Efficient Task-Graph Scheduling for Parallel QR Factorization in SLSQP

Soumyajit Chatterjee[1] Rahul Utkoor[2] Uppu Eshwar[1] **Sathya Peri[1]** V. K. Nandivada[3]

[1]Indian Institute of Technology Hyderabad
[2]QUALCOMM India Private Limited
[3]Indian Institute of Technology Madras

Agenda

Agenda

Talk Roadmap

1. Motivation: Optimization in Practice
2. Why Focus on QR Factorization?
3. The Sequential QR Algorithm
4. A Naive Approach: Parallelism with Barriers
5. Problem Formulation: QR as a Task DAG
6. Research Question and Contributions
7. Our Solution: Asynchronous Two-Queue Scheduling

Motivation: Optimization in Practice

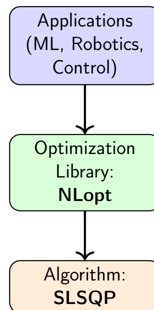
[1] Steven G. Johnson. *The NLopt nonlinear-optimization package*. https://nlopt.readthedocs.io/. See also https://github.com/stevengj/nlopt. 2007

[2] Jacob Williams. *SLSQP — Procedure Documentation*. https://jacobwilliams.github.io/slsqp/proc/slsqp.html. 2016–

Motivation: Optimization in Practice

Context

- Many applications in **ML, robotics, and scientific computing** rely on nonlinear optimization.
- **NLopt**[1] is a widely used library providing optimization algorithms.
- Among them, SLSQP [2](Sequential Least Squares Quadratic Programming) is popular for constrained problems.



[1] Steven G. Johnson. *The NLopt nonlinear-optimization package*. https://nlopt.readthedocs.io/. See also https://github.com/stevengj/nlopt. 2007

[2] Jacob Williams. *SLSQP — Procedure Documentation*. https://jacobwilliams.github.io/slsqp/proc/slsqp.html. 2016–

Why Focus on QR Factorization?

Why Focus on QR Factorization?

Runtime Cost

- Each **SLSQP iteration** requires solving a linearized subproblem.
- This involves a **QR factorization** of the constraint Jacobian.
- QR calls appear repeatedly, making them the **dominant runtime cost**.
- Naïve parallel implementations rely on **global barriers**, which stall threads and waste resources.

Why Focus on QR Factorization?

Runtime Cost

- Each **SLSQP iteration** requires solving a linearized subproblem.
- This involves a **QR factorization** of the constraint Jacobian.
- QR calls appear repeatedly, making them the **dominant runtime cost**.
- Naïve parallel implementations rely on **global barriers**, which stall threads and waste resources.

Opportunity

The QR factorization step is not only the **bottleneck** in SLSQP, but also a prime opportunity for optimization:

- It has a **structured task graph**.
- Dependencies are clear and exploitable.
- Unlocking fine-grained parallelism can **reduce barrier overheads**.

Research Question and Contributions

Research Question and Contributions

Our Research Question

- How can we efficiently parallelize **QR factorization** inside **SLSQP**?

Research Question and Contributions

Our Research Question

- How can we efficiently parallelize **QR factorization** inside **SLSQP**?

Goals

- Preserve correctness of factorization
- Reduce synchronization overhead
- Exploit task-graph scheduling for better parallelism

Research Question and Contributions

Our Research Question

- How can we efficiently parallelize **QR factorization** inside **SLSQP**?

Goals

- Preserve correctness of factorization
- Reduce synchronization overhead
- Exploit task-graph scheduling for better parallelism

Our Contributions

- Model QR factorization as a task DAG
- Design a barrier-free, asynchronous two-queue scheduler
- Demonstrate performance gains on multi-threaded environments
- Show real impact: faster SLSQP within **NLopt**

The Sequential QR Algorithm

The Sequential QR Algorithm

Triangular loop execution pattern

1: **Input:** A, a $m \times n$ non-singular real matrix.
2: **for** $i = 1$ **to** m **do**
3: $(up, b) \leftarrow$ UPDATE_PIVOT_ROW(A, i)
4: **for** $j = i + 1$ to n **do**
5: UPDATE_TRAILING_NON_PIVOT_ROW(A, i, j, up)
6: **end for**
7: **end for**
8: **Output:** Matrix A in upper-triangular form.

The Sequential QR Algorithm

Triangular loop execution pattern

1: **Input:** A, a $m \times n$ non-singular real matrix.
2: **for** $i = 1$ **to** m **do**
3: $(up, b) \leftarrow$ UPDATE_PIVOT_ROW(A, i)
4: **for** $j = i + 1$ to n **do**
5: UPDATE_TRAILING_NON_PIVOT_ROW(A, i, j, up)
6: **end for**
7: **end for**
8: **Output:** Matrix A in upper-triangular form.

Key Characteristics

This is the algorithm at the core of the SLSQP bottleneck. It performs QR factorization using two main computational kernels:

- `update_pivot_row`
- `update_trailing_non_pivot_row`

It has a classic nested-loop structure with a critical **data dependency** across iterations.

What Do The Kernels Actually Do?

What Do The Kernels Actually Do?

1. Pivot Kernel – Compute the Transformation

- Examines the current pivot row.
- Computes a transformation to introduce zeros above the diagonal.
- Updates the row in-place and outputs transformation parameters.

$$
\begin{matrix}
(d \quad c \quad g) & \xrightarrow{\texttt{update_pivot_row}} & (d' \quad 0 \quad 0) \\
(a \quad e \quad h) & & (a \quad e \quad h) \\
(b \quad c \quad f) & & (b \quad c \quad f)
\end{matrix}
$$

What Do The Kernels Actually Do?

1. Pivot Kernel – Compute the Transformation

- Examines the current pivot row.
- Computes a transformation to introduce zeros above the diagonal.
- Updates the row in-place and outputs transformation parameters.

$$\begin{pmatrix} d & c & g \end{pmatrix} \xrightarrow{\texttt{update_pivot_row}} \begin{pmatrix} d' & 0 & 0 \end{pmatrix}$$
$$\begin{pmatrix} a & e & h \end{pmatrix} \qquad \begin{pmatrix} a & e & h \end{pmatrix}$$
$$\begin{pmatrix} b & c & f \end{pmatrix} \qquad \begin{pmatrix} b & c & f \end{pmatrix}$$

2. Update Kernel – Apply the Transformation

- Receives parameters from the pivot kernel.
- Applies the transformation to each trailing row.
- Enables parallel execution across multiple rows.

$$\begin{pmatrix} d' & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} d' & 0 & 0 \end{pmatrix}$$
$$\begin{pmatrix} a & e & h \end{pmatrix} \xrightarrow[\texttt{update_trailing_non-pivot_row}]{} \begin{pmatrix} a' & e' & h' \end{pmatrix}$$
$$\begin{pmatrix} b & c & f \end{pmatrix} \longrightarrow \begin{pmatrix} b' & c' & f' \end{pmatrix}$$

What Do The Kernels Actually Do?

1. Pivot Kernel – Compute the Transformation

- Examines the current pivot row.
- Computes a transformation to introduce zeros above the diagonal.
- Updates the row in-place and outputs transformation parameters.

$$\begin{pmatrix} d & c & g \end{pmatrix} \xrightarrow{\texttt{update_pivot_row}} \begin{pmatrix} d' & 0 & 0 \end{pmatrix}$$
$$\begin{pmatrix} a & e & h \end{pmatrix} \qquad \begin{pmatrix} a & e & h \end{pmatrix}$$
$$\begin{pmatrix} b & c & f \end{pmatrix} \qquad \begin{pmatrix} b & c & f \end{pmatrix}$$

2. Update Kernel – Apply the Transformation

- Receives parameters from the pivot kernel.
- Applies the transformation to each trailing row.
- Enables parallel execution across multiple rows.

$$\begin{pmatrix} d' & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} d' & 0 & 0 \end{pmatrix}$$
$$\begin{pmatrix} a & e & h \end{pmatrix} \xrightarrow[\texttt{update_trailing_non-pivot_row}]{} \begin{pmatrix} a' & e' & h' \end{pmatrix}$$
$$\begin{pmatrix} b & c & f \end{pmatrix} \longrightarrow \begin{pmatrix} b' & c' & f' \end{pmatrix}$$

The pivot kernel computes transformation parameters, which the update kernel applies to construct the lower-triangular matrix required by SLSQP.

Example showing the working of QR Decomposition

Example showing the working of QR Decomposition

Iteration 1

$$\begin{matrix} v1 & v2 & v3 \\ v4 & v5 & v6 \\ v7 & v8 & v9 \end{matrix} \rightarrow \begin{matrix} \boxed{v1} & v2 & v3 \\ v4 & v5 & v6 \\ v7 & v8 & v9 \end{matrix} \rightarrow \begin{matrix} v1' & 0 & 0 \\ v4' & v5' & v6' \\ v7' & v8' & v9' \end{matrix}$$

Iteration 2

$$\begin{matrix} v1' & 0 & 0 \\ v4' & v5' & v6' \\ v7' & v8' & v9' \end{matrix} \rightarrow \begin{matrix} v1' & 0 & 0 \\ v4' & \boxed{v5'} & v6' \\ v7' & v8' & v9' \end{matrix} \rightarrow \begin{matrix} v1' & 0 & 0 \\ v4' & v5'' & 0 \\ v7' & v8'' & v9'' \end{matrix}$$

A Naive Approach: Parallelism with Barriers

A Naive Approach: Parallelism with Barriers

Barrier-Based Parallel Algorithm

1: **Input:** A, a $m \times n$ non-singular real matrix.
2: **for** $i = 1$ **to** m **do**
3: $(up, b) \leftarrow$ UPDATE_PIVOT_ROW(A, i)
4: — **BARRIER 1** —
5: **for** $j = i + 1$ to n **do** ▷ Parallel execution
6: UPDATE_TRAILING_NON_PIVOT_ROW(A, i, j, up)
7: **end for**
8: — **BARRIER 2** —
9: **end for**
10: **Output:** Matrix A in upper-triangular form.

List Scheduling with Barriers

List Scheduling with Barriers

The Barrier-Based Approach

- Model QR as a graph of tasks ($T_{i,j}$).
- **Barrier 1**: All threads wait after pivot task ($T_{i,i}$) is completed.
- **Barrier 2**: All threads wait after all update tasks ($T_{i,j}$) finish before the next iteration.

List Scheduling with Barriers

The Barrier-Based Approach

- Model QR as a graph of tasks ($T_{i,j}$).
- **Barrier 1**: All threads wait after pivot task ($T_{i,i}$) is completed.
- **Barrier 2**: All threads wait after all update tasks ($T_{i,j}$) finish before the next iteration.

Why Barriers Hurt Performance

- Barriers force strictly synchronous execution.
- Faster threads idle while the slowest thread finishes.
- The idle time reduces overall throughput.

List Scheduling with Barriers

The Barrier-Based Approach

- Model QR as a graph of tasks ($T_{i,j}$).
- **Barrier 1**: All threads wait after pivot task ($T_{i,i}$) is completed.
- **Barrier 2**: All threads wait after all update tasks ($T_{i,j}$) finish before the next iteration.

Why Barriers Hurt Performance

- Barriers force strictly synchronous execution.
- Faster threads idle while the slowest thread finishes.
- The idle time reduces overall throughput.

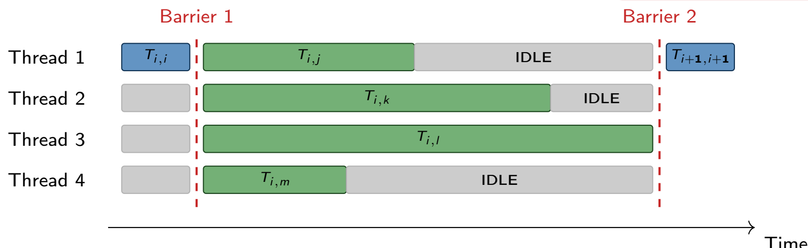


Fig : Global barriers introduce idle time and squander parallel resources.

Problem Formulation: QR as a Task DAG

Problem Formulation: QR as a Task DAG

Task & Dependency Model

The QR algorithm is modeled as a DAG of tasks, $T_{i,j}$:

- **Pivot Task ($T_{i,i}$):** Runs `update_pivot_row`.
- **Update Task ($T_{i,j}, j > i$):** Runs `update_trailing_non_pivot_row`.
- **Dependencies:** A task $T_{i,j}$ can only run after its parents, $T_{i,i}$ (pivot) and $T_{i-1,j}$ (previous row), are complete.

Problem Formulation: QR as a Task DAG

Task & Dependency Model

The QR algorithm is modeled as a DAG of tasks, $T_{i,j}$:

- **Pivot Task ($T_{i,i}$):** Runs `update_pivot_row`.
- **Update Task ($T_{i,j}, j > i$):** Runs `update_trailing_non_pivot_row`.
- **Dependencies:** A task $T_{i,j}$ can only run after its parents, $T_{i,i}$ (pivot) and $T_{i-1,j}$ (previous row), are complete.

Scheduling Goal

- Execute the task DAG in parallel.
- Respect all task dependencies.

Problem Formulation: QR as a Task DAG

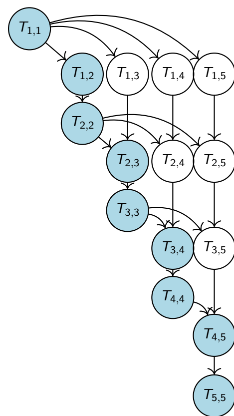
Task & Dependency Model

The QR algorithm is modeled as a DAG of tasks, $T_{i,j}$:

- **Pivot Task ($T_{i,i}$):** Runs `update_pivot_row`.
- **Update Task ($T_{i,j}, j > i$):** Runs `update_trailing_non_pivot_row`.
- **Dependencies:** A task $T_{i,j}$ can only run after its parents, $T_{i,i}$ (pivot) and $T_{i-1,j}$ (previous row), are complete.

Scheduling Goal

- Execute the task DAG in parallel.
- Respect all task dependencies.



TaskGraph for Triangular System

Our Solution: The Two-Queue Logic

Our Solution: The Two-Queue Logic

The Two-Queue Flow

The scheduler's logic is a simple loop:

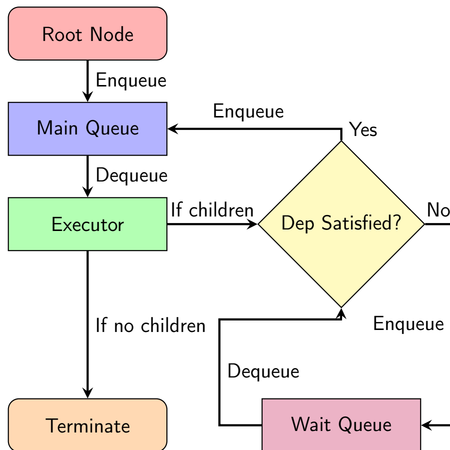
- A thread dequeues a task from the **Main Queue** and executes it.
- Then it checks if the dependencies are satisfied for its child tasks.
- **If YES:** The child task is enqueued in the main queue.
- **If NO:** The child task is enqueued to the **Wait Queue** to be re-checked again by any thread.

Our Solution: The Two-Queue Logic

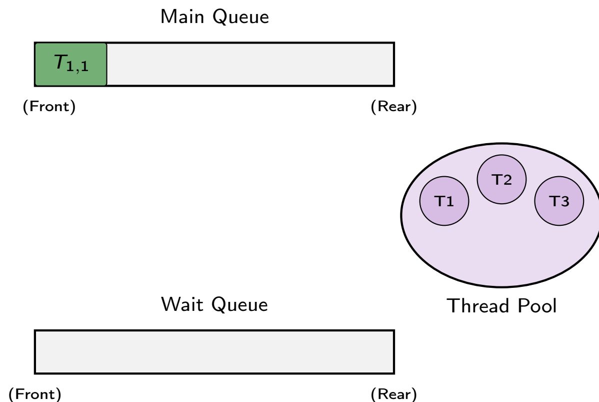
The Two-Queue Flow

The scheduler's logic is a simple loop:

- A thread dequeues a task from the **Main Queue** and executes it.
- Then it checks if the dependencies are satisfied for its child tasks.
- **If YES:** The child task is enqueued in the main queue.
- **If NO:** The child task is enqueued to the **Wait Queue** to be re-checked again by any thread.



Simulation: Execution & Spawning



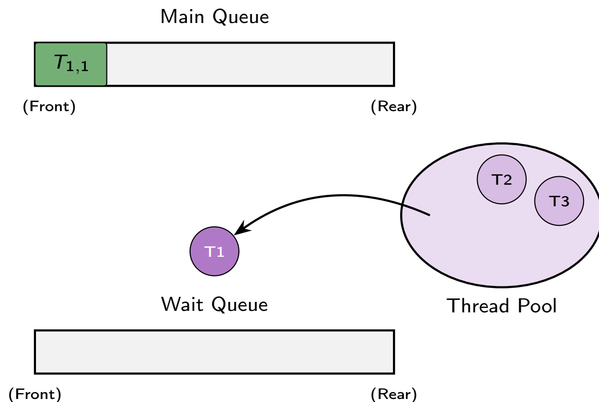
Step 1: Initial Task

The system starts with $T_{1,1}$ ready at the front of the Main Queue.

Simulation: Execution & Spawning

Step 2: Dequeue & Execute

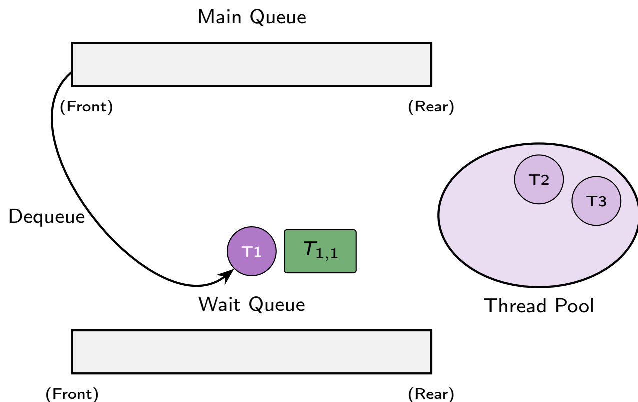
An available thread, **T1**, from the pool dequeues the task from the Main Queue.



Simulation: Execution & Spawning

Step 2: Dequeue & Execute

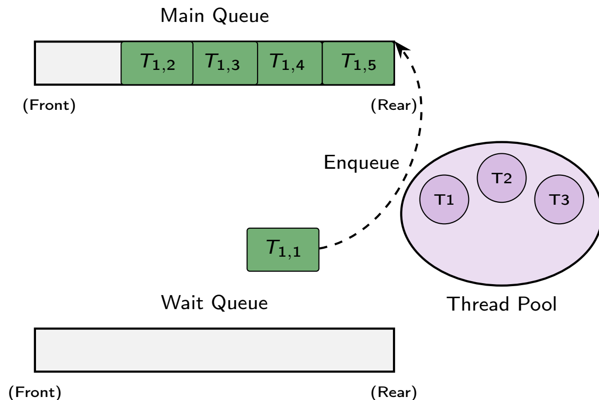
An available thread, **T1**, from the pool dequeues the task from the Main Queue.



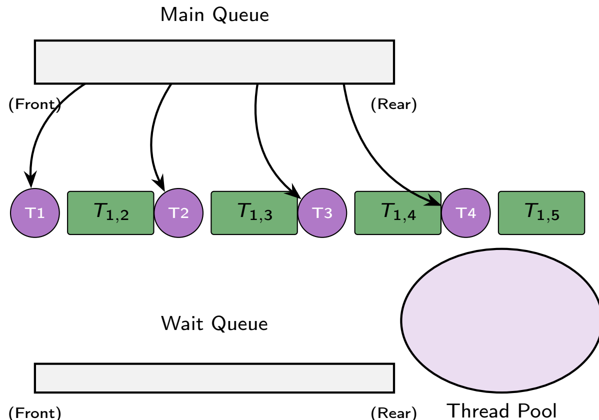
Simulation: Execution & Spawning

Step 3: Enqueue Children

After executing the task, T1 enqueues the ready child tasks (with dependencies satisfied) $T_{1,2}, \ldots$ to the Main Queue.

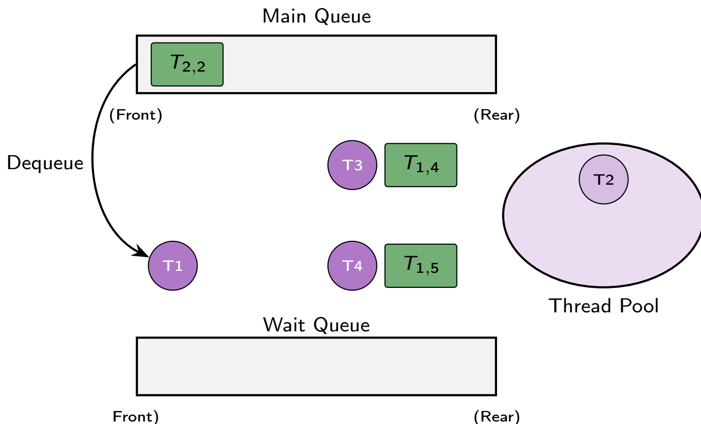


Simulation: Parallel Dequeue



Step 4: Peak Parallel Execution

Each thread independently dequeues a task, from the readily executable tasks from the main queue thereby achieving maximum parallelism.

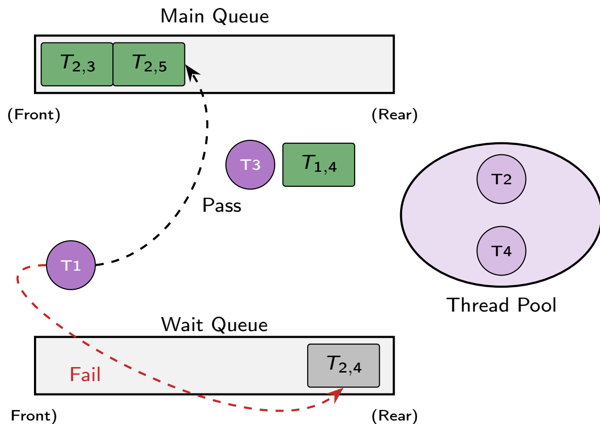


Mid-Execution Step

After executing $T_{2,2}$ its children's dependencies are checked and enqueued to the main queue if its dependency is satisfied or else its enqueued into the wait queue.

Step 5: Dependency Aware Enqueue Process

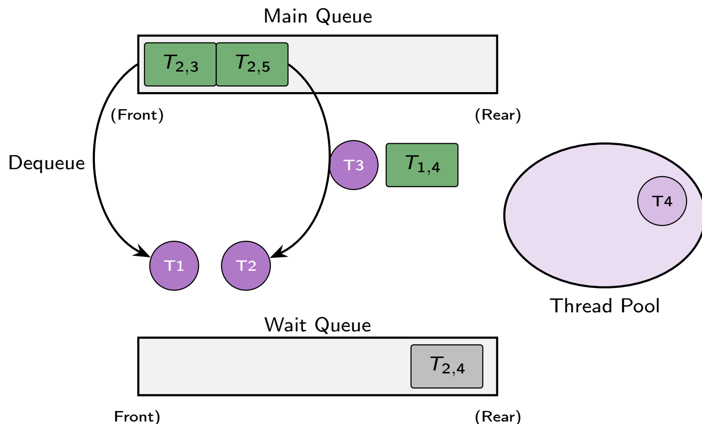
Thread **T3** is processing its task from the previous level. So the children task $T_{2,4}$ gets pushed to the wait queue as its parent task is incomplete.



Simulation: Navigating the Dependency Graph

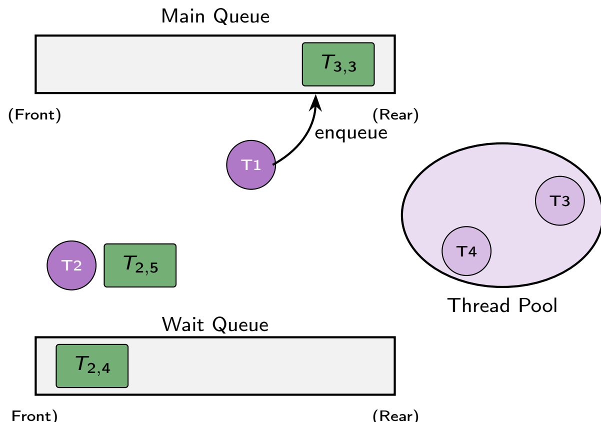
Step 6: Continued Execution with Readily Available Tasks

The remaining threads can pick up other immediately executable tasks from the main queue.



Step 7: Next Pivot Push

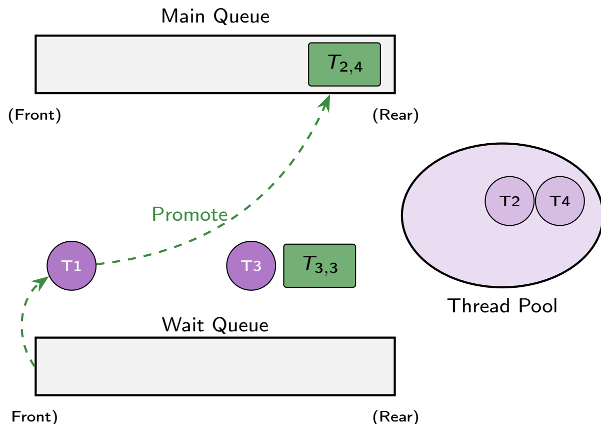
T1 finishes $T_{2,3}$ and enqueues its child, the new pivot $T_{3,3}$. Also, $T_{1,4}$ is complete in the mean time so its child $T_{2,4}$ can be safely pushed to the main queue.



Simulation: Navigating the Dependency Graph

Step 8: Promotion Admist Work

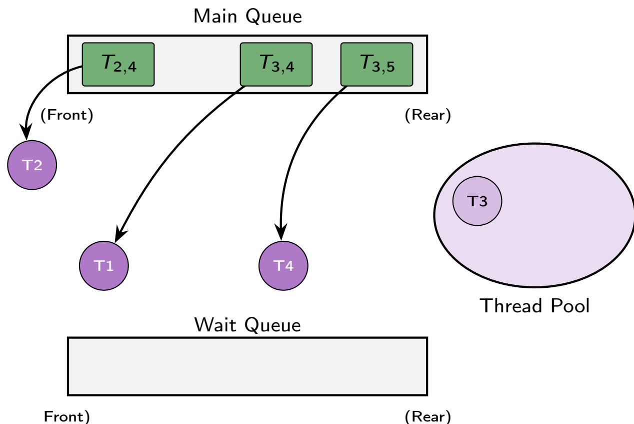
Since, the parent of $T_{2,4}$ was complete, $T_{2,4}$ got promoted from the wait queue to the main queue as an immediately executable task.



Simulation: Navigating the Dependency Graph

Step 9: Further Execution with Immediately Executable Tasks

The threads can pick up more executable tasks from the main queue and continue executing the nodes of the DAG in a non-blocking manner.



Experimental Setup & Configurations

Experimental Setup & Configurations

The Goal of Our Experiments

We conducted a series of experiments to answer three key questions:

- How do we find the optimal task granularity (α, β)?
- How does our two-queue scheduler scale compared to the barrier scheduling?
- What is the final, end-to-end impact on the SLSQP solver?

Experimental Setup & Configurations

The Goal of Our Experiments

We conducted a series of experiments to answer three key questions:

- How do we find the optimal task granularity (α, β)?
- How does our two-queue scheduler scale compared to the barrier scheduling?
- What is the final, end-to-end impact on the SLSQP solver?

Implementation & Configurations

We used Intel Threading Building Blocks (TBB) library for scheduling.

We evaluate two primary configurations of our scheduler:

- **Without Priority:** Uses TBB's "concurrent_queue". Tasks are processed in a first-in, first-out (FIFO) order.
- **With Priority:** Uses TBB's "concurrent_priority_queue". Tasks on the critical path are assigned higher priority to be executed sooner.

Tuning Task Granularity with α and β

Tuning Task Granularity with α and β

The Problem: High Scheduling Overhead

The original task DAG contains many small, fine-grained tasks. Managing these individually creates significant scheduling overhead.

Tuning Task Granularity with α and β

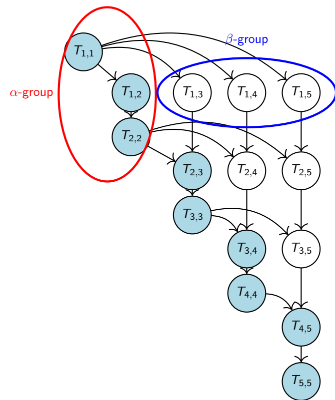
The Problem: High Scheduling Overhead

The original task DAG contains many small, fine-grained tasks. Managing these individually creates significant scheduling overhead.

Our Solution: Coalescing Tasks

We introduce two control parameters to group the fine-grained tasks into larger, more efficient chunks of work:

- α **(Alpha):** Controls the number of pivot computations that are performed as a single batch.
- β **(Beta):** Determines how many trailing rows are grouped together for a thread to update simultaneously.



α**-group:** Groups **dependent** tasks along the critical path.
β**-group:** Groups tasks that can be **parallelized**.

Priority Scheduling & Performance Trade-offs

[3] Muthu Manikandan Baskaran et al. "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors". In: *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 219–228. doi: 10.1145/1504176.1504209

Priority Scheduling & Performance Trade-offs

Prioritizing the Critical Path

To further improve performance, we can execute more important tasks first.

- Inspired by the work of **Baskaran et al.**[3], we assign each task a priority.
- The priority is based on the task's **Bottom Level** ('bottomL')—the longest path of work remaining from that task.
- This forces tasks on the critical path to be scheduled sooner, unlocking more parallelism.

[3] Muthu Manikandan Baskaran et al. "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors". In: *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 219–228. doi: 10.1145/1504176.1504209

Priority Scheduling & Performance Trade-offs

Prioritizing the Critical Path

To further improve performance, we can execute more important tasks first.

- Inspired by the work of **Baskaran et al.**[3], we assign each task a priority.
- The priority is based on the task's **Bottom Level** ('bottomL')—the longest path of work remaining from that task.
- This forces tasks on the **critical path** to be scheduled sooner, unlocking more parallelism.

The Trade-off: Granularity vs. Overhead

The effectiveness of the priority queue depends on the task size (α, β):

- **Many small tasks:** High overhead from frequent queue rebalancing.
- **Fewer large tasks:** Low overhead, allowing the benefit of priority ordering to dominate.

[3] Muthu Manikandan Baskaran et al. "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors". In: *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 219–228. doi: 10.1145/1504176.1504209

Tuning Results: Heatmaps

Tuning Results: Heatmaps

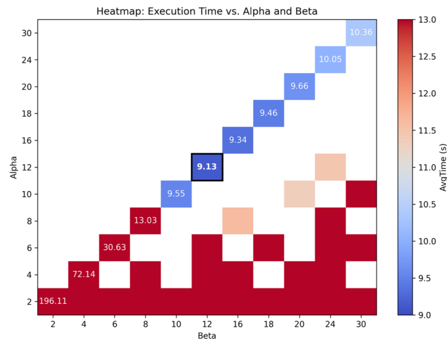


Figure: **Without Priority:** Optimal performance with smaller task chunks, at $(\alpha, \beta) \approx (12, 12)$.

Tuning Results: Heatmaps

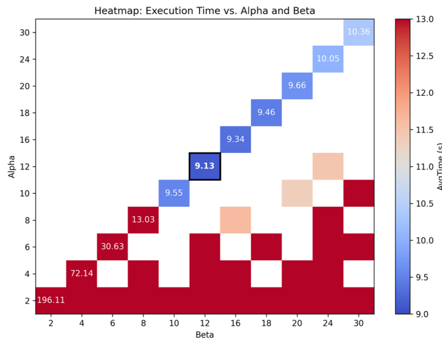


Figure: **Without Priority:** Optimal performance with smaller task chunks, at $(\alpha, \beta) \approx (12, 12)$.

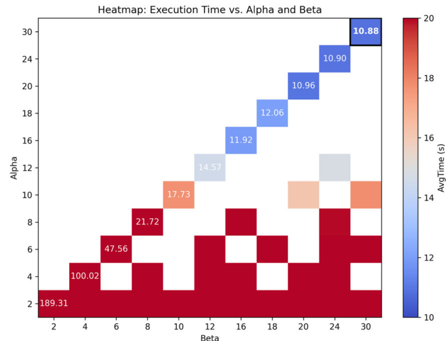
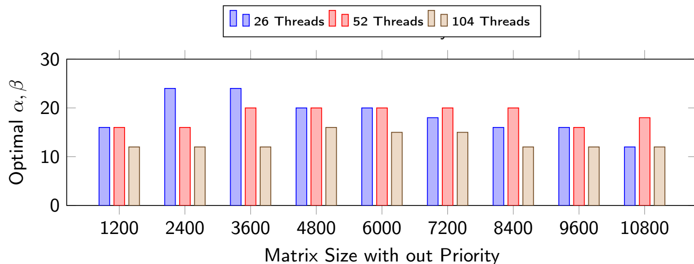


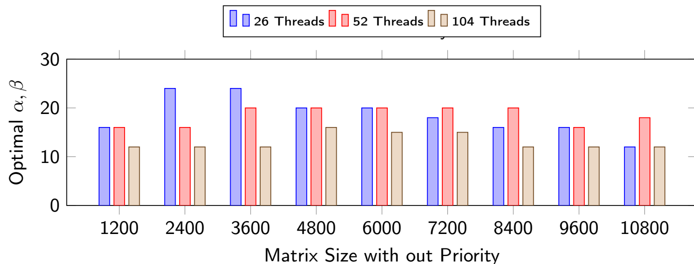
Figure: **With Priority:** Optimal performance with larger task chunks, at $(\alpha, \beta) \approx (30, 30)$, to reduce priority queue overhead.

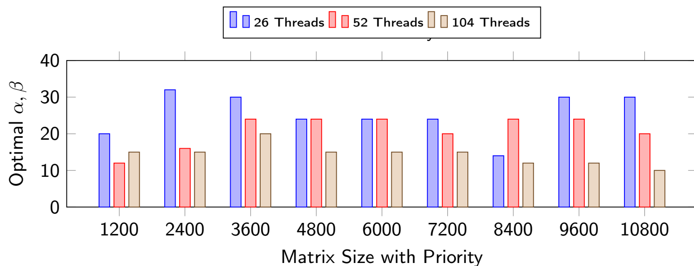
Tuning Results: Optimal Values vs. Matrix Size

Tuning Results: Optimal Values vs. Matrix Size



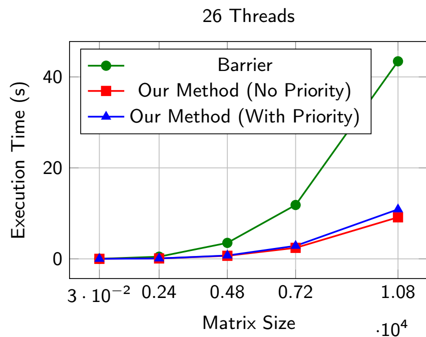
Tuning Results: Optimal Values vs. Matrix Size



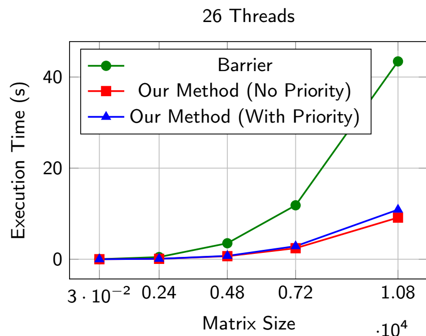


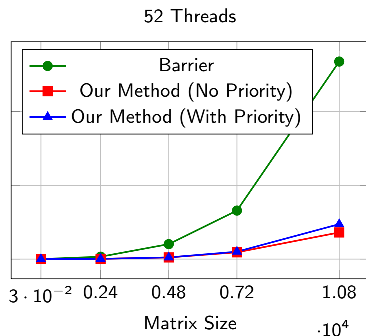
Results: Scalability with Matrix Size

Results: Scalability with Matrix Size

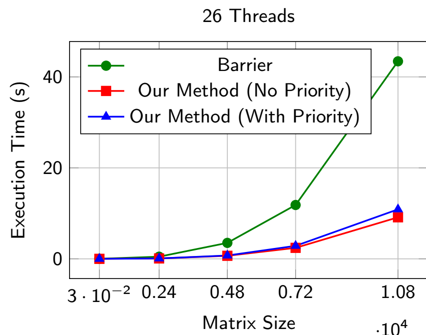


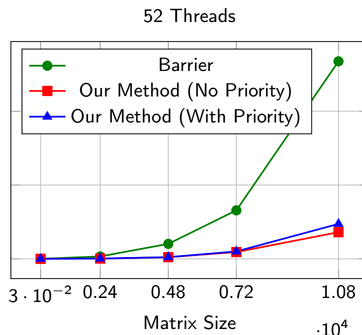
Results: Scalability with Matrix Size





Results: Scalability with Matrix Size



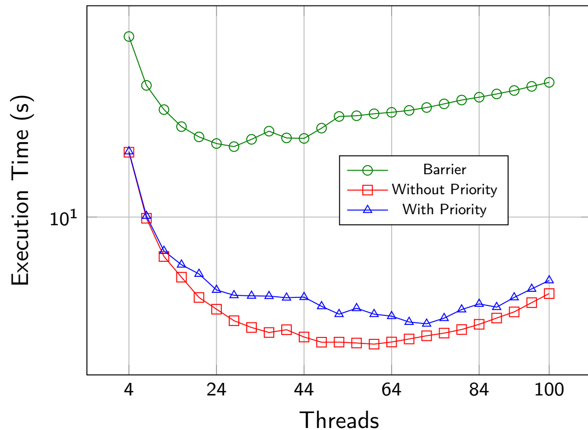


Key Finding: Two-Queue Scheduler Outperforms Barriers

Both configurations of our scheduler are significantly faster than the baseline barrier-based approach. The performance gap widens as the matrix size and thread count increase.

Results: Throughput Evaluation

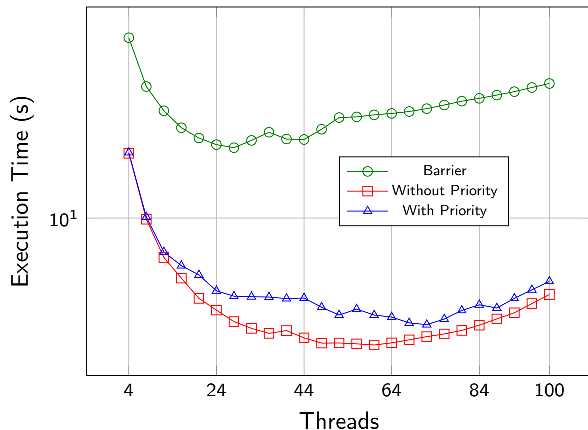
Results: Throughput Evaluation



The Experiment

We measure execution time on a fixed-size matrix (8192×8192) while increasing the number of threads.

Results: Throughput Evaluation



The Experiment

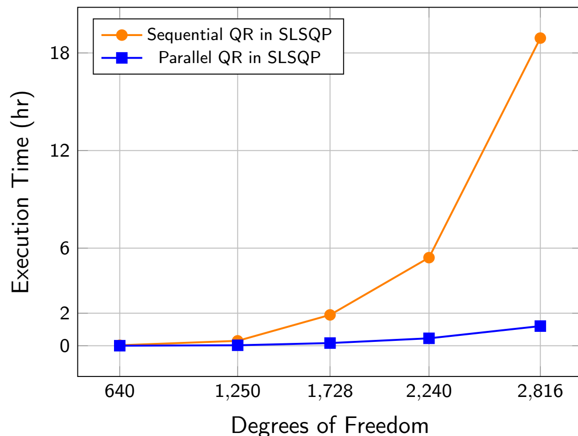
We measure execution time on a fixed-size matrix (8192×8192) while increasing the number of threads.

Key Findings

- Our two-queue schedulers show strong scaling, with performance improving significantly as threads are added.
- The barrier-based method scales poorly due to synchronization overhead, becoming slower after an initial improvement.

Results: End-to-End Impact on SLSQP

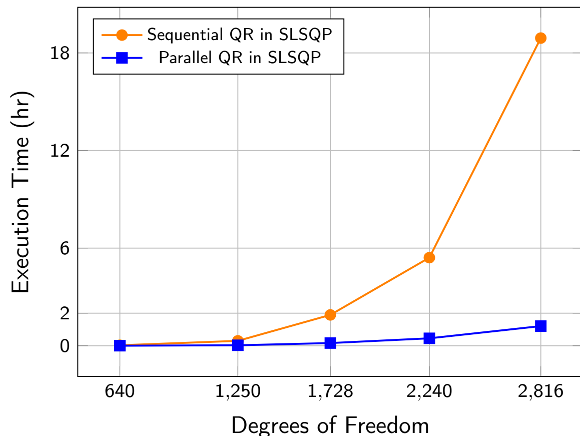
Results: End-to-End Impact on SLSQP



Context

- We integrated our parallel QR into the full SLSQP solver.
- **DOF (Degrees of Freedom):** Total problem variables. More DOF = larger matrix.

Results: End-to-End Impact on SLSQP



Context

- We integrated our parallel QR into the full SLSQP solver.
- **DOF (Degrees of Freedom):** Total problem variables. More DOF = larger matrix.

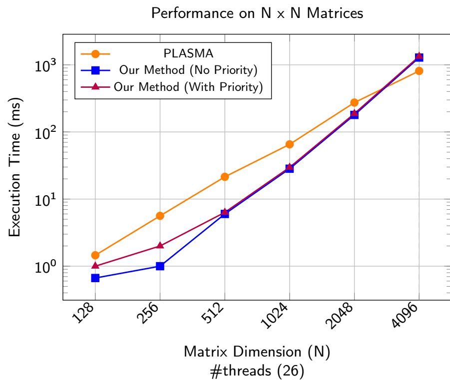
Final Result

At 2,816 DOF:

- Sequential: **18.9 hours**
- Parallel: **1.2 hours**
- **> 15x Speedup**

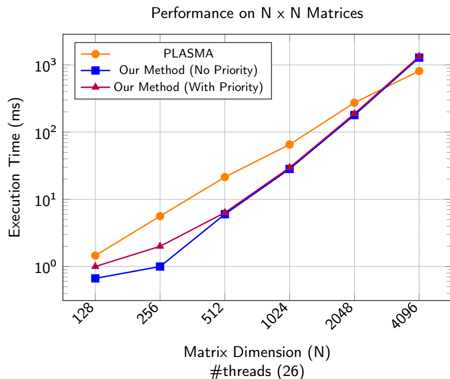
[4] Jack Dongarra et al. "PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP". In: *ACM Transactions on Mathematical Software* 45.2 (2019). doi: 10.1145/3264491

SOTA - Plasma Comparison



[4] Jack Dongarra et al. "PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP". In: *ACM Transactions on Mathematical Software* 45.2 (2019). doi: 10.1145/3264491

SOTA - Plasma Comparison



Key Takeaways

- Our approach achieves a **2–3× speedup** over PLASMA[4] (current SOTA) for sizes 128×128 to 2048×2048.
- Relies only on **regular compiler optimizations** and our scheduling logic.
- PLASMA depends on **BLAS routines**, which are hand-optimized using intrinsics.
- Performance drops for >2048, but the relevant range (up to 2048) directly matches **structural engineering and non-linear programming workloads**, where solver speed is a critical bottleneck.

[4] Jack Dongarra et al. "PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP". In: *ACM Transactions on Mathematical Software* 45.2 (2019). doi: 10.1145/3264491

[5] Alfredo Buttari et al. "Parallel tiled QR factorization for multicore architectures". In: *Concurrency and Computation: Practice and Experience* 20.13 (2008), pp. 1573–1590. doi: 10.1002/cpe.1301

[6] Muthu Manikandan Baskaran et al. "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors". In: *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 219–228. doi: 10.1145/1504176.1504209

Related Work & Our Contribution

Advances in Optimization Solvers

- Solvers like **IPOPT** and **SNOPT** are continuously evolving.
- NLOPT's **SLSQP**, while popular, has lacked recent performance updates.

Advances in Parallel QR Factorization

- Significant research exists on parallelizing QR, often using task-based DAG scheduling.
- Key works (Buttari et al.[5], Baskaran et al.[6] have explored tiled and dynamic scheduling methods

[5] Alfredo Buttari et al. "Parallel tiled QR factorization for multicore architectures". In: *Concurrency and Computation: Practice and Experience* 20.13 (2008), pp. 1573–1590. doi: 10.1002/cpe.1301

[6] Muthu Manikandan Baskaran et al. "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors". In: *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 219–228. doi: 10.1145/1504176.1504209

Related Work & Our Contribution

Advances in Optimization Solvers

- Solvers like **IPOPT** and **SNOPT** are continuously evolving.
- NLOPT's **SLSQP**, while popular, has lacked recent performance updates.

Advances in Parallel QR Factorization

- Significant research exists on parallelizing QR, often using task-based DAG scheduling.
- Key works (Buttari et al.[5], Baskaran et al.[6]have explored tiled and dynamic scheduling methods

The Gap & Our Contribution

- **The Challenge:** Achieving low-overhead, high-performance QR factorization for the **small-to-medium matrix sizes** common in iterative solvers.
- **Our Contribution:** Our lightweight scheduler is specifically designed for this scenario, delivering a **2-3x speedup** over mature libraries in this critical range.

[5] Alfredo Buttari et al. "Parallel tiled QR factorization for multicore architectures". In: *Concurrency and Computation: Practice and Experience* 20.13 (2008), pp. 1573–1590. doi: 10.1002/cpe.1301

[6] Muthu Manikandan Baskaran et al. "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors". In: *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 219–228. doi: 10.1145/1504176.1504209

Conclusion

Conclusion

What We Achieved

- Modeled the sequential QR algorithm as a task-based DAG to expose parallelism.
- Designed a novel, asynchronous two-queue scheduler to execute the DAG without barriers.
- Integrated our high-performance parallel QR into the NLOPT SLSQP solver.
- Demonstrated a **>15x end-to-end speedup** on large-scale problems.

Conclusion

What We Achieved

- Modeled the sequential QR algorithm as a task-based DAG to expose parallelism.
- Designed a novel, asynchronous two-queue scheduler to execute the DAG without barriers.
- Integrated our high-performance parallel QR into the NLOPT SLSQP solver.
- Demonstrated a **>15x end-to-end speedup** on large-scale problems.

Overall Impact

Our work delivers a lightweight, open-source scheduler that significantly accelerates SLSQP for the common, medium-sized workloads where performance is most critical.

Questions?

Team:
Soumyajit Chatterjee, Rahul Utkoor, Uppu Eshwar,
Sathya Peri, V.K. Nandivada

Artifact

github.com/PDCRL/ParSQP