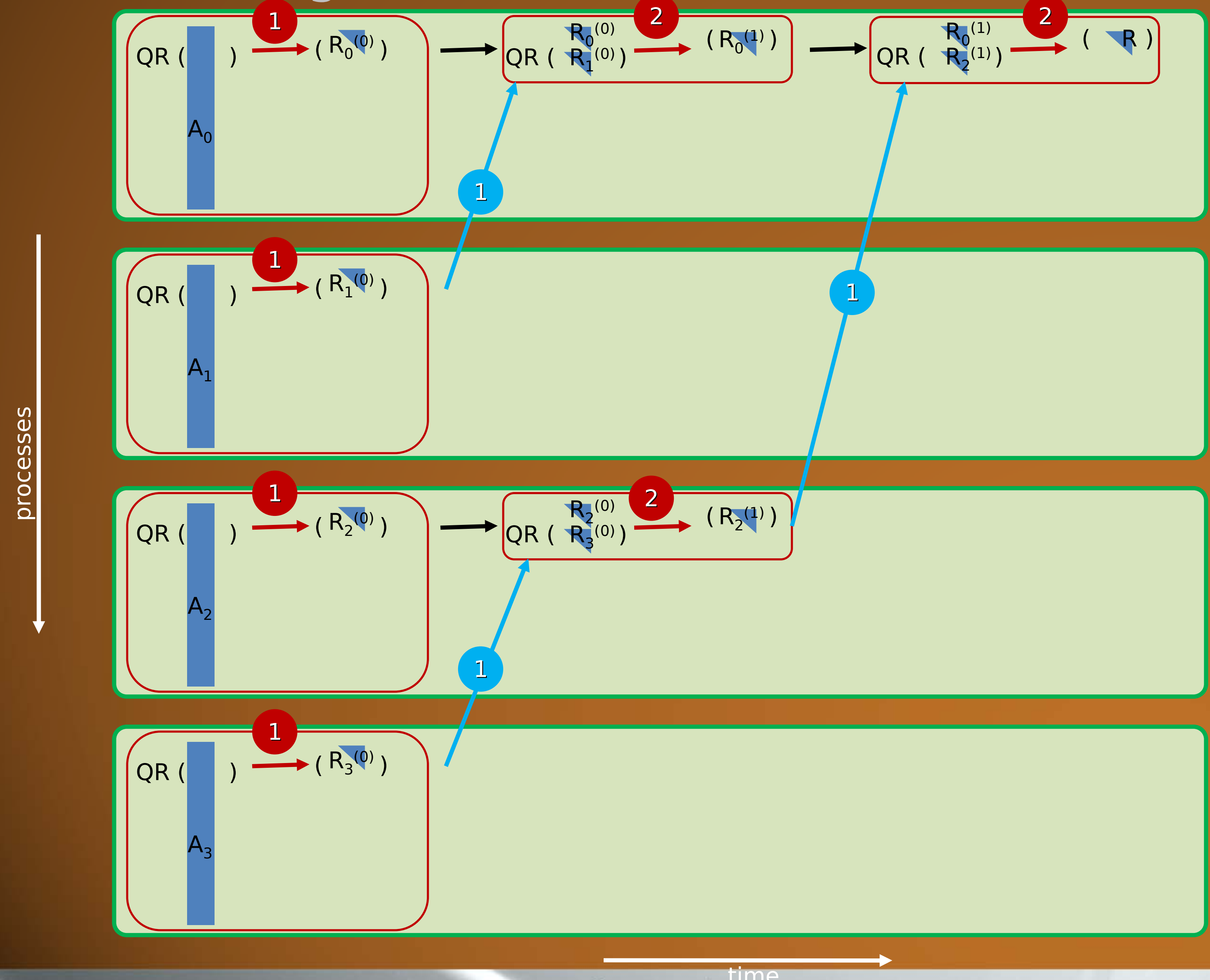
AllReduce algorithms



AllReduce algorithms



AllReduce algorithms



AllReduce algorithms: performance

Weak Scalability

Strong Scalability