Parallel Processing Letters

Scaling Up Parallel Computation of Tiled QR Factorizations by a Distributed Scheduling Runtime System and Analytical Modeling

Weijian Zheng, Fengguang Song

Indiana University-Purdue University Indianapolis,
723 W. Michigan St, Indianapolis, Indiana 46202, USA
{*wz26, fgsong*}*@iupui.edu*

Lan Lin

Ball State University,
2000 W. University Ave, Muncie, Indiana 47306, USA
llin4@bsu.edu

Zizhong Chen

University of California at Riverside,
900 University Ave, Riverside, California 92521, USA
chen@cs.ucr.edu

ABSTRACT

Implementing parallel software for QR factorizations to achieve scalable performance on massively parallel manycore systems requires a comprehensive design that includes algorithm redesign, efficient runtime systems, synchronization and communication reduction, and analytical performance modeling. This paper presents a piece of tiled communication-avoiding QR factorization software that is able to scale efficiently for matrices with general dimensions. We design a tiled communication-avoiding QR factorization algorithm and implement it with a fully distributed dynamic scheduling runtime system to minimize both synchronization and communication. The whole class of communication-avoiding QR factorization algorithms uses an important parameter of D (i.e., the number of domains), whose best solution is still unknown so far and requires manual tuning and empirical searching to find it. To that end, we introduce a simplified analytical performance model to determine an optimal number of domains D^*. The experimental results show that our new parallel implementation is faster than a state-of-the-art multicore-based numerical library by up to 30%, and faster than ScaLAPACK by up to 30 times with thousands of CPU cores. Furthermore, using the new analytical model to predict an optimal number of domains is as competitive as exhaustive searching, and exhibits an average performance difference of 1%.

Keywords: High performance computing; numerical libraries; analytical performance modeling

1. Introduction

QR factorization has been offered by a variety of numerical libraries because it can be used to not only solve scientific and engineering problems such as linear

systems and least-squares problems, but also solve big data analytics problems such as linear regression problems, low-rank factorization data analysis, and production function modeling, as well as assessing the conditioning of these problems [1–3]. QR factorization of an $m \times n$ matrix A 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. Since QR factorization is a fundamental kernel for many important scientific, engineering, and big data analytics applications, a more scalable QR factorization library will accelerate a wide range of domain applications.

Today's most widely used parallel algorithm to solve QR factorizations is the *block QR factorization algorithm* adopted by LAPACK [4] and ScaLAPACK [5,6], as illustrated 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 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, the block algorithm does not scale well for *tall and skinny* matrices (i.e., matrices whose number of rows is much bigger than the number of columns).

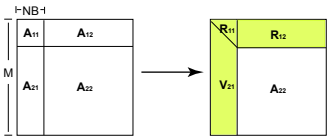


Fig. 1. The classic block QR factorization algorithm.

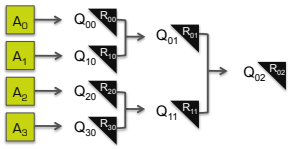


Fig. 2. Communication-Avoiding QR (CAQR) performs level 3 BLAS on the panel (i.e., A_0, A_1, A_2, A_3) followed by a parallel reduction.

To improve the library's performance to solve tall and skinny matrices, James Demmel et al. designed the *Communication-Avoiding QR factorization* (*CAQR*) algorithm [7–9]. As explained in Figure 2, CAQR computes a set of faster level 3 BLAS operations, instead of computing a sequence of slow column-by-column level 1 BLAS operations in the panel as used by ScaLAPACK. 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.

However, generally there may be many different approaches to implementing a *theoretical* algorithm. In this paper, we design and develop a *distributed-memory tiled CAQR* algorithm that aims to reduce not only communication but also synchronizations. Our previous work [10] provided its first implementation but only performed well on tall and skinny matrices. This paper will extend the work, and provide a complete solution. Basically, distributed tiled CAQR factorization can be regarded as a combination of CAQR and tiled QR factorization. We describe the idea briefly here. Suppose an $m \times n$ matrix consists of $m_b \times n_b$ tiles, and b is the

tile size for which $m_b = \frac{m}{b}$ and $n_b = \frac{n}{b}$, our algorithm partitions m rows into D blocks or groups: $A = [A_1; A_2; \ldots; A_D]$, where A_i is of dimension $\frac{m}{D} \times n$ and is called "domain i." A tile-represented matrix A that is divided into D horizontal domains can be expressed as follows:

$$A = \left(\begin{array}{cccc} A_{1,1} & A_{1,2} & \cdots & A_{1,n_b} \\ \cdots & \cdots & \ddots & \cdots \\ A_{\frac{m_b}{D},1} & A_{\frac{m_b}{D},2} & \cdots & A_{\frac{m_b}{D},n_b} \\ \hline A_{\frac{m_b}{D}+1,1} & A_{\frac{m_b}{D}+1,2} & \cdots & A_{\frac{m_b}{D}+1,n_b} \\ \cdots & \cdots & \ddots & \cdots \\ A_{\frac{2m_b}{D},1} & A_{\frac{2m_b}{D},2} & \cdots & A_{\frac{2m_b}{D},n_b} \\ \hline \vdots & \vdots & \ddots & \vdots \\ A_{m_b,1} & A_{m_b,2} & \cdots & A_{m_b,n_b} \end{array}\right),$$

where $A_{i,j}$ is a tile of size $b \times b$. At the beginning, all the domains start to execute the tiled QR factorization on the first panel and the associated updates concurrently. There is no data dependency between different domains. After the tiled QR factorization within each domain is finished, each domain d gets a $b \times b$ upper triangular factor $\hat{R}_d$ located at $A_{(d-1)\times\frac{m_b}{D}+1,1}$. Next, CAQR factorization performs a reduction among all the $\hat{R}_d$'s, where $d \in \{1, \ldots, D\}$. The output of the reduction is the final factor of $R_{1,1}$. Then the final $R_{1,1}$ will be applied to the first row $\{A_{1,2}, \ldots, A_{1,n_b}\}$ to compute the final result of $\{R_{1,2}, \ldots, R_{1,n_b}\}$.

We have implemented a library called "*scalable universal communication-avoiding QR factorization*" (*suCAQR*) to support the distributed tiled CAQR algorithm. This library implementation has the advantages of being simple (e.g., a simplified design), generic (e.g., suitable for matrices of any shapes), and scalable. The *suCAQR* implementation uses the following optimizations for all matrix shapes: (1) *suCAQR* uses a communication-avoiding algorithm to the logical cyclic distribution to achieve load balancing while using a tiled algorithm to the physical contiguous distribution to minimize communications; (2) *suCAQR* uses a fully distributed dynamic scheduling runtime system to support efficient synchronization-reducing executions and communication hiding; (3) The new software design maintains a good tradeoff between the degree of parallelism and the number of fastest kernels for matrices of different shapes by using an appropriate number of domains; And (4) the software introduces an analytical performance model to automatically determine an optimal number of domains without any searching or empirical tuning.

2. Related Work

Morven Gentleman introduced for sparse matrices [11] the approach of splitting a matrix into submatrices allowing the reduction to be done independently and recursively for the submatrices. Then, Pothen and Raghavan [12] developed the idea of parallelizing the factorization of a panel by implementing distributed orthogonal factorizations using Householder and Givens algorithms. Their approach divides the

columns into P subcolumns (where P is the number of processors) and performs factorizations locally from which the final triangular factors are merged. Based on Pothen and Raghavan's work, Demmel et al. [7] proposed a class of algorithms using parallel panel factorizations, which are named communication-avoiding algorithms.

Later on, more communication-avoiding algorithms have been designed and developed for a variety of applications. Yelick et al. developed a shared-memory communication avoiding GMRES solver [13]. Khabou et al. designed a communication avoiding LU factorization version with panel rank revealing pivoting [14]. Our previous work designed a distributed-memory CAQR for multicore cluster systems [10], but it did not scale well on square matrices. In this work, we design and implement a new parallel *suCAQR* library to support matrices of any shapes, and augment it with an analytical model. DPLASMA [15] has implemented a CAQR algorithm, but it is built on a different runtime system and requires more parameter tuning efforts than our implementation. For instance, *suCAQR* does not require empirical tuning of an optimal tree and searching for an optimal number of domains, thanks to a simplified algorithm design and an embedded analytical performance model.

3. The Parallel suCAQR Algorithm

This section introduces the data structure we use to store the matrix, and the algorithm design and pseudocode of *suCAQR*.

Given a matrix, we divide it into square blocks (also known as *tiles*). Each tile is stored in a contiguous memory block. For 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 contiguous memory block that stores a single tile.

In our *task parallelism* implementation, every computational task takes as input several individual tiles (i.e., the basic unit) and computes new output.

Before the real computation starts, the input matrix is distributed across different processes in a static way. Also, we choose to use the simple block cyclic distribution method in our implementation.

3.1. *Computation Tasks in the Algorithm*

Assuming a matrix has $m_b \times n_b$ tiles, the algorithm will execute n_b iterations (as shown in Algorithm 1). In each iteration, there are two phases of computations: 1) Every process performs a local factorization independently; Then 2) Processes perform a parallel reduction to merge partial results computed by Phase 1 to get the final result. Details of the two phases are described as follows.

Phase 1: In the first phase (lines 3–9 in Algorithm 1), every process computes its beginning row that has not gotten the final result in the local data layout (i.e., a number of tile rows stored locally). Next, each process computes a local CAQR factorization on its local matrix, which spans from the beginning row to the last local row. The two subroutines used by Phase 1 are introduced as follows:

Algorithm 1 Parallel *suCAQR* Algorithm

```
1: suCAQR(A, m_b, n_b, P, D)
2: for each tile column k ← 0 to n_b-1 do
3:     ▷ Phase 1: local CAQR factorization in each process
4:     for each process pid ← 0 to P-1 do
5:         phys_1st_row ← get_first_row_position(k, B, P, pid);
6:         if (phys_1st_row < ⌊m_b/P⌋) then
7:             local_caqr(A,phys_1st_row, k, B, m_b,n_b,P,pid,D);
8:         end if
9:     end for
10:    ▷ Phase 2: binary-tree merge among processes
11:    root_pid ← ⌊k/B⌋ % P;
12:    num_active_procs ← ⌈(m_b − k)/B⌉;
13:    if (num_active_procs ≥ P) then
14:        num_active_procs ← P;
15:    end if
16:    for (hgt ← 1 to ⌈log_2 num_active_procs⌉) do
17:        d_1 ← 0; d_2 ← 0+2^(hgt−1);
18:        while (d_2 < num_active_procs) do
19:            p_1 ← (d_1+root_pid)%P;
20:            p_2 ← (d_2+root_pid)%P;
21:            i_1 ← get_first_row_position(k, B, P, p_1);
22:            i_2 ← get_first_row_position(k, B, P, p_2);
23:            merge_two_rows(A, i_1, i_2, p_1, p_2, k, B ,n_b, P);
24:            d_1+=2^hgt; d_2+=2^hgt;
25:        end while
26:    end for
27: end for
```

- *get_first_row_position*: This function tries to find the current process's first row location in the process's local data layout. Given a column index k, it first decides which process the tile $[k, k]$ belongs (this special process is deemed the *root process*). Based on the relative position of the current process to the root process (e.g., above, same, below), the current process's first row that has not computed the final result will be decided adaptively according to the relative positions (i.e., either above, same, or below).
- *local_caqr (in Algorithm 2)*: It calls six computational kernels, which are `dgeqrt`, `dormqr`, `dtsqrt`, `dtsssmqr`, `dttqrt`, and `dttssmqr`. The mathematical details of the kernels can be referred to our previous work [10]. To make it easier to understand, we 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 partitioned into 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

Algorithm 2 Local CAQR Factorization

```
1: local_caqr(A, phys_1st_row, k, B, m_b, n_b, P, pid, D)
2: ds ← ⌊m_b/P/D⌋ /*rows per domain*/
3: ▷ step 1: do local factorization for each domain
4: for each domain d ← 0 to D-1 do
5:     1st_row ← phys_1st_row + d*rows_per_domain
6:     R[1st_row,k], V[1st_row,k], T[1st_row,k]
7:       ← dgeqrt (A[1st_row,k]);
8:     for j ← k+1 to n_b-1 do
9:         A[1st_row,j] ← dormqr
10:          (V[1st_row,k],T[1st_row,k],A[1st_row,j]);
11:    end for
12:    for i ← 1st_row+1 to ⌊m_b/P⌋-1 do
13:        R[i,k], V[i,k], T[i,k]
14:          ← dtsqrt (A[1st_row,k],A[i,k]);
15:    end for
16:    for i ← 1st_row+1 to 1st_row + ⌊m_b/P/D⌋ - 1 do
17:        for j ← k+1 to n_b-1 do
18:            R[1st_row,j], A[i,j] ←
19:              dtsssmqr (V[i,k],T[i,k], R[1st_row,j],A[i,j]);
20:        end for
21:    end for
22: end for
23: ▷ step 2: merge results from the D local domains
24: root_domain ← ⌊phys_1st_row/ds⌋ ;
25: for (hgt ← 1 to ⌈log_2 D − root_domain⌉) do
26:    d_1 ← root_domain; d_2 ← d_1+2^(hgt−1);
27:    while (d_2 < D) do;
28:        i_1 ← d_1 × ds;
29:        i_2 ← d_2 × ds;
30:        if (d_1 = root_domain) then
31:            i_1 ← phys_1st_row;
32:        end if
33:        merge_two_rows(A, i_1, i_2, pid, pid, k, B ,n_b, P);
34:        d_1+=2^hgt; d_2+=2^hgt;
35:    end while
36: end for
```

binary-tree merge. The computations from the D domains can be executed in an embarrassingly parallel way. Section 5 will introduce how to determine an optimal number of domains D^*.

Phase 2: In the second phase (lines 10–26 in Algorithm 1), a global binary-tree reduction computation is conducted by a set of processes in parallel. The algorithm first decides the root of the parallel reduction tree, which changes dynamically in a block-cyclic manner. Next, it decides which processes are involved in the trailing submatrix (lines 12–15). We refer to the involved processes as *active processes*. Only the active processes will participate in the global binary-tree reduction, to which each process contributes one row of tiles. The binary tree is among processes, and has a height of $\lceil \log_2(ActiveProcesse) \rceil$.

Algorithm 3 Merge Partial Results from Two Block Rows

```
merge_two_rows(A, i1, i2, p1, p2, k, B ,nb, P)
Input: i1 and i2 are the physical row indices
logic_i1 ← physical_2_logical(i1, B, P, p1);
logic_i2 ← physical_2_logical(i2, B, P, p2);
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 nb-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
```

The subroutine of `merge_two_rows` (in Algorithm 3) is responsible for merging the partial results from two processes. It merges the i_1-th row of process p_1, and the i_2-th row of process p_2 to obtain an intermediate result. `merge_two_rows` is called by both Algorithm 2 as a local operation, and Algorithm 1 as a distributed operation. A runtime system can detect whether the task is a local operation or a global operation, and use shared memory or message passing to compute the task automatically at runtime.

The merge subroutine needs the following *physical_2_logical* function to translate a row index, from a physical *contiguous* data layout to a logical *block cyclic* data layout, in order to to find out where a process's local row is located in a global logical cyclic view.

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

4. A Distributed Scheduling Runtime System

We have implemented the parallel *suCAQR* algorithm by extending a task scheduling runtime system TBLAS [16]. The TBLAS runtime system can support distributed dynamic directed acyclic graph (DAG) scheduling on distributed systems with multicore compute nodes. Given an input matrix, its data is distributed across different compute nodes (see Figure 3). The corresponding task graph is also implicitly partitioned into different compute nodes. That is, every task has a home compute node and is executed by any CPU core on the home compute node. From a high level standpoint, every compute node is executing a runtime system instance which collaborate with each other to solve the same single problem in parallel. The data dependency correctness is also guaranteed by following a distributed computing protocol [17].

To utilize the TBLAS runtime system, 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 program, we launch a number of MPI processes on different multicore compute nodes. Every MPI process is managed by one instance of 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 design of the runtime system running on each multicore compute node is shown in Figure 4. It consists of three types of threads: task-generation thread, task-computing thread, and communication thread. The task-generation thread executes a sequential task-based program and generates fine-grain tasks to fill in a number of 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 TBLAS runtime system does not require constructing a task graph by users in advance or require a new compiler; instead it can automatically resolve data dependencies at runtime among all distributed compute nodes.

The new *suCAQR* tasks are scheduled based on their priorities. In our extended runtime system, the binary-tree Merge tasks have the highest priority. At iteration i, the tasks located between the i-th column and the $(i+d)$-th column have the second highest priority given a lookahead depth of d. The remaining tasks have a regular priority. Also, all the *suCAQR* tasks' input and output tiles are indexed by a logical layout to facilitate solving data dependencies. When executing the task, its input/output tiles will be converted to the local data layout to read/write data.

When the task-based *suCAQR* program is being executed, the frontier of the

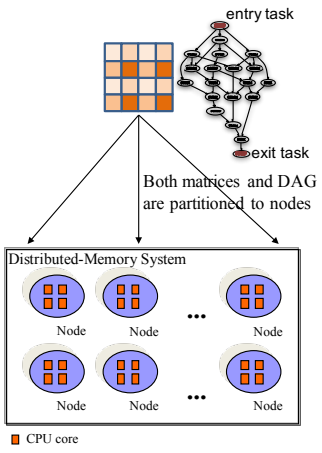


Fig. 3. A high level view of the distributed suCAQR software.

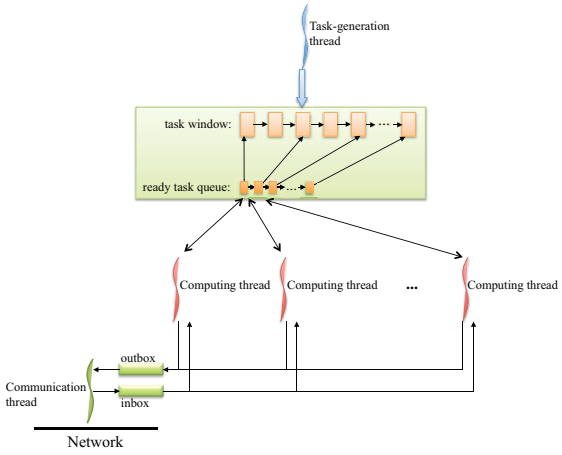


Fig. 4. The runtime system on each compute node.

DAG is unrolled dynamically by the runtime system. The size of the active frontier 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 5 shows an example of the DAG for a *suCAQR* execution, where three processes execute from the left to the right, and communicate with each other for very few times only when necessary. Also, there are no global synchronizations.

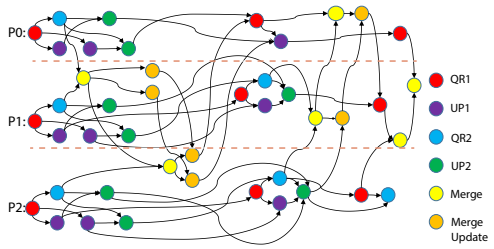


Fig. 5. A DAG execution for the parallel *suCAQR* algorithm using 3 processes.

5. Analytical Model to Determine an Optimal Number of Domains

As listed in Algorithm 2, each process requires a number of D domains of data to compute. However, D could be any number between one and $\frac{m_b}{P}$, where $D = \frac{m_b}{P}$ means that every block row is a domain. The problem we target is how to decide an optimal number of domains directly without trying all the possibilities of D. This section introduces an analytical model to determine an optimal number of domains.

5.1. *Effect of Domains, Our Observations, and Analysis*

We reveal that the number of domains has a significant impact on the overall program performance. For instance, in Figure 7. (a), a different value of D leads to distinct performance on the same input size (e.g., 128 block rows and an increasing number of block columns). Each line in the figure represents the performance a specific D, where D is between one and 32. As an example, for an input matrix of size 128×8 blocks, $D = 1$ is 230% slower than $D = 8$ on 64 CPU cores. Hence, determining an optimal number of domains is critical to program performance.

In order to search for an optimal number of domains, we analyze the algorithm and give the following insights into the problem:

(1) Within each domain, the *degree of task parallelism* is decided by the number of block columns assuming a matrix has $m_b \times n_b$ blocks. That is, there are $n_b - k$ parallel tasks in the k-th iteration. Note that there is no parallelism between different block rows that belong to the same domain.
(2) When n_b is much larger than the number of threads per process (e.g., T threads), using one domain will achieve the minimum number of operations and maximum

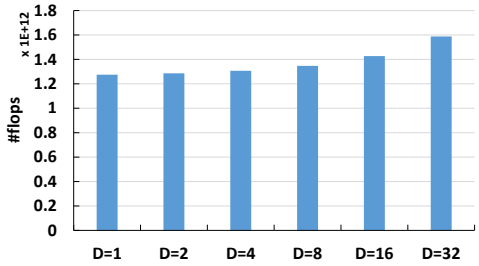


Fig. 6. An increasing number of floating point operations as the number of domains increases. #flops are measured by PAPI counters.

number of higher-performance `UP2` tasks, hence leading to the best performance. We used PAPI to measure the total number of floating point operations when we increase the number of domains. The PAPI measurement of *number of floating point operations (#flops)* is shown in Figure 6. As D increases from one to 32, the number of *flops* increases accordingly. This result empirically proves that a larger D increases the number of operations.

(3) When D is increased, the degree of task parallelism within each process will be increased by D times correspondingly because all the D domains can be executed in parallel. However, when D increases too much, the number of the fastest `UP2` tasks will decrease (replaced by the slower `Merge` and `MergeUpdate` tasks) and the number of operations will increase, so that the overall performance starts to drop instead. Therefore, we have to find a good tradeoff to choose between a large D and a small D.

5.2. *The Analytical Performance Model*

Based on the above analysis, we build a new analytical performance model to determine the optimal number of domains.

First, it is important to have more tasks than the number of CPU cores at any time on manycore systems. Otherwise, CPU cores will become idle due to the lack of tasks to compute.

Hence, to achieve high performance, we require the degree of task parallelism within a multi-threaded process must be greater than the number of threads per process (T) to make fully use of all CPU cores. We use one process per compute node. Assuming a process has n_b block columns and D domains, the process can have up to $n_b \times D$ independent tasks. This desired condition of *sufficient task parallelism degree* can be expressed as follows:

$$(\text{degree}_{parallel} = n_b \times D) \geq c \times T \implies D \geq \lceil \frac{c \times T}{n_b} \rceil$$

In the above formula, c is a simple coefficient and is set to 4 based on our collected performance data. The formula has a special boundary case: when $n_b = c \times T$, D

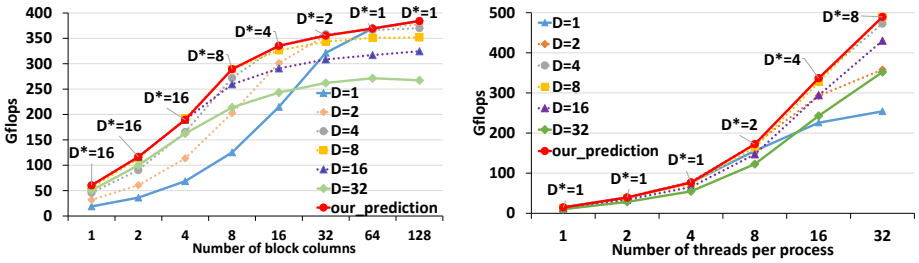
will always satisfy the condition of sufficient task parallelism degree because D is at least equal to 1. In other words, as long as $n_b \geq c \times T$, using one domain can satisfy the condition and produce the best performance, regardless of the matrix shape being skinny or square.

On the other hand, n_b may become smaller and smaller and eventually $D = 1$ cannot satisfy the condition. In such a situation, we need to make D bigger and bigger to compensate for the diminishing degree of task parallelism. However, if n_b is too small (e.g., only 2 or 3 columns), we require $D = T$ to let T threads compute T domains in an embarrassingly parallel way. The reason we do not make $D > T$ is the following. Aggressively creating unnecessarily more domains than the number of threads per process will slow down the performance due to: 1) an increased number of floating point operations; 2) lesser fast `UP2` tasks; and 3) an increase in the working set from more irregular cross-domain merges.

Our final analytical model is defined as follows:

$$D^* = \begin{cases} \lceil \frac{c \times T}{n_b} \rceil & \text{if } n_b \geq c \\ \\ \min\{T, \frac{m_b}{P}\} & \text{if } n_b < c, \text{where c=4.} \end{cases}$$

Next, we performed two experiments to verify the analytical model. Figure 7.a shows the actual performance using different numbers of domains. Each D has a distinct line. Also, there is a special line showing the performance of using the predicted D^* derived from the analytical model. We execute 4 processes and each process consists of 16 threads. The input matrix size goes from 128×1 blocks to 128×128 blocks. For each matrix, we use all possible numbers of domains and collect their performance. From the data, we can see that the model-based prediction (D^*) closely matches the actual best number of domains among all possibilities. Note that the line of D^* is directly calculated using the analytical model.



(a) All input matrices have 128 block rows but a different number of block columns (ranging from 1 to 128).

(b) The input matrix has 128 block rows and 16 block columns. It is computed using different numbers of threads.

Fig. 7. Use an analytical model to predict an optimal number of domains D^*. The model-predicted D^* is compared to all different possible D.

In the second experiment, we take the same matrix size of 128×16 blocks but change the number of threads per process from one to 32. As shown in Figure 7.b, the model-based prediction again coincides with the actual best number of domains. We also did a set of large-scale experiments to verify the analytical model's predictions. The large-scale experimental results are presented in the next Section 6.

6. Experimental Results

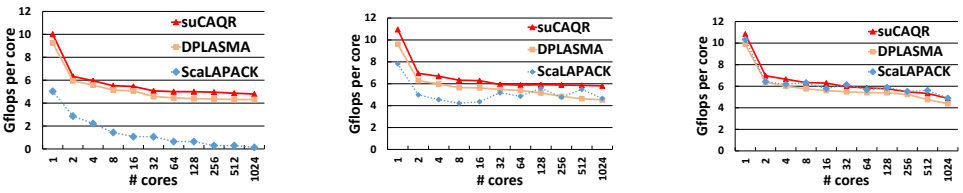
In this section, we compare *suCAQR* with ScaLAPACK and DPLASMA [15] on two different HPC systems. ScaLAPACK is a de facto standard MPI-based linear algebra library for distributed memory systems, and DPLASMA is a newer linear algebra library for distributed multicore systems. We conduct experiments on the Big Red II Cray XE6/XK7 supercomputer [18] at Indiana University, and the Comet supercomputer at San Diego Supercomputing Center [19]. On Big Red II, each compute node has 32 CPU cores and 64 GB of memory, and runs a Cray Linux OS. On Comet, each compute node has 24 CPU cores and 128 GB of memory, and is connected with an Infiniband network.

6.1. *Performance Evaluation*

We conduct experiments to measure the weak scalability performance of different libraries. In the experiments, whenever we double the number of CPU cores, we also double the total amount of computation accordingly. The number of matrix elements per CPU core is kept as a constant in each experiment. Weak scalability is often used to measure a program's capability to solve larger problems when a user has access to more computing resources.

On Big Red II: Figure 8 shows the measured performance of *Gflops Per Core* on Big Red II using 1 to 1,024 CPU cores. There are three subfigures. They correspond to three different matrix shapes, ranging from extremely tall&skinny matrices, matrices whose rows are four times as many as columns, to square matrices. Overall from the three subfigures, we can see that *suCAQR* provide better performance than DPLASMA. In particular, in subfigure *a*, *suCAQR* is 15% faster than DPLASMA on 1,024 cores. In subfigure *b*, *suCAQR* is 30% faster than DPLASMA on 1,024 cores. And in subfigure *c*, when the matrix is square, *suCAQR* is 11% faster. In comparison with ScaLAPACK, the *suCAQR* program is 30 times and 30% faster, as shown in subfigures *a* and *b*, respectively. However, ScaLAPACK provides the best performance — *suCAQR* is comparable to it — when solving square matrices as displayed in subfigure *c*.

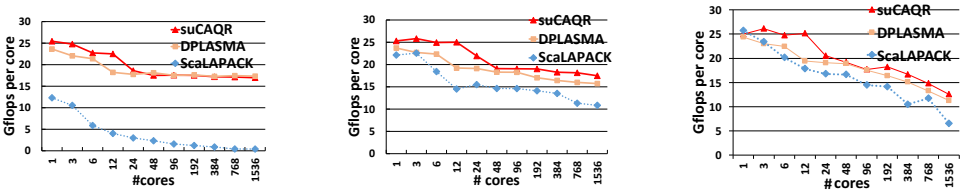
On Comet: We did weak scalability experiments using up to 1,536 CPU cores on the Comet system. Figure 9.a shows that *suCAQR* is faster than DPLASMA by up to 25% from 1 to 12 cores; then becomes similar to DPLASMA after 24 cores. The performance drop from 12 to 24 cores is due to the fact that each Comet compute node has two 12-core sockets and two NUMA memory nodes, and the slow NUMA memory accesses decrease the overall performance. From Figure 9.b, we can see



(a) extremely tall (b) $\frac{\#columns}{\#rows} = \frac{1}{4}$ (c) square

Fig. 8. Experiments of weak scalability on Big Red II.

that *suCAQR* is faster than DPLASMA by up to 32% from 1 to 24 cores, then continues to be faster than DPLASMA by up to 26% from 48 to 1536 cores. In Figure 9.c, *suCAQR* outperforms both DPLASMA and ScaLAPACK. But none of them can attain a constant Gflops-per-core performance. We believe the reason is related to the characteristics of computer system balance. That is, Comet's compute node is twice faster than BigRedII's compute node, but its network performance is relatively less than Cray's Gemini interconnect so that the communication time is hard to hide and dominates the total execution time.



(a) extremely tall (b) $\frac{\#columns}{\#rows} = \frac{1}{4}$ (c) square

Fig. 9. Experiments of weak scalability on Comet.

6.2. *Evaluation of Analytical Modeling: Strong Scalability*

We run large scale experiments to verify whether our analytical model can find the best number of domains or not. For the experiments, we compare the *model predicted* optimal number of domains (i.e., D^*) to the empirically *searched* best number of domains (i.e., $D^{*\prime}$).

We first use a set of strong scalability experiments to verify the analytical model. The next section will use a set of weak scalability experiments to verify the model. Table 1 shows the performance differences between using the predicted D^* and the empirically searched $D^{*\prime}$ on Big Red II. As shown in the table, there are four groups of experiments with distinct matrix shapes: 1) extremely tall and skinny matrices, 2) matrices whose rows are 16 times as many as columns, 3) matrices whose rows are 4 times are many as columns, and 4) square matrices.

For each matrix input, we use different numbers of CPU cores from 1 to 512. Provided with n cores, we display both the predicted D^* and the searched best

Table 1. Analytical modeling for strong scalability experiments on BigRedII.

	extremely tall and skinny				$\frac{\#columns}{\#rows} = \frac{1}{16}$			
#Cores	D*	D*′	perf of D*	perf of D*′	D*	D*′	perf of D*	perf of D*
1	1	1	10.4 Gflops	10.4 Gflops	1	1	12	12
2	2	1	12.1	12.9	1	1	14.6	14.6
4	4	2	23.9	24	1	1	27.4	27.4
8	8	16	44.2	44.3	2	1	50.4	51.2
16	16	16	88.4	88.4	4	1	99.4	101.4
32	16	16	158.6	158.6	4	8	189.4	190.6
64	16	16	322.3	322.3	4	4	377.1	377.1
128	16	16	623.9	623.9	4	4	716	716
256	16	16	1257.9	1257.9	3	3	1423.9	1423.9
512	16	16	2316.7	2316.7	3	3	2606.9	2606.9
	$\frac{\#columns}{\#rows} = \frac{1}{4}$				square			
1	1	1	12.1	12.1	1	1	12.2	12.2
2	1	1	14.7	14.7	1	1	14.8	14.8
4	1	1	27.7	27.7	1	1	27.9	27.9
8	1	1	51.6	51.6	1	1	52	52
16	2	1	99.1	102.5	1	1	102.8	102.8
32	2	2	188.8	188.8	1	1	195.2	195.2
64	2	2	374.7	374.7	1	2	373	377.3
128	1	1	699.6	699.6	1	1	727.7	727.7
256	2	2	1438	1438	1	1	1354.7	1354.7
512	2	2	2448	2448	1	1	2223.7	2223.7

$D^{*\prime}$ as well as their corresponding performances. From Table 1, we can see that the model-predicted D^* is equal to the empirically searched $D^{*\prime}$ in totally 32 out of 40 different experiments. Even for the rest of the experiments (i.e., 8 out of 40), the predicted D* is still close to the empirically searched best D*′, and the maximum performance difference (in terms of Gflops) is 6%.

6.3. *Evaluation of Analytical Modeling: Weak Scalability*

We also verify the model using a set of weak scalability experiments. Similarly, we compare the predicted number of domains (D^*) to the empirically searched number of domains ($D^{*\prime}$).

Table 2 lists the results on Big Red II using four different matrix shapes from extremely tall and skinny matrices to square matrices. Based on the table, we find that the predicted D^* is the same as the empirically searched best $D^{*\prime}$ in 29 out of 44 cases. Among the rest of the 15 cases, the worst performance loss occurs when running the 256-core experiment with the matrix shape of $\frac{1}{16}$, in which D^*=1 differs from $D^{*\prime}$=2, affecting the performance by 3.9%. Although not identical, the predicted D^* is still comparable to the searched best number. As a result, the performance of using D^* is almost the same as that using the empirical best $D^{*\prime}$. On average, the overall performance difference among all the 44 test cases is 0.7%.

7. Conclusion

We target the distributed-memory multicore computer architecture and provide a simple, efficient, and scalable parallel implementation to compute QR factorizations for various matrix shapes. The parallel *suCAQR* library we have developed uses

Table 2. Analytical modeling for weak scalability experiments on BigRedII.

	extremely tall and skinny				$\frac{\#columns}{\#rows} = \frac{1}{16}$			
#Cores	D*	D*′	perf of D*	perf of D*′	D*	D*′	perf of D*	perf of D*
1	1	1	10.4Glops	10.4Gflops	1	1	10.3	10.3
2	2	2	12.7	12.7	2	1	12.8	13.2
4	4	8	23.2	23.7	2	1	25.3	26
8	8	8	44.1	44.1	4	4	47.1	47.1
16	16	16	87.3	87.3	4	4	94.7	94.7
32	16	32	165	167	4	4	186	186
64	16	16	324	324	2	4	366	377
128	16	16	641	641	2	4	743.5	753.4
256	16	32	1259	1267.8	1	2	1461.1	1518.4
512	16	16	2490.1	2490.1	1	2	2963.2	3027.1
1024	16	32	4898	4907	1	2	6048.4	6071.1
	$\frac{\#columns}{\#rows} = \frac{1}{4}$				square			
1	1	1	11	11	1	1	10.8	10.8
2	1	1	13.9	13.9	1	1	14	14
4	2	1	26	26.6	1	1	26.6	26.6
8	2	1	49.2	50.6	1	1	50.7	50.7
16	4	2	97.5	100.6	1	1	100.4	100.4
32	2	2	189.7	189.7	1	1	190.3	190.3
64	2	2	376.3	376.3	1	1	374.1	374.1
128	1	2	743.6	756.4	1	1	737.3	737.3
256	1	1	1506.6	1506.6	1	1	1403.9	1403.9
512	1	1	2997	2997	1	1	2729.3	2729.3
1024	1	1	5945.2	5945.2	1	1	4765	4765

a logical data layout on which a dynamic-root parallel tree reduction method is deployed. It also leverages the architectural strength of a cluster of multicore nodes. That is, within each multicore node, a shared-memory tiled QR is carried out on the local matrix data from the first row to the last row of the local matrix, where the front line of the data flow (where it shows a "domino effect") keeps all the cores busy. Between different compute nodes, a binary-tree reduction is conducted among all P processes in parallel. Such a simple design simplifies our algorithm design and library implementation.

We build a new analytical model to determine the important factor of the number of domains without any searching. From the perspective of the software design, we have designed a distributed dynamic scheduling runtime system, and realized a synchronization-reducing version of the tiled QR factorization algorithm. The experimental results have shown that *suCAQR* can perform better than the state-of-the-art libraries with different matrices using up to 1,536 cores on two distinct HPC systems. Since the task-based dynamic parallelism approach is particularly effective in avoiding CPU idle cycles and hiding expensive communications, our future work will generalize and extend it to other domains such as large-scale graphs, computational fluid dynamics, and machine learning.

References

[1] E Anderson, Zhaojun Bai, and J Dongarra. Generalized QR factorization and its applications. *Linear Algebra and its Applications*, 162:243–271, 1992.
[2] Ake Björck. *Numerical methods for least squares problems*. SIAM, 1996.
[3] James W Demmel. *Applied numerical linear algebra*. SIAM, 1997.

[4] E. Anderson, Z. Bai, C. Bischof, L. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.

[5] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide*, volume 4. SIAM, 1997.

[6] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R Clinton 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*, pages 95–106. Springer, 1996.

[7] James W. Demmel, Laura Grigori, Mark Frederick Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. LAPACK Working Note 204, UTK, August 2008.

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

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

[10] Fengguang Song, Hatem Ltaief, Bilel Hadri, and Jack 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)*, pages 1–11, 2010.

[11] W. Morven Gentleman. Row elimination for solving sparse linear systems and least squares problems. *Numer. Anal., Lect Notes Math*, 506:122–133, 1976.

[12] A. Pothen and P. Raghavan. Distributed orthogonal factorization: Givens and Householder algorithms. *SIAM Journal on Scientific and Statistical Computing*, 10:1113–1134, 1989.

[13] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 36. ACM, 2009.

[14] Amal Khabou, James W Demmel, Laura Grigori, and Ming Gu. LU factorization with panel rank revealing pivoting and its communication avoiding version. *SIAM Journal on Matrix Analysis and Applications*, 34(3):1401–1429, 2013.

[15] PaRSEC. http://icl.utk.edu/parsec.

[16] Fengguang Song, Asim YarKhan, and Jack 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)*, pages 1–11. IEEE, 2009.

[17] Fengguang Song and Jack Dongarra. A scalable approach to solving dense linear algebra problems on hybrid CPU-GPU systems. *Concurrency and Computation: Practice and Experience*, 27(14):3702–3723, 2015.

[18] BigRedII. http://rt.uits.iu.edu/bigred2.

[19] Comet. https://portal.xsede.org/sdsc-comet.