2016 IEEE 22nd International Conference on Parallel and Distributed Systems

suCAQR: A Simplified Communication-Avoiding QR Factorization Solver Using the TBLAS Framework

Weijian Zheng and Fengguang Song
Department of Computer Science
Indiana University Purdue University Indianapolis
Email: {wz26, fgsong}@iupui.edu

Lan Lin
Department of Computer Science
Ball State University
Email: llin4@bsu.edu

Zizhong Chen
Department of Computer Science
University of California at Riverside
Email: chen@cs.ucr.edu

***Abstract*—The scope of this paper is to design and implement a scalable QR factorization solver that can deliver the fastest performance for tall and skinny matrices and square matrices on modern supercomputers. The new solver, named *scalable universal communication-avoiding QR factorization (suCAQR)*, introduces a simplified and tuning-less way to realize the communication-avoiding QR factorization algorithm to support matrices of any shapes. The software design includes a mixed usage of physical and logical data layouts, a simplified method of dynamic-root binary-tree reduction, and a dynamic dataflow implementation. Compared with the existing communication avoiding QR factorization implementations, *suCAQR* has the benefits of being simpler, more general, and more efficient. By balancing the degree of parallelism and the proportion of faster computational kernels, it is able to achieve scalable performance on clusters of multicore nodes. The software essentially combines the strengths of both synchronization-reducing approach and communication-avoiding approach to achieve high performance. Based on the experimental results using 1,024 CPU cores, *suCAQR* is faster than DPLASMA by up to 30%, and faster than ScaLAPACK by up to 30 times.**

***Index Terms*—high performance computing; computational science application; performance analysis and optimization; dataflow runtime system.**

I. Introduction

QR factorization is a fundamental computational kernel for many important scientific, engineering, and big data analytics applications. It has been applied to solving linear systems, least-squares problems, linear regression problems, and the production function modeling, as well as assessing the conditioning of these problems [1]–[3]. The QR factorization of a matrix A of dimension $m \times n$ ($m \geq n$) takes the form of $A = QR$, where Q is an $m \times m$ orthogonal matrix, and R (=$Q^T A$) is an upper triangular matrix with zeros below its diagonal. The design and implementation of more scalable QR factorizations will accelerate a wide range of domain applications.

The most widely used parallel algorithm to solve QR factorizations is the *block QR factorization algorithm* used by the de facto standard ScaLAPACK library [4], [5]. As displayed in Figure 1, matrix A is divided into a thin *panel* (i.e., $\frac{A_{11}}{A_{21}}$) of dimension $M \times$ NB, a block of rows A_{12}, and

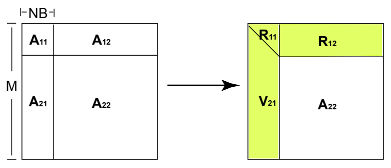


Fig. 1. The classic block QR factorization algorithm used by ScaLAPACK.

a *trailing submatrix* A_{22}. The block algorithm first applies *level 1* PBLAS subroutines to the panel ($\frac{A_{11}}{A_{21}}$), next it forms the triangular factor from the panel, finally it uses *level 3* PBLAS to factor A_{12} and update A_{22}. However, since the panel is computed one column after another—resulting in a large communication overhead and surface-to-volume ratio — the block algorithm does not scale well for *tall and skinny* matrices (i.e., matrices with much more rows than columns).

To reduce the large communication overhead with tall and skinny matrices, Demmel et al. then designed an algorithm called *Communication-Avoiding QR factorization* (*CAQR*) [6]–[8]. As explained in Figure 2, instead of computing a sequence of column-by-column operations, CAQR can perform a set of level 3 BLAS operations in the panel. Then it merges the output of the level 3 BLAS operations to get the final factor R. Not only does the algorithm convert level 1 BLAS to level 3 BLAS, but also it significantly reduces the number of communication messages. The original work of Demmel et al. [6] offered an estimated performance speedup of CAQR. Later, Song et al. [9] developed the first parallel implementation of CAQR, which is referred to as *distributed tiled CAQR* [9].

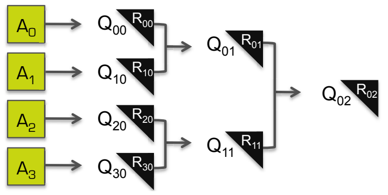


Fig. 2. Communication-Avoiding QR (CAQR) performs level 3 BLAS on the panel (i.e., A_0, A_1, A_2, A_3) followed by a parallel reduction.

1521-9097/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPADS.2016.142

While the distributed tiled CAQR demonstrates good strong scalability and weak scalability on different HPC systems, it only attains good performance for tall and skinny matrices. The issue is that its performance starts to decrease when the input matrix goes from skinny to square. The distributed tiled CAQR algorithm uses a static block distribution method to map blocks of rows to different processes. Although the block distribution method can minimize the inter-node communication, it makes certain processes turn into idle processes on non-skinny matrices. Another work of Hierarchical QR [10] investigates the effect of using a wide variety of reduction trees to improve the CAQR performance. However, determining the optimal tree and the optimal number of CAQR domains is dependent on the actual input and the number of CPUs, which are not known until executing the program. In this paper, we target designing a new solver that has a simpler design and implementation, does not require a lot of parameter tuning, and scales well on matrices of any shapes, ranging from extremely tall and skinny to square matrices.

To achieve the goal, we design a QR factorization solver called "Scalable Universal Communication-Avoiding QR Factorization" (*suCAQR*). The *suCAQR* solver consists of two types of computations: 1) process-local computation, and 2) global binary-tree reduction among all processes. It also mixes two types of data layout: 1) *physical block data layout*, and 2) *logical block-cyclic data layout*. The *suCAQR* employs a logic block cyclic data distribution method to achieve load balancing while applying a tiled QR factorization method to the physical data storage directly to avoid communication. It uses a dynamic-root binary-tree to merge results from different factorization domains (domain is a group of computational tasks that is independent of other groups). Also, *suCAQR* can maintain a good tradeoff between the degree of parallelism and the number of faster kernels for matrices of different shapes by changing the number of factorization domains. Furthermore, *suCAQR* uses a dynamic dataflow-driven TBLAS runtime [11] to enable efficient synchronization-reducing executions on a specific communication-avoiding algorithm.

We measure and evaluate the weak scalability for three different libraries: ScaLAPACK, DPLASMA [10], [12], and *suCAQR*. We also test different matrix shapes ranging from extremely tall and skinny to square using 1,024 cores. Based on the experimental results, *suCAQR* can outperform ScaLAPACK by 30 times and DPLASMA by 30%. Based on the runtime efficiency analysis, we also find that *suCAQR*'s total wall clock execution time is almost equal to its computation time (i.e., the CPU time taken by the computational kernels). This implies that there is no CPU idle time and the communication cost is totally hidden by computations (further discussed in Section IV).

To the best of our knowledge, this work makes the following contributions: 1) we design and implement a scalable *suCAQR* solver that works efficiently for both tall and skinny and square matrices; 2) it demonstrates that we can simplify the software implementation by using a unique reduction tree meanwhile we can provide scalable performance; and 3) detailed performance analysis reveals that the task-based dynamic scheduling approach is efficient in avoiding CPU idle cycles and hiding expensive communications, and should be adopted into more parallel software.

II. The suCAQR Algorithm

suCAQR essentially consists of three components: 1) the tile data storage, 2) the logical/physical data layouts, and 3) the *suCAQR* computation. After a matrix is distributed to different processes, the data can be viewed from two perspectives: a logical data layout and a physical data layout. The rest of the section describes the three components.

A. Tile Data Storage

suCAQR uses a tile data storage. We divide a matrix into $b \times b$ square blocks (also known as *tiles*). Each tile is stored in a contiguous memory block. If the tile size is tuned appropriately, we can fit multiple tiles in caches and maximize the cache hit rate. For instance, a 12×12 matrix can be represented as follows:

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}, \text{ where}$$

$a_{i,j}$ represents a tile of size 4×4. In the algorithm, every computational kernel takes as input several individual tiles (i.e., the basic unit) and computes new output.

Given an input matrix with $m_b \times n_b$ tiles, *suCAQR* allocates a 2-D $m_b \times n_b$ array of pointers, each of which points to a memory block that stores a distinct tile.

B. Usage of a Mixed Logical/Physical Layout

We use a block-cyclic distribution method to distribute m_b rows of tiles to P processes. The method first divides all the tile rows into groups of size B, then distributes the groups to different processes. With the block-cyclic distribution method, the i-th tile row is stored in process $\frac{i}{B} \mod P$. After the distribution, each process has $\frac{m_b}{P}$ rows of tiles.

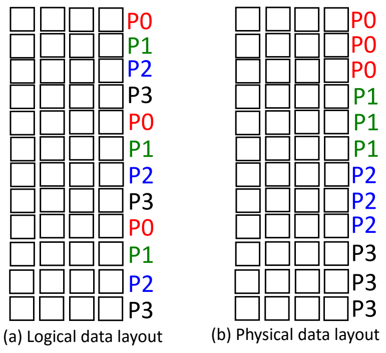


Fig. 3. After distributing a matrix with 12×4 tiles to four processes in a block cyclic way, we can view the same matrix from two perspectives: a logical data layout and a physical data layout. Both the logical data layout and the physical data layout are utilized by the *suCAQR* algorithm.

As shown in Figure 3. (a), twelve rows of a matrix are distributed to four processes (P_0, P_1, P_2, and P_3) in a block-cyclic way, where the group size $B = 1$. We call this data layout a logical (or "virtual") data layout. However, physically, each process stores three virtually interleaved rows together in its local memory. Figure 3. (b) shows the corresponding physical data layout where each process owns three rows. Note that *suCAQR* uses both physical layout and logical layout. While most parallel solvers only need to use the physical data layout, *suCAQR* must follow the logical cyclic pattern (e.g., P_0–P_3, P_0–P_3, …) such that the trailing submatrices consistently involve all the processes to achieve load balancing. It might appear that only one layout is needed here, but two types of computations working on two different layouts require that we deal with both of them.

As detailed in next subsection, *suCAQR* applies local CAQR factorizations to the physical data layout and applies inter-process binary tree reduction to the logical data layout at each iteration. To keep track of both physical data layout and logical data layout, we use the subroutine of *physical_2_logical* (as shown in Algorithm 1) to translate a row index from a physical data layout to a logical data layout (similar to the *virtual memory* idea used in operating systems).

Algorithm 1 Physical to Logical Index Translation

```
int physical_2_logical(i, B, P, pid)
Input: physical row number i, group size B, P processes.
Output: logical row number.
cycle_size ← B×P; /*#rows per cycle*/
/*number of logical rows before the i-th physical row*/
begin_row ← $\lfloor i/B \rfloor$×cycle_size;
return (begin_row+pid×B+i%B);
```

C. suCAQR Computation

The parallel implementation of *suCAQR* is shown in Algorithm 2. Given a matrix with $m_b \times n_b$ tiles, the algorithm goes through n_b iterations. At each iteration, there are two stages of computations: 1) every process performs a local CAQR factorization independently, followed by 2) a parallel binary-tree reduction to merge the partial results from stage 1 to get the final result.

Stage 1: In the first stage (lines 3–9), every process decides the beginning row that has not computed the final result in the physical data layout. As the iteration number increases, more and more rows in the top portion of each process get the final result. Next, each process computes a local CAQR factorization on its local matrix, which spans from the beginning row to the last local row. The subroutines to determine the beginning row and perform local CAQR are briefly introduced as follows, respectively.

- *get_first_row_position*: In Algorithm 3, given a column number k, it first checks which process the tile $[k, k]$ belongs to (i.e., the *root process* `root_pid`). Depending on the relative position of the current process to the root process (e.g., above, same, below), the first row that has not computed result may lie in one of the three different locations (corresponding to the three conditional statements in Algorithm 3), respectively.
- *local_caqr*: Algorithm 4 calls six basic computational kernels, which are `dgeqrt`, `dormqr`, `dtsqrt`, `dtsssmqr`, `dttqrt`, and `dttssmqr`. To simplify our illustration, we will use the notations of `QR1`, `UP1` (stands for update), `QR2`, `UP2`, `Merge`, and `MergeUpdate` to represent the six kernels correspondingly. The local CAQR factorization will be applied to the submatrix whose local rows are between `phys_1st_row` and the (m_b/P)-th row, and whose columns are between the k-th column and the n_b-th column. The local submatrix can be regarded as D domains of rows. The parameter D is an argument passed to the solver, which can vary from one to the number of rows per process. Local CAQR first computes a partial result for each domain, and then summarizes the results by using a local binary-tree merge. The computations from the D domains can be executed in an embarrassingly parallel way. Section **??** introduces how to determine the optimal number of domains.

Algorithm 2 Parallel *suCAQR* Algorithm

```
suCAQR(A, $m_b$, $n_b$, P, D)
for each tile column k ← 0 to $n_b$-1 do
    ▷ stage 1: local CAQR factorization in each process
    for each process pid ← 0 to P-1 do
        phys_1st_row ← get_first_row_position(k, B, P, pid);
        if (phys_1st_row < $\lfloor m_b/P \rfloor$) then
            local_caqr(A,phys_1st_row, k, B, $m_b$,$n_b$,P,pid,D);
        end if
    end for
    ▷ stage 2: binary-tree merge among processes
    root_pid ← $\lfloor k/B \rfloor$ % P;
    num_active_procs ← $\lceil (m_b - k)/B \rceil$;
    if (num_active_procs ≥ P) then
        num_active_procs ← P;
    end if
    for (hgt ← 1 to $\lceil \log_2 num\_active\_procs \rceil$) do
        $d_1$ ← 0; $d_2$ ← 0+$2^{\text{hgt}-1}$;
        while ($d_2$ < num_active_procs) do
            $p_1$ ← ($d_1$+root_pid)%P;
            $p_2$ ← ($d_2$+root_pid)%P;
            $i_1$ ← get_first_row_position(k, B, P, $p_1$);
            $i_2$ ← get_first_row_position(k, B, P, $p_2$);
            merge_two_rows(A, $i_1$, $i_2$, $p_1$, $p_2$, k, B ,$n_b$, P);
            $d_1$+=$2^{\text{hgt}}$; $d_2$+=$2^{\text{hgt}}$;
        end while
    end for
end for
```

Stage 2: In the second stage (lines 10–26 in Algorithm 2), a parallel binary-tree reduction operation is conducted by a number of processes. It first decides the root of the parallel reduction tree, which changes in a block-cyclic manner. Next, it decides which processes are involved in computing the trailing submatrix (lines 12–15). We refer to the involved processes as *active processes*. The set of active processes will participate in the parallel reduction operation and each process contributes one row of tiles. The binary tree has a height of $\lceil \log_2(ActiveProcesse) \rceil$. It merges the results from

Algorithm 3 Decide the First Row in Physical Layout

```
int get_first_row_position(k, B, P, pid)
num_cycles ← $\lfloor\lfloor k/B\rfloor/P\rfloor$;
root_pid ← $\lfloor k/B\rfloor$ % P;
if (pid < root_pid) then
    phys_1st_row ← B+num_cycles×B;
end if
if (pid = root_pid) then
    phys_1st_row ← k%B+num_cycles×B;
end if
if (pid > root_pid) then
    phys_1st_row ← num_cycles×B;
end if
return phys_1st_row
```

Algorithm 4 Local CAQR Factorization

```
local_caqr(A, phys_1st_row, k, B, $m_b$, $n_b$, P, pid, D)
ds ← $\lfloor m_b/P/D\rfloor$ /*rows per domain*/
▷ step 1: do local factorization for each domain
for each domain d ← 0 to D-1 do
    1st_row ← phys_1st_row + d*rows_per_domain
    R[1st_row,k], V[1st_row,k], T[1st_row,k]
      ← dgeqrt (A[1st_row,k]);
    for j ← k+1 to $n_b$-1 do
        A[1st_row,j] ← dormqr
          (V[1st_row,k],T[1st_row,k],A[1st_row,j]);
    end for
    for i ← 1st_row+1 to $\lfloor m_b/P\rfloor$-1 do
        R[i,k], V[i,k], T[i,k]
          ← dtsqrt (A[1st_row,k],A[i,k]);
    end for
    for i ← 1st_row+1 to 1st_row + $\lfloor m_b/P/D\rfloor$ - 1 do
        for j ← k+1 to $n_b$-1 do
            R[1st_row,j], A[i,j] ←
              dtsssmqr (V[i,k],T[i,k], R[1st_row,j],A[i,j]);
        end for
    end for
end for
▷ step 2: merge results from the D local domains
root_domain ← $\lfloor phys\_1st\_row/ds\rfloor$ ;
for (hgt ← 1 to $\lceil\log_2 D - root\_domain\rceil$) do
    $d_1$ ← root_domain; $d_2$ ← $d_1$+$2^{hgt-1}$;
    while ($d_2$ < D) do;
        $i_1$ ← $d_1$ × ds;
        $i_2$ ← $d_2$ × ds;
        if ($d_1$ = $root\_domain$) then
            $i_1$ ← phys_1st_row;
        end if
        merge_two_rows(A, $i_1$, $i_2$, pid, pid, k, B ,$n_b$, P);
        $d_1$+=$2^{hgt}$; $d_2$+=$2^{hgt}$;
    end while
end for
```

all the leaves up to the proper root identified by `root_pid`. Subroutine `merge_two_rows` is responsible for merging the partial results from two processes. As shown in Algorithm 5, it merges the i_1-th row in process p_1 and the i_2-th row in process p_2 to obtain a new result.

Dynamic Trees: The parallel reduction operation is a global operation that involves all the active processes. Since the *suCAQR* factorization proceeds from the top left corner to the bottom right corner of the matrix, and due to the block cyclic data layout, the root of the binary tree must change in a block cyclic way. Figure 4 shows how the root of a parallel reduction tree may change dynamically. Suppose a matrix is distributed to four processes (P0, P1, P2, P3), the four processes need to participate in a global parallel reduction at every iteration. Each parallel reduction tree has a different root and the root changes in a round robin manner. If there are more than four iterations, the same pattern will be repeated for multiple times.

Algorithm 5 Merge Partial Results from Two Block Rows

```
merge_two_rows(A, $i_1$, $i_2$, $p_1$, $p_2$, k, B ,$n_b$, P)
Input: i1 and i2 are the physical row indices
logic_i1 ← physical_2_logical($i_1$, B, P, $p_1$);
logic_i2 ← physical_2_logical($i_2$, B, P, $p_2$);
R[logic_i1,k], V[logic_i2,k], T[logic_i2,k]
  ← dttqrt (R[logic_i1,k],R[logic_i2,k]);
for j ← k+1 to $n_b$-1 do;
    A[logic_i1,j], A[logic_i2,j] ← dttssmqr (V[logic_i2,k],
      T[logic_i2,k],A[logic_i1,j],A[logic_i2,j]);
end for
```

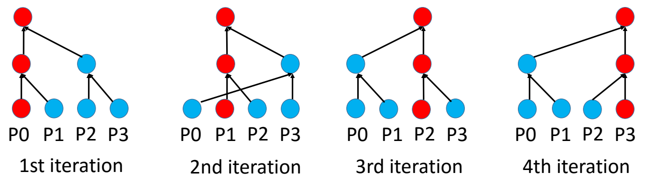


Fig. 4. An example of the parallel reduction operation. Given a matrix with four columns and four processes, *suCAQR* performs the global parallel reduction operation four times (once per iteration). Each iteration corresponds to a different parallel tree with a different root (e.g., P0 is the root (shown in red) in 1st iteration, P1 is the root (shown in red) in 2nd iteration, etc.).

An Example: We use Figure 5 to show a simple example of the parallel *suCAQR* algorithm. In Figure 5. (a), the input matrix has 12 × 4 tiles and is distributed to P0–P3 using a block cyclic distribution method, where the group size B equals one and the number of domains per process equals one. Figure 5. (b) shows the physical data layout of the matrix, where each process stores three rows from three separated rows from subfigure (a). At the first iteration, in Figure 5. (b), every process performs a local CAQR factorization. The local factorization applies QR1 and QR2 to the first column, applies UP1 to all tiles on the first row, and applies UP2 to all tiles on the trailing submatrix. All processes are performing the same computation in parallel but on different data.

After local factorizations are completed, the global binary-tree reduction will start. Since the binary tree reduction is based on a logical data layout, Figure 5. (c), (d), and (e) use the logical data layout to display the matrix. After (b), all processes have obtained their partial R factors which are located on their first rows. Figure 5. (c) shows the status of the binary tree at the bottom level (i.e., depth=2), where P0 and P1 merge results meanwhile P2 and P3 merge results. Figure 5. (d) shows the second level of the binary tree (i.e., depth=1), where P0 and P2 merge results and store the final result to the root process P0.

As shown in Figure 5. (e), the second iteration will start with

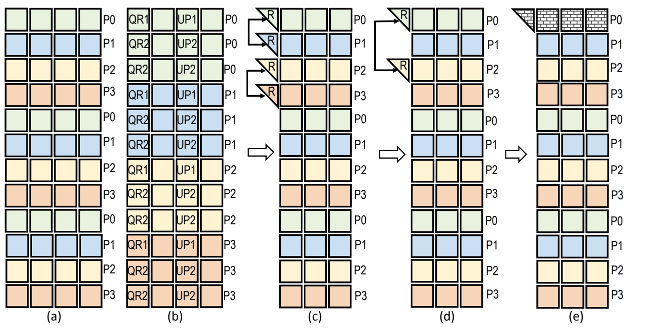


Fig. 5. An example of the parallel *suCAQR* algorithm given a matrix with 12×4 tiles and four processes. Different processes are denoted by different colors. (a) Input matrix in a logical data layout. (b) Physical data layout and corresponding local CAQR factorization in each process. (c) Merging in the third level of the binary tree. (d) Merging in the second level of the binary tree. (e) Matrix status at the beginning of the second iteration.

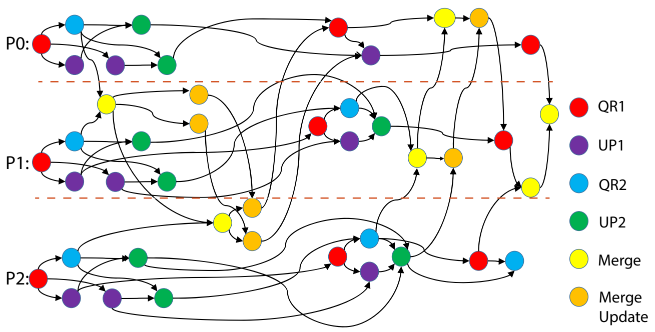


Fig. 6. The corresponding DAG of the *suCAQR* algorithm computing a matrix of 6×3 tiles with 3 processes. Six tile rows are distributed to the three processes in a cyclic way such that each process has two rows of tiles.

a trailing submatrix with one less row and one less column. In the second iteration, P1 becomes the root of the parallel reduction tree, and the same steps will be repeated on the smaller trailing submatrix.

III. The Asynchronous Dataflow Implementation

The parallel algorithm of *suCAQR* has been implemented with the TBLAS [11] runtime system. The TBLAS runtime system is able to support distributed and dynamic DAG scheduling using all CPU cores. It consists of three types of threads: task-generation thread, compute thread, and communication thread. The task-generation thread executes a task-based *suCAQR* program and generates fine-grain tasks to fill in multiple priority-based task queues. Whenever becoming idle, a compute thread picks up a ready task from the ready task queue and executes it. The communication thread is responsible for sending and receiving data between a parent task and its children to satisfy the data dependency requirement. The major distinction of TBLAS is that it does not require representing or constructing a task graph by users but dynamically solves data dependencies among all distributed compute nodes.

With TBLAS, our implementation work is to write a sequential program to generate tasks that are needed in the *suCAQR* algorithm. The rest of the parallel implementation is handled by the TBLAS runtime system, which can automatically detect data dependencies by simply matching a task's output to another task's input on the fly, and schedule tasks to CPUs dynamically. In the *suCAQR* program, we create six types of tasks: QR1, QR2, UP1, UP2, Merge, and Merge Update. Each type of task takes multiple tiles as input and writes new result to the output tiles. To execute the *suCAQR* program, we launch a number of MPI processes on different multicore compute nodes. Every MPI process is managed by one TBLAS runtime system. The runtime systems coordinate with each other to solve data dependencies, dispatch tasks, and send the output of a parent task to its children tasks which are waiting for their input.

The execution of the task-based *suCAQR* program is driven by data availability. When the task-based program is being executed, the frontier portion of the DAG is unrolled dynamically by the runtime system. The size of the active DAG is controlled by a *task window* size. Each runtime system has its own execution point, which follows the data-availability path to reach a different place in the DAG. Figure 6 shows the complete DAG of *suCAQR* which solves a matrix of 6×3 tiles. The task graph is executed by three processes, each of which stores two rows of tiles. From the figure, we can see that three processes execute from the left to the right and communicate with each other only when it is necessary. Note that there is no global synchronization point in the execution, and no artificial order between tasks except for the real data dependencies.

IV. Experimental Results

We first evaluate the impact of tile size on *suCAQR*'s computational kernels, and compare *suCAQR* with the distributed tiled CAQR to show the performance improvement. Next, we present the breakdown of the total execution time to analyze the overhead of different components in our implementation. Finally, we compare *suCAQR* with ScaLAPACK and DPLASMA [12] in weak scalability. For DPLASMA, we have tuned and selected the best tree type and the best D. Both *suCAQR* and DPLASMA use the tuned tile size b=300.

We conducted experiments on the Big Red II Cray XE6/XK7 supercomputer [13] at Indiana University. Table I shows the specifications of the computer system. Each compute node has 32 cores and 64 GB of memory, and runs a Cray Linux OS. Since the mathematics library of Cray LibSci provides a faster performance than Intel MKL (see Figure 7), we use Cray LibSci to run the experiments.

A. Impact of Tile Size on Sequential Computational Kernels

In our implementation, the tile size plays an important role in the overall performance. To search for the best tile size, we test a number of tile sizes on the sequential computational kernels. Figure 7 shows the performance of the `UP2` and `MergeUpdate` kernels in *suCAQR*, as well as DGEMM provided by Cray LibSci and Intel MKL. DGEMM demonstrates

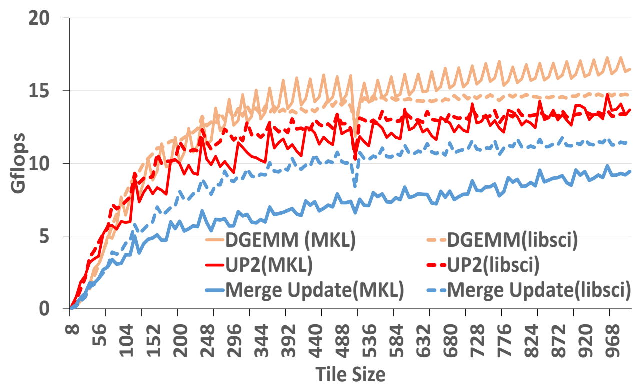


Fig. 7. Performance of sequential suCAQR kernels with different tile sizes using Intel MKL and Cray LibSci. DGEMM implies a practical upper bound.

the practical maximum performance on the computer system. From the figure, we can see that the performance of the kernels increases as the tile size becomes larger. In general, we want to have a bigger tile size to maximize the kernel performance. However, a bigger tile size results in less number of tasks and degraded load balance. Hence, a good tile size is often a number between 200 and 400. In our implementation, we choose 300 as the tile size.

The other important observation is that the kernel of `UP2` *is much faster than the kernel of* `MergeUpdate`. As the number of domains per process (D) increases, the number of `MergeUpdate` tasks will increase. As a result, the overall performance will drop. This is one of the reasons that we do not want D to be large.

B. Comparison with Previous Distributed Tiled CAQR

We compare *suCAQR* with distributed tiled CAQR to evaluate the performance gain of our new solver. In the experiment, we use 512 cores and fix the number of rows for the input matrices. Each matrix is of dimension 57,600 $\times$ n, where n is increased from 900 to 57,600. Figure 8 shows the performance of distributed tiled CAQR and *suCAQR*. When n is greater than

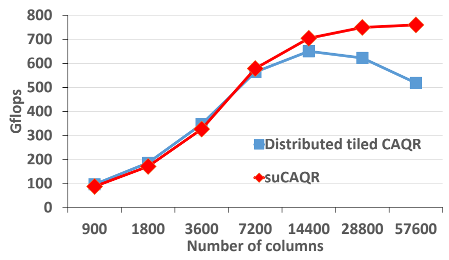


Fig. 8. Comparison of suCAQR with distributed tiled CAQR. All the input has a fixed number of 57,600 rows but a varying number of columns.

or equal to 14,400 (i.e., $\frac{\#columns}{\#rows} = \frac{1}{4}, \frac{1}{2}$, and 1), *suCAQR* starts to be faster than distributed tiled CAQR.

The reason can be illustrated by a simple example shown in Figure 9. The example shows how the tiled CAQR algorithm and the *suCAQR* algorithm would use four processes to factor an 8×5 matrix, respectively. In Figure 9. (a), P0 becomes idle after two iterations as shown in (2), then P1 also becomes idle after another two iterations as shown in (3). Note that half of the processes have no work to do in (3). In Figure 9. (b), P0-P3 are kept busy all the time from the first iteration to the last iteration.

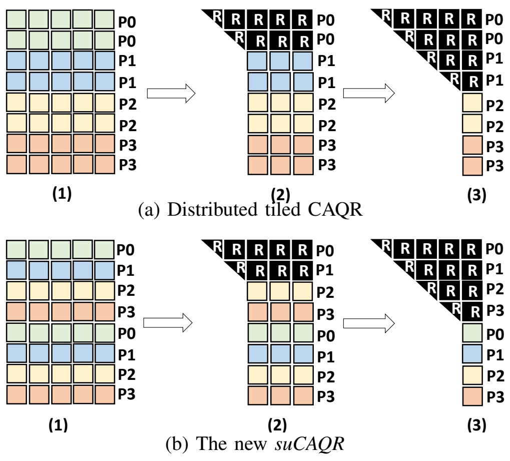
C. Runtime Efficiency Analysis

To analyze the runtime efficiency, we measure the time spent on each component of the *suCAQR* implementation. As presented in Section III, our implementation uses one generation thread to i) create new tasks, one communication thread to ii) send/receive messages, and a number of compute threads to iii) get ready tasks, iv) execute the ready tasks, and v) fire new tasks that are waiting for the completed tasks. We instrument the source code to measure the time spent on each of the five components.

Figure 10 lists the breakdown of the total execution time using 512 cores. We consider four different matrix shapes

TABLE I
SPECIFICATIONS OF THE EXPERIMENTS

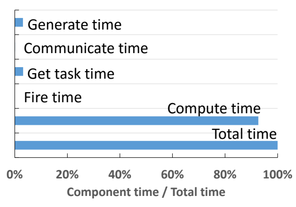
OS	Cray Linux Environment
Nodes	1,020 (a maximum of 2048 cores per job)
Memory per node	64 GB
Processors per node	2
Cores per processor	16
FPU info	two cores share one FPU
Processor	AMD Opteron 6380 2.5GHz
Peak performance/core	13.6 Gflops (with AMD turbo core)
MPI	Cray-MPICH 7.2.5
suCAQR	linked with Cray LibSci 13.2
ScaLAPACK	Cray LibSci 13.2
DPLASMA's HQR	PaRSEC / DPLASMA 2.0.0, PLASMA 2.7.1, linked with Cray LibSci 13.2, **Tuned and selected the best tree and D (between 1 and 4), tile size b=300**.



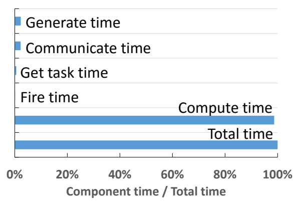
(a) Distributed tiled CAQR

(b) The new *suCAQR*

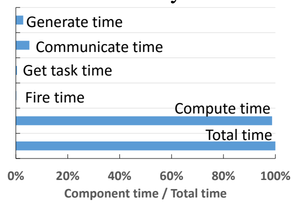
Fig. 9. (a) shows that more and more processes become idle as distributed tiled CAQR proceeds. (b) shows that there is no idle process in the new suCAQR algorithm. Black color represents the completed result.



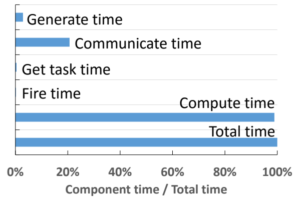
(a) extremely tall and skinny



(b) $\frac{\#columns}{\#rows} = \frac{1}{16}$

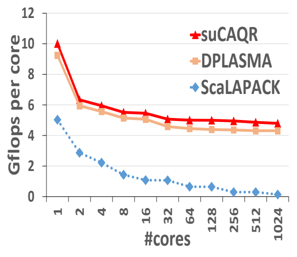


(c) $\frac{\#columns}{\#rows} = \frac{1}{4}$

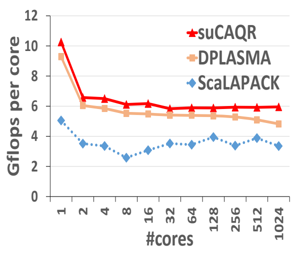


(d) square

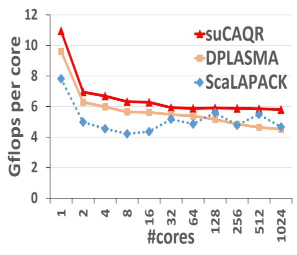
Fig. 10. Breakdown of the total execution time. The total wall clock time is nearly equal to the compute time taken by the computational kernels.



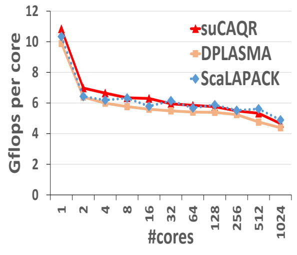
(a) extremely tall and skinny



(b) $\frac{\#columns}{\#rows} = \frac{1}{16}$



(c) $\frac{\#columns}{\#rows} = \frac{1}{4}$



(d) square

Fig. 11. Experiments of weak scalability on 1024 cores.

for which the ratio of the number of columns to the number of rows is: extremely tall and skinny, $\frac{1}{16}$, $\frac{1}{4}$, and square, respectively. As shown in Figure 10, the percentages of the generate-task time, get-task time, and fire-task time are small from (a) to (d). Although the percentage of the communication time increases, it is hidden by a large number of dynamic scheduling computational tasks. For different matrices, the percentage of the compute time relative to the total clock time is 92.7%, 98.7%, 98.7%, and 98.9%, respectively. It implies that the wall clock time is almost equal to the CPU time that is taken by the computational kernels. Also, all the other costs including runtime overhead and communication only occupy at most 7% of the total time. Therefore, this result demonstrates the good efficiency of the *suCAQR* solver.

D. Performance Evaluation in Weak Scalability

We use weak scalability to measure the capability of *suCAQR* to solve larger problems if a user has access to more CPUs. In the experiments, whenever we double the number of cores, we also double the total amount of computation accordingly. The number of matrix elements per core is thus kept as a constant in each experiment.

Figure 11 displays the metric of *Gflops-per-core* for four different matrix shapes using up to 1,024 cores. From all four subfigures, we can see that *suCAQR* maintains good scalability with a nearly constant Gflops-per-core except for the 40% drop from one core to two cores. The reason for the performance drop is that two AMD CPU cores are sharing one floating point unit (FPU) by using the *dual stream* mode.

As shown in Figure 11. (a), when the matrix is extremely tall and skinny (having only 6 block columns but many block rows), *suCAQR* can outperform ScaLAPACK by 30 times and outperform DPLASMA by 13% on 1024 cores. For the matrix shape of $\frac{1}{16}$, *suCAQR* outperforms ScaLAPACK by 78% and outperforms DPLASMA by 23%. For the $\frac{1}{4}$ shape, *suCAQR* outperforms ScaLAPACK by 25% and outperforms DPLASMA by 30%. When the matrix is purely square (in Figure 11. (d)), *suCAQR* is as good as ScaLAPACK, and is 6% better than DPLASMA.

We can also see that the performance of ScaLAPACK keeps rising from tall and skinny matrices to square matrices. This is because a wider matrix will produce a larger trailing submatrix which is more abundant with the vendor-optimized DGEMM kernel which is faster than *suCAQR*'s kernels (see Figure 7). Another observation is that the performance of ScaLAPACK fluctuates a little bit when its 2-D process grid $P \times Q$ is not in a square shape (i.e., $P \neq Q$).

Reasons for the Performance Improvement: The *suCAQR* is faster than DPLASMA due to the following reasons. *First*, we minimize the number of domains to maximize the number of invocations of the fastest kernel. Whenever possible, *suCAQR* uses one domain per process to make sure that there is a sufficient number of `UP2` kernels. *Second*, the single-domain's consecutive row-by-row update has a smaller working set size and better locality than multiple domains which work on multiple different rows. *Third*, the TBLAS runtime we use is efficient in hiding communication by computation such that the wall clock time is simply equal to the pure computation time, which is almost an ideal case. At last, we modify DPLASMA to use the same binary tree and same parameters as *suCAQR*. The profiling results show that *suCAQR* is still faster than DPLASMA because TBLAS has a better data reuse between tasks, which results in reduced computation time.

V. Related Work

Different communication-avoiding algorithms have been applied to a variety of applications. Yelick et al. developed a communication avoiding GMRES solver on shared-memory multicore computers [14]. Grigori et al. designed the communication optimal LU factorization (CALU) algorithm [15]. Khabou et al. also designed a communication avoiding LU factorization version with panel rank revealing pivoting [16]. Ballard et al. created a communication avoiding algorithm for Strassen's matrix multiplication [17]. In this paper, we provide a faster parallel implementation of the CAQR algorithm with a simpler design, and is able to handle matrices of any shapes.

There are also other implementations of communication avoiding QR factorization that target different computing systems. Anderson et al. implemented the communication avoiding QR factorization entirely on a single GPU to achieve high performance [18]. Agullo et al. [19] developed a topology-aware approach that can adapt to a grid environment to compute the CAQR factorization by confining communications to local sites.

DPLASMA's HQR [10] studies different tree options such as flat tree, greedy tree, fibonacci tree, binary tree, and so on. DPLASMA also has an optional fourth level tree called *domain-level tree*. But it is not known when to include and how to optimize the fourth domain-level tree. Also, what type of tree is the best is totally dependent on the result of trying all the different trees at four different levels given a specific input and a certain number of CPUs. Differently, our work reveals that these types of complexities can be avoided without any loss of performance. The elimination of many trees, the simpler software implementation, and the better performance are distinct from the other QR implementations.

VI. Conclusion

Communication-avoiding QR factorization is a generic parallel algorithm and potentially has many different approaches to implementing and optimizing it. In this paper, we target the distributed multicore computer architecture and give a simple, efficient, and scalable design to handle various matrix shapes.

From the perspective of the algorithm, *suCAQR* uses a logical block cyclic data layout on which a dynamic-root parallel tree reduction method is deployed. It also uses a much simplified parallel reduction method to leverage the specific architectural strength of a cluster of multicore compute nodes. *From the perspective of the software design*, a simple unique binary tree can significantly simplify the software implementation and reduce the cost of parameter tuning. *From the perspective of the implementation*, *suCAQR* generates a large number of fine-grain tasks to achieve a high degree of parallelism, and allows for a fully dynamic execution to completely hide communication by computation using the dynamic scheduling TBLAS runtime. The experimental results have shown that *suCAQR* can perform better than the state-of-the-art QR factorization libraries on different matrices using up to 1,024 cores. The runtime efficiency analysis has also revealed a near optimal circumstance (i.e., wall clock time equals computation time) for efficient parallel executions.

Acknowledgement

We would like to thank George Bosilca, Mathieu Faverge, Thomas Herault, and Chongxiao Cao for their great help on installing, tuning, and profiling the PaRSEC/DPLASMA software on Big Red II.

References

[1] E. Anderson, Z. Bai, and J. Dongarra, "Generalized QR factorization and its applications," *Linear Algebra and its Applications*, vol. 162, pp. 243–271, 1992.

[2] A. Björck, *Numerical methods for least squares problems*. SIAM, 1996.

[3] J. W. Demmel, *Applied numerical linear algebra*. SIAM, 1997.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK users' guide*. SIAM, 1997, vol. 4.

[5] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "ScaLAPACK: A portable linear algebra library for distributed memory computers: Design issues and performance," in *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*. Springer, 1996, pp. 95–106.

[6] J. W. Demmel, L. Grigori, M. F. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," UTK, LAPACK Working Note 204, August 2008.

[7] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, N. Knight, and H. Nguyen, "Reconstructing householder vectors from tall-skinny QR," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 3 – 31, 2015, IPDPS 2014 Selected Papers on Numerical and Combinatorial Algorithms.

[8] M. Hoemmen, "A communication-avoiding, hybrid-parallel, rank-revealing orthogonalization method," in *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, May 2011, pp. 966–977.

[9] F. Song, H. Ltaief, B. Hadri, and J. Dongarra, "Scalable tile communication-avoiding QR factorization on multicore cluster systems," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, 2010, pp. 1–11.

[10] J. Dongarra, M. Faverge, T. Herault, M. Jacquelin, J. Langou, and Y. Robert, "Hierarchical QR factorization algorithms for multi-core clusters," *Parallel Computing*, vol. 39, no. 4, pp. 212–232, 2013.

[11] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *Proceedings of the 2009 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'09)*. IEEE, 2009, pp. 1–11.

[12] PaRSEC, "http://icl.utk.edu/parsec."

[13] BigRedII, "http://rt.uits.iu.edu/bigred2."

[14] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 36.

[15] L. Grigori, J. W. Demmel, and H. Xiang, "CALU: A communication optimal LU factorization algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1317–1350, 2011.

[16] A. Khabou, J. W. Demmel, L. Grigori, and M. Gu, "LU factorization with panel rank revealing pivoting and its communication avoiding version," *SIAM Journal on Matrix Analysis and Applications*, vol. 34, no. 3, pp. 1401–1429, 2013.

[17] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, "Communication-optimal parallel algorithm for Strassen's matrix multiplication," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 193–204.

[18] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-avoiding QR decomposition for GPUs," in *2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2011, pp. 48–58.

[19] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. Langou, "QR factorization of tall and skinny matrices in a grid computing environment," in *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*. IEEE, 2010.