Dynamic Task Scheduling with Data Dependency Awareness Using Julia

Rabab Alomairy[1,2,4], Felipe Tome[1,3,5], Julian Samaroo[1,6], and Alan Edelman[1,7]

[1]Computer Science & Artificial Intelligence Laboratory,
Massachusetts Institute of Technology, USA.
[2]College of Computer Science and Engineering, University of Jeddah, KSA.
[3]São Carlos School of Engineering, University of São Paulo, BR.
[4]*rabab.alomairy@mit.edu* [5]*felipe0@mit.edu* [6]*jsamaroo@mit.edu* [7]*edelman@mit.edu*

***Abstract*—Dynamic task scheduling is vital for optimizing performance and resource utilization, particularly in heterogeneous computing environments. The `LLVM`-based `Julia` programming language offers a unique opportunity for developing efficient task-based runtime systems. This paper introduces the `Dagger.jl` package, a Julia-native implementation for dynamic task scheduling with data dependency awareness. We design a high-performance scheduler that leverages `Julia`'s type inference capabilities to support various computational tasks and data types. Our approach provides an unified API, facilitating the development and deployment of applications across different architectures. We evaluate the performance and overhead of `Dagger` through several tiled dense linear algebra computations on shared memory systems. Notably, our results show that `Dagger` with data dependency awareness outperforms other parallel paradigms in Julia and achieves performance comparable to vendor-optimized operations. `Dagger` also leverages the implementation of the QR communication-avoiding algorithm, delivering significant performance improvements, and highlighting its potential for scalable and efficient parallel computing.**

I. Introduction

Task-based programming models have emerged as a promising approach to harness the computational power of modern supercomputers. These models allow for the expression of parallelism at a fine-grained level, enabling dynamic scheduling and efficient resource utilization. They optimize the orchestration of computational resources, particularly enhancing efficiency in dense linear algebra algorithms. These algorithms, vital for solving large-scale problems in physics simulations, computational biology, and machine learning, are among the foundational *dwarfs* of high-performance computing.

Several task-based runtime systems have been developed over the past decade, including `Legion` [5], `Kokkos` [13], `OpenMP` [19], `StarPU` [4], `QUARK` [26], and `PaRSEC` [7]. While these systems offer significant performance improvements, they often rely on low-level programming languages like `C` and `C++`, which can complicate optimization efforts for developers and task scheduling interactions for users. Task-based programming options can also be found in `PyCOMPSs` [25] and `Pygion` [23]. These frameworks provide task interfaces with `Python`, but their runtime engines are implemented in Java and C++, which can complicate code production and force developers to manage multiple languages, a challenge faced by many simulation efforts today.

The `Julia` programming language, designed to bridge the gap between scientific computing and data science, offers a powerful platform for developing task-based runtime systems. `Julia`'s high-level syntax, combined with its performance capabilities, makes it an attractive alternative to traditional tools such as `MATLAB` and `Python`. `Julia`'s compact, readable code is well-suited for rapid prototyping, while its rich set of scientific libraries, including tools for numerical computation, data processing, and visualization, further enhances its appeal.

`Dagger.jl` [22] is a Julia-native implementation for dynamic task scheduling, simplifying the development of computational workflows in `Julia`, particularly for distributed systems with diverse accelerators. To our knowledge, `Dagger` is the first task-based engine written entirely in a high-level language, offering unprecedented flexibility and performance in task-based programming. It employs a sequential programming model, allowing users to write straightforward `Julia` code annotated for asynchronous parallel tasks. `Dagger` automatically detects concurrency, orchestrates data dependencies between tasks, and optimally distributes them across available resources, maximizing efficiency and performance. This paper demonstrates the potential of high-level languages, such as `Julia`, for implementing asynchronous runtime systems, paving the way for more flexible, scalable, and user-friendly parallel computing solutions.

We evaluated `Dagger` using dense linear algebra operations: Cholesky and QR factorization, and general matrix-matrix multiplication (GEMM). `Dagger` significantly reduces synchronous overhead compared to other threading mechanisms in `Julia`, achieving performance on par with vendor-optimized `BLAS`/`LAPACK` operations. Additionally, we introduced the Semi-Parallel Communication-Avoiding QR (SP-CAQR) [16], which minimizes synchronization in factorizing tall and skinny matrices. Our implementation surpasses `LAPACK` 's performance when the height of the matrix is doubled. This paper presents the first high-performance implementation of dense linear algebra in a dynamic language like `Julia`. While our benchmarks focus on shared-memory CPUs, `Dagger` supports GPU and distributed-

memory scheduling. All implementations are publically available in the Dagger.jl [1] package.

The remainder of the paper is structured as follows: Section II overviews task-based programming models and dense linear algebra computation. Section III highlights Julia's parallel computing ecosystem. Section IV introduces Julia's task-based scheduler and DArray data layout. Section VI evaluates Dagger's performance and overhead compared to other Julia's threading mechanisms and vendor operations.

II. Related Work

A. Task-based Programming Model

In the last decade, several engines have emerged to support task-based programming models, enabling scientific applications to scale on today's fastest supercomputers [9], [3]. Some of these engines use compiler-based technologies embedded in their parallel programming models, such as, to cite a few, Charm++ [21], Legion [5], and Kokkos [13]. There are also runtimes that utilize pragma-based tasking approaches, such as OmpSs [12], [1], which have eventually been integrated into the OpenMP standard specifications [18], and later in [19] with inter-task dependencies.

Moreover, many dynamic runtime systems rely on user-defined APIs, and make them portable across various software and hardware configurations, such as StarPU [4], QUARK [26], and ParalleX [17]. These systems are capable of scheduling complex Directed Acyclic Graphs (DAGs), whose nodes represent computational tasks whereas edges define the data dependencies, while leveraging applications' performance on multiple hardware systems. Additionally, task-based frameworks such as PaRSEC [7] offer a domain specific language for advanced users, supported by a customized compiler technology, to execute parameterized task graphs [10].

Task-based programming is no longer confined to C and C++; users of high-level languages such as Python can also benefit from asynchronous execution. PyCOMPSs [25], for instance, is designed to operate on top of the COMPSs Java runtime system, and Pygion [23] provides a Python interface to the C++ Legion task-based runtime. Despite offering high-level interfaces, their engines are written in low-level languages, which complicates both optimization efforts for developers and task scheduling interactions for users.

In contrast, the Dagger runtime system is natively implemented in Julia, marking a significant advancement in task-based engines. Julia's design uniquely combines the ease of use of high-level languages with the performance of low-level languages, thanks to its advanced type system and Just-In-Time (JIT) compilation via LLVM. This not only simplifies optimization but also enhances the accessibility of the task scheduling mechanism for users.

B. Tiled Dense Linear Algebra Computation

Scientific applications from diverse sources rely on dense matrix operations. These operations arise in: Schur complements, integral equations, covariances in spatial statistics, ridge regression, radial basis functions from unstructured meshes, and kernel matrices from machine learning, among others.

LAPACK and ScaLAPACK have been the de facto standard dense linear algebra libraries for decades, primarily due to their effective use of vendor-optimized BLAS libraries to achieve high performance. These libraries employ block algorithm approaches to parallelize computations, adhering to a Bulk Synchronous Parallel (BSP) paradigm. However, this approach introduces unnecessary synchronization points between the panel and update phases.

PLASMA [27] and recently SLATE [14] adopt tile computation techniques, which are crucial for architectures featuring high parallelism and deep memory hierarchies. SLATE advances tile formats by allocating distributed tiles with no correlation between their positions in the matrix and their memory locations. Similarly, Dagger adopts a strategy akin to SLATE by providing DArray, or distributed array format, which partitions a larger array into smaller "tiles" or "chunks" that may be located anywhere in the memory subsystem.

Dagger leverages DArray as the standard block format to efficiently manage tasks and transfer data across different computing resources. This enables highly performant tiled dense linear algebra computations within Julia's high-level programming environment. Notably, this marks the first instance of transferring tile computation from low-level languages, such as C and C++, used by PLASMA and SLATE, respectively, to a high-level language like Julia. This tile approach has now been successfully implemented in Julia, showcasing a significant advancement in making sophisticated computational techniques more accessible and efficient in modern programming languages.

III. Overview of Julia in HPC

Julia leverages the LLVM (Low-Level Virtual Machine) compiler infrastructure to achieve high performance and flexibility in its Just-In-Time (JIT) compilation approach. It combines the ease and expressiveness of high-level numerical computing languages like R, MATLAB, and Python with robust support for general programming. Features such as type inference and specialization, multiple dispatch, interoperability, and advanced optimizations like loop unrolling and vectorization enable Julia to achieve performance close to low-level native code [15], [20]. Type inference in Julia determines the types of variables and expressions, which allows LLVM to generate highly optimized machine code for specific types. Julia also supports intuitive polymorphic behavior, defining function behavior across diverse argument types to enhance flexibility, expressiveness, and performance.

Julia supports various parallel computing paradigms, including multithreading, distributed computing, and GPU acceleration. It provides built-in support for multi-threading with the *Threads* standard library. The @threads/@spawn macros simplify parallelizing loops across multiple threads, facilitating easy concurrency without explicit management of thread creation and synchronization. Julia's *Distributed* standard library enables distributed computing across multiple

[1] https://github.com/JuliaParallel/Dagger.jl

processes, either on a single machine or across a network of machines. This allows for scaling computations across different architectures. Julia integrates well with all major GPUs vendors through packages like `CUDA.jl`, `AMDGPU.jl`, `oneAPI.jl`, `Metal.jl` and `GPUArrays.jl` [6].

Each of those paradigms follow different parallelism approaches and employ different APIs, posing challenges in heterogeneous computing environments. This disparity has highlighted the need for a composable solution to unify the way tasks are coded and scheduled. To address this, `Dagger` has been developed to support asynchronous, fine-grained computations across heterogeneous execution environments. `Dagger` offers a unified framework for task scheduling and execution, making it easier to harness the full potential of `Julia`'s parallel capabilities across diverse hardware systems.

IV. Julia Task based Scheduler

`Julia` is a `LLVM`-based dynamic programming language, making it an excellent environment for designing native task-based runtime systems similar to the OpenMP standard used in Fortran, C, and C++. `Julia`'s `Dagger` is built on the principle of separating algorithmic design from hardware complexity, facilitating the deployment of software implementations on massively parallel systems. `Dagger` achieves this by providing `DArray` as a basic block for defining and managing data movement. It also manages task execution by creating `DTask` objects and scheduling tasks on designated hardware at runtime based on resource and function availability.

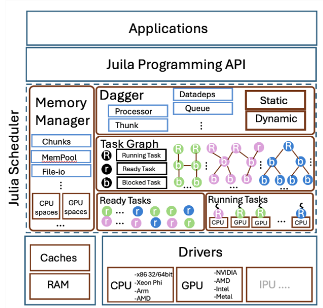


Fig. 1: Julia Dagger Software Stack.

A. DArray

`Julia` uses `Array` extensively due to their simplicity, storage efficiency, and broad functionality. However, parallel and heterogeneous `Array` operations are not fully supported, as operations like sum, reduce, and broadcasting do not utilize multi-threading even with multiple CPUs. `Dagger`'s `DArray` supports parallelism for nearly all operations and allows for type- and hardware-agnostic distributed arrays on both personal computers and large-scale supercomputers. It also enables `MPI` implementations of common array operations and supports UCX and InfiniBand network protocols.

To integrate with the numerical linear algebra ecosystem, `DArray` uses the `Blocks` data type (e.g. analogous to tile) to divide itself into `chunks` of its underlying array, whether it be a vector, matrix, or n-ranked tensor. Users can pass `Blocks` as an argument to the array generation function to create a `DArray` instead of a standard `Julia` array. For instance, `randn(10,10)` returns a 10×10 matrix of `Float64` numbers, while `randn(Blocks(2,2),10,10)` returns the same matrix divided into 22 `chunks`. `Blocks` can have user-specified dimensions or be an `AutoBlocks` type for automatic column-major distribution. Each `DArray` partition is treated as a task (`DTask`) with execution info, function, task IDs, and scheduling data. For example, generating a `DArray` and performing a broadcast sum retains references to the allocation function and summation step, stored in the chunks field for efficient task execution and tracking.

B. Dagger

`Dagger` is a powerful `Julia` package designed to simplify parallelization. It allows users to write `Dagger`-based applications in a manner as close as possible to sequential code. In this model, users focus on (i) identifying tasks for asynchronous execution, and (ii) annotating these tasks with data direction. Dagger analyzes this information and builds a task execution flow represented as a Directed Acyclic Graph (`DAG`), where nodes are computational tasks and edges are data dependencies. The `DAG`-based runtime system then effectively exposes concurrency, reduces synchronization points, ensures load balancing, shortens the critical path, and abstracts hardware complexity, similar to out-of-order task scheduling on superscalar processors.

Fig. 1 illustrates the software stack of `Dagger`. At the bottom layer lies the heterogeneous hardware environment, which includes various CPU architectures and GPU vendors, as well as potential support for IPUs. Dagger treats tasks as flexible units that can be scheduled on any available resource, based on kernel availability. The `Julia` scheduler layer is responsible for managing both the memory movements of `DArray chunks` and the scheduling and execution phases of tasks. The memory manager handles memory pools, file I/O, and the allocation of CPU and GPU spaces. The task graph component visually represents running, ready, and blocked tasks, facilitating efficient task management. `Dagger` abstracts the complexity of hardware and scheduling from the applications.

`Dagger` provides unified and user-friendly APIs to boost productivity and ensure code portability. These APIs allow users to focus on their applications without dealing with the intricacies of the underlying hardware and schedulers. In `Dagger`, tasks are created using the `@spawn` macro, which generates a `DTask` object and submits it to `Dagger`'s scheduler by calling the `enqueue` function. Task dependencies in `Dagger` are managed by chaining tasks based on their tasks dependencies. For example, if we define two tasks one to add two matrices A and B and store the results to B and another one to copy B to C as following:

```
A = rand(1000)
B = rand(1000)
C = zeros(1000)
```

```
function add!(B, A)
    B .= A .+ B
    return
end
B = Dagger.@spawn add!(B, A)
R = Dagger.@spawn copyto!(C, B)
```

In this case the second task depends on first task because it waits for the DTask holding B to be done before start executing its function. In this case Dagger queues tasks based on the task dependency and this is called Dagger TaskDeps. Dagger provides as well data awareness mechanism by chaining task graph according to it's data dependency as similar to OpenMP depend clause. The dependency direction can be either In for read dependencies, Out for write dependencies, and InOut for read and write dependencies. Therefore, above code can be rewritten as following:

```
A = rand(1000)
B = rand(1000)
C = zeros(1000)
function add!(B, A)
    B .= A .+ B
    return
end
Dagger.spawn_datadeps() do
    Dagger.@spawn add!(InOut(B), In(A))
    Dagger.@spawn copyto!(Out(C), In(B))
end
```

This mechanism is called Dagger Datadeps. Dagger tasks need to be enclosed within the spawn_datadeps function to define a Datadeps region, with parallelism controlled via dependencies. In the code above, the add task specifies that A is only being read from, while B is being read from and written to. Similarly, the copyto task specifies that B is being read from, and C is only being written to.

The recent development of Dagger introduced an eager dynamic scheduler that abstracts processing units, such as CPUs and GPUs, as workers and manages data movement based on task spawning. This approach allows tasks to execute immediately as soon as workers become available, without requiring synchronization (i.e. lazy execution). One of the latest efforts by the authors was to transition the DArray from using the legacy lazy scheduler to the new, more composable eager scheduler. Dagger efficiently manages the memory spaces of each argument and performs necessary data movement or copies to the appropriate workers.

V. Numerical Linear Algebra in Dagger

All the aforementioned infrastructure constitutes a robust toolset for numerical linear algebra. With Dagger's data-aware capabilities, it is possible to implement in-place, tiled, and parallel data decomposition algorithms with ease and high productivity. This is particularly effective for tiled algorithms such as Cholesky, GEMM, QR, and CAQR, which are extensively studied in [8], [16], [24], [14], [2]. In Dagger, we provide an implementation of those algorithms and use multiple dispatch to ensure seamless integration with Julia.

Dagger 's linear algebra implementation respects the APIs of Julia's LinearAlgebra.jl base ecosystem to ensure easy adoption. For the Cholesky factorization, users simply need to create a DArray and use it in the LinearAlgebra.cholesky API function. Julia's multiple dispatch will then automatically call the function designed for DArray instead of the one for Array. The LinearAlgebra.cholesky function will return an object containing the upper and lower factors.

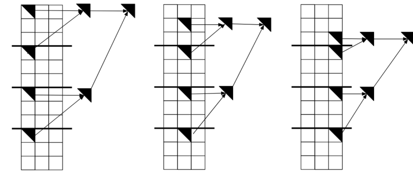


Fig. 2: Asynchronous merging phases of R for SP-CAQR

TABLE I: Hardware specifications.

Vendor	Intel	AMD
Family	Ice Lake	EPYC Milan
Model	6330	7713
Socket(s)	2	2
Cores per Socket	28	64
GHz	2	2
DDR4	1 TB	256GB
L3 Cache	84 MiB	256MiB

We implemented as well the Semi-Parallel Communication-Avoiding QR (SP-CAQR) algorithm, known for its efficiency with tall and skinny (TS) matrices [11]. SP-CAQR offers the advantage of increased asynchrony while maintaining good data locality. In this algorithm, the matrix is divided into subdomains, as illustrated in fig 2. Each subdomain performs local QR factorization independently. Subsequently, the triangular factors are merged in parallel using tree-like communication, enhancing parallelism for this matrix structure and mitigating the idleness observed in traditional tile QR algorithms, which require the entire panel to be factorized before proceeding. Julia's Base defaults all matrices of different shapes to the same underlying QR function. However, Dagger employs an implementation tailored to the matrix dimensions, leveraging Julia's multiple dispatch capability.

VI. Performance Results

In this section, we analyze the performance of the Dagger scheduler using three dense linear algebra operations: GEMM, and Cholesky and QR factorization.

A. Apparatus

We conducted our experiments using Julia (v1.10.4), Dagger (v0.18.12), and oneMKL (v2020.0.166). To avoid conflicts with BLAS multithreading, we set BLAS.set_num_threads(1). For BLAS and LAPACK from oneMKL, we BLAS.set_num_threads(n) to the available threads. The tile size for tiled computation was fixed at 2048. Results were obtained on Intel and AMD machines (see TABLE I). We also varied floating-point precisions to demonstrate Julia's type inference capabilities and the simplicity of the API.

B. Julia Parallel Paradigms

Analyzing the performance of Julia's parallel paradigms is crucial for advancing optimization in parallel scheduling. Fig. 3 shows incremental performance improvements of three Julia-based parallelization strategies for Cholesky decomposition: Threading, Dagger with TaskDeps, and Dagger with Datadeps. As matrix sizes increase, Dagger with DataDeps shows a speedup of approximately 2x over Julia's Threading. Dagger with Datadeps consistently achieves the best results, demonstrating a speedup of about 3x. This highlights the significant efficiency gains of Dagger, especially with Datadeps, for high-performance linear algebra.

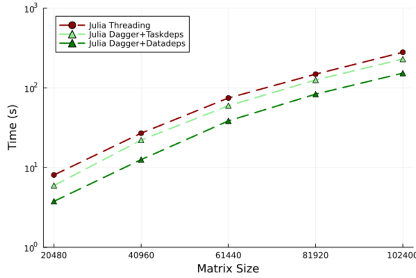


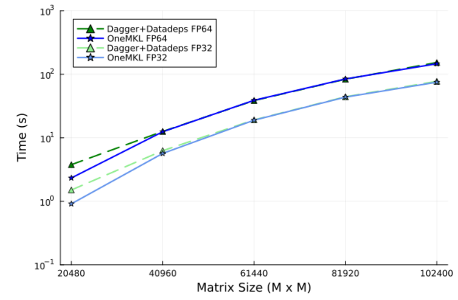
Fig. 3: Comparing Julia's parallel paradigms on Intel.

C. Performance Compared to Vendor Implementation

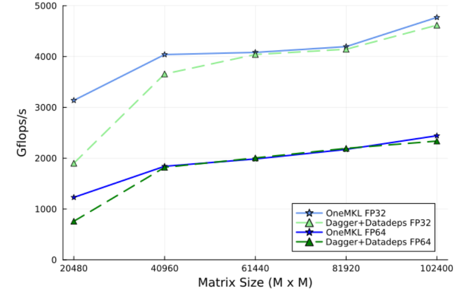
It is crucial to understand the performance of Dagger Datadeps compared to vendor optimized implementation. It schedules fine-grained computation as soon as their dependencies are satisfied. Fig. 4 shows the performance of Cholesky factorization in terms of time-to-solution (Fig. 4a) and GFLOP/s (Fig. 4b), compared to LAPACK implementation in oneMKL. For relatively small matrix sizes, Dagger exhibits a slight performance slowdown. However, as matrix sizes increase, Dagger DataDeps becomes more competitive, achieving performance levels closer to those of the vendor-optimized oneMKL. This suggests that Dagger Datadeps is a viable alternative for more computationally intensive tasks.

GEMM is a cornerstone of AI operations. For example, transformers, foundational to most large language models, heavily rely on this computation. Dagger utilizes a data-aware mechanism to achieve competitive performance comparable to oneMKL GEMM across various data types and different matrix shapes, as depicted in Fig. 5 and 6. This provides a robust foundation for further optimization considerations particularly in supporting fine-grained mixed-precision operations, as discussed in [9].

QR factorization is essential for solving linear systems, finding best-fit solutions, and computing eigenvalues. Fig. 7 illustrates performance of QR in Float64 using square matrices. It compares tiled QR using Dagger Datadeps to two QR kernels from oneMKL. GEQRT computes relies on triangular block reflectors factors, whereas GEQRF employs scalar reflectors of elementary reflectors. Dagger implements a tiled QR based on triangular block reflectors as GEQRT. It

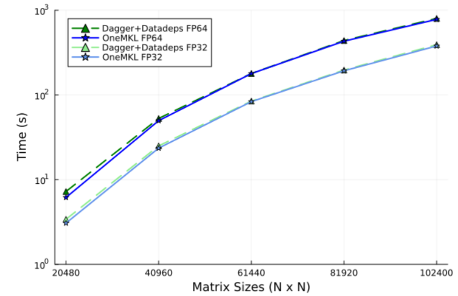


(a) Execution Time on Intel.

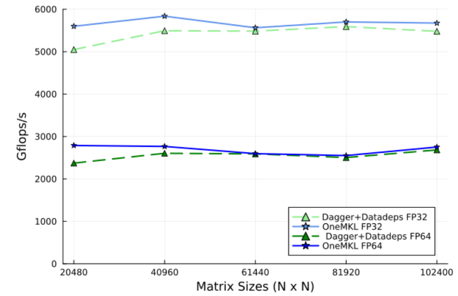


(b) Achieved GFLOP/s on Intel.

Fig. 4: Performance of Cholesky Factorization



(a) Execution Time on Intel.

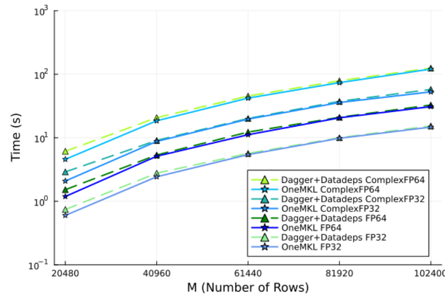


(b) Achieved GFLOP/s on Intel.

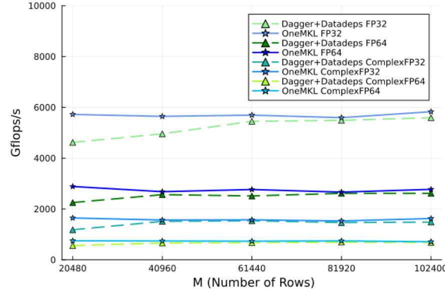
Fig. 5: Performance of GEMM using square matrices.

outperforms GEQRT from oneMKL by up to 1.5x and matches GEQRF's performance as the matrix size increases.

Fig 8 illustrates SP-CAQR's performance in Float64, comparing it with GEQRT and GEQRF from oneMKL. Dagger implementation with a single subdomain (SP1), execution defaults to regular tiled QR methods without communication avoidance. However, this approach does not yield optimal



(a) Execution Time on Intel.



(b) Achieved GFLOP/s on Intel.

Fig. 6: Performance of GEMM using tall and wide matrices. A of size $M \times 4096$, B of size $4096 \times M$, and C is square of size $M \times M$.

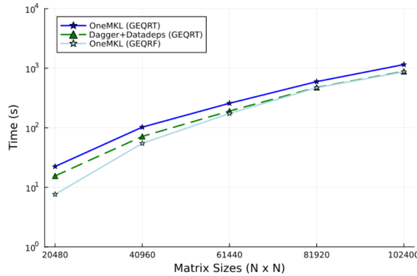


Fig. 7: Performance of QR Factorization using square matrices.

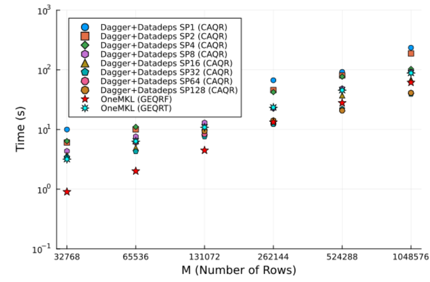


Fig. 8: Performance of SP-CAQR using TS Matrices.

performance for matrices of this shape. Regular tiled QR methods introduce heightened synchronization demands during the panel phase and the constrained BLAS-3 phase (trailing update), which is essential for achieving superior performance.

Dagger begins to exhibit superior performance as the number of subdomains increases, particularly beyond 32. For matrices with more rows than columns, Dagger SP-CAQR achieves lower latency and reduced bandwidth costs compared to the existing LAPACK implementation, outperforming GEQRF with up to a 2x speedup. This advantage stems from SP-CAQR's ability to execute the panel and reflector application phases of each subdomain in an embarrassingly parallel manner, followed by a *tree-like* merge step. This approach mitigates the synchronous aspects typical in QR factorization's panel computation.

D. Strong Thread Scalability Results

To demonstrate the efficiency of Dagger Datadeps when scaling number of cores within a node, Fig. 9 shows performance advancement when doubling the number of threads, reaching up to 128 threads for Cholesky operations using DaggerDatadeps and oneMKL, with a matrix size of 61440. Dagger exhibits strong scalability comparable to well-optimized vendor operations, highlighting the performance efficiency of the Dagger scheduler.

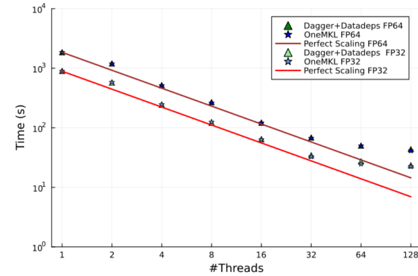


Fig. 9: Strong thread scaling on AMD.

VII. Conclusion

We introduced the Dagger task-based scheduler with data dependency awareness, marking the first implementation of an asynchronous runtime system natively in a high-level language like Julia. Dagger leverages dependency analysis of the data flow within function arguments to construct a DAG of tasks, ensuring that dependencies are respected during scheduling and execution. Our demonstrations using tiled dense linear algebra computations with the Dagger scheduler have showcased its capability to achieve high performance in dynamic languages. Notably, Dagger Datadeps has outperformed existing parallel paradigms within Julia and has shown comparable results to vendor-optimized operations. Dagger also shows significant performance improvement when factorizing TS matrices using SP-CAQR. Moving forward, we plan to harness the power of GPUs and distributed environments to further enhance its efficiency.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant no. OAC-1835443, grant

no. SII-2029670, grant no. ECCS-2029670, grant no. OAC-2103804, and grant no. PHY-2021825. The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0001211 and DE-AR0001222. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. This material was supported by the Research Council of Norway and Equinor ASA through Research Council project "308817 - Digital wells for optimal production and drainage". Research was sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. The authors would like to also thank the support and resources provided by KAUST. This material is also supported by grant #2022/07810-7, São Paulo Research Foundation (FAPESP), furthermore the opinions, hypotheses, conclusions, or recommendations expressed in this material are the responsibility of the authors and do not necessarily reflect the views of FAPESP. Additionally, authors extend their appreciation to the KAUST Ibn Rushd Fellowship and the São Paulo Research Foundation for their support.

REFERENCES

[1] Rabab Al-Omairy, Guillermo Miranda, Hatem Ltaief, Rosa M Badia, Xavier Martorell, Jesus Labarta, and David Keyes. Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing. *Supercomputing Frontiers and Innovations*, 2(1):49–72, 2015.

[2] Rabab Alomairy, Wael Bader, Hatem Ltaief, Youssef Mesri, and David Keyes. High-Performance 3D Unstructured Mesh Deformation Using Rank Structured Matrix Computations. *ACM Transactions on Parallel Computing*, 9(1):1–23, 2022.

[3] Rabab Alomairy, Hatem Ltaief, Mustafa Abduljabbar, and David Keyes. Abstraction Layer for Standardizing APIs of Task-Based Engines. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2482–2495, 2020.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012.

[6] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter. Rapid Software Prototyping for Heterogeneous and Distributed Platforms. *Advances in Engineering Software*, 132:29–46, 2019.

[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[8] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. LAPACK Working Note 191, University of Tennessee, 2007.

[9] Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G Genton, David E Keyes, Hatem Ltaief, and Others. Reshaping Geostatistical Modeling and Prediction for Extreme-Scale Environmental Applications. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2022.

[10] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. PTG: an Abstraction for Unhindered Parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30. IEEE, 2014.

[11] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-Optimal Parallel and Sequential QR and LU Factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[12] A. Duran, R. Ferrer, E. Ayguade, R. M. Badia, and J. Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3), 2009.

[13] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, 74(12), 2014.

[14] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18, 2019.

[15] Mosè Giordano, Milan Klöwer, and Valentin Churavy. Productivity Meets Performance: Julia on A64FX. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 549–555. IEEE, 2022.

[16] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tile QR Factorization with Parallel Panel Processing for Multicore Architectures. *2010 IEEE International Symposium on Parallel & Amp; Distributed Processing (IPDPS)*, 2010.

[17] T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX Execution Model to Stencil-Based Problems. *Computer Science - Research and Development*, 28(2-3), 2013.

[18] OpenMP. OpenMP 3.0 Complete Specifications, 2008.

[19] OpenMP. OpenMP 4.0 Complete Specifications, 2013.

[20] Hendrik Ranocha, Michael Schlottke-Lakemper, Andrew R Winters, Erik Faulhaber, Jesse Chan, and Gregor J Gassner. Adaptive Numerical Simulations with Trixi. Jl: A Case Study of Julia for Scientific Computing. *ArXiv Preprint ArXiv:2108.06476*, 2021.

[21] Michael P. Robson, Ronak Buch, and Laxmikant V. Kale. Runtime Coordinated Heterogeneous Tasks in Charm++. In *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware*, ESPM2, Piscataway, NJ, USA, 2016. IEEE Press.

[22] Julian Samaroo, Rabab Alomairy, and Felipe Tome. Dagger.jl. https://github.com/JuliaParallel/Dagger.jl, 2024.

[23] Elliott Slaughter and Alex Aiken. PyGion: Flexible, Scalable Task-Based Parallelism with Python. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives to MPI (PAW-ATM)*, pages 58–72. IEEE, 2019.

[24] F. Song, H. Ltaief, B. Hadri, and J. Dongarra. Scalable tile communication-avoiding qr factorization on multicore cluster systems. *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010.

[25] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. PyCOMPSs: Parallel Computational Workflows in Python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.

[26] Asim Yarkhan. Dynamic Task Execution on Shared and Distributed Memory Architectures. *PhD Dissertation*, 2012.

[27] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the PLASMA Numerical Library to the OpenMP Standard. *International Journal of Parallel Programming*, 45:612–633, 2017.