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

**Problem**
- power consumption
- heat dissipation
- pins

**Solution**
- reduce clock and increase execution units = Multicore

**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

**Shared Memory**

- LAPACK
  - Threaded BLAS
    - PThreads
    - OpenMP

**Distributed Memory**

- ScalAPACK
  - PBLAS
    - BLACS + MPI

---

Parallelism in Linear Algebra software so far

**Shared Memory**

- LAPACK
  - Threaded BLAS
    - PThreads
    - OpenMP

---
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 $\begin{bmatrix} \end{bmatrix}$ is the part of the matrix that has been already factorized and $\begin{bmatrix} \end{bmatrix}$ 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.

$= \text{DGEQR2}(\ldots)$
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 $2/3 \, n^3$
Parallelism in LAPACK: LU/QR factorizations

- **DGEQR2**: BLAS-2 non-blocked panel factorization
- **DLARFB**: BLAS-3 updates U with transformation computed in DGETF2

- **Z** strict synchronization
- **Z** poor parallelism
- **Z** poor scalability

<table>
<thead>
<tr>
<th># cores</th>
<th>DGEQF2 (Gflop/s)</th>
<th>DGEQRF (Gflop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.45</td>
<td>3.31</td>
</tr>
<tr>
<td>2</td>
<td>0.46</td>
<td>5.51</td>
</tr>
<tr>
<td>4</td>
<td>0.46</td>
<td>9.69</td>
</tr>
<tr>
<td>8</td>
<td>0.45</td>
<td>10.58</td>
</tr>
</tbody>
</table>

Time
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.

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.

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

Parallel tiled QR factorization
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.
Parallel tiled QR factorization
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.

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.
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

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

*we are working on a way to remove these extra flops.
Parallel tiled QR factorization

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

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

Parallel tiled QR factorization: block data layout

\text{Column-Major} \quad \text{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
The use of block data layout storage can significantly improve performance.

Graph showing blocking speedup with block sizes of 64, 128, and 256.

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:
- 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

QR -- 8-way dual Opteron

Time

Gflop/s

# cores
Parallel tiled QR factorization: results

QR -- 8-way dual Opteron

Gflop/s vs. # cores

async 2d
async 2d raw
LAPACK+th

Parallel tiled QR factorization: results

QE -- 8-way Dual Opteron

Gflop/s vs. problem size

async 2d
async 2d raw
LAPACK+th
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

```c
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[i][j], A[i][i] );
    }
    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)
```
for (i = 0; i < DIM; ++i) {
    for (j = i - 1; j > -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.”

• E. L. Yip.
  “FORTRAN Subroutines for Out-of-Core Solutions of Large Complex Linear Systems”.

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