Out-of-Core Computation of the QR Factorization on Multi-core Processors

Mercedes Marqués[1], Gregorio Quintana-Ortí[1], Enrique S. Quintana-Ortí[1], and Robert van de Geijn[2]

[1] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12.071–Castellón, Spain
{mmarques, gquintan, quintana}@icc.uji.es

[2] Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
rvdg@cs.utexas.edu

Abstract. We target the development of high-performance algorithms for dense matrix operations where data resides on disk and has to be explicitly moved in and out of the main memory. We provide strong evidence that, even for a complex operation like the QR factorization, the use of a run-time system creates a separation of concerns between the matrix computations and I/O operations with the result that no significant changes need to be introduced to existing in-core algorithms. The library developer can thus focus on the design of algorithms-by-blocks, addressing disk memory as just another level of the memory hierarchy. Experimental results for the out-of-core computation of the QR factorization on a multi-core processor reveal the potential of this approach.

Keywords: Dense linear algebra, out-of-core computation, QR factorization, multi-core processors, high performance.

1 Introduction

Practical efforts to solve very large dense linear systems employ message-passing libraries on distributed-memory systems, extending the memory hierarchy to include secondary memory; see, e.g., [1,2,3,4]. However, the constant evolution of computer architectures and, more recently, the uprise of general-purpose multi-core processors and hardware accelerators (Cell B.E., GPUs, etc.) is changing the scale of what is considered a *large* problem.

In a previous paper [5] we employed a simple operation like the Cholesky factorization to introduce a high-level approach for computing out-of-core (OOC) dense matrix operations on multi-core processors. We showed there that a matrix with 100,000 rows and columns can be factorized in less than one hour using an eight-core processor. (This problem would have been solved using a distributed-memory platform just a couple of years ago.) Key to our approach is a run-time system which deals with I/O from/to disk, implements a software cache, and overlaps I/O with computation. The most remarkable property, however, is that no significant change is needed to the in-core `libflame` library code [6].

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 809–820, 2009.

The major contribution of this paper is to provide much stronger and practical evidence of the validity of our approach, analyzing the programmability and performance issues in detail using a more complex operation, the QR factorization. We will demonstrate that the run-time completely hides secondary memory to the library developer, who can focus on developing and parallelizing algorithms-by-blocks (also called tiled algorithms; see, e.g., [4]), with a notable increase in the programmer's productivity.

Our results also show that, provided the user is willing to wait a few hours for the answer, the approach may become a highly cost-effective solution for most dense matrix operations, making it possible that projects with moderate budgets address problems of moderate scale (dimensions of $O(10,000 - 100,000)$) using multi-core processors. Examples of problems that require the solution of large dense linear systems or linear least-squares problems of this dimension include the estimation of Earth's gravitational field, boundary element formulations in electromagnetism and acoustics, and molecular dynamics simulations [7,8,9,10,11].

The rest of the paper is structured as follows. In Section 2 we describe the tiled left-looking algorithm for computing an OOC QR factorization proposed in [12]. Algorithms for this operation are presented using the FLAME notation there, and the multi-threaded parallelization of the basic building kernels is also analyzed in that section. The traditional OOC implementation of the tiled QR factorization and the new run-time are described in Section 3. Finally, experimental results on a multi-core processor with two Intel Xeon QuadCore processors are reported in Section 4, and concluding remarks follow in Section 5.

2 A Tiled Algorithm for the QR Factorization

The QR factorization of a matrix $A \in \mathbb{R}^{m \times n}$ decomposes this matrix into the product

$$A = QR,$$

where $Q \in \mathbb{R}^{m \times m}$ is orthogonal and $R \in \mathbb{R}^{m \times n}$ is upper triangular.

Traditional in-core algorithms for the QR factorization employ Householder reflectors [13] to annihilate the subdiagonal elements of the matrix, processing one column per iteration (from left to right), and effectively reducing A to the upper triangular factor R. In practice, the elements of R overwrite the corresponding entries of A and Q is not formed explicitly; instead, the reflectors are stored in compact form using the strictly lower triangle of the matrix (plus some negligible work space). Blocked algorithms build upon this procedure to improve data locality: at each iteration, the current panel (or slab) of columns is factored, and the columns to its right are updated using efficient level-3 *Basic Linear Algebra Subprograms* (BLAS) [14].

2.1 The Tiled Left-Looking Algorithm

While right-looking blocked algorithms which proceed by slabs in general yield high performance for in-core operations, tiled left-looking algorithms are usually

Algorithm: $[A] := \text{QR\_B}(A)$

Partition $A \rightarrow \left(A_L \,\|\, A_R \right)$
  where A_L is 0 tiles wide
while $n(A_L) < n(A)$ **do**
  Repartition
    $\left(A_L \,\|\, A_R \right) \rightarrow \left(A_0 \,\|\, A_1 \,|\, A_2 \right)$
    where A_1 is 1 tile wide

$A_1 := \text{QR\_B1}(A_0, A_1)$

  Continue with
    $\left(A_L \,\|\, A_R \right) \leftarrow \left(A_0 \,|\, A_1 \,\|\, A_2 \right)$
endwhile

Algorithm: $[B] := \text{QR\_B1}(A, B)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right), B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right)$
  where A_{TL} has 0×0 tiles,
    B_T is 0 tiles high
while $n(A_{TL}) < n(A)$ **do**
  Repartition
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right),$$
$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array} \right)$$
  where A_{11} and B_1 are tiles

$$\begin{aligned} B_1 &:= \textsc{Apply\_Q}(A_{11},\ B_1) \\ [B_1,\ B_2] &:= \text{QR\_B2}(A_{11},\ B_1,\ A_{21},\ B_2) \\ B_1 &:= \text{QR}(B_1) \\ [B_1,\ B_2] &:= \text{QR\_B3}(B_1, B_2) \end{aligned}$$

  Continue with
$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right),$$
$$\left(\begin{array}{c} B_T \\ \hline B_B \end{array} \right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array} \right)$$
endwhile

Fig. 1. Tiled algorithms for computing the QR factorization

preferred for OOC computations due to their higher scalability and reduced I/O. We next describe the tiled left-looking algorithm for the QR factorization introduced in [12].

Consider a partitioning of the matrix $A \in \mathbb{R}^{m \times n}$ into square tiles of size t, $A \rightarrow (A_{ij}) \in \mathbb{R}^{t \times t}$. (For simplicity, we assume that both the row and column dimensions of the matrix are integer multiples of t.) Figures 1 and 2 show the tiled algorithm QR_B for computing the QR factorization of this matrix using the FLAME notation [15]. There $m(X)/n(X)$ stand for the number of row/column tiles of a matrix X. We believe the rest of the notation is intuitive.

FLAME also comprises a set of high-level applications programming interfaces (APIs) which allow to easily code algorithms in FLAME notation [15]. Using the `Spark` web site (`http://www.cs.utexas.edu/users/flame`) C codes for the algorithms in Figures 1 and 2 can be obtained in a matter of minutes.

Algorithm: $[C, D] := \text{QR\_B2}(A, B, C, D)$

Partition $C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$, $D \rightarrow \left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right)$
  where C_T and D_T are 0 tiles high
while $m(C_T) < m(C)$ **do**
  Repartition
  $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$, $\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right)$
  where C_1 and D_1 are tiles

$\left(\begin{array}{c} B \\ D_1 \end{array}\right) := \boxed{\textsc{Apply\_Qtd}}\left(\left(\begin{array}{c} A \\ C_1 \end{array}\right), \left(\begin{array}{c} B \\ D_1 \end{array}\right)\right)$

  Continue with
  $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$, $\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \leftarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right)$
endwhile

Algorithm: $[A, C] := \text{QR\_B3}(A, C)$

Partition $C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$
  where C_T is 0 tiles high
while $m(C_T) < m(C)$ **do**
  Repartition
  $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$
  where C_1 is a tile

$\left(\begin{array}{c} A \\ C_1 \end{array}\right) := \boxed{\textsc{QRtd}}\left(\left(\begin{array}{c} A \\ C_1 \end{array}\right)\right)$

  Continue with
  $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$
endwhile

Fig. 2. Tiled algorithms for computing the QR factorization (continued)

2.2 Sequential Basic Building Kernels

Four basic building kernels (BK) appear highlighted in the previous algorithms: the QR factorization of a full dense matrix, the application of orthogonal transformations resulting from it, the QR factorization of a 2×1 blocked matrix with the top submatrix being upper triangular, and the application of the corresponding transformations to a 2×1 blocked matrix:

$$\begin{array}{lll} \text{BK1.} & A & := \text{QR}(A), \\ \text{BK2.} & B & := \textsc{Apply\_Q}(Q, B), \\ \text{BK3.} & \left(\begin{array}{c} R \\ C \end{array}\right) & := \text{QR}\textsc{td}\left(\left(\begin{array}{c} R \\ C \end{array}\right)\right), \quad \text{and} \\ \text{BK4.} & \left(\begin{array}{c} B \\ D \end{array}\right) & := \textsc{Apply\_Qtd}\left(Q, \left(\begin{array}{c} B \\ D \end{array}\right)\right), \end{array}$$

respectively. We note here that in the invocation of BK2 and BK4, Q is replaced by the matrix which contains the appropriate orthogonal transformations (stored in compact form in the strictly lower triangle).

The building kernels BK1 (QR) and BK2 (Apply_Q) are well-known dense linear algebra operations, for which sequential efficient implementations exist as part of `libflame` and LAPACK legacy code (routines `geqrf` and `ormqr`) [16].

The key that makes the discussed tiled QR algorithm practical is the use of structure-aware implementations of BK3 (QRtd) and BK4 (Apply_Qtd) that

Algorithm: $\left[\begin{pmatrix} R \\ C \end{pmatrix}\right] := \text{QRTD}\left(\begin{pmatrix} R \\ C \end{pmatrix}\right)$

Partition $R \rightarrow \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right)$, $C \rightarrow \left(C_L \,\|\, C_R \right)$

where R_{TL} is 0×0, C_L has 0 columns

while $m(R_{TL}) < m(R)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right), \left(C_L \,\|\, C_R \right) \rightarrow \left(C_0 \,\|\, C_1 \,|\, C_2 \right)$$

where R_{11} is $b \times b$, C_1 has b columns

$$\begin{pmatrix} R_{11} \\ C_1 \end{pmatrix} := \text{QR}\left(\begin{pmatrix} R_{11} \\ C_1 \end{pmatrix}\right)$$

$$\begin{pmatrix} R_{12} \\ C_2 \end{pmatrix} := \text{APPLY\_Q}\left(\begin{pmatrix} R_{11} \\ C_1 \end{pmatrix}, \begin{pmatrix} R_{12} \\ C_2 \end{pmatrix}\right)$$

Continue with

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right), \left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,|\, C_1 \,\|\, C_2 \right)$$

endwhile

Fig. 3. Blocked algorithm for the building block BK3

exploit the upper triangular form of the top submatrix. Figure 3 shows how to do so for BK3. Provided $b \ll t$, the procedure there requires $2t^3$ floating-point arithmetic operations (flops), which is considerably lower than the $8t^3/3$ flops required to compute the factorization if the structure of R is not considered and a general QR factorization of $\begin{pmatrix} R \\ C \end{pmatrix}$ was computed. The procedure for BK4 is shown in Figure 4, reducing the cost from $8t^3$ for the general procedure to $4t^3$ flops for the structure-aware one (provided $b \ll t$). For details, see [12].

2.3 Multi-threaded Basic Building Kernels

For multi-core processors, parallel implementations of the kernels BK1 and BK2 can be obtained by just linking the appropriate `libflame`/LAPACK routines with a multi-threaded implementation of BLAS. The result is a code that will extract all its parallelism from the invocations to BLAS from within the routines. Alternatively, one can also use multi-threaded implementations of `geqrf` and `ormqr` that are part of MKL for these two building kernels, or the parallel data-driven algorithm with dynamic scheduling described in [15].

The parallelization of the structure-aware building kernels BK3 and BK4 is more challenging. Of course, one can still link the sequential codes with a

Algorithm: $\left[\begin{pmatrix} B \\ D \end{pmatrix}\right] := \textsc{Apply\_Qtd}\left(\begin{pmatrix} R \\ C \end{pmatrix}, \begin{pmatrix} B \\ D \end{pmatrix}\right)$

Partition $R \rightarrow \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$, $C \rightarrow \left(C_L \,\|\, C_R \right)$

where R_{TL} is 0×0, B_T has 0 rows C_L has 0 columns,

while $m(R_{TL}) < m(R)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right),$$

$$\left(C_L \,\|\, C_R \right) \rightarrow \left(C_0 \,\|\, C_1 \,|\, C_2 \right)$$

where R_{11} is $b \times b$, B_1 has b rows, C_1 has b columns

---

$$\begin{pmatrix} B_1 \\ D \end{pmatrix} := \textsc{Apply\_Q}\left(\begin{pmatrix} R_{11} \\ C_1 \end{pmatrix}, \begin{pmatrix} B_1 \\ D \end{pmatrix}\right)$$

Continue with

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right), \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right),$$

$$\left(C_L \,\|\, C_R \right) \leftarrow \left(C_0 \,|\, C_1 \,\|\, C_2 \right)$$

endwhile

Fig. 4. Blocked algorithm for the building block BK4

multi-threaded implementation of BLAS, yielding a parallel version of the codes that extracts all parallelism from within the calls to BLAS. However, given that b is small, low performance can be expected from this. We will refer to this first variant as *intra-tile column parallel*.

On the other hand, the application of transformations only occurs from the left, making the updates independent by slabs of columns. Thus, a highly parallel multi-thread implementation can be obtained by splitting the columns of the matrix that need to be updated into several slabs (as many as threads are being used) and then computing the update concurrently. We will refer to this second variant as *column parallel*.

3 OOC Algorithms for the QR Factorization

3.1 A Traditional OOC Implementation

Developers of OOC codes usually decide first how many tiles will be kept in-core, then carefully design their algorithms to reduce the number of data movements between main memory and disk, and finally insert the appropriate invocations

to I/O calls in the codes. In general, the tile size t is set to occupy a fraction of the main memory as large as possible: provided t is large enough, the I/O overhead necessary to move the tiles involved in a given operation is negligible compared with the cost of the computations ($O(t^2)$ accesses to disk vs. $O(t^3)$ flops). OOC algorithms that keep three to four tiles in-core are common for many dense linear algebra operations. To attain higher performance, computation and I/O have to be overlapped, and space is needed to store data in-core which will be involved in future computations (*double-buffering*). However, overlapping for performance also complicates programming, as asynchronous I/O routines need to be employed.

We explain next how to unburden the library developer from explicitly managing I/O and overlapping it with the computation.

3.2 A Run-Time System for OOC Dense Linear Algebra Operations

Our approach employs a run-time system which executes the codes corresponding to the algorithms for the tiled QR factorization (see Figures 1–4) in two stages. In the first stage, the run-time does a symbolic execution of the code creating a *list of pending tasks*: every time an invocation to a routine that corresponds to one of the building kernels is detected, a new annotation is introduced in the list which identifies the task to be computed and the operands (tiles) which are involved. Upon completion, tasks appear in this list in the same order as they are encountered in the codes.

During the second stage, the real I/O and computations occur. Here a *scout* thread and a *worker* tread collaborate to perform I/O and computations. The scout thread extracts the next task from the pending list, bringing in-core the tiles involved by the task. To hide memory latency, this thread uses a *software cache* with capacity to store a few tiles and the corresponding replacement policies/mechanisms. Once data is in-core for a given operation, the scout thread moves the task to the *list of ready tasks*, which only contains operations with all data in main memory.

The worker thread extracts the tasks from the list of ready tasks in order, one task at a time, and executes the corresponding operation using as many threads as cores are available in the system. Thus, in our current approach, parallelism is only exploited within the computations of a single task. However, in case there are two or more tasks in the ready list with all its input data updated (i.e., no previous task in the list will overwrite them), we could have also split the set of computational threads to execute them in parallel.

We have previously used the idea of a run-time/two-stage execution, with an initial symbolic analysis of the code, to extract more parallelism and improve the scalability in multi-core systems; see, e.g., [15]. The purpose is different here. In particular, we employ the run-time to overlap I/O done by the scout thread and computation performed by the worker thread without using asynchronous I/O routines. The fact that the list of tasks (or part of it) is known in advance, is equivalent in practice to having a perfect prefetch engine as the tiles that will be needed in future operations (tasks) are known *a priori*.

To ensure the correct operation of the worker thread, the replacement policy for the software cache only selects tiles which are not involved in any operation in the list of ready tasks. If there are no candidates which satisfy this criterion, the scout thread blocks till the execution of new tasks is completed.

The bottom line is that proceeding in this manner, the scout thread completely hides I/O from the library developer so that no changes are necessary in the in-core codes. Also, both threads conspire to hide asynchronous I/O from the developer.

The concurrent execution of the scout and the worker thread implies a first level of parallelism in the system. The use of multiple threads to perform the actual computations on the data reflects an additional, nested level of parallelism.

4 Experimental Results

The target architecture for the experiments is a workstation with two Intel Xeon QuadCore E5405 processors (8 cores) at 2.0 GHz with 8 GBytes of DDR2 RAM (peak performance is 128 billions of flops per second or GFLOPS). The Intel 5400 chipset provides an I/O interface with a peak bandwidth of 1.5 Gbits/second. The disk is a SATA-I with a total capacity of 160 Gbytes. All experiments were performed using MKL 10.0.1 and single precision.

We first evaluate the performance of the multi-threaded in-core building kernels, operating on matrices (tiles) of size $t \times t$ (BK1 and BK2) and $2t \times t$ (BK3 and BK4). Table 1 reports the number of invocations to each building kernel during the factorization of a square matrix of order n using the algorithms for the tiled (left-looking) QR factorization in Figures 1 and 2, with tile size t and $k = n/t$. There we also report the performance of the basic building kernels, measured in terms of GFLOPS, using the counts of $4t^3/3$, $2n^3$, $2t^3$ and $4t^3$ flops[1] for BK1, BK2, BK3 and BK4, respectively. In this analysis we set t=5,120 which, in a separate study, was found to be an fair value for the tiled OOC algorithm. Clearly, the performance of BK4 will determine the efficiency of the overall OOC algorithm and, therefore, we tried to optimize this building kernel carefully. In particular, the table shows the results of two parallelization strategies for BK4: one with all parallelism being extracted from calls to multi-threaded BLAS and an alternative one with a parallelization by blocks of columns (see subsection 2.3). In the experiments we also found that the block size b=64 (see Figures 3 and 4) was optimal in most cases.

Our second experiment evaluates the performance of several in-core and OOC routines for the QR factorization:

In-core MKL: The (in-core) multi-threaded implementation of the QR factorization in MKL 10.0.1.

In-core LAPACK: The (in-core) LAPACK legacy code linked with the multi-threaded BLAS in MKL 10.0.1.

[1] We emphasize that we are only counting "useful computations" and do not count additional operations that are artificially introduced in order to expose better parallelism and/or improve the use of matrix-multiplications.

Table 1. Number of invocations and performance of different implementations of the multi-threaded in-core building kernels operating on tiles of size t=5,120

Building kernel	#invocations	Performance	
		GFLOPS	Parallelization strategy
BK1	k	45	Multi-threaded BLAS
BK2	$k^3/2$	67	Multi-threaded BLAS
BK3	$k^2/2$	39	Multi-threaded BLAS
BK4	$k^3/3$	53	Multi-threaded BLAS
		65	Column parallel

OOC explicit+intra-tile column parallel: OOC implementation with explicit invocations to I/O routines (see subsection 3.1). The tile size was set in this routine to t=5,120. Although, in theory, using a large tile size makes the cost of moving data between disk and main memory ($O(t^2)$ disk accesses) negligible compared with the computational cost ($O(t^3)$ flops), in our experimentation we found out a large drop in the disk transfer rate for tiles of dimension larger than $5,120 \times 5,120$. (We experienced similar behaviour for several other current desktop systems equipped with different disks.) In this routine, parallelism is extracted implicitly by linking to a multi-threaded BLAS (see subsection 2.3).

OOC cache+column parallel: OOC implementation with a software cache in place to reduce the number of I/O transfers (see Section 3.2). The cache occupies 6 GBytes in RAM, and is organized as a k-way set associative with t=5,120 and $k = n/t$. LRU was implemented as the replacement policy. Parallelism is extracted explicitly by implementing column parallel variants for the building kernels BK3 and BK4 (see subsection 2.3). I/O is synchronous.

OOC cache+I/O overlap+column parallel: OOC implementation that includes all mechanisms of the previous routine plus overlap of I/O and computation (see Section 3.2).

Figure 5 reports the performance of these routines measured in terms of GFLOPS, with the usual count of $4n^3/3$ flops for the QR factorization of a square matrix of order n. (Experiments with nonsquare matrices offered similar results. We also note here that the tiled QR factorization performs a larger number of flops than the in-core factorization routines in MKL and LAPACK; see [12] for details.)

The results in the figure show a practical peak performance for the in-core QR factorization that is slightly over 74 GFLOPS. Using routine **OOC column parallel+cache+I/O overlap**, the tiled QR factorization which operates on OOC data yields a performance that is around 65 GFLOPS and, therefore, close to that of the in-core algorithm. As expected, the performance of the tiled OOC algorithm matches that of the building kernel BK4, in a practical demonstration that the disk latency is mostly hidden. The performance results reveal that the GFLOPS rate for the OOC algorithm is maintained as the problem size is increased, thus confirming the scalability of the solution.

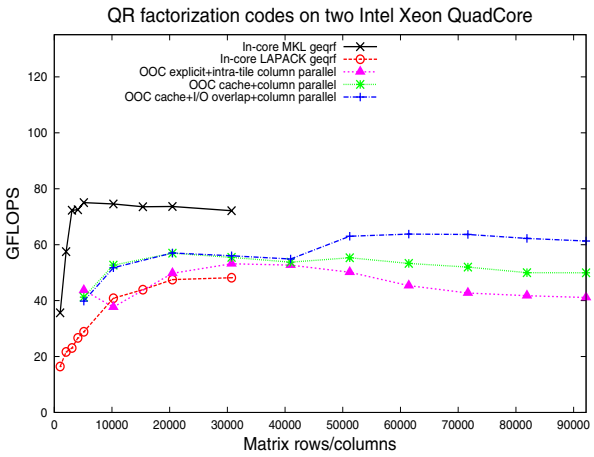

Fig. 5. Performance of the QR factorization codes on a multi-core processor

To asses the benefits contributed by the use of the software cache, Table 2 shows the number of tiles read from or written to disk for the OOC routine with explicit I/O calls and the ones that employ a software cache. The results demonstrate that the software cache greatly reduces the number of tiles that are transferred between the disk and RAM.

Table 3 reports the execution time required to compute the tiled QR factorization for routine **OOC column parallel+cache+I/O overlap** as well as the amount of memory that is needed to store the full dense matrix. The results show that what would have been considered a very large problem only a few years ago was solved in less than 5 hours. Solving a linear least squares problem or a linear system (with a few right-hand sides) once the orthogonal matrix and the triangular factor are available is computationally much less expensive

Table 2. Reduction in the number of disk accesses (in terms of number of tiles reads/written) attained by the use of a software cache

Matrix size (square)	**OOC explicit**		**OOC cache**	
	#reads	#writes	#reads	#writes
51,200	1,045	715	237	100
92,160	5,985	4,047	2,170	324

Table 3. Execution time (in hours, minutes, and seconds) of the tiled QR factorization code using the OOC run-time and amount of memory needed to hold the matrix

Matrix size (square)	Time	MBytes
51,200	54min 20.9sec	10,000
92,160	4h 43min 12.3sec	40,000

than the factorization procedure and does not represent a challenge from the viewpoint of an OOC implementation.

5 Concluding Remarks

In this paper we have described and evaluated a run-time system which deals with I/O, overlaps computation and data movement from disk, and implements a software cache and the associated mechanisms. The case study is the QR factorization, a complex operation for which tiled in-core left-looking codes exist. The results obtained by combining these codes with the run-time system show that the extension of the memory hierarchy to include the disk can be made transparent to the library developer at the expense of little overhead. Our approach thus greatly increases the programmer's productivity without significantly hurting performance.

This work also demonstrates that multi-core processors are a cost-effective approach for the solution of many dense linear algebra operations of moderate scale. The implicit message is that, for these problems, it is not necessary to utilize expensive distributed-memory architectures and design complex message-passing algorithms, provided one is willing to wait longer.

Acknowledgements

The researchers at the Universidad Jaime I were supported by projects CICYT TIN2005-09037-C02-02, TIN2008-06570-C04-01 and FEDER, and P1B-2007-19 of the *Fundación Caixa-Castellón/Bancaixa* and UJI.

References

1. Baboulin, M., Giraud, L., Gratton, S., Langou, J.: Parallel tools for solving incremental dense least squares problems. application to space geodesy. Technical Report UT-CS-06-582; TR/PA/06/63, University of Tennessee; CERFACS (2006); To appear in J. of Algorithms and Computational Technology 3(1) (2009)
2. D'Azevedo, E.F., Dongarra, J.J.: The design and implementation of the parallel out-of-core scalapack LU, QR, and Cholesky factorization routines. LAPACK Working Note 118 CS-97-247, University of Tennessee, Knoxville (1997)
3. Reiley, W.C., van de Geijn, R.A.: POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin (1999)
4. Toledo, S.: A survey of out-of-core algorithms in numerical linear algebra. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science (1999)
5. Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.: Solving "large" dense matrix problems on multi-core processors. In: 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing – PDSEC 2009 (to appear, 2009)
6. Van Zee, F.G.: The complete reference (2008) (in preparation), `http://www.cs.utexas.edu/users/flame`

7. Baboulin, M.: Solving large dense linear least squares problems on parallel distributed computers. Application to the Earth's gravity field computation. Ph.D. dissertation, INPT, TH/PA/06/22 (2006)
8. Gunter, B.C.: Computational methods and processing strategies for estimating Earth's gravity field. PhD thesis, The University of Texas at Austin (2004)
9. Geng, P., Oden, J.T., van de Geijn, R.: Massively parallel computation for acoustical scattering problems using boundary element methods. Journal of Sound and Vibration 191(1), 145–165 (1996)
10. Schafer, N., Serban, R., Negrut, D.: Implicit integration in molecular dynamics simulation. In: ASME International Mechanical Engineering Congress & Exposition (2008) (IMECE2008-66438)
11. Zhang, Y., Sarkar, T.K., van de Geijn, R.A., Taylor, M.C.: Parallel MoM using higher order basis function and PLAPACK in-core and out-of-core solvers for challenging EM simulations. In: IEEE AP-S & USNC/URSI Symposium (2008)
12. Gunter, B.C., van de Geijn, R.A.: Parallel out-of-core computation and updating the QR factorization. ACM Transactions on Mathematical Software 31(1), 60–78 (2005)
13. Watkins, D.S.: Fundamentals of Matrix Computations, 2nd edn. John Wiley & Sons, Inc., New York (2002)
14. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software 16(1), 1–17 (1990)
15. Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R., Zee, F.V., Chan, E.: Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Transactions on Mathematical Software (2008) (to appear), FLAME Working Note #32, `http://www.cs.utexas.edu/users/flame/`
16. Anderson, E., Bai, Z., Demmel, J., Dongarra, J.E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A.E., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992)