Impact of Data Distribution and Schedulers for LU Factorization on Clusters

Otho José Sirtoli Marcondes*, Lucas Mello Schnorr*
* Institute of Informatics/PPGC/UFRGS, Porto Alegre, Brazil

***Abstract*—As the demand for more computational resources grows, the usage of clusters has become one of the main options to satisfy this need. In order to exploit these resources efficiently, the distribution of data between nodes must be considered as an important factor in application performance. This study aims to analyze the impact of the static block-cyclic data distribution and different dynamic schedulers of the linear algebra LU factorization on clusters. The analysis focuses on the execution time to explain how the application's behavior is influenced by the data distribution and scheduling strategy.**

I. Introduction

High-Performance Computing (HPC) systems, particularly computing clusters, are essential for solving large-scale scientific and engineering problems. These clusters consist of multiple interconnected nodes, each with its own processor units and memory. In order to maximize application performance on clusters, it is essential to consider both inter-node communication efficiency and workload balance across the computing nodes.

A crucial aspect of achieving efficient parallel performance is data partitioning, which determines how data is divided and distributed across the computing nodes. Among various strategies, static data partitioning is commonly used due to its simplicity and low runtime overhead. One of the examples of static data distribution is the block-cyclic (BC) distribution, a method that was popularized by the ScaLAPACK [**?**] library.

This paper focuses on a scenario that combines static data partitioning with dynamic task scheduling. By leveraging task-based runtimes, we aim to dynamically schedule tasks at runtime while maintaining a static block layout of data. This approach enables better adaptability to runtime variations, such as load imbalance and communication delays, while preserving the advantages of a static data map.

As a case study, we explore the LU factorization, a fundamental operation in linear algebra widely used in scientific computing. We adopt a block cyclic distribution scheme for the input matrix, a method that balances the computational load and spreads data evenly across processes. Our goal is to evaluate how dynamic scheduling of tasks can improve the performance of LU factorization in clusters.

Throughout the development of this work, several challenges were encountered related to the use of MPI for executing applications across multiple nodes. These included: configuration challenges with Guix for package management across distributed nodes; issues related to the TCP interface in the MPI NewMadeleine implementation; and errors when using StarVZ [**?**] visualization framework with the traces collected from the executions (still not resolved).

The paper is structured as follows. Section~II presents some related work on matrix distribution and modern task-based runtimes. Section~III details our methodology and explains how we conducted the experiments in our investigation. Section~IV presents the experiments and their results. Section~V concludes this work with some considerations.

II. Related Work

A. Matrix distribution

ScaLAPACK [**?**] is the message passing version of LAPACK [**?**], and also the standard library for linear algebra operations over parallel distributed platforms. In this paper, the LU factorization will be the focus, as its parallelization strategy is similar to others.

As shown in the Figure~1, the LU factorization of a given matrix A is defined as $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix. The LU algorithm relies on three different LAPACK kernels: `DGTRF-NOPIV`, `DTRSM` and `DGEMM`. This application has a tendency to be dominated by `DGEMM` kernels when N is large, which makes it mandatory to have well distributed sub-matrixes between the nodes.

```
for (k = 0; k < N; k++)
  DGTRF-NOPIV(RW, A[k][k]);
  for (m = k+1; m < N; m++)
    DTRSM(RW, A[m][k], R, A[k][k]);
    DTRSM(RW, A[k][m], R, A[k][k]);
  for (n = k+1; n < N; n++) // Update
    for (m = k+1; m < N; m++)
      DGEMM(RW, A[m][n], R, A[m][k],
                         R, A[k][n]);
```

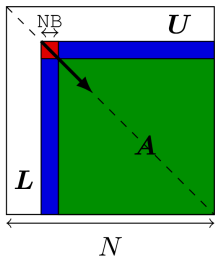


Fig. 1: The LU algorithm (left) without pivoting, and the regions of A updated at iteration k (right). [**?**]

The block cyclic distribution, popularized by the ScaLAPACK [**?**] depends on the PxQ parameters and the number of available nodes. Based on the P value, the matrix will be partitioned differently across the nodes. In the Figure~2 we can visualize that while P is 1 (each panel show a PxQ distribution), there is only one node per row, as in reverse of the 8x1 distribution, where there is only one node per column. For the 2x4 and 4x2 cases, the distribution is interleaved.

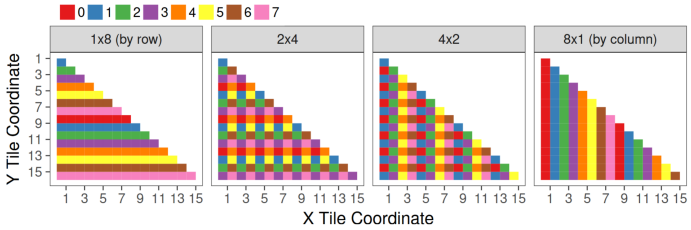


Fig. 2: Example of a block cyclic distribution across 8 nodes [**?**]

B. Task-based paradigm

As the computers used in HPC environments became more complex, adapting and exploiting them to their full potential has become increasingly challenging. The task-based paradigm was designed to solve these new challenges. It relies on a DAG (Directed Acyclic Graph) to represent the relation between tasks and their dependencies (edges). The scheduler then can dynamically allocate these tasks during execution time, according to the dependencies of the graph and the scheduler heuristic [**?**]. Chameleon [**?**], as other linear algebra libraries such as DPLASMA [**?**] are built on task-based runtimes, which allows them to efficiently exploit their computational resources of clusters. The scheduler heuristics studied in this work are the following:

- `lws`: stands for locality work stealing. When a worker becomes idle, it steals a task from a neighboring worker;
- `random`: tasks are distributed randomly according the assumed worker overall performance;
- `dmda`: takes task execution performance models and data transfer time into account;
- `dmdas`: same as `dmda`, but also take into account task priorities and data buffer availability on the target device.

III. **TODO** Experimental Methodology

To enable execution on multiple nodes, we employed StarPU [**?**], which provides a variety of scheduling policies and built-in support for application tracing.

StarPU is a task-based runtime system for heterogeneous platforms, being multicore or multinode. The StarPU uses the Sequential Task-Flow (STF) [**?**], where the tasks are sequentially submitted to the runtime that is responsible for their scheduling. Each task can have one implementation for a respective computational resource (CPU, GPU) called worker, and the scheduler must assign a task to one of the available workers during the program execution. To enable multi-node execution, the StarPU-MPI extension was used [**?**].

We utilized Chameleon [**?**] implementation of the LU factorization, with a matrix size of 16000x16000 block size for all experiments. This value was taken from a preliminary execution only varying the block size as shown in Figure~3, that depicts different blocks dimensions and their respective execution times. It is possible to observe that the 360 block size had the best performance among the other values.

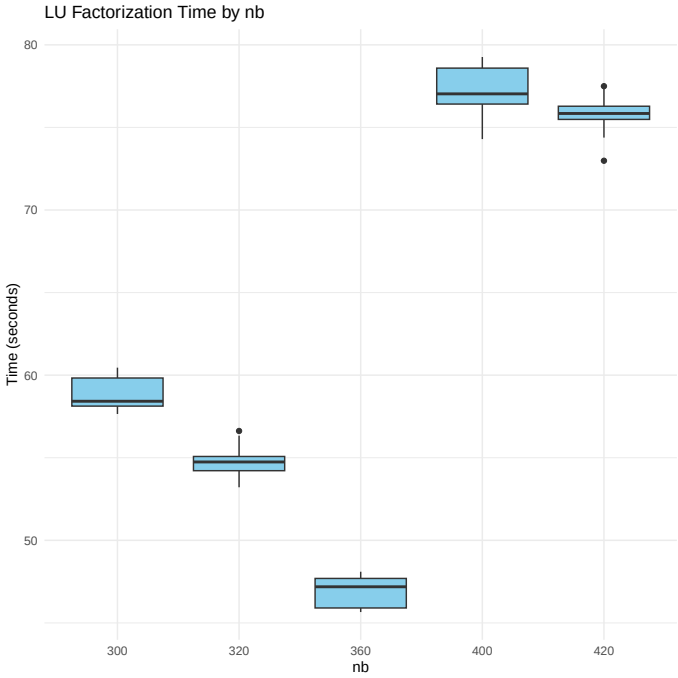


Fig. 3: Execution times per block dimension

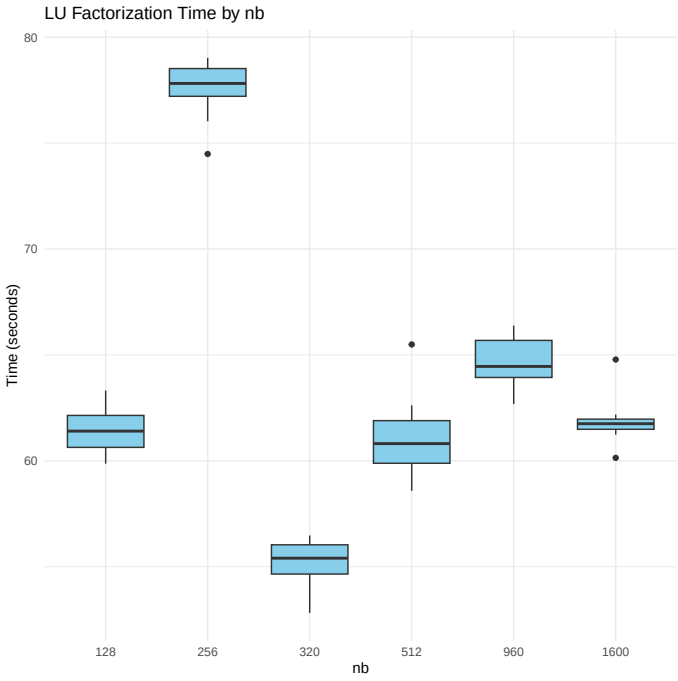


Fig. 4: Execution times per block dimension

The executions were performed in the PCAD from UFRGS using the `Cei` partition. `Cei` comprises six nodes, each one with two Intel Xeon Silver 4116 (24 cores/CPU). We used the 1.4.7 StarPU and 1.3.0 Chameleon version. We also used NewMadeleine [**?**] MPI implementation as OpenMPI [**?**] presented significant idle times during the executions. The

NewMadeleine version used was from commit `6e1a64d0` from June 2025, which resolved a TCP interface issue that we reported. For the execution time evaluation, each execution was run 10 times and the standard deviation was lower than 5\

IV. **TODO** Results

Figure~5 depicts four panels, aligned in the X dimension (time), each showing the execution time of a different scheduler (`random`, `lws`, `dmdas`, `dmda`) with a fixed PxQ configuration. The standard deviation is represented by the black error bars on each bar. We can see that the `lws` and `random` schedulers did not present much variation when changing the PxQ configuration. As for the `dmdas` and `dmda`, both of them showed significantly better performance when utilizing the $P = 2$ $Q = 3$ and $P = 3$ $Q = 2$ configurations.

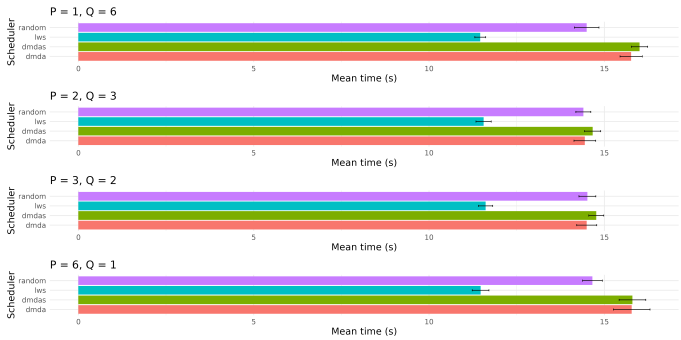


Fig. 5: Execution times based on the PxQ configuration

Figure~6 depicts four panels, aligned in the X dimension (time), each of them showing the execution time of a PxQ configuration with a fixed scheduler heuristic. The standard deviation is represented by the black error bars on each bar. We can see that the `lws` scheduler had the best results among the schedulers fallowed by the `random` scheduler. The `dmda` and `dmdas` had similar performance, with performance gains when P and Q are interleaved.

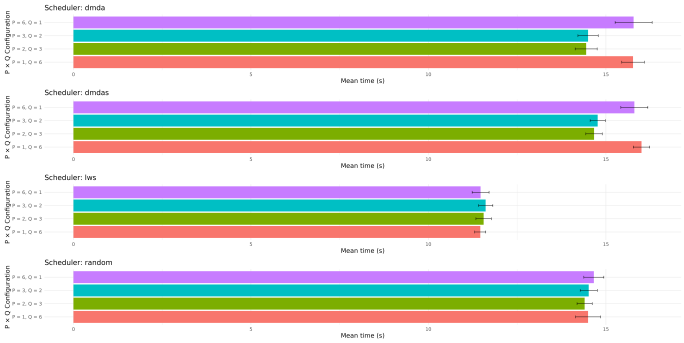


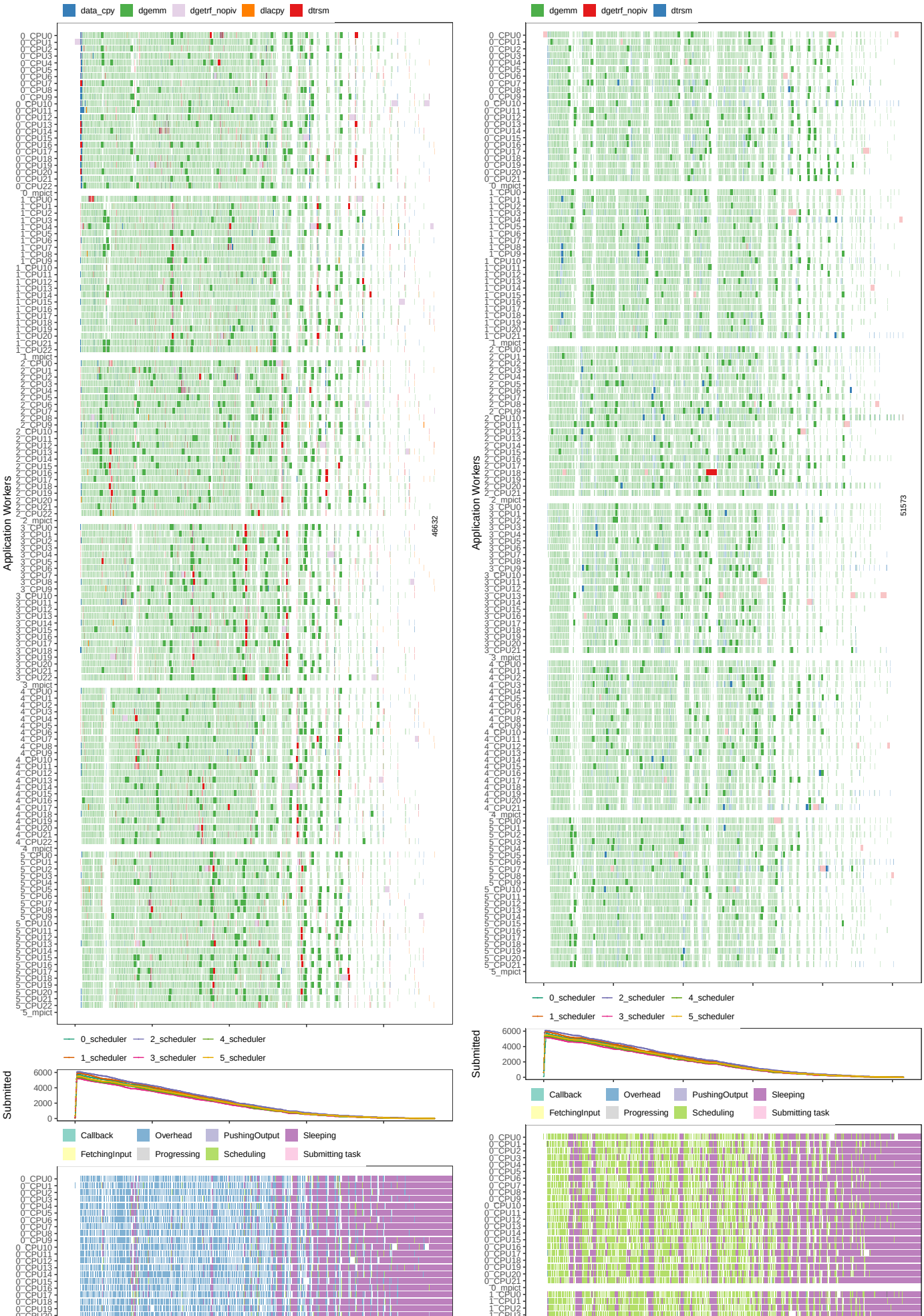
Fig. 6: Execution times based on the scheduler heuristic

V. **TODO** Conclusion

The study examines the impact of data distribution using Block cyclic and also the impact of different scheduler heuristics in the context of task-based runtime in clusters. The linear algebra LU factorization application provided by Chameleon was used as a means to analyze how these configurations impact performance. The `dmda` and `dmdas` heuristics presented similar behavior in their execution times, showing performance gains when the P and Q were interleaved. The `lws` heuristic presented the best results in terms of performance, although the P and Q parameters did not have significant impact in it. The `random` heuristic also showed no significant impact on its performance based on the P and Q parameters.

Acknowledgements

The experiments in this work used the PCAD infrastructure, http://gppd-hpc.inf.ufrgs.br, at INF/UFRGS.


data_cpy
dgemm
dgetrf_nopiv
dlacpy
dtrsm
Application Workers
46632
0_scheduler
1_scheduler
2_scheduler
3_scheduler
4_scheduler
5_scheduler
Submitted
6000
4000
2000
0
Callback
Overhead
PushingOutput
Sleeping
FetchingInput
Progressing
Scheduling
Submitting task
dgemm
dgetrf_nopiv
dtrsm
Application Workers
51573
0_scheduler
1_scheduler
2_scheduler
3_scheduler
4_scheduler
5_scheduler
Submitted
6000
4000
2000
0
Callback
Overhead
PushingOutput
Sleeping
FetchingInput
Progressing
Scheduling
Submitting task