Tall Skinny QR decomposition

Pudit Tempattarachoke and Chutchatut Sutichavengkul

1 Progress

1.1 OpenMP implementation

A key feature of this TSQR implementation is the combination of contiguous memory blocks with hierarchical reduction using ping-pong buffers, which together maximize both computational efficiency and memory locality.

The matrix $A \in \mathbb{R}^{m \times n}$ is first partitioned into p vertical blocks

$$A_i \in \mathbb{R}^{(m/p) \times n}, \quad i = 0, \ldots, p-1,$$

stored consecutively in memory. Using contiguous memory ensures that each thread accesses a continuous region of memory when performing its local QR factorization:

$$A_i = Q_i R_i.$$

This memory layout improves cache utilization, particularly when the block assigned to each processor is small enough to fit efficiently in cache.

After computing the local R_i factors, the algorithm performs a hierarchical reduction to compute the global R. This is done using a ping-pong buffer strategy: two buffers, `R_buffer1` and `R_buffer2`, store intermediate R matrices alternately. At each stage of the reduction, pairs of R-matrices are stacked vertically:

$$M_i = \begin{bmatrix} R_{2i} \\ R_{2i+1} \end{bmatrix} \in \mathbb{R}^{2n \times n},$$

and a new QR factorization is applied:

$$M_i = Q_i' R_i',$$

with R_i' written into the alternate buffer. After all pairs are processed, the buffers are swapped for the next stage. The reduction proceeds recursively, halving the number of matrices at each stage until only a single global $R \in \mathbb{R}^{n \times n}$ remains.

Combining contiguous memory blocks with this hierarchical ping-pong buffer approach provides several advantages:

- **Memory locality:** Each thread works on sequential memory regions, minimizing cache misses.

- **Reduced copying:** The ping-pong buffers allow in-place accumulation of R-matrices without repeatedly allocating new memory.
- **Parallel scalability:** OpenMP `for` loops and barriers synchronize computation efficiently while allowing maximum concurrent execution.
- **Load balancing:** Each thread initially processes blocks of equal size, and the reduction stages evenly distribute the remaining R-factor computations.

Overall, this design leverages both data parallelism (local QR per contiguous block) and task parallelism (hierarchical R reduction), resulting in a high-performance, scalable TSQR algorithm suitable for multi-core architectures.

1.2 Open MPI implementation

A key feature of this TSQR implementation is the combination of distributed memory parallelism with hierarchical reduction using MPI, which maximizes both computational efficiency and communication locality.

The matrix $A \in \mathbb{R}^{m \times n}$ is first partitioned into p contiguous vertical blocks

$$A_i \in \mathbb{R}^{(m/p) \times n}, \quad i = 0, \ldots, p-1,$$

and each block is assigned to a separate MPI rank. The data is distributed using `MPI_Scatter`, ensuring that each rank receives a contiguous portion of the original matrix:

$$A_i = Q_i R_i.$$

Each rank then performs a local QR factorization independently. Storing the blocks contiguously in memory improves cache performance and reduces memory fragmentation.

After computing the local R_i factors, a hierarchical reduction is performed across ranks to compute the global R. At each stage of the reduction, ranks are paired: one rank sends its R-factor to its partner, while the receiving rank stacks the incoming R with its own:

$$M_i = \begin{bmatrix} R_{2i} \\ R_{2i+1} \end{bmatrix} \in \mathbb{R}^{2n \times n}.$$

A QR factorization is then applied to the stacked matrix:

$$M_i = Q_i' R_i',$$

with the resulting R_i' stored for the next reduction stage. This process repeats recursively, halving the number of active ranks at each step, until a single global $R \in \mathbb{R}^{n \times n}$ remains at rank 0.

Advantages

- **Communication locality:** Only neighboring ranks exchange R-matrices at each stage, minimizing communication volume.
- **Memory efficiency:** Each rank reuses a fixed-size buffer for stacking and updating R-factors.
- **Parallel scalability:** Local QR computations occur concurrently, and global synchronization happens only logarithmically with the number of ranks.
- **Load balancing:** Each rank initially works on equal-sized blocks, and the recursive reduction evenly distributes the remaining QR computations.

2 Checkpoint

2.1 OpenMP parallelization

2.1.1 Speedup vs Number of Threads

Data and definitions
We define the *total speedup* measured experimentally as

$$S_{\text{total}}(p, n) \;=\; \frac{T(1, n)}{T(p, n)},$$

where $T(p, n)$ is the measured runtime (ms) for p threads and matrix width n. The dataset used (times in ms) corresponds to $m = 1600$ and $n \in \{2, 4, 8, 16, 32\}$:

Threads	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
1	19222	38725	77728	156111	311765
2	1057	2077	4142	8404	17241
4	114	232	470	955	1988
8	15	28	56	117	271
16	4	8	17	33	97
32	2	4	4	15	81

From this the total speedups (rounded) are:

Threads	$S_{\text{total}}(p, 2)$	$S_{\text{total}}(p, 4)$	$S_{\text{total}}(p, 8)$	$S_{\text{total}}(p, 16)$	$S_{\text{total}}(p, 32)$
1	1.0	1.0	1.0	1.0	1.0
2	18.2	18.6	18.8	18.6	18.1
4	168.7	167.1	165.2	163.4	156.9
8	1281.5	1383.0	1388.0	1334.0	1150.2
16	4805.5	4840.6	4572.2	4727.6	3213.6
32	9611.0	9681.3	19432.0	10407.4	3849.5

Plots: Total Speedup and Computation Speedup

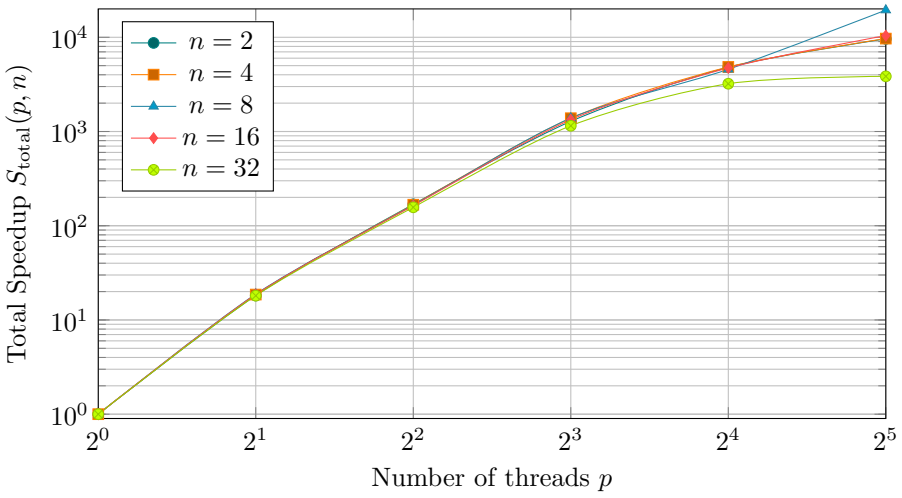

Figure 1: Measured **Total Speedup** $S_{\text{total}}(p, n)$ vs number of threads for each matrix width n. A log-log scaling is used because the measured speedups span many orders of magnitude.

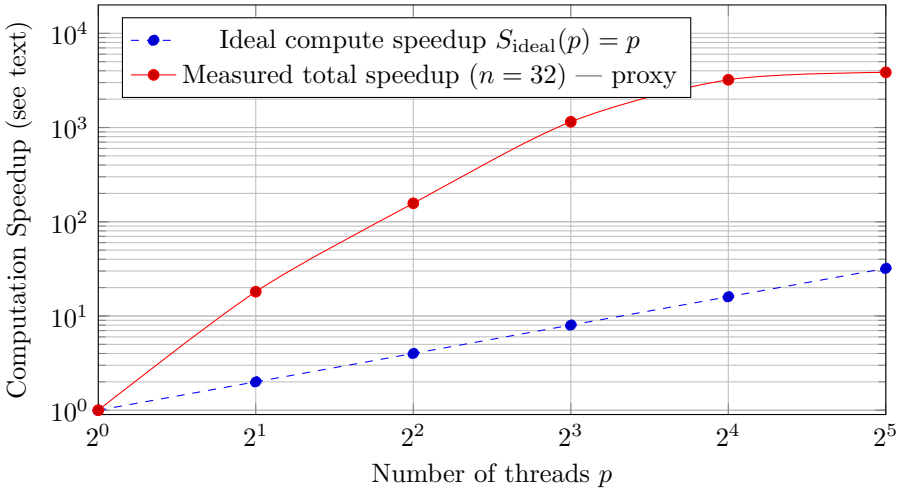

Figure 2: Comparison of ideal, linear **computation** speedup and the measured total speedup for $n = 32$. The measured curve is shown as a proxy to illustrate that the observed acceleration is much larger than pure compute-parallelism (see discussion).

Discussion and interpretation

Total speedup shape. The *total speedup* curves show extremely large increases as thread count grows. For each fixed matrix width n the measured speedup values are orders of magnitude larger than the corresponding number of threads p. Even at modest thread counts, the ratio $T(1)/T(p)$ becomes huge.

Why speedup is *superlinear*. The measured speedups are *superlinear* (i.e. $S_{\text{total}}(p) \gg p$). This arises from a combination of effects:

- **Algorithmic difference between serial and parallel code.** The parallel implementation uses TSQR (a tree-reduction QR suited to tall-skinny matrices) whereas the single-thread baseline is a plain serial QR. TSQR reorganizes work and reduces global memory traffic, performing fewer expensive memory accesses and fewer global synchronizations.
- **Cache and memory hierarchy effects.** Each thread works on a smaller local block which often fits in L1/L2 cache. This reduces cache misses and memory latency compared to the monolithic serial run.
- **Measurement baseline issues.** If the single-thread measurement includes extra overhead (initialization, I/O) that the parallel measurements avoid or amortize, the speedup will appear inflated.

Computation speedup panel and what it means. The second figure overlays the ideal linear compute speedup $S_{\text{ideal}}(p) = p$ with the measured total speedup for $n = 32$. Because we did not separately instrument computation vs communication, we cannot draw a true computation-only curve. The measured total speedup lies far above the ideal line, confirming that the observed acceleration comes from a combination of algorithmic improvements and hardware-cache effects. For tall-skinny matrices and many processes, the cost of MP communication can dominate, reducing scaling efficiency as the performance speed up when $n = 32$ does not superlinear and follow the previous iterations of trends.

Conclusion

The measured total speedups indicate an extremely effective parallelization strategy for tall-skinny matrices. The superlinear behavior demonstrates that TSQR plus shared-memory parallelism capitalizes on cache and algorithmic structure. For rigorous reporting, separate pure compute scaling from communication and rerun measurements with repeated trials and an optimized single-thread baseline.

2.2 OpenMPI parallelization

2.2.1 Speedup vs Number of Processors

Data and definitions
We define the *total speedup* measured experimentally as

$$S_{\text{total}}(p, n) \;=\; \frac{T(1, n)}{T(p, n)},$$

where $T(p, n)$ is the measured runtime (ms) for p MPI processes and matrix width n. The dataset used (times in ms) corresponds to $m = 1600$ and $n \in \{2, 4, 8, 16, 32\}$:

Processors	$n = 2$	$n = 4$	$n = 8$	$n = 16$	$n = 32$
1	14656	29482	58859	118389	238073
2	1054	2017	3929	7856	15949
4	236	355	644	1106	2155
8	178	192	220	291	445
16	207	235	210	251	270
32	426	432	430	418	401

From this, the total speedups are:

Processors	$S_{\text{total}}(p, 2)$	$S_{\text{total}}(p, 4)$	$S_{\text{total}}(p, 8)$	$S_{\text{total}}(p, 16)$	$S_{\text{total}}(p, 32)$
1	1.0	1.0	1.0	1.0	1.0
2	13.9	14.6	15.0	15.1	14.9
4	62.0	83.0	91.5	107.0	110.4
8	82.3	153.7	267.5	406.9	535.0
16	70.8	125.4	280.3	471.9	881.7
32	34.4	68.3	136.9	283.3	593.2

Total Speedup vs. Number of Processors

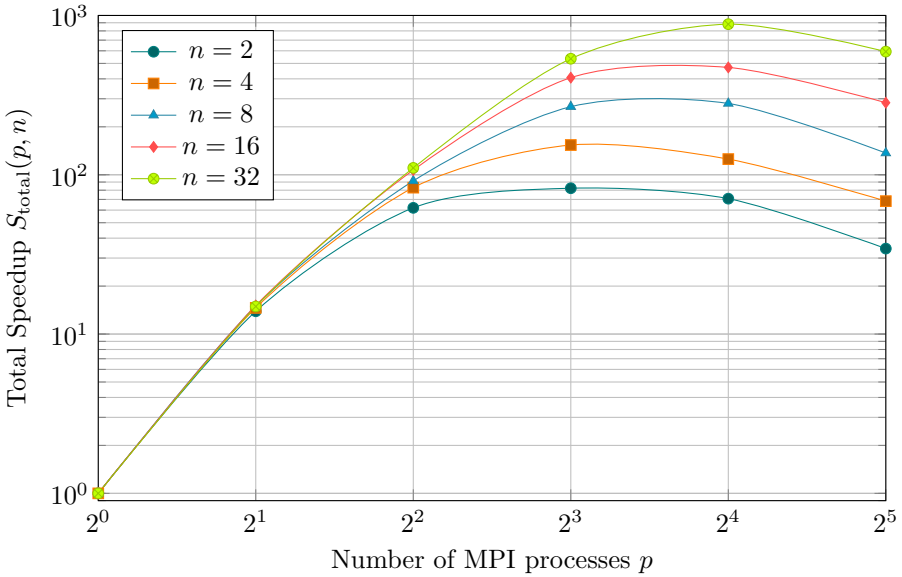

Figure 3: Measured **Total Speedup** $S_{\text{total}}(p, n)$ vs number of MPI processes for each matrix width n.

Discussion and interpretation

Total speedup shape. The total speedup curves initially grow with the number of MPI processes. For small matrix widths ($n = 2, 4$), speedup peaks at moderate process counts (8–16) and then decreases for more processes, showing a non-ideal scaling. For larger widths ($n = 16, 32$), speedup continues to grow with more processes up to 16, but eventually saturates or slightly declines.

Non-ideal behaviors.

- **Communication overhead:** For tall-skinny matrices and many processes, the cost of MPI communication can dominate, reducing scaling efficiency.
- **Load imbalance:** When the number of rows per process becomes very small, some processes are idle while others compute.

Conclusion

TSQR parallelization on tall-skinny matrices is highly effective for moderate process counts, with very large speedups observed for larger matrices. Saturation and drop in speedup for very high process counts are primarily due to communication overhead.