

Parallel Tiled Algorithms for Multicore Architectures

Alfredo Buttari, Jack Dongarra, Jakub Kurzak and Julien Langou

SciDAC CScADS Summer Workshop on Libraries and Algorithms for Petascale Applications

Snowbird UT, July 31st 2007



The free lunch is over

Hardware

Problem

- power consumption
- heat dissipation
- pins

Solution

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...)
- memory?
 - how is it arranged?
- bus?
 - is it going to be fast enough?
- cache?
 - shared? (Intel/AMD)
 - non present at all? (STI Cell)
- communications?

What is a Multicore processor, BTW?

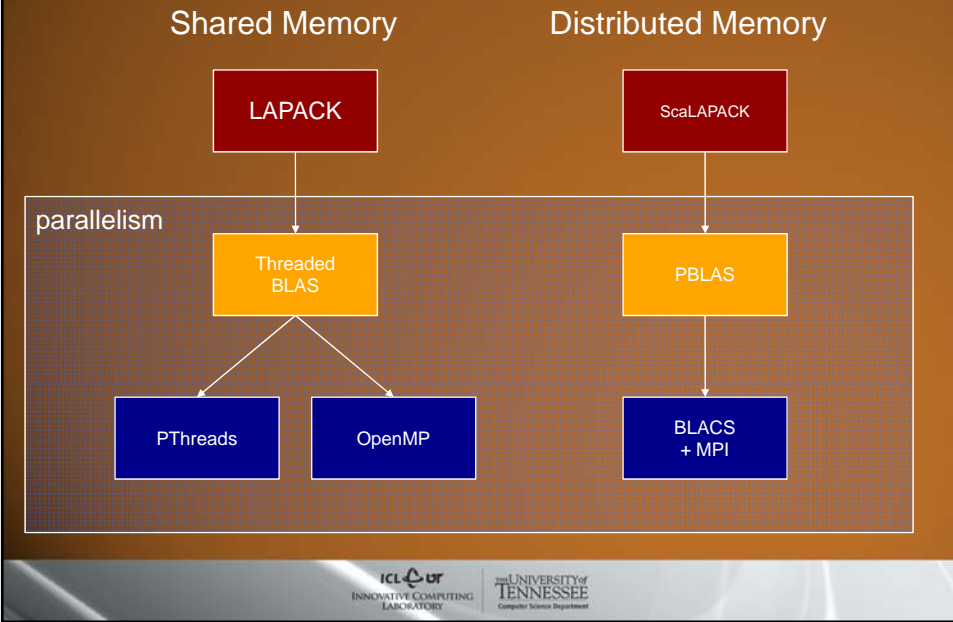
Parallel software for multicores should have two characteristics:

• **fine granularity:**

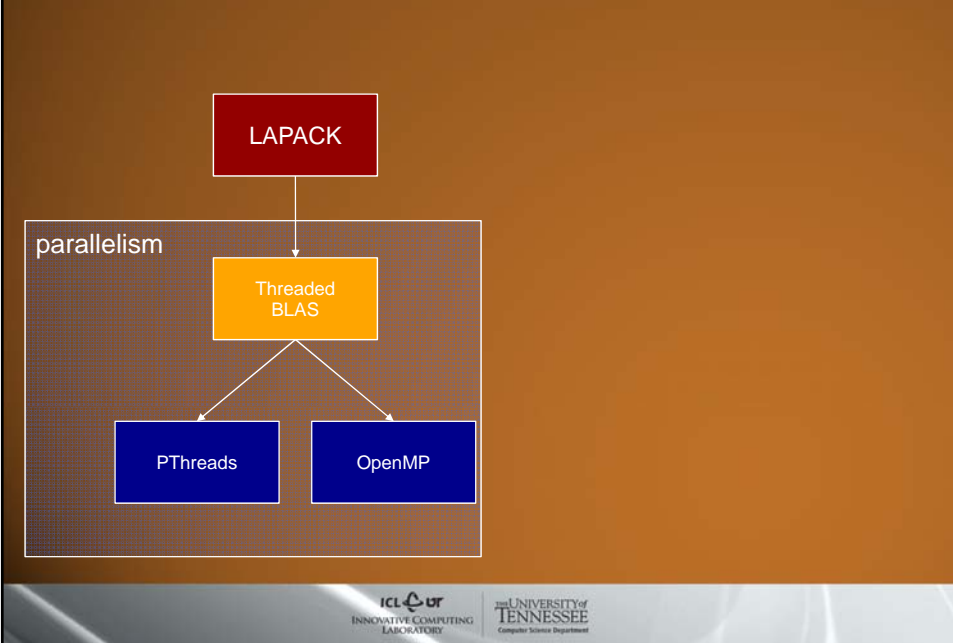
- high parallelism degree is needed
- cores are (and probably will be) associated with relatively small local memories. This requires splitting an operation into tasks that operate on small portions of data in order to reduce bus traffic and improve data locality.

• **asynchronicity:** as the degree of TLP grows and granularity of the operations becomes smaller, the presence of synchronization points in a parallel execution seriously affects the efficiency of an algorithm.

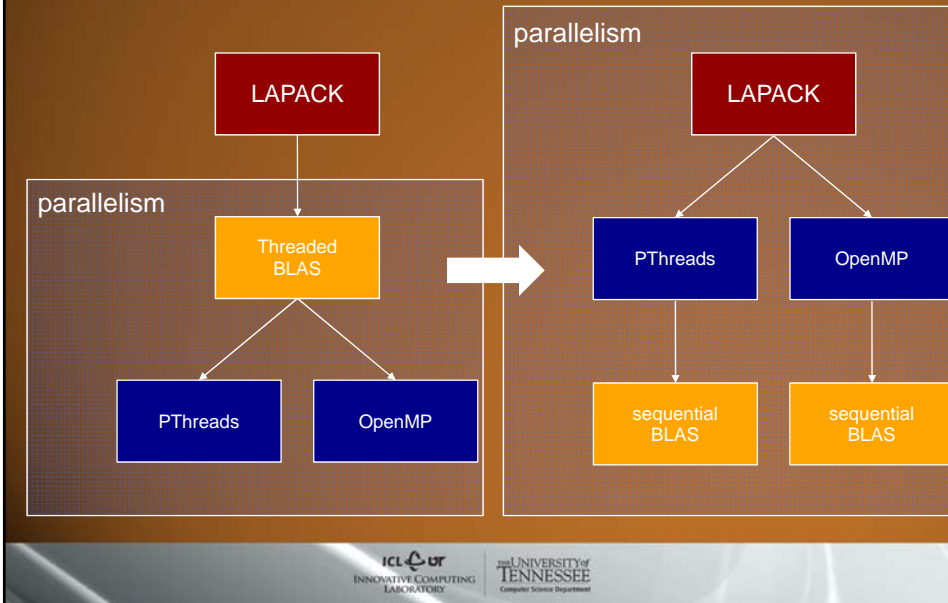
Parallelism in Linear Algebra software so far



Parallelism in Linear Algebra software so far



Parallelism in Linear Algebra software so far



The LAPACK algorithm for QR factorization

The QR factorization in LAPACK

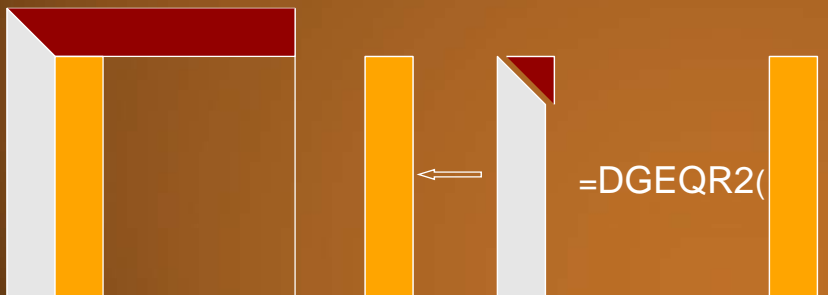
The QR transformation factorizes a matrix A into the factors Q and R where Q is unitary and R is upper triangular. It is based on Householder reflections.



Assume that A_{11} is the part of the matrix that has been already factorized and A_{21} contains the Householder reflectors that determine the matrix Q .

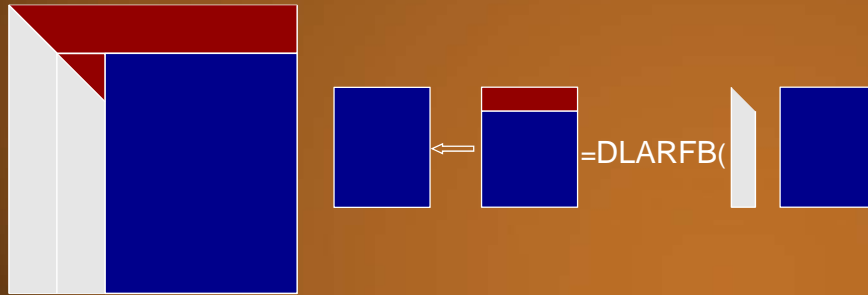
The QR factorization in LAPACK

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.



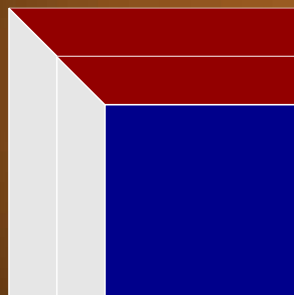
The QR factorization in LAPACK

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

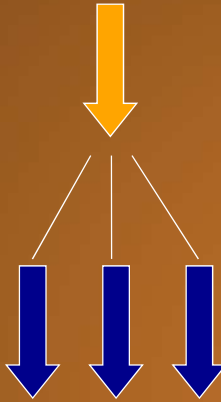
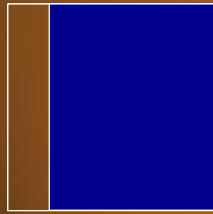
The QR transformation factorizes a matrix A into the factors Q and R where Q is unitary and R is upper triangular. It is based on Householder reflections.



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 $\frac{4}{3}n^3$ versus $\frac{2}{3}n^3$

Parallelism in LAPACK: LU/QR factorizations



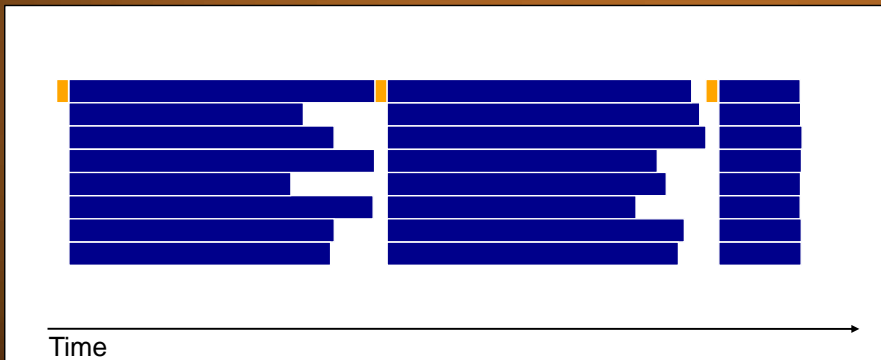
DGEQR2: BLAS-2
non-blocked panel
factorization

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

Parallelism in LAPACK: LU/QR factorizations

- strict synchronization
- poor parallelism
- poor scalability

# cores	DGEQF2	DGEQRF
	(Gflop/s)	(Gflop/s)
1	0.45	3.31
2	0.46	5.51
4	0.46	9.69
8	0.45	10.58



A parallel tiled algorithm for QR factorization

Parallel tiled QR factorization

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

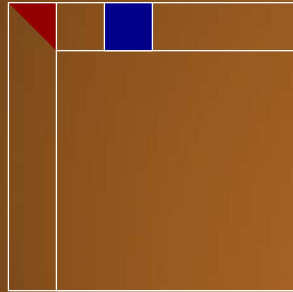


$$\text{[Red Square]} \leftarrow \text{[Red Triangle]} = \text{DGEQT2}(\text{[Red Square]})$$

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

Parallel tiled QR factorization

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

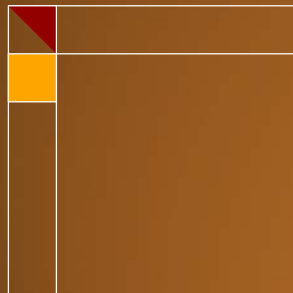


$$\text{blue square} \leftarrow \text{blue square} = \text{DLARFB}(\text{white triangle}, \text{blue square})$$

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

Parallel tiled QR factorization

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

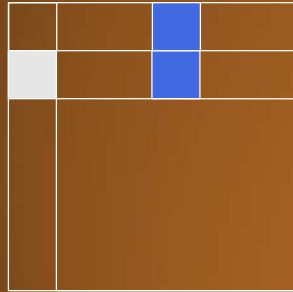


$$\text{red triangle} \leftarrow \text{red triangle} = \text{DTSQT2}(\text{white square}, \text{yellow square})$$


The R factor DTSQT2 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.

Parallel tiled QR factorization

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

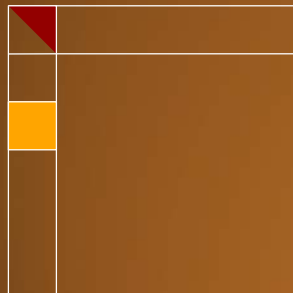


$$\begin{bmatrix} \text{blue} \\ \text{blue} \end{bmatrix} \leftarrow \begin{bmatrix} \text{blue} \\ \text{blue} \end{bmatrix} = \text{DSSRFB}(1, \begin{bmatrix} \text{blue} \\ \text{blue} \end{bmatrix}, \text{grey})$$

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.

Parallel tiled QR factorization

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

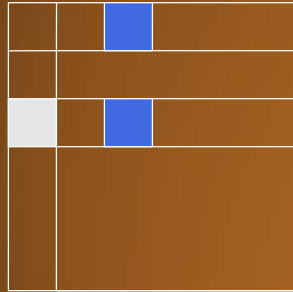


$$\begin{bmatrix} \text{red} \\ \text{yellow} \end{bmatrix} \leftarrow \begin{bmatrix} \text{red} \\ \text{yellow} \end{bmatrix} = \text{DTSQT2}(1, \begin{bmatrix} \text{red} \\ \text{yellow} \end{bmatrix}, \text{grey})$$

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

Parallel tiled QR factorization

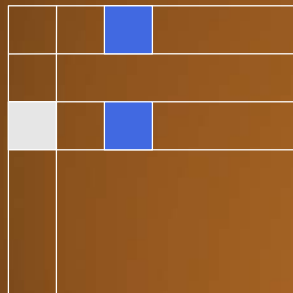
A different algorithm can be used where operations can be broken down into tiles.



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

Parallel tiled QR factorization

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.

Parallel tiled QR factorization

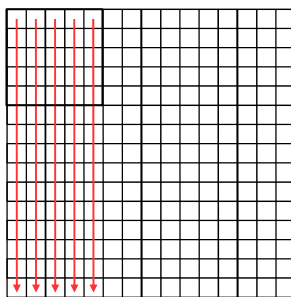
$$\begin{pmatrix} A_{11} & A_{12} & \dots & A_{1q} \\ A_{21} & A_{22} & \dots & A_{2q} \\ \vdots & & \ddots & \vdots \\ A_{p1} & A_{p2} & \dots & A_{pq} \end{pmatrix}$$

```

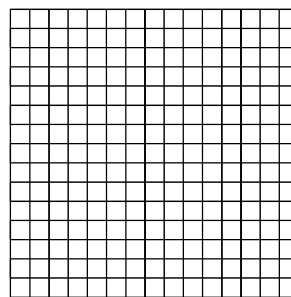
1: for  $k = 1, 2, \dots, \min(p, q)$  do
2:   DGEQT2( $A_{kk}, T_{kk}$ );
3:   for  $j = k + 1, k + 2, \dots, q$  do
4:     DLARFB( $A_{kj}, V_{kk}, T_{kk}$ );
5:   end for
6:   for  $i = k + 1, k + 1, \dots, p$  do
7:     DTSQT2( $R_{ik}, A_{ik}, T_{ik}$ );
8:     for  $j = k + 1, k + 2, \dots, q$  do
9:       DSSRFB( $A_{kj}, A_{ij}, V_{ik}, T_{ik}$ );
10:    end for
11:  end for
12: end for
    
```

Parallel tiled QR factorization: block data layout

Column-Major

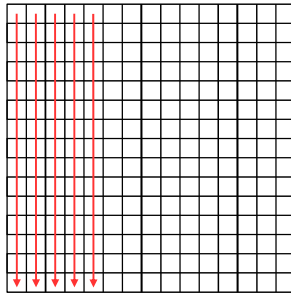


Block data layout

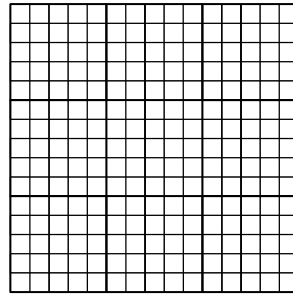


Parallel tiled QR factorization: block data layout

Column-Major

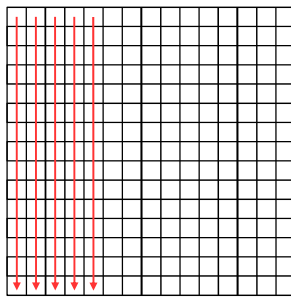


Block data layout

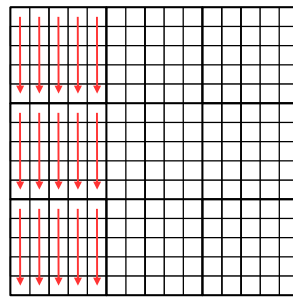


Parallel tiled QR factorization: block data layout

Column-Major

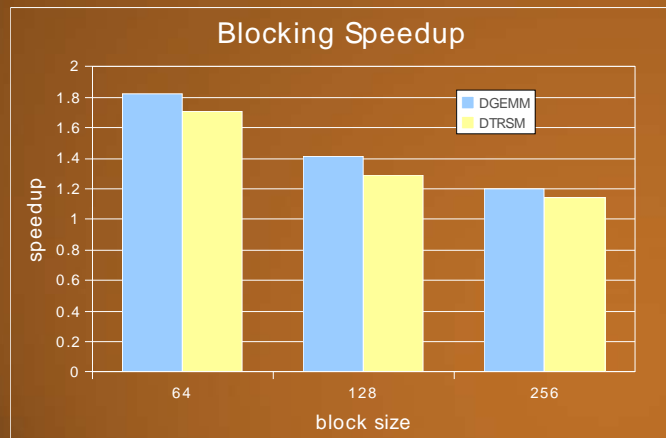


Block data layout



Parallel tiled QR factorization: block data layout

The use of block data layout storage can significantly improve performance

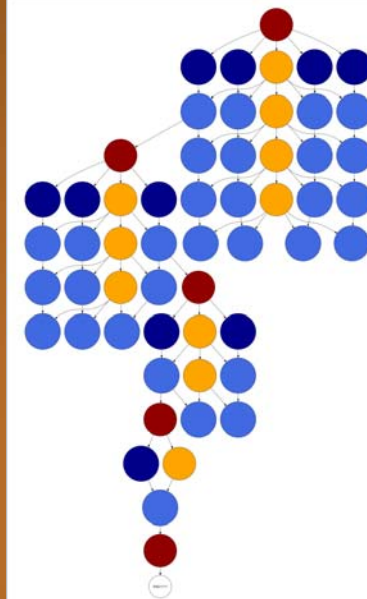


Parallel tiled QR factorization: scheduling

The whole factorization can be represented as a DAG:

- nodes: tasks that operate on tiles
- edges: dependencies among tasks

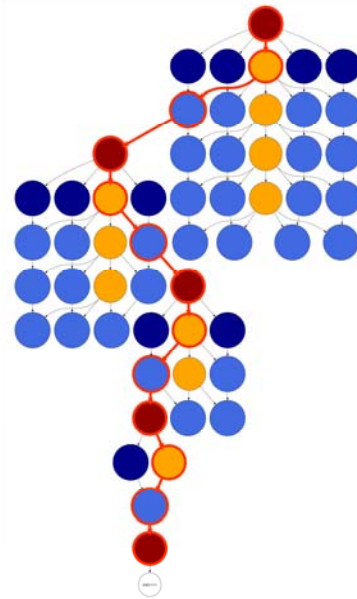
Tasks can be scheduled asynchronously and in any order as long as dependencies are not violated.



Parallel tiled QR factorization: scheduling

A critical path can be defined as the shortest path that connects all the nodes with the higher number of outgoing edges.

Priorities:



Parallel tiled QR factorization

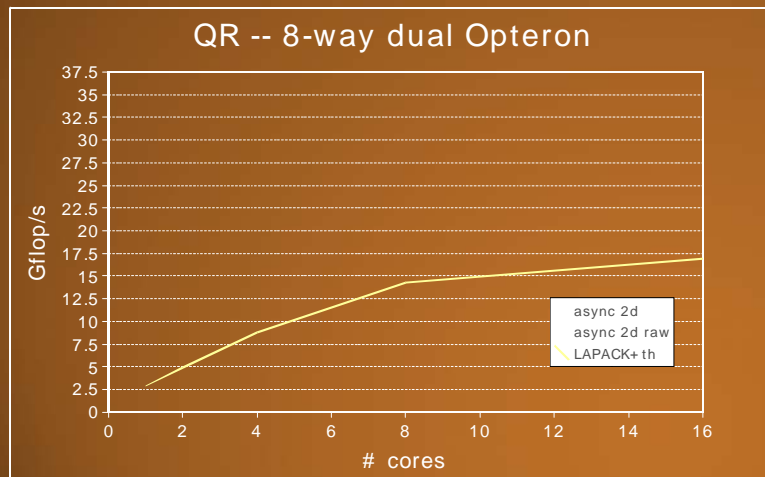
- very fine granularity
- few dependencies, i.e., high flexibility for the scheduling of tasks →asynchronous scheduling
- no idle times
- some degree of adaptativity
- better locality thanks to block data layout

Parallel tiled QR factorization

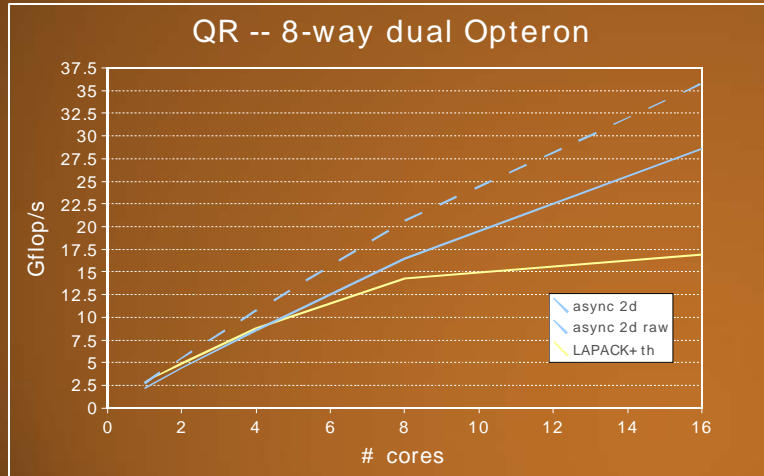
Execution flow on a 8-way dual core Opteron.



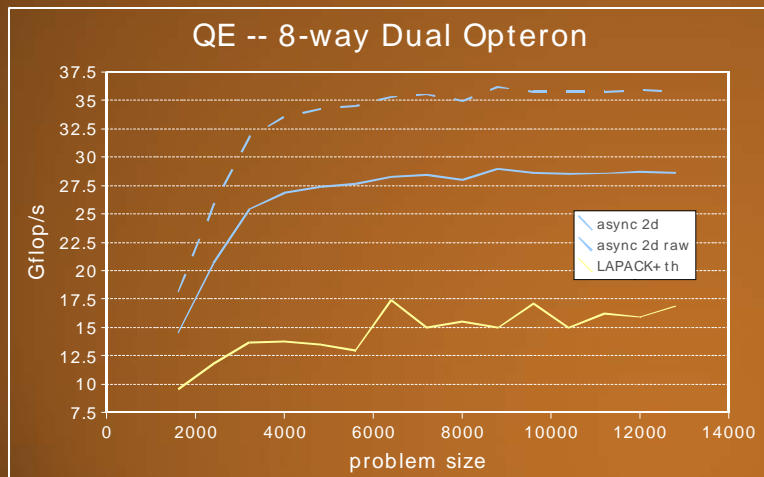
Parallel tiled QR factorization: results



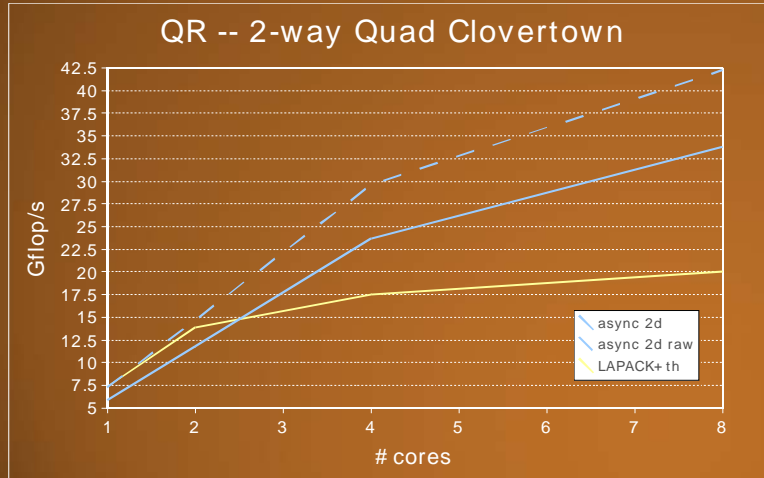
Parallel tiled QR factorization: results



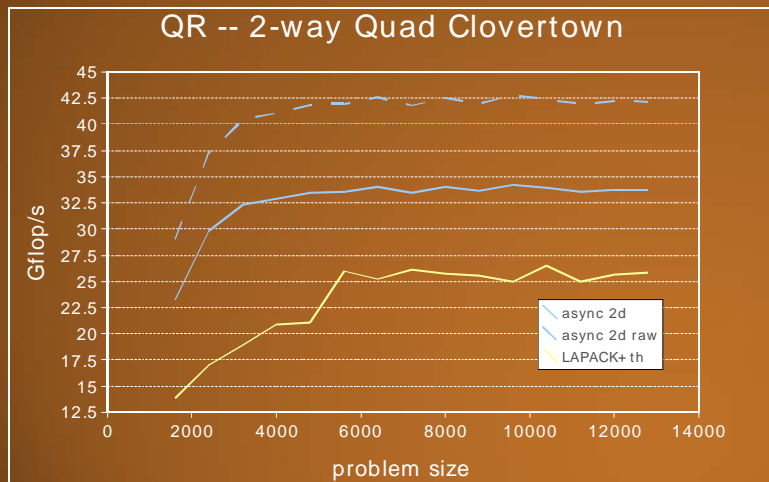
Parallel tiled QR factorization: results



Parallel tiled QR factorization: results



Parallel tiled QR factorization: results



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



<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 instructing the code with pragmas and, thus, is transparent to other compilers



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)
```



Conclusions

- Fine granularity and loose synchronism are key features for multicore-friendly algorithms
- Is it worth paying the cost of higher opcounts for the sake of scalability?

YES

- parallel tiled algorithms
- OSKI
- low latency iterative solvers



- Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra
“*Parallel Tiled QR Factorization for Multicore Architectures*”.
LAWN #190, UT-CS-07-598,
July 2007.

- Brian Gunter and Robert van de Geijn.
“*Parallel Out-of-Core Computation and Updating of the QR Factorization*”.
ACM Transactions on Mathematical Software, 31(1):60-78, March 2005.

- E. L. Yip.
“*FORTRAN Subroutines for Out-of-Core Solutions of Large Complex Linear Systems*”.
Technical Report CR-159142, NASA,
November 1979.

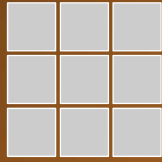


Thank you

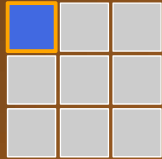
<http://icl.cs.utk.edu>



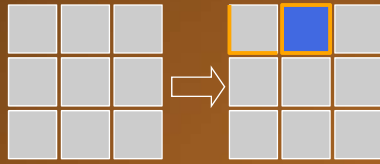
Parallel tiled QR factorization



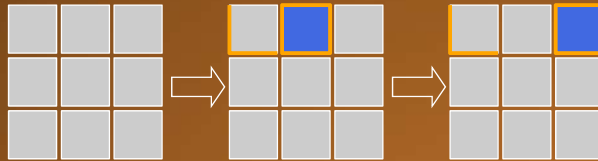
Parallel tiled QR factorization



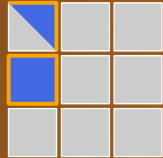
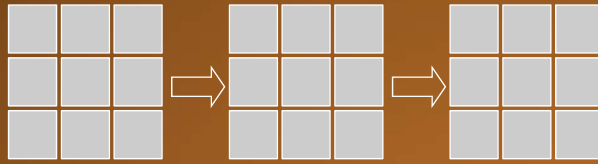
Parallel tiled QR factorization



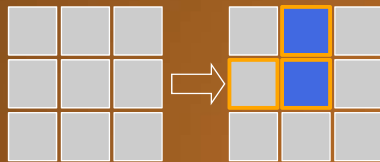
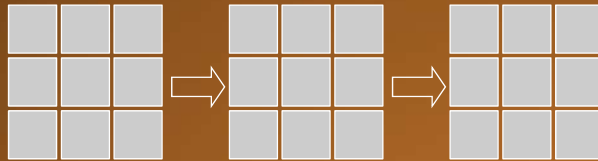
Parallel tiled QR factorization



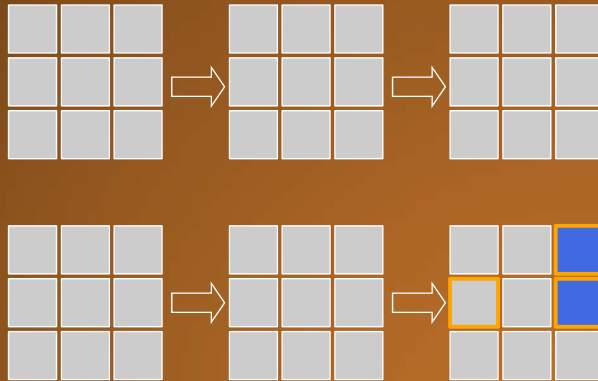
Parallel tiled QR factorization



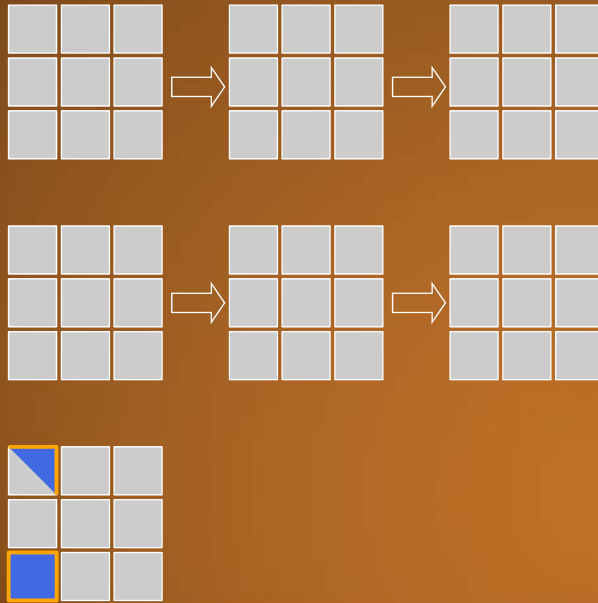
Parallel tiled QR factorization



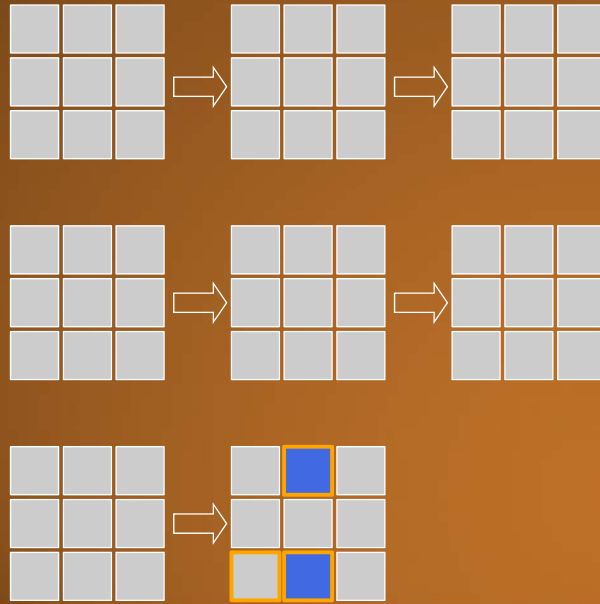
Parallel tiled QR factorization



Parallel tiled QR factorization



Parallel tiled QR factorization



Parallel tiled QR factorization

