Efficient Task Graph Scheduling for Parallel QR Factorization in SLSQP*

Soumyajit Chatterjee[1] ✉, Rahul Utkoor[2], Uppu Eshwar[1], Sathya Peri[1], and V.Krishna Nandivada[3]

[1] Indian Institute of Technology, Hyderabad
{ai22mtech02005@, ch21btech11034@, sathya_p@cse}.iith.ac.in
[2] QUALCOMM India Private Limited
rutkoor@qti.qualcomm.com
[3] Indian Institute of Technology, Madras
nvk@cse.iitm.ac.in

Abstract. Efficient task scheduling is paramount in parallel programming on multi-core architectures, where tasks are fundamental computational units. QR factorization is a critical sub-routine in Sequential Least Squares Quadratic Programming (SLSQP) for solving non-linear programming (NLP) problems. QR factorization decomposes a matrix into an orthogonal matrix Q and an upper triangular matrix R, which are essential for solving systems of linear equations arising from optimization problems. SLSQP uses an in-place version of QR factorization, which requires storing intermediate results for the next steps of the algorithm. Although DAG-based approaches for QR factorization are prevalent in the literature, they often lack control over the intermediate kernel results, providing only the final output matrices Q and R. This limitation is particularly challenging in SLSQP, where intermediate results of QR factorization are crucial for back-substitution logic at each iteration. Our work introduces novel scheduling techniques using a two-queue approach to execute the QR factorization kernel effectively. This approach, implemented in high-level C++ programming language, facilitates compiler optimizations and allows storing intermediate results required by back-substitution logic. Empirical evaluations demonstrate substantial performance gains, including a 10x improvement over the sequential QR version of the SLSQP algorithm.

Keywords: Non Linear Programming · Parallel Computing · DAG Scheduling.

1 Introduction

In modern engineering, the demand for efficient optimization techniques is critical across disciplines such as structural engineering, material sciences, and molecular dynamics. Nonlinear programming (NLP) has emerged as a fundamental

* GitHub Repository: https://github.com/PDCRL/ParSQP

tool for addressing complex design challenges, as these problems involve intricate, nonlinear relationships that govern system performance and reliability. In structural engineering, NLP facilitates the optimization of large-scale structures, while in material sciences, it enables the development of novel materials with tailored properties. Similarly, molecular dynamics relies on nonlinear equations to model particle interactions, requiring advanced optimization techniques to accurately capture complex behaviors. As the complexity and dimensionality of design spaces continue to expand, advanced computational methods are essential for achieving efficient and scalable solutions.

SLSQP: Sequential Least Squares Quadratic Programming [**?**] is a well established algorithm for solving NLP problems involving constrained, smoo-th, and differentiable functions. At each iteration, SLSQP constructs a quadratic approximation of the nonlinear objective function while employing a linearization of the constraints, resulting in a quadratic programming (QP) subproblem that determines the search direction for updating decision variables. Despite its effectiveness, the sequential execution of core linear algebra operations poses computational challenges, particularly in high-dimensional optimization problems. As problem dimensionality increases, these sequential computations introduce performance bottlenecks, necessitating the exploration of more efficient and scalable approaches[**?**][**?**][**?**].

QR Factorization: Enhancing algorithm efficiency is key for progress in optimization driven fields. The QR Factorization is a mathematical technique used to decompose a matrix A into an orthogonal matrix Q and an upper triangular matrix R. Methods like Householder transformations achieve this by iteratively applying a sequence of reflection matrices H_k to A. Each H_k is constructed from the current state of the k-th column. The *critical intermediate results* of this process are the components defining these Householder reflectors. The final R matrix resides in the upper triangle, while Q is implicitly represented as the product of the H_k transformations ($Q = H_1 H_2 \dots H_m$). In algorithms like SLSQP, which iteratively solve systems of equations or least-squares problems arising from Quadratic Programming (QP) subproblems, these stored intermediate results are paramount. They allow for the efficient application of Q or Q^T to various matrices and vectors without explicitly forming the (potentially dense) matrix Q. This repeated application is fundamental to updating solutions and Lagrange multipliers within SLSQP. QR Factorization is a critical sub-routine of SLSQP that is invoked multiple times, making the management and parallel computation of these intermediate results crucial for overall algorithmic performance.

This study presents parallel techniques for QR factorization that harness the advantages of concurrent task execution. By decomposing QR factorization into smaller, independent tasks suitable for parallel processing, an asynchronous Directed Acyclic Task Graph (DATG) scheduling mechanism is employed to optimize execution while preserving task dependencies.

Empirical evaluations demonstrate substantial performance gains in solving large-scale NLP problems using the SLSQP algorithm. The results underscore

the transformative impact of parallel computing techniques on QR factorization, reinforcing the necessity for high-performance numerical methods within open-source optimization frameworks.

Our Contributions: The key contributions of this work are as follows.

- Developed a dynamic algorithm that schedules QR factorization into smaller tasks using asynchronous DATG schedulingAlg. 5.
- Integrated the optimized parallel QR factorization method into the SLSQP implementation of the open-source NLOPT library.
- Comprehensive evaluations demonstrated a 10x improvement of the parallel QR technique over the sequential QR version of the SLSQP algorithm.

2 Background

2.1 SLSQP → Descent Direction Computation → QR

The Sequential Least Squares Programming (SLSQP) problem is formulated as a constrained optimization problem, typically expressed in the following mathematical form:

$$\min_x f(x), \tag{1}$$

$$\text{subject to:} \quad h(x) = 0, \quad h : \mathbb{R}^n \to \mathbb{R}^m, \tag{2}$$

$$g(x) \leq 0, \quad g : \mathbb{R}^n \to \mathbb{R}^p. \tag{3}$$

SLSQP is an efficient algorithm for solving NLP problems subject to equality and inequality constraints. It seeks to find the minimum of a non-linear objective function while ensuring the satisfaction of constraints. The core of SLSQP involves iteratively approximating the solution using a line search approach to compute the descent direction.

SLSQP determines the descent direction by solving a QP sub-problem at each iteration. This QP is derived from the first and second derivatives, gradients and Hessians, of the Lagrangian function associated with the objective function and constraints. QR factorization plays a pivotal role in this process by providing an efficient method to solve the system of linear equations arising from the Karush-Kuhn-Tucker (KKT) conditions[**?**].

Using QR factorization, SLSQP ensures numerical stability and efficiency in solving the KKT system, thereby accelerating the computation of descent directions in constrained optimization problems.

2.2 QR Factorization using Householder Transformations

Given a matrix A of size $m \times n$, the goal is to compute an orthogonal matrix Q and an upper triangular matrix R such that: $A = QR$. The Householder[**?**] algorithm achieves this by iteratively constructing and applying reflection vectors to transform A into Q and R, either in-place or out-of-place. Considering the importance of QR Factorization in SLSQP, we next consider efficient ways to parallelize QR Factorization.

Algorithm 1 QR Factorization using Householder Reflections

Initialize: Set $Q = I$ (identity matrix) and $R = A$.
for each column index $j = 1$ to $\min(m, n)$ **do**
  Extract column vector $x = R[j : m, j]$.
  Compute $\alpha = -x_1 \cdot \|x\|$.
  Set $u = x + \alpha e_1$, where e_1 is the first standard basis vector.
  Normalize $u = u/\|u\|$.
  Compute Householder matrix $W = I - 2\frac{uu^T}{\|u\|^2}$.
  Apply transformation: $R \leftarrow WR$.
  Update $Q \leftarrow QW^T$.
end for
return Q, R satisfying $A = QR$.

3 Our Methodology: Efficient Parallel QR Factorization

Algorithm 1 outlines QR decomposition's mathematical logic via Householder reflections. This formulation is further expressed in Algorithm 2, representing the standard computational structure frequently employed in linear algebra kernels such as QR factorization, Cholesky decomposition, and LU decomposition. The SLSQP algorithm from the NLOPT library uses an in-place QR factorization technique based on Householder transformations.

Algorithm 2 represents the in-place transformation of matrix A into the upper triangular matrix R. The algorithm relies on two key computational kernels: 1) `update_pivot_row` and 2) `update_trailing_non_pivot_row`. Both kernels operate at the row level, updating individual elements with a linear time complexity, each involving a fixed number of arithmetic operations.

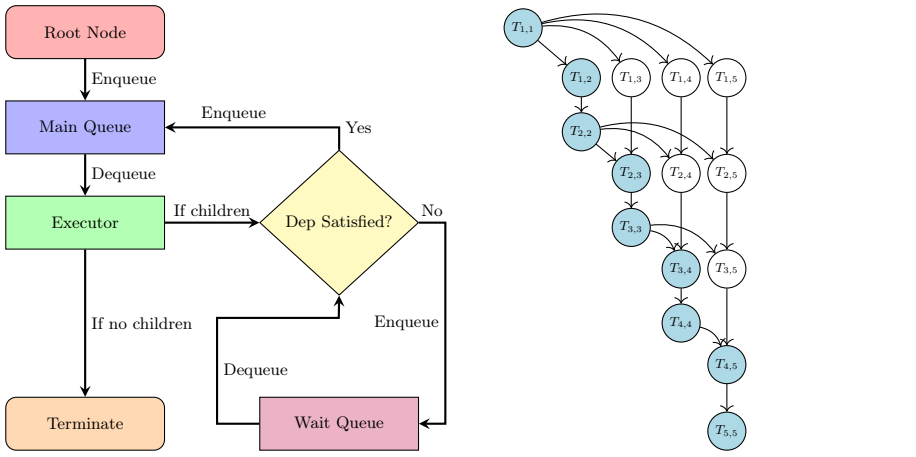
Algorithm 2 Transform matrix A to upper-triangular form.

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

In Algorithm 2, for a given value of i, $1 \leq i \leq m$, all tasks $T_{i,*}$ represent computations at the i^{th} iteration of the outer loop. The pivot update calculation of the i^{th} row is performed first (task $T_{i,i}$), representing a call to the kernel `update_pivot_row`. Then all the rows of the entire trailing sub-matrix, with $j > i$, are updated (task $T_{i,j}$), by a call to `update_trailing_non_pivot_row`.

To model the dependency constraints between tasks, we construct a directed acyclic graph (DAG) Fig. 1(b), where the vertices represent tasks, and the edges

encode dependencies. An edge $e : T \to T'$ indicates that T' can start only after T is completed, regardless of the availability of resources. Each $T_{i,j}$ task in Fig. 1(b) represents a call to the kernel `update_trailing_non_pivot_row`, which can be executed in parallel once its parent tasks are complete. Therefore, any task $T_{i,j}$ always has two parent nodes $T_{i,i}$ and $T_{i-1,j}$ (except for all tasks $T_{i,j}$ in the first level of Fig. 1(b) where it has only one parent) whose execution must be completed before the execution of task $T_{i,j}$ can begin. The dependency of $T_{i,j}$ on $T_{i,i}$ denotes the dependency within a single iteration according to Algorithm 2 where the pivot row update needs to be completed first before proceeding with the row updates of the non-pivot rows. However, the dependency of $T_{i,j}$ on $T_{i-i,j}$ denotes the dependency across iterations where a row in the given input matrix can only be updated in the current iteration if it had been successfully updated by a call to the kernel `update_trailing_non_pivot_row` in the previous iteration.



(a) Flowchart for Task Execution (b) TaskGraph for Triangular System

Fig. 1: Task Execution Flowchart and Task Graph

3.1 Optimizing Thread Workload for Parallel Task Execution

Even though all the tasks $T_{i,j}$ describe the availability of parallel tasks for the threads, however, the call to the kernel `update_trailing_non_pivot_row` only updates a single non-pivot row. We, therefore, want a mechanism to control the amount of work per thread. We introduce two control parameters α and β and design two new kernels Task 1 and Task 2, which allows us to increase/decrease the amount of work available per thread by coalescing smaller tasks into larger chunks. The parameter β determines how many non-pivot rows each thread updates simultaneously using the `update_trailing_non_pivot_row` kernel, rather

than processing one row at a time. The parameter α controls the number of iterations in which these β rows are updated once the pivot computations for the α rows are complete. Accordingly, Task 1 performs α pivot computations, enabling efficient batched updates of non-pivot rows in chunks of β (Task 2) over α iterations.

3.2 DAG Scheduling using Barriers

Given the task graph $G = (V, E)$ in Fig. 1(b), where V represents the set of nodes and E represents the set of directed edges. A directed edge (i, j) between two task nodes t_i and t_j indicates that t_i must be completed before t_j can commence. A straightforward approach to schedule the DAG in Fig. 1(b) across multiple processors while preserving the dependencies is to use **barriers**—a synchronization mechanism that ensures all threads reach a specific point before proceeding further. In Fig. 1(b), each task $T_{i,j}$ depends on both $T_{i,i}$ and $T_{i-1,j}$, necessitating two synchronization points:

Algorithm 3 Function: Task 1

```
1: Input:
     – Matrix mat of size m × n, pivot_start, row_chunk_start.
2: Global: global_up_array, global_b_array, α, β.
3: pivot_end ← pivot_start + α
4: row_chunk_end ← row_chunk_start + β
5: for lpivot = pivot_start to pivot_end − 1 do
6:     (up, b) ← UPDATE_PIVOT_ROW(mat, n, lpivot)
7:     if up or b is undefined then
8:         continue
9:     end if
10:    global_up_array[lpivot] ← up
11:    global_b_array[lpivot] ← b
12:    for j = lpivot + 1 to row_chunk_end − 1 do
13:        UPDATE_TRAILING_NON_PIVOT_ROW(mat, n, lpivot, j, up, b)
14:    end for
15: end for
16: Output: Updated matrix mat.
```

- A barrier after $T_{i,i}$ ensures that the `update_pivot_row` kernel completes before executing parallel tasks $T_{i,j}$.
- A second barrier at the end of each iteration ensures that all tasks $T_{i,j}$ complete before proceeding to the next iteration, as $T_{i+1,j}$ depends on $T_{i,j}$.

However, barriers impose a rigid execution order, limiting parallel efficiency. For instance, if $T_{i,j}$ belonging to a critical path completes, the next $T_{i+1,i+1}$ could begin execution immediately, which after completion can unleash more parallel tasks from the next level. Yet, due to barriers, all threads must wait for the slowest task to complete, even when additional work is available.

Algorithm 4 Function: Task 2

Input:
- Matrix `mat` of size $m \times n$, $pivot\_start$, row_chunk_start.

Global: `global_up_array`, `global_b_array`, α, β.
$pivot\_end \leftarrow pivot\_start + \alpha$
$row\_chunk\_end \leftarrow row\_chunk\_start + \beta$
for $lpivot = pivot\_start$ **to** $pivot\_end - 1$ **do**
    $up \leftarrow global\_up\_array[lpivot]$
    $b \leftarrow global\_b\_array[lpivot]$
    for $j = lpivot + 1$ **to** $row\_chunk\_end - 1$ **do**
        UPDATE_TRAILING_NON_PIVOT_ROW(mat, n, lpivot, j, up, b)
    end for
end for
Output: Updated matrix `mat`.

3.3 DAG Scheduling using LockFree Queues

A key observation from Fig. 1(b) is that traversing the critical path (highlighted nodes) allows more tasks $T_{i,j}$ to be executed in parallel across different levels. This approach reduces synchronization overhead by requiring only a single dependency check, specifically in $T_{i-1,j}$. Based on this insight, we propose a dual-queue scheduling mechanism for DAG execution: one queue handles the parallel generation of tasks, while the other ensures that dependencies are satisfied before execution.

As illustrated in Fig. 1(a), our approach leverages two centralized, lock-free global queues—`main_queue` and `wait_queue`—which are shared among all threads for task scheduling. The main queue contains tasks that any available thread can immediately execute. When a critical path task $T_{i,i}$ is completed, it loads its child tasks $T_{i,j}$ into the `main_queue` after verifying the completion of their parent $T_{i-1,j}$. If the parent has already completed the task, the child task is immediately available for execution. Otherwise, the task is placed in the wait queue, allowing threads to continue executing readily available tasks from the main queue instead of waiting for the pending parent task to complete. This strategy, as depicted in Algorithm 5 ensures that threads prioritize active execution over spinning or waiting for dependencies to resolve, thereby improving overall responsiveness. Upon completing a task from the main queue, a thread checks the wait queue for deferred tasks. If a task's parent has completed, it is moved to the main queue for immediate execution. Otherwise, it is enqueued back into the wait queue for re-evaluation in subsequent iterations.

3.4 DAG scheduling using Priority queues

In Baskaran's work [**?**], each vertex in the DAG is associated with two metrics: top level (`topL`) and bottom level (`bottomL`). For any vertex v in DAG G, the `topL(v)` is defined as the longest length of the path from the root node to the vertex v, excluding v.

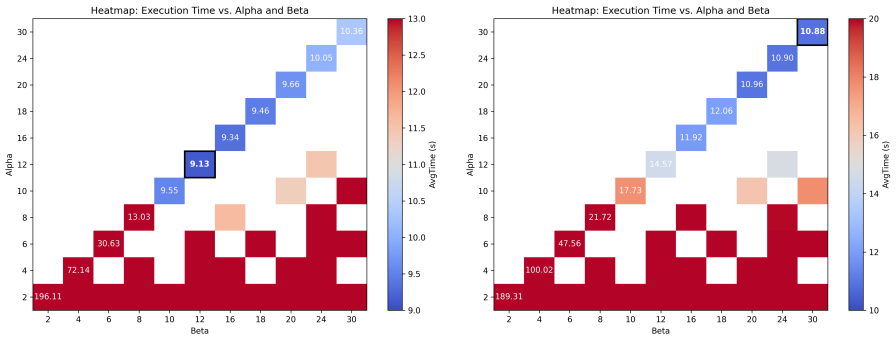
Algorithm 5 Thread Work

```
1: Global: lockfree main_queue, wait_queue; dependency_table tb
2: while true do
3:     if main_queue ≠ ∅ then
4:         curr_task ← main_queue.pop()
5:         if curr_task.type = 1 then
6:             Task1(curr_task.params)
7:             tb[curr_task] = True
8:             for child ∈ curr_task.children do
9:                 if ∀p ∈ child.parent, tb[p] = True then
10:                    main_queue.push(child)
11:                else
12:                    wait_queue.push(child)
13:                end if
14:            end for
15:        else if curr_task.type = 2 then
16:            Task2(curr_task.params)
17:            tb[curr_task] = True
18:            if curr_task ∈ CriticalPath then
19:                task1 = curr_task.children[0]
20:                main_queue.push(task1)
21:            end if
22:        end if
23:    end if
24:    if wait_queue ≠ ∅ then
25:        old_task ← wait_queue.pop()
26:        if ∀p ∈ old_task.parent, tb[p] = True then
27:            main_queue.push(old_task)
28:        else
29:            wait_queue.push(old_task)
30:        end if
31:    end if
32:    if ∃ task ∈ tb, tb[task] = False then
33:        continue
34:    else
35:        break
36:    end if
37: end while
```

Similarly, `bottomL(v)` is defined as the length of the longest path from v to the leaf node (vertex with no children). The tasks are prioritized based on the sum of `topL(v)` and `bottomL(v)` or just the `bottomL(v)`. Nodes that are part of the critical path will have higher priority, and as we move away from the critical path, the priority value of the nodes decreases. We use this technique to assign priority values to each node in the task graph, ensuring that critical-path tasks are executed earlier to accelerate the release of dependent parallel tasks.

Our proposed approach in Section 3.3 employs standard lock-free queues that execute DAG nodes without prioritization. Replacing them with global lock-free **priority queues** allows nodes to be ordered based on predefined criteria. This prioritization ensures that critical-path nodes execute earlier, thereby accelerating the release of dependent parallel tasks. However, maintaining priority order introduces overhead from rebalancing the data structure, which can degrade performance for large queues. Our implementation utilizes Intel TBB **concurrent priority queues** and **concurrent queues** to optimize task scheduling.

4 Experimental Results



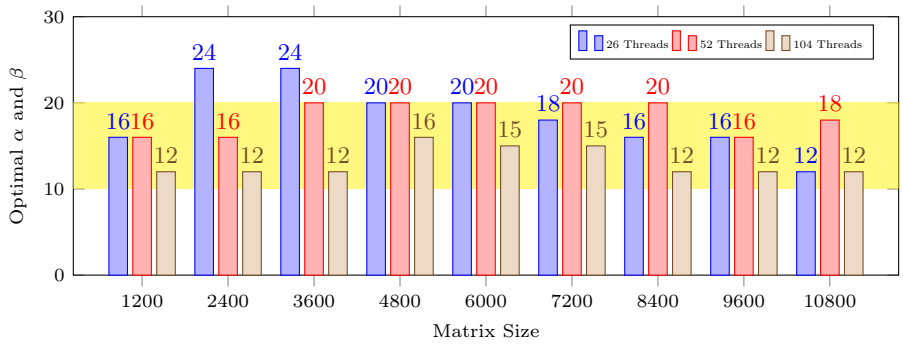
(a) Without Priority (b) With Priority

Fig. 2: Heatmap views for the parameter sweep.

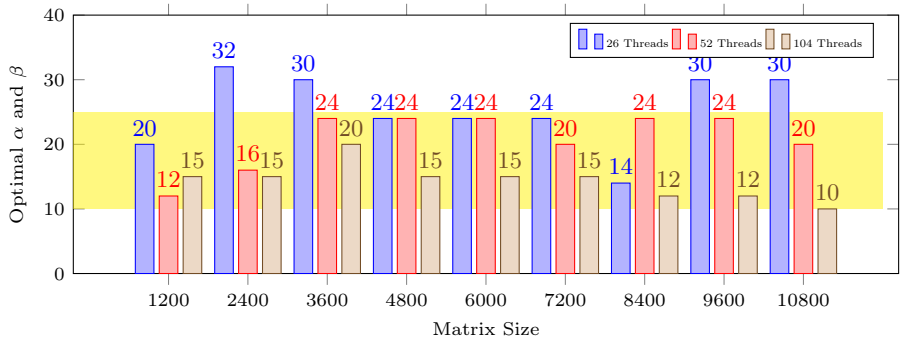
4.1 Parameter Tuning for Parallel QR Factorization

(a) Heatmap Analysis: In this experiment, an exhaustive sweep over the parameters α and β (ranging from 2 to 32) was conducted on a fixed matrix size of 10800×10800 using 26 threads. The primary objective was to minimize execution time. Figure 2 presents heatmap visualizations for two cases: without priority scheduling and with priority scheduling. The results reveal that the optimal configuration occurs when α equals β, with $\alpha = \beta = 12$ in the absence of priority scheduling and $\alpha = \beta = 30$ under priority-based scheduling.

(b) Bar Graph Analysis: Figure 3 illustrates the evolution of optimal $\alpha = \beta$ settings across various matrix sizes and thread counts (26, 52, and 104 threads), highlighting how computational load influences parameter tuning. The yellow highlighted band indicates the range in which the optimal parameter values consistently fall across different matrix sizes. A key insight from the results is that, as the number of threads increases, the variability in the optimal α–β values diminishes. This convergence suggests that, under higher parallelism, finer granularity (i.e., lower α and β) is preferred to maximize workload distribution and minimize execution time.



(a) Without Priority



(b) With Priority

Fig. 3: Bar graphs of best configurations across different matrix sizes.

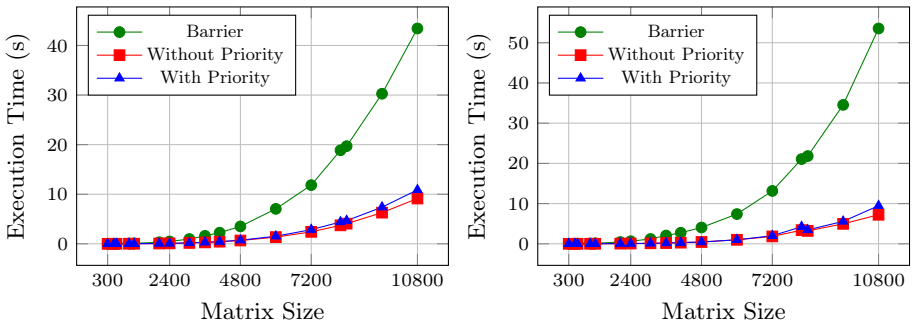
4.2 Scalability Analysis

In this experiment, we assesses the scalability of our proposed algorithm on dense square matrices with dimensions ranging from 300×300 to 10800×10800. The optimal α and β values determined in Experiment 4.1 were employed consis-

tently. To capture the impact of parallelism, tests were performed using 26 and 52 threads.

To assess the effectiveness of our approach, we compare three different methods: (i) a parallel DAG execution method that employs synchronization barriers and (ii) two variants of our proposed approach—one incorporating a priority-based scheduling mechanism and another without priority. This comparative analysis provides insights into the efficiency and scalability of the proposed methodology under varying computational loads.

Figure 4a (26 threads) and Figure 4b (52 threads) display the execution time trends as the matrix size increases. The results clearly demonstrate that both variants of our method (with and without priority scheduling) significantly outperform the barrier-based parallel DAG execution. However, the priority-free variant performs slightly better than priority-based due to the overhead introduced by priority queues. The additional overhead stems from the fact that priority queues reorder nodes according to their priority values and require more frequent data structure re-balancing.



(a) Exec. Time vs Matrix Size (26 Threads) (b) Exec. Time vs Matrix Size (52 Threads)

Fig. 4: Scalability comparison of the proposed algorithm for different matrix sizes.

4.3 Throughput Evaluation

In this experiment, we evaluate the throughput of various algorithms by incrementally increasing the number of threads (in multiples of 4) for a fixed matrix size of 8192×8192, while using the optimal α and β values from Experiment 4.1. Figure 5 plots the execution time (on a logarithmic scale) against thread count for three methods: barrier-based, without priority, and with priority scheduling.

The results align with the previous experiment, confirming that our proposed method (with and without priority) outperforms the parallel DAG execution with barriers. However, the priority-based variant is slightly less efficient due to its additional overhead.

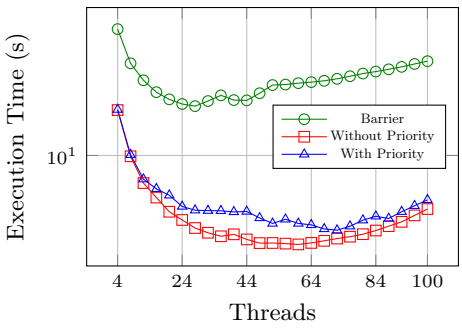


Fig. 5: Throughput Evaluation

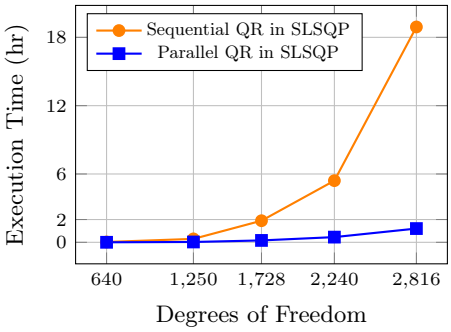


Fig. 6: SLSQP Performance

4.4 Impact of Parallel QR Factorization in SLSQP

In this experiment, we examine the influence of integrating our proposed parallel QR approach into the SLSQP framework within the NLOPT library. Our focus is on large-scale boundary value problems involving implicit constitutive relations under both elastic and inelastic responses.[?].

To assess performance, we compare a baseline SLSQP implementation equipped with sequential QR factorization against an SLSQP version that leverages our parallel QR factorization. We evaluated these approaches over a range of problem sizes defined by degrees of freedom (DOF) equal to 640, 1250, 1728, 2240, and 2816. Here, the term "degrees of freedom" denotes the total number of unknowns, such as nodal displacements, stresses, or auxiliary state variables, emerging from the discretizations of the governing partial differential equations and boundary conditions. Consequently, an increase in DOF results in a proportional increase in the dimension of the system matrix factorized by SLSQP.

Figure 6 shows the execution times (in hours) corresponding to each DOF level for both sequential and parallel QR implementations. Notably, when the DOF reaches 2816, the parallel QR variant completes its tasks in approximately 1.21 hours, whereas the sequential counterpart requires nearly 18.91 hours. This substantial improvement in computational efficiency becomes increasingly pronounced as the DOF grows, reflecting the strong scalability of the parallel approach.

5 Related Works

The NLOPT[?] library's SLSQP[?] algorithm is a recognized open-source, numerically stable solver for nonlinear optimization, valued for its customizability. However, unlike continuously evolving contemporaries such as IPOPT[?] (open-source) and SNOPT[?] (commercial), SLSQP has seen limited recent advancements. To address this, Joshy et al. introduced the PySLSQP[?] package, enhancing SLSQP's utility by bridging Python with the original Fortran code,

thereby facilitating easier modification and addressing limitations in current formulations, particularly those in NLOPT. Concurrently, significant progress has been made in parallel QR decomposition. Buttari et al.[**?**] introduced a Parallel Tiled QR factorization method using a DAG for scheduling. Baskaran et al.[**?**] contributed with compiler-assisted dynamic scheduling using lock-based priority queues, and building on this, Roshan et al. (Dathathri et al.)[**?**] developed dynamic scheduling techniques with lock-free priority queues for QR factorization. This research extends existing approaches by enhancing the NLOPT SLSQP solver through the integration of advanced asynchronous parallel algorithms. Departing from traditional synchronous strategies, this method incorporates novel parallel scheduling techniques inspired by the dynamic and lock-free approaches developed for QR factorization, aiming to improve both performance and scalability. For managing the interdependent tasks within QR decomposition, this work employs a variation of DAG scheduling, drawing upon established efficient DAG scheduling methodologies [**?**][**?**].

6 Conclusions and Future Work

This study integrates a highly parallel QR factorization into NLOPT's SLSQP routine by decomposing the factorization into numerous independent micro-tasks and coordinating them with an asynchronous, dependency-aware DAG scheduler. The resulting workflow markedly cuts computational overhead and achieves consistent speed-ups across a broad suite of benchmark problems, underscoring the value of fine-grained parallelism in nonlinear constrained optimization. In future work, we will substitute the classical Householder QR with a tiled implementation [**?**][**?**][**?**]. Tiling will expose even finer parallel granularity, facilitate NUMA-aware task placement, and enable closed-form selection of tuning parameters α and β, thereby eliminating costly parameter sweeps. Moreover, tiling naturally reduces queue-transfer contention between wait and execution queues by improving spatial locality and minimizing remote memory traffic. We further intend to generalize this tiling and scheduling strategy to the full suite of linear-algebra kernels invoked by SLSQP—such as rank-update, triangular solve, and Cholesky-related operations—ultimately delivering a solver that scales robustly on diverse multicore and many-core architectures while addressing increasingly complex optimization problems in modern science and engineering

7 Acknowledgment

We extend our deepest gratitude towards professor Dr. Saravanan, Dept. of Civil Engineering, Indian Institute of Technology, Madras, for providing us with the dataset for the experiment 4.5. We also thank the anonymous reviewers for their useful comments.

Disclosure of Interests. This work is partly funded by CRG 009391 & 001090, GoI (Government of India).