Implementing a systolic algorithm for QR factorization on multicore clusters with PaRSEC

Guillaume Aupy, Mathieu Faverge, Yves Robert, Jakub Kurzak, Piotr Luszczek, Jack Dongarra

Implementing a Systolic Algorithm for QR Factorization on Multicore Clusters with PaRSEC

Guillaume Aupy[1], Mathieu Faverge[2], Yves Robert[1,3],
Jakub Kurzak[3], Piotr Luszczek[3], and Jack Dongarra[3]

[1] LIP laboratory - École Normale Supérieure de Lyon,
CNRS, INRIA, UCBL, and Université de Lyon, France
[2] Laboratoire LaBRI, IPB ENSEIRB-MatMeca, Bordeaux, France
[3] University of Tennessee Knoxville, USA

Abstract This article introduces a new systolic algorithm for QR factorization, and its implementation on a supercomputing cluster of multicore nodes. The algorithm targets a virtual 3D-array and requires only local communications. The implementation of the algorithm uses threads at the node level, and MPI for inter-node communications. The complexity of the implementation is addressed with the PaRSEC software, which takes as input a parametrized dependence graph, which is derived from the algorithm, and only requires the user to decide, at the high-level, the allocation of tasks to nodes. We show that the new algorithm exhibits competitive performance with state-of-the-art QR routines on a supercomputer called Kraken, which shows that high-level programming environments, such as PaRSEC, provide a viable alternative to enhance the production of quality software on complex and hierarchical architectures.

1 Introduction

Future exascale machines are projected be massively parallel architectures, with 10^5 to 10^6 nodes, each node itself being equipped with 10^3 to 10^4 cores. At the node level, the architecture is a shared-memory machine, running many parallel threads on the cores. At the machine level, the architecture is a distributed-memory machine. This additional level of hierarchy dramatically complicates the design of new versions of the standard factorization algorithms, that are central to many scientific applications. In particular, the performance of numerical linear algebra kernels are at the heart of many grand challenge applications, and it is of key importance to provide highly efficient implementations of these kernels to leverage the potential of exascale platforms.

This article introduces a new systolic algorithm for QR factorization on clusters of multicore nodes. The main motivation is to enhance the state-of-the-art algorithms, that use tile kernels and several elimination domains per panel, that enforce the inter-node communication between neighbors only. The systolic algorithm targets a 3D torus, which is the underlying interconnection topology of the contemporary and up-coming HPC systems. For instance, Blue Gene/L is a 3D torus of size $64 \times 32 \times 32$ [1], Kraken, a Cray XT 5, is a 3D torus of size $25 \times 16 \times 24$ [2]. In addition, the Cray XT3 and XT4 also are architectures based on a 3D torus [3]. Our systolic algorithm uses a two-level allocation of tile rows to the faces of the torus cube, in order to restrict the reduction tree for each panel to only local communication.

Implementing such a complex algorithm with low-level primitives would require non-trivial and error-prone programming effort. However, using the PaRSEC software [4] has enabled us to implement, validate, and evaluate the algorithm on Kraken, within a few weeks of development. Although we use a high-level environment, we report competitive performance results with state-of-the-art QR routines, thereby showing that PaRSEC provides a viable alternative to enhance the production of quality software prototypes on complex hierarchical architectures.

The rest of the paper is organized as follows. Section 2 provides background information on QR factorization algorithms, and surveys state-of-the-art algorithms in the literature. Section 4 presents the new systolic algorithm, while Section 5 provides additional details of its implementation using the PaRSEC software. Section 6 presents experimental results obtained on the Kraken supercomputer. Finally, we close with concluding remarks in Section 7.

2 Background

2.1 Tiled-QR Factorization

The general shape of a tiled QR algorithm for a tiled matrix of $m \times n$ tiles, whose rows and columns are indexed from 0, is given in Algorithm 1. Here i and k are tile indices, and operate on square $b \times b$ tiles, where b is the block size. Thus, the actual size of the matrix is $M \times N$, with $M \equiv m \times b$ and $N \equiv n \times b$. The first loop index, k, is the panel index, and $elim(i, \mathbf{CurPiv}(i,k), k)$ is an orthogonal transformation that combines rows i and $\mathbf{CurPiv}(i,k)$ to zero out the tile in position (i,k). Each elimination $elim(i, \mathbf{CurPiv}(i,k), k)$ consists of two sub-steps: (i) first in column k, tile (i,k) is zeroed out (or eliminated) by tile $(\mathbf{CurPiv}(i,k), k)$, which is called the pivot; and (ii) in each subsequent column $j > k$, tiles (i, j) and $(\mathbf{CurPiv}(i,k), j)$ are updated; all these updates are independent and can be triggered as soon as the elimination is completed. The algorithm is entirely characterized by its *elimination list*, which is the ordered list of all the eliminations $elim(i, \mathbf{CurPiv}(i,k), k)$ that are executed.

Algorithm 1: Generic QR algorithm.

```
begin
    for k = 0 to min(m,n) − 1 do
        for i = k + 1 to m − 1 do
            elim(i, CurPiv(i,k), k)
```

Algorithm 2: Elimination $elim(i, \mathbf{CurPiv}(i,k), k)$.

```
begin
    (a) With TS (Triangle on top of square) kernels
    GEQRT(CurPiv(i,k), k)
    TSQRT(i, CurPiv(i,k), k)
    for j = k + 1 to n − 1 do
        UNMQR(CurPiv(i,k), k, j)
        TSMQR(i, CurPiv(i,k), k, j)

    (b) With TT (Triangle on top of triangle) kernels
    GEQRT(CurPiv(i,k), k)
    GEQRT(i, k)
    for j = k + 1 to n − 1 do
        UNMQR(CurPiv(i,k), k, j)
        UNMQR(i, k, j)
    TTQRT(i, CurPiv(i,k), k)
    for j = k + 1 to n − 1 do
        TTMQR(i, CurPiv(i,k), k, j)
```

To implement an orthogonal transformation $elim(i, \mathbf{CurPiv}(i,k), k)$, we can use either *TT* kernels or *TS* kernels, as shown in Algorithm 2. *TT* kernels are needed to allow for several eliminator tiles in a given column, but are less efficient than *TS* kernels. More detailed information on the various kernels is provided elsewhere [5]. In a nutshell, a

tile can have three states: square, triangle, and zero. Transitions between these states occur through the following kernels:

- *GEQRT* is the transformation of one square tile to a triangle tile,
- $TSQRT(i, \mathbf{CurPiv}(i,k), k)$ is the transformation of a square tile (tile i) into a zero tile, using a triangle tile (tile $\mathbf{CurPiv}(i,k)$) at step k,
- $TTQRT(i, \mathbf{CurPiv}(i,k), k)$ is the transformation of a triangle tile (tile i) into a zero tile using a triangle tile (tile $\mathbf{CurPiv}(i,k)$) at step k.

3 Related Work

While the advent of multi-core machines is somewhat recent, there is a long line of papers related to tiled QR factorization. Tiled QR algorithms have first been introduced in Buttari et al. [6,7] and Quintana-Ortí et al. [8] for shared-memory (multi-core) environments, with an initial focus on square matrices. The sequence of eliminations presented in these papers is analogous to the prior work [9], and corresponds to reducing each matrix panel with a flat tree: in each column, there is a unique eliminator, namely the diagonal tile.

The introduction of several eliminators in a given column has a long history [9,10,11,12,13,14]. For shared-memory (multi-core) environments, recent work advocates the use of domain trees [15] to expose more parallelism with several eliminators while enforcing some locality within domains. A recent paper [16] introduces tiled versions of the Greedy algorithm [17] and the Fibonacci scheme [10], it shows that these algorithms are asymptotically optimal.

There are recent efforts for distributed-memory environments. The algorithm of [18] uses a hierarchical approach: for each matrix panel, it combines two levels of reduction trees. First, several local binary trees are applied in parallel, one within each node, and then a global binary tree is applied for the final reduction across nodes. Yet another implementation [19] also uses a hierarchical approach, and it also uses a 1D block distribution. The main difference is that the first level of reduction is performed with a flat tree within each node. Note that the hierarchical algorithm (HQR) used previously [5] can be parametrized to implement this original algorithm [19] as well as a more efficient variant with cyclic layout. The HQR algorithm [5] is the reference algorithm for multi-level clusters: it provides a flexible approach, and allows one to use various elimination trees (Flat, Binary, Fibonacci or Greedy) at each level.

4 The SYSTOLIC-3D algorithm

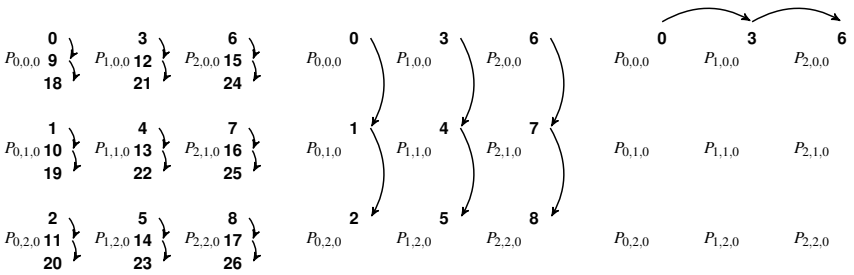
Platform and data layout – We first detail the 3D torus architecture. Within a $p \times q \times r$ 3D torus, processor $P_{a,b,c}$ has a direct communication link with processors $P_{a-1 \mod p,b,c}$, $P_{a+1 \mod p,b,c}$, with processors $P_{a,b-1 \mod q,c}$, $P_{a,b+1 \mod q,c}$, and with processors $P_{a,b,c-1 \mod r}$, $P_{a,b,c+1 \mod r}$. We have a $m \times n$ tile matrix. Tiles are mapped as follows: we use a two-level cyclic distribution for the rows (directions a and b in the torus) and a cyclic distribution for the columns (direction c in the torus). The mapping is defined formally as follows: proc P_{abc} is assigned all tiles $T_{t,s}$ such that $t \equiv b \mod q$, $\frac{t-b}{q} \equiv a \mod p$ and $s \equiv c \mod r$. We give an example of the two-level cyclic distribution for the rows in Figure 1a, for a matrix with 27 rows mapped onto a $3 \times 3 \times r$ torus.

General description – As stated in Section 2.1, a tiled-QR algorithm is entirely defined by the ordered list of eliminations. The algorithm eliminates the tiles using a hierarchical approach, using the 3D torus to minimize inter-processor communication contention. In order to do this, pivots should be propagated across physical links in the torus, and only to neighbor nodes, before each elimination. Figure 1 describes the elimination of the first column of the matrix.

Consider a given step k of the factorization. The k-th tile column is distributed across a face of the cube, i.e. a square of $p \times q$ processors (those whose third index is $c_0 \equiv k \mod r$). Let dimension a be "horizontal" and dimension b be "vertical". There are three levels of elimination in the algorithm:

1. The first level of elimination corresponds to local tiles and uses *TS* kernels. There are pq pivots in this step, one for each processor in the square, and they correspond to rows numbered $k, k+1, \cdots, k+pq-1$. These pivots are used to eliminate all local tiles within each processor, hence they do not require any communication across the square. We use a flat tree reduction for this step, but other elimination trees could be chosen freely. This first elimination level is illustrated in Figure 1a when $k = 0$.
2. The second level of elimination consists of concurrent flat trees along the vertical dimension, and uses *TT* kernels (see Figure 1b). There are p pivots for this level, namely the k^{th} elements of rows $k, k+q, \cdots, k+q(q-1)$. Each of these pivots will sequentially eliminate the $q-1$ subsequent tiles, which are located in the corresponding grid column.
3. The third level of the elimination consists of a single flat tree along the horizontal dimension (see Figure 1c). There remains a single pivot, in row k, that sequentially eliminates with *TT* kernels the $q-1$ remaining tiles.

At the end of step k, row number k will have been routed through at most $p+q-2$ physical communication links. The communication pattern is the same for the other faces of the cube. The whole algorithm is summarized in Algorithm 3.



(a) 1st elimination level (b) 2nd elimination level (c) 3rd elimination level

Figure 1: Elimination in the first panel (panel 0) of all tiles below the diagonal (rows 1 to 27) on a 3×3 processor square (face 0 of the $3D$-torus).

Algorithm 3: The SYSTOLIC-3D algorithm

```
begin
    for k = 0 to min(m,n) − 1 do
        define i2 ← k mod d ;
        /* Local FlatTree */
        for l = k to k + d^2 − 1 do
            GEQRT(l,k);
            for x = l + d^2 to m − 1 by d^2 do
                TSQRT(x,l,k);
                for j = k + 1 to n − 1 do
                    UNMQR(l,k, j);
                    TSMQR(x,l,k, j);
        // Note that from now on, we
           do not need to use GEQRT
           anymore, all the
           remaining tiles are
           triangles.
        /* Vertical FlatTree */
        for l2 = i2 + 1 to d − 1 do
            TTQRT_V(k + (l2 − i2),k,k);
            for j = k + 1 to n − 1 do
                TTMQR_V(k + (l2 −
                i2),k,k, j);
        for l2 = 0 to i2 − 1 do
            TTQRT_V(k + d^2 + (l2 − i2),k,k);
            for j = k + 1 to n − 1 do
                TTMQR_V(k + d^2 + (l2 −
                i2),k,k, j);
        for l = k + d to k + d^2 − 1 by d do
            for x = l + 1 to (l − i2) + d − 1 do
                TTQRT_V(x,l,k);
                for j = k + 1 to n − 1 do
                    TTMQR_V(x,l,k, j);
            for x = l − i2 to l − 1 do
                TTQRT_V(x,l,k);
                for j = k + 1 to n − 1 do
                    TTMQR_V(x,l,k, j);
        /* Horizontal FlatTree */
        for x = k + d to k + d^2 − 1 by d do
            TTQRT_H(x,k,k);
            for j = k + 1 to n − 1 do
                TTMQR_H(x,k,k, j);
```

5 Implementation with PaRSEC

This section details the implementation of the SYSTOLIC-3D algorithm using PaRSEC. With an infinite number of resources, the scheduling could follow a greedy heuristic: the execution would progress as fast as possible.

The elimination list of the algorithm is the composition of the reduction trees at all of the different levels. All eliminators are known before the execution. Each component of an elimination is triggered as soon as possible, i.e., as soon as all dependencies are satisfied: first we have the elimination of the tile, and then the updates in the trailing panels. Note that the overall elimination scheme is complex, and mixes the elimination of tiles at all levels. However, with a fixed number of resources, it is necessary to decide an order of execution of the tasks, hence to schedule them: this is achieved through the PaRSEC environment.

5.1 Introduction to PaRSEC

PaRSEC is a high-performance fully-distributed scheduling environment for systems of micro-tasks. It takes as input a problem-size-independent, symbolic representation of a Direct Acyclic Graph (DAG) of tasks, and schedules them at runtime on a distributed parallel machine of multi-cores. Data movements are expressed implicitly by the data flow between the tasks in the DAG representation. The runtime engine is then responsible for actually moving the data from one machine (node) to another, using an underlying communication mechanism such as MPI. A full description of PaRSEC, and the implementation of classical linear algebra factorizations in this environment, is provided elsewhere [20,21].

To implement any QR algorithm in PaRSEC, it is sufficient to give an abstract representation of all the tasks (eliminations and updates) that constitute the QR factorization, and how data flows from one task to the other. Since a tiled QR algorithm is fully determined by its elimination list, it suffices to provide a function, that the runtime engine

is capable of evaluating, and that computes this elimination list. The PaRSEC object obtained in this way is generic: when instantiating a PaRSEC QR factorization, the user simply gives the size of the platform ($p \times q \times r$), defining a new DAG at each instantiation. Note that this DAG is not fully generated: PaRSEC keeps only a parametric representation of the DAG in memory, and interprets symbolic expressions at runtime to explicitly represent only the ready tasks at any given time. This technique is similar to the Parametrized Tasks Graphs [20], and to HQR [5].

At runtime, task executions trigger data movements, and create new ready tasks, following the dependencies defined by the elimination list. Tasks that are ready to compute are scheduled according to a data-reuse heuristic: each core will try to execute close successors of the last task it ran under the assumption, that these tasks require data that was recently touched by the completed task. This policy is tuned by the user through a priority function: among the ready tasks for a given core, the choice is done by following a hint from this function. To balance the load between cores, tasks on the same node in the algorithm (residing on the same shared memory machine) are shared between the computing cores, and a NUMA-aware job stealing policy is implemented. The user is responsible for defining the affinity between data and tasks, and to distribute the data between the computing nodes. Thus, he defines which tasks should execute on which node, and he is responsible for this level of load balancing. In our case, the data distribution is the data layout given in Section 4. Since all kernel operations modify a single tile (or a tile and its reflectors, which are distributed in the same way), we chose the strategy "owner computes" for the tasks: tasks' affinity is set to the node that owns the data that is going to be modified, and the input data might need to be transferred from other nodes.

5.2 Implementation Details

The implementation of SYSTOLIC-3D in PaRSEC involves limited effort compared with other software strategies, that we are aware of. We only implemented a few functions that are used by PaRSEC to generate the dependency graph. They depend on the current elimination step k as follows:

1. **CurPiv**(i,k), returns the pivot to use for the row i at step k;
2. **NextPiv**$(\text{pivot},k,\text{start})$, returns the next row which will use the row "pivot" as a pivot in step k after it has been used by row "start";
3. **PrevPiv**$(\text{pivot},k,\text{start})$, returns the previous row which used the row "pivot" as a pivot in step k before it has been used by row "start";

We have decomposed each one of these functions in two sub-functions: (i) a low-level function, which takes all the *TS* operations into account, and which calls the *local FlatTree* because operations are local to each node; and (ii) a high-level function, which takes all the *TT* operations into account, and where the pivot will "move" across the architecture. Using these functions, PaRSEC is able to construct a dependency graph between the different tiles in order to run the algorithm as efficiently as possible.

6 Experimental Evaluation

In this section, we report experimental results obtained on Kraken. We compare the SYSTOLIC-3D algorithm with a number of competing implementations such as vendor library routines and recent algorithms from literature.

6.1 Experimental Setup

All runs were done on the Kraken supercomputer at the National Institute for Computational Science [2]. The Kraken machine is a Cray XT5 system operated by the University of Tennessee and located in Oak Ridge, Tennessee, U.S.A. The entire system consists of 9048 computing nodes. The experiments presented here used up to 1989 nodes, which is about one fifth of the machine. Each node contains two 2.6 GHz six-core AMD Opteron (Istanbul) processors, 16 GB of memory and the Cray SeaStar2+ interconnect.

We have compared SYSTOLIC-3D with several state-of-the-art algorithms, using three matrix sizes: (i) small matrices, of size $M = N = 10,368$; (ii) medium matrices, of size $M = N = 20,736$; and large matrices, of size $M = N = 41,472$. Here is the list of the algorithms used for comparison:

- SYSTOLIC-3D is the algorithm described in this paper. Table 1 shows the 3D grid configuration (p,q,r) used for each matrix size ($M = N$) and for each total number of nodes T, where $T = p \times q \times r$. Note, that there is no guarantee, that the nodes assigned to the experiment will indeed form the desired 3D torus. They can be scattered across the machine. To the best of our knowledge, the only way to guarantee that assigned nodes indeed form a 3D torus would be to reserve the entire Kraken machine: something beyond our capabilities.
- HQR is the hierarchical QR factorization algorithm [5], which was also implemented using the PaRSEC software. We compare several variants of HQR, which all use the same FLAT-TREE low-level reduction tree, but which use different high level (or distributed) reduction trees [5]:
  1. HQR-FLAT uses the FLATTREE reduction;
  2. HQR-FIBO uses the FIBONACCI reduction;
  3. HQR-BINARY uses the BINARYTREE reduction;
  4. HQR-GREEDY uses the GREEDY reduction.

  Because HQR uses a 2D-processor grid, we use T nodes configured as a $(pq) \times r$ 2D grid.
- SYSTOLIC-2D is a variant of SYSTOLIC-3D where q is set to 1 and then runs on a 2D grid of size $(pq) \times r$. We introduced it for the sake of comparison with the HQR variants – SYSTOLIC-2D can be viewed as yet another HQR variant with a new high-level reduction tree.

We compare all the previous algorithms that were implemented with PaRSEC with the following algorithms from the literature [22] on the very same hardware:

- SYSTOLIC-1D is the virtual systolic array decomposition [22]. As its name indicates, it targets a 1D-linear array of processors. Note that SYSTOLIC-1D has been implemented using a hand-written communication engine over MPI – not PaRSEC.
- HPL 4/3 N^3 is the virtual performance of the High Performance Linpack LU factorization using the flops count of QR: $O(\frac{4}{3}N^3)$.
- LIBSCI QR is the QR factorization from ScaLAPACK used in the Cray Scientific Library.
- HPL 2/3 N^3 is the High Performance Linpack LU factorization with the actual flops count of LU: $O(\frac{2}{3}N^3)$.

$M=N=10,368$				$M=N=20,736$				$M=N=41,472$			
T	$p\times q\times r$	T	$p\times q\times r$	T	$p\times q\times r$	T	$p\times q\times r$	T	$p\times q\times r$	T	$p\times q\times r$
4	$2\times2\times1$	52	$6\times3\times3$	16	$4\times2\times2$	210	$6\times5\times7$	64	$4\times4\times4$	840	$10\times6\times14$
12	$3\times2\times2$	80	$5\times4\times4$	48	$4\times4\times3$	320	$8\times5\times8$	192	$8\times4\times6$	1232	$11\times7\times16$
18	$3\times3\times2$	96	$6\times4\times4$	80	$5\times4\times4$	405	$9\times5\times9$	336	$7\times6\times8$	1632	$12\times8\times17$
28	$5\times2\times3$	128	$8\times4\times4$	112	$6\times4\times5$	486	$9\times6\times9$	480	$8\times6\times10$	1989	$13\times9\times17$
42	$7\times2\times3$			168	$7\times4\times6$			648	$9\times6\times12$		

Table 1: Partition of the nodes into a 3D torus for each matrix size and each total number of nodes T.

For each set of results, we ran the different algorithms ten times, and we take the average performance over all these executions.

Our decision to include performance numbers for HPL's LU factorization might seem controversial due to the fundamental differences between the LU and QR factorization algorithms including their numerical properties, operation-count, and the runtime behavior. However, from the end-user perspective, both LU and QR solve a system of linear equations, both are backward stable, and only an explicitly stated rule [23] prohibits QR from scoring the entrants to the TOP500 ranking. With this in mind, we include results for the LU factorization, and include the case when we pretend that LU performs as many Flops as QR: $O(\frac{4}{3}N^3)$ (this may be simply treated as time-to-run comparison) as well as the case where we report the actual performance rate based on the actual amount of floating point operations LU: $O(\frac{2}{3}N^3)$.

6.2 Performance Results

The first observation is that PaRSEC-based algorithms (SYSTOLIC-3D, SYSTOLIC-2D and all HQR variants) always perform better than LIBSCI QR and HPL 4/3 N^3, the QR factorization algorithms from Kraken's standard software stack. This observation holds for all matrix sizes, and this makes our main point: owing to the PaRSEC system, we have been able to experiment with a variety of complex, hierarchical algorithms, without paying the price of lengthy development effort.

Note, how SYSTOLIC-3D , HQR variants, and SYSTOLIC-1D compare with each other. SYSTOLIC-3D has approximatively the same efficiency as HQR-BINARY and HQR-GREEDY on all matrix sizes. For matrices of size $M=N=10,368$, HQR-FLAT is 54% more efficient on 1536 cores ($\approx$ 1700 GFlops compared to $\approx$ 1100 GFlops), and SYSTOLIC-1D ($\approx$ 1900) GFlops is 73% more efficient. This difference increases with the size of the matrix: for $M=N=41,472$, HQR-FLAT reaches $\approx$ 16000 GFlops, and SYSTOLIC-1D reaches $\approx$ 21000 GFlops on 23868 cores, where SYSTOLIC-3D is bound by $\approx$ 10600 GFlops (half the performance of SYSTOLIC-1D). However, for $M=N=41,472$, and with a small number of cores, SYSTOLIC-3D performs better than SYSTOLIC-1D.

As mentioned earlier, it is infeasible to guarantee, that the assignment of Kraken nodes from our batch queue submissions form a true 3D torus. We expected the constraints to be less stringent when using a 2D torus, and this turns out quite true: SYSTOLIC-2D, the implementation of SYSTOLIC-3D on a 2D torus, performs very well for all matrix sizes, and is the best algorithm for larger matrices of size $M=N=41,472$.

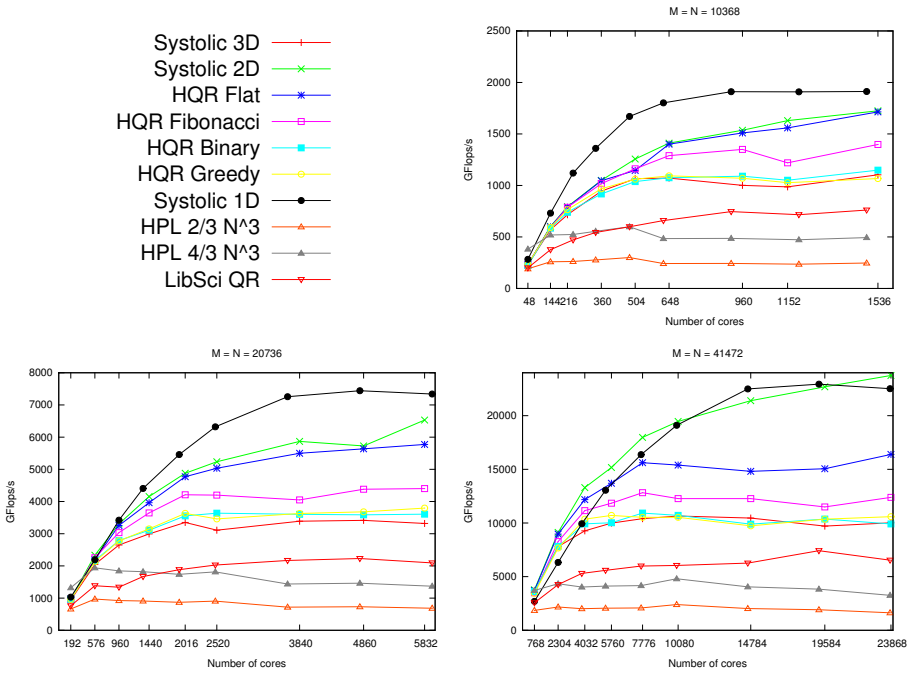


Figure 2: Performance of the various algorithms for different problem sizes.

7 Conclusion

In this article, we have presented a systolic QR factorization algorithm, SYSTOLIC-3D, which aims to minimize the amount of communication in the reduction trees. We have shown that mapping this systolic algorithm onto a 3D torus leads to a competitive factorization kernel with strong scaling capabilities. As of today, the main limitation to fully validate the experiments is the lack of possibility to reserve an actual 3D torus architecture on the Kraken supercomputer. Still, the performance of the new algorithm, together with its 2D counterpart are very encouraging. Both versions dramatically outperform LIBSCI QR and HPL 4/3 N^3, the vendor QR factorization implementations on Kraken, and also HPL 2/3 N^3, the widely-used LU factorization routine (despite its favorable flop count). This last observation fully demonstrates the usefulness of the PaRSEC system, which has enabled us to experiment with complex, hierarchical QR algorithms, without paying the price of lengthy and complex development effort of distributed memory software engineering.

Acknowledgements Yves Robert is with Institut Universitaire de France. This work was supported in part by the ANR *RESCUE* project. It was also supported in part by grant #SHF-1117062: "Parallel Unified Linear algebra with Systolic ARrays (PULSAR)" from the *National Science Foundation* (NSF). The authors would like to thank the *National Institute for Computational Sciences* (NICS) for a generous time allocation on the Kraken supercomputer. Finally, the authors would also like to thank Thomas Hérault for his help with PaRSEC.

References

1. Adiga, N.R., Almási, G., Almasi, G.S., Aridor, Y., Barik, R., Beece, D., Bellofatto, R., et al.: An overview of the BlueGene/L supercomputer. In: Supercomputing Conference. (2002)
2. The National Institute for Computational Sciences: Kraken machine size. http://www.nics.tennessee.edu/computing-resources/machine_size
3. Bhatele, A., Kale, L.V.: Application-specific topology-aware mapping for three dimensional topologies. In: IPDPS. (2008)
4. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing. Parallel Computing **38**(1) (2012) 37–51
5. Dongarra, J., Faverge, M., Herault, T., Jacquelin, M., Langou, J., Robert, Y.: Hierarchical QR factorization algorithms for multi-core clusters. Parallel Computing (2013)
6. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: Parallel tiled QR factorization for multicore architectures. Concurrency: Practice and Experience **20**(13) (2008) 1573–1590
7. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing **35** (2009) 38–53
8. Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.A., Zee, F.G.V., Chan, E.: Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Transactions on Mathematical Software **36**(3) (2009)
9. Sameh, A., Kuck, D.: On stable parallel linear systems solvers. J. ACM **25** (1978) 81–91
10. Modi, J., Clarke, M.: An alternative Givens ordering. Numerische Mathematik **43** (1984) 83–90
11. Pothen, A., Raghavan, P.: Distributed orthogonal factorization: Givens and Householder algorithms. SIAM J. Scientific Computing **10**(6) (1989) 1113–1134
12. da Cunha, R., Becker, D., Patterson, J.: New parallel (rank-revealing) QR factorization algorithms. In: Euro-Par. (2002)
13. Demmel, J.W., Grigori, L., Hoemmen, M., Langou, J.: Communication-avoiding parallel and sequential QR and LU factorizations: theory and practice. Technical Report 204, LAPACK Working Note (2008)
14. Langou, J.: Computing the R of the QR factorization of tall and skinny matrices using MPI_Reduce. Technical Report 1002.4250, arXiv (2010)
15. Hadri, B., Ltaief, H., Agullo, E., Dongarra, J.: Tile QR factorization with parallel panel processing for multicore architectures. In: IPDPS. (2010)
16. Bouwmeester, H., Jacquelin, M., Langou, J., Robert, Y.: Tiled QR factorization algorithms. In: SC, ACM/ IEEE Computer Society Press (2011)
17. Cosnard, M., Robert, Y.: Complexity of parallel QR factorization. Journal of the A.C.M. **33**(4) (1986) 712–723
18. Agullo, E., Coti, C., Dongarra, J., Herault, T., Langou, J.: QR factorization of tall and skinny matrices in a grid computing environment. In: IPDPS. (2010)
19. Song, F., Ltaief, H., Hadri, B., Dongarra, J.: Scalable tile communication-avoiding QR factorization on multicore cluster systems. In: SC, ACM/IEEE Computer Society Press (2010)
20. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing. In: HIPS. (2011)
21. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., et al.: Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In: PDSEC. (2011)
22. Kurzak, J., Luszczek, P., Gates, M., Yamazaki, I., Dongarra, J.: Virtual systolic array for QR decomposition. In: IPDPS, IEEE Computer Society Press (2013)
23. Dongarra, J.J., Luszczek, P., Petitet, A.: The LINPACK benchmark: Past, present, and future. Concurrency and Computation: Practice and Experience **15** (2003) 1–18