# Data-Driven execution of the Tile LU Decomposition

George Matheou
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
Email: geomat@cs.ucy.ac.cy

Costas Kyriacou
Department of Computer
Science and Engineering
Frederick University
Nicosia, Cyprus
Email: eng.kc@frederick.ac.cy

Paraskevas Evripidou
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
Email: skevos@cs.ucy.ac.cy

***Abstract*—The objective of this paper is to analyze, develop and evaluate the tile LU Decomposition using the FREDDO framework. FREDDO is a C++ framework, based on the DDM model of execution, that supports efficient data-driven execution on conventional processors. The performance evaluation shows that FREDDO scales well and tolerates scheduling overheads and memory latencies effectively. The LU implementation is evaluated in both single-node and distributed execution environments. In both cases our framework achieves very good speedups, especially in the larger problem sizes. Particularly, our framework achieves up to 97% of the maximum possible speedup on a single-node and up to 90% of the maximum possible speedup on a 4-node cluster with a total of 128 cores.**

***Index Terms*—Data-Driven Multithreading, Multi-core Systems, Distributed Systems, Tile LU Decomposition**

## I. Introduction

Contemporary systems, like High Performance Computers and Supercomputers, are now based on multi-core and many-core architectures. Programming of such machines is mainly done through parallel extensions of the sequential models like MPI [1] and OpenMP [2]. These extensions do facilitate high productivity parallel programming, but also suffer from the limitations of the sequential synchronization and their inability to tolerate long latencies [3], [4], [5]. Thus, as the number of cores increases in the future, new parallel programming models and architectures must be developed to overcome these limitations.

The components of the traditional software stack, like numerical libraries, that are used by the scientific applications, should be redesigned to take advantage of the new chip architecture (hundreds of thousands of computing nodes, a million or more cores, etc.). High performance dense linear algebra software libraries, like LAPACK [6], have shown limitations on multi-core architectures [7], [8], [9]. This is because the parallelism is based on the expensive fork-join paradigm [8]. As such, several projects, like PLASMA [10], DPLASMA [11] and FLAME [12], developed new algorithms for dense linear algebra based on tile algorithms.

Tile algorithms can be represented by Directed Acyclic Graphs (DAGs), where nodes are the tasks and edges are the dependencies between the tasks [8]. The paramount key is to implement a runtime system to efficiently schedule the DAG across a parallel architecture. Such a runtime system can be provided by the Data-Driven Multithreading (DDM) model [13].

DDM is an execution model that combines the benefits of the data-flow model in exploiting concurrency with the efficient execution of the control-flow model. DDM decouples the execution from the synchronization part of a program and allows them to execute asynchronously, thus, tolerating synchronization and communication latencies efficiently. The core of the DDM model is the Thread Scheduling Unit (TSU) [14] which is responsible for scheduling threads/tasks at run-time based on data-availability on conventional processors.

The DDM model was evaluated in the past by several implementations. The first implementation of DDM was targeting Networks of Workstations [13], called the Data-Driven Network of Workstations ($D^2$NOW). That was followed by two other implementations, TFlux [15] and the Data-Driven Multithreading Virtual Machine (DDM-VM) [16], [17], [18]. DDM was also evaluated by a hardware implementation [19], [20] where the TSU was implemented as a hardware peripheral for sequential multi-core systems. Finally, the latest software implementation of DDM, FREDDO [21], [22], is a C++ framework that supports efficient data-driven execution on conventional processors through object-oriented programming.

This work presents the implementation details, performance and scalability of the tile LU decomposition, using the FREDDO framework (version 0.89.0). LU decomposition (also called LU factorization) is an important algorithm used for solving systems of linear equations efficiently [23]. The LU kernel factors a dense matrix into the product of a lower triangular $L$ and an upper triangular $U$ matrix [24]. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ tiles ($n$ = $NB$). This helps to exploit temporal locality on sub-matrix elements.

LU has been evaluated on different number of cores and problem sizes, on a 32-core AMD processor. The evaluation showed that the FREDDO implementation of LU scales very well and achieves very good speedups. For an $8192 \times 8192$ matrix size, FREDDO achieves an average speedup of 3.94 out of 4, 7.86 out of 8, 15.63 out of 16 and 30.09 out of 31.

Furthermore, we have evaluated the LU implementation

on a 4-node multi-core cluster with a total of 128 cores. For this configuration we are using a preliminary distributed implementation of the FREDDO framework. Our results show that our framework scales well in the distributed environment, achieving 90% of the maximum possible speedup, for a $32768 \times 32768$ matrix size.

The performance results show that Data-Driven/Data-Flow based systems, like FREDDO, effectively leverage the decoupling of synchronization and execution for the maximum tolerance of synchronization overheads. Thus, these systems can be used for the efficient implementation of dense linear algorithms, like LU, where in most of the cases they have high-complexity Dependency Graphs.

The remainder of the paper is organized as follows. Section II describes the FREDDO framework. Section III describes and analyses the tile LU algorithm and its dependency graph, and also it represents the DDM code for the algorithm. Finally, Section IV gives the experimental results and Section V provides the conclusions and future work.

## II. The FREDDO Framework

FREDDO [21], [22] is an efficient object-oriented implementation of the Data-Driven Multithreading (DDM) model [13]. It is a C++ framework that supports efficient data-driven execution on conventional multi-core and many-core processors. In FREDDO, a program consists of several threads of instructions (called DThreads) that have producer-consumer relationships. A DThread is scheduled for execution after all of its required data have been produced, thus, no synchronization or communication latencies are experienced after a DThread begins its execution. The DThreads' instructions are fetched by the CPU sequentially in control-flow order, thus, exploiting any optimization available by the CPU hardware.

### A. DThread Instances

FREDDO allows multiple instances of the same DThread to co-exist in the system. Each DThread instance is identified uniquely by the tuple: Thread ID (TID) and *Context*. Reentrant constructs, such as loops and function calls, can be parallelized by mapping them into DThreads. For instance, the iterations of a parallel loop can be executed by the instances of a specific DThread.

This idea was based on the U-Interpreter's tagging system [25] which provides a formal distributed mechanism for the generation and management of the tags at execution time. This system was used in Dynamic Data-flow architectures to allow loop iterations and subprogram invocations to proceed in parallel via the tagging of data tokens [26].

### B. The FREDDO API

FREDDO provides a C++ API (Application Programming Interface) that enables the programmers to develop DDM applications. The API includes a set of runtime functions and classes which are grouped together in a C++ namespace called *ddm*. The user creates and manages DThreads by creating and accessing objects of special C++ classes. Seven basic C++ classes are provided:

1) *SimpleDThread*: indicates a DThread with only one instance. It can be used to execute code without loops and/or recursion.
2) *MultipleDThread*: indicates a DThread with multiple instances which is used to parallelize one-level loops.
3) *MultipleDThread2D*: indicates a DThread with multiple instances which is used to parallelize two-level nested loops.
4) *MultipleDThread3D*: indicates a DThread with multiple instances which is used to parallelize three-level nested loops.
5) *RecursiveDThreadWithContinuation*: a special template class which provides functionalities for algorithms with multiple recursion. It is used when the number of instances of a recursive function is known at compile time.
6) *RecursiveDThread*: a special class that allows parallelizing recursive functions that their number of instances is not known at compile time. This class allocates/deallocates the arguments and the return values of the instances at runtime. It can be used for different types of recursion, such as, linear, tail, and so on.
7) *ContinuationDThread*: it can be used in combination with the RecursiveDThread to implement algorithms with multiple recursion (or any similar algorithms).

### C. The FREDDO Components

FREDDO allows efficient DDM execution by utilizing three different components: the TSU, the Kernels and the Runtime Support.

*1) The Thread Scheduling Unit (TSU):* The TSU is responsible for the management of DThreads. For each DThread, the TSU collects meta-data (also called Thread Templates) that enable the management of the dependencies among the DThreads and determine when a DThread instance can be scheduled for execution.

In particular, TSU schedules a DThread instance for execution when all its producer-instances have completed their execution. This ensures that all the data that this DThread instance needs are available. The ready DThread instances are dispatched to the Kernels for execution.

*2) The Kernels:* A Kernel is a POSIX Thread (PThread) that is pinned in a specific core until the end of the DDM execution. This eliminates the overheads of the context-switching between the Kernels in the system. The Kernel is responsible for executing the ready DThread instances that are received from the TSU.

In FREDDO, *m* Kernels are created, where *m* is the maximum number of DThread instances that can be executed in parallel in a system. Usually, *m* is equal to *N-1*, where *N* is the number of cores of the system. This is because one of the cores is reserved for the execution of the TSU code.

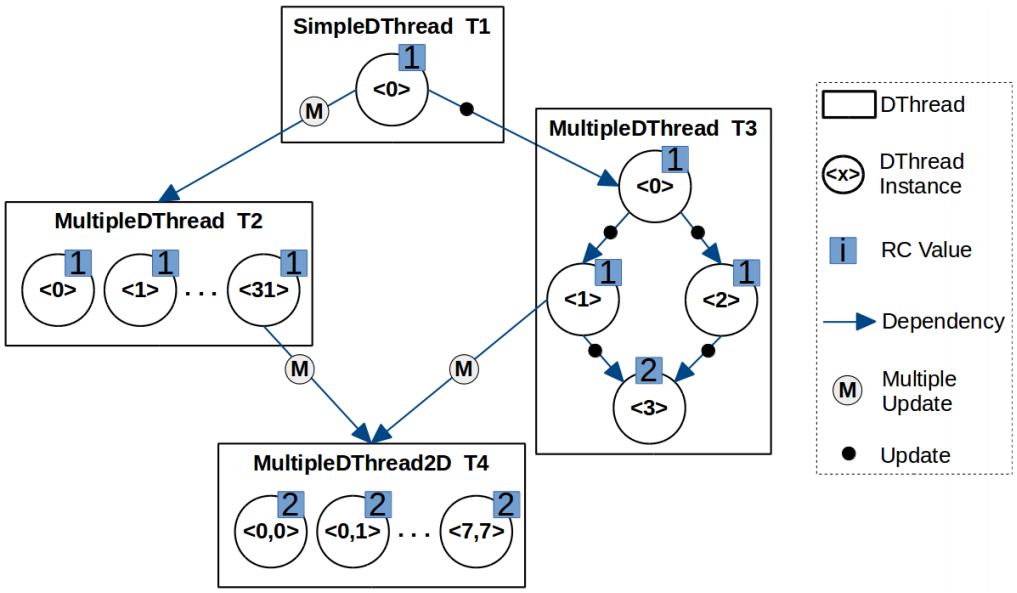


Fig. 1: Example of a FREDDO Dependency Graph.

*3) The Runtime System:* The Runtime system enables the communication between the Kernels and the TSU through the Main Memory. It is also responsible for loading the Thread Templates in the TSU, for creating and running the Kernels, and for deallocating the resources allocated by DDM programs.

### D. The FREDDO Dependency Graph

In DDM implementations, the Dependency Graph is a directed graph where the nodes represent the DThreads and the arcs represent the data-dependencies amongst them. Each instance of a DThread is paired with a special value called *Ready Count (RC)* that represents the number of its producers.

An example of a Dependency Graph is shown in Figure 1 which consists of four DThreads (T1-T4). The RC values are depicted as shaded values next to the nodes. For example, the instance 3 of T3 has RC=2 because it has two producers. Also, all the instances of T2 have RC=1 because they are waiting for only one producer, the instance 0 of T1. Notice that the number inside each node indicates its Context value.

The RC value is initiated statically and is dynamically decremented by the TSU each time a producer completes its execution. A DThread's instance is deemed executable when its RC value reaches zero. In FREDDO, the operation used for decreasing the RC value is called *Update*. Update operations can be considered as tokens that are moving from the producer to consumer instances through the arcs of the graph.

*Multiple Updates* are introduced in order to decrease multiple RC values of a DThread at the same time. This reduces the number of tokens in the graph. For instance, DThread T1 sends a Multiple Update to DThread T2 in order to spawn all its instances, instead of sending 32 single Updates.

Moreover, T4 is a MultipleDThread2D object and consists of 64 instances (with Contexts from <0,0> to <7,7>). In this case, a Context value contains two parts, the leftmost (or outer) which holds an index of the outer loop and the rightmost (or inner) which holds an index of the inner loop. Recall that a MultipleDThread2D object is used to parallelize a two-level nested loop. Similarly, the instances of a MultipleDThread3D object have Contexts values with three parts: outer, middle and inner.

## III. Tile LU Decomposition

In this section we describe and analyze the tile LU algorithm. After that, we provide the DDM Dependency Graph of the algorithm as well as the DDM code based on the FREDDO framework. The code of the original tile LU Decomposition is shown in Listing 1. The code is composed of five nested loops that perform four basic operations on a tiled matrix.

For demonstration purposes we choose the following indicative names for the operations: *diag*, *front*, *down* and *comb*. The presented algorithm is based on the LU implementation of the SPLASH-2 Application Suite [24]. However, the four operations can also be replaced by the following LAPACK and BLAS routines: *DGETRF, DTSTRF, DGESSM, DSSSSM* [11], [9].

### A. Benchmark Analysis

In every iteration of the outermost loop, the *diag* operation takes as input the diagonal tile that corresponds to the iteration number to produce its new value. The *front* operation produces the remaining tiles on the same row as the diagonal tile. For each one of those tiles, it takes as input the result of the *diag* in addition to the current tile to produce its new value. Similarly, the *down* operation produces the remaining tiles on the same column as the diagonal tile.

The *comb* operation produces the rest of the tiles for that LU iteration. For every tile it produces, it takes as input three

```
double AOrig[n*n]; // The original matrix
double *A[N][N];   // Each entry of A is a
                      pointer to a tile

for (kk = 0; kk < N; kk++) {        // Loop 1
  // A[kk][kk]: inout
  diag(A[kk][kk]);

 for (jj = kk + 1; jj < N; jj++)   // Loop 2
  // A[kk][kk]: input
  // A[kk][jj]: output
  front(A[kk][kk], A[kk][jj]);

 for (ii = kk + 1; ii < N; ii++)   // Loop 3
  // A[kk][kk]: input
  // A[ii][kk]: output
  down(A[kk][kk], A[ii][kk]);

 for (ii = kk + 1; ii < N; ii++)   // Loop 4
  for (jj = kk + 1; jj < N; jj++) // Loop 5
   // A[ii][kk]: input
   // A[kk][jj]: input
   // A[ii][jj]: ouput
   comb(A[ii][kk], A[kk][jj], A[ii][jj]);
}
```

Listing 1: Tile LU Decomposition (Original Code).

tiles: the current tile, the tile produced by the front operation and the tile produced by the down operation. It multiplies the second and third tiles and adds the result to the first tile to produce the final resulting tile. This computational pattern is repeated in the next LU iteration on a subset of the resulting matrix that excludes the first row and column and continues for as much iterations as the diagonal tiles of the matrix.

Figure 2 depicts the tiles produced by the four operations for the first iteration of LU decomposition on a $4 \times 4$ tile Matrix. Each tile is labeled with the first letter of its operation. The tile produced by the *diag* operation is labeled as diag. The arrows in the figure indicate the input tiles needed by each operation to produce its result.

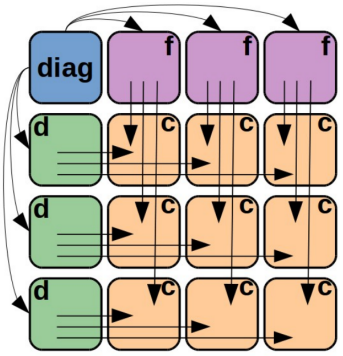


Fig. 2: LU Decomposition: dependencies between operations for the first iteration.

### B. Dependency Graph

The loops implementing the control-flow in the original application are mapped into five DThreads, called *loop_1_thread*, *diag_thread*, *front_thread*, *down_thread* and *comb_thread*. The first DThread implements the outermost loop of the algorithm while the other DThreads are responsible for executing the four operations. The following data dependencies are observed:

- The DThreads that execute the operations depend on the *loop_1_thread* since the index of the outermost loop is used in the four operations.
- The *front_thread* and *down_thread* DThreads depend on the *diag_thread*.
- The *comb_thread* depends on the *front_thread* and *down_thread* DThreads.
- The next LU iteration depends on the results of the previous iteration. Particularly, the results produced by the *comb_thread* invocations, in the current iteration, are consumed by the invocations of the *diag_thread*, *front_thread*, *down_thread* and *comb_thread*, of the next LU iteration.

The Dependency Graph shown in Figure 3 illustrates the dependencies among the instances of the DThreads for the first two iterations of LU. For simplicity, we show the operations on only two tiles of the matrix. Each DThread's instance is labeled with the value of its Context.

### C. The DDM Code

Listing 2 depicts the code of the five DThreads that implement the algorithm. In FREDDO, the code of the DThreads can be embodied in any callable target (called DFunction) like: (i) standard C++ functions, (ii) Lambda expressions and (iii) functors. Each DFunction has one input argument, the Context value. Different Context structures (ContextArg, Context2DArg and Context3DArg) are provided based on the type of the DThread class. In this example we place the code of the DThreads in standard C/C++ functions.

Firstly, the DThread objects are declared in lines 5-7 (will be created/allocated later in the main function of the program). Each call of an Update command in the DThreads' code corresponds to one dependency arrow in Figure 3.

Each DThread object (DObject) can call several different types of Update commands [22]. A DObject is able to Update itself or all its consumers. In both cases a user is able to specify a Context (Single Update) or a range of Contexts (Multiple Update). For example, the Update operation in line 11 indicates a Single Update which decrements the RC value of the instance kk of the diag_thread by one.

Furthermore, in line 25, a Multiple Update is used to decrement the RC value of multiple instances of the front_thread by one. In this case the following instances will be updated: <kk, kk+1>, <kk, kk+2>, ..., <kk, N-1>. Moreover, the Update operations at the end of the comb_thread_code implement a switch actor, which depending on the Context of the instance, it updates a different consumer-instance.

Listing 3 illustrates the main function of the DDM program. The *init* runtime function (line 7) initializes the DDM execution environment and it starts NUM_OF_KERNELS Kernels.

For each DObject we specify its DFunction, its RC value and the number of its instances. The RC value specified in the

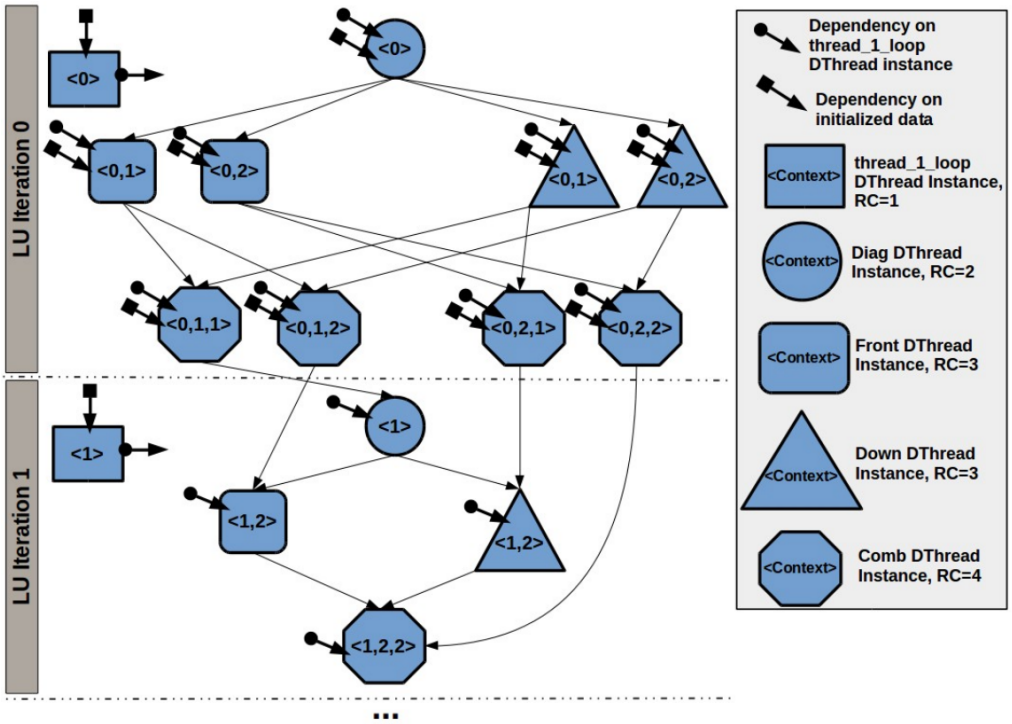


Fig. 3: The LU's DDM Dependency Graph.

constructor of a DObject indicates that all its instances will have this RC value. For instance, the diag_thread has DFunction=diag_thread_code, RC=2 and N number of instances. Notice that for the MultipleDThread2D and MultipleDThread3D objects we specify the number of instances in each level. For example, the comb_thread has N*N*N number of instances.

After the creation of the DObjects we are sending the initial updates to the TSU (lines 17-21). These updates correspond to the arrows of Figure 3 that describe dependencies on initialized data. The run() function starts the DDM scheduling. When the scheduling finishes, the DThreads are removed from the TSU (lines 26-30) and all the resources allocated by the FREDDO framework are deallocated (line 33).

## IV. Experimental Results

### A. Experimental Setup

The LU decomposition benchmark is evaluated in both single-node and distributed execution environments using the FREDDO framework (version 0.89.0).

The compute node is an HP server machine with 2 AMD Opteron 6276 processors running at 1.4GHz that supports 32 hardware threads. Each processor is an 8-core 64-bit Clustered Multi-Threaded (CMT) architecture with the capacity of running 16 threads simultaneously. Each core has a 16KB 4-way set associative L1 data cache, a 64K 2-way set associative L1 instruction cache and a 2MB 16-way set associative L2 cache. Also, each processor utilizes a 6MB 64-way set associative L3 cache. The server is equipped with a shared 48GB DDR3 RAM clocked at 1333MHz. Out of the 32 threads, one is used to run the TSU, while the rest are used for executing DThreads. The server runs the Ubuntu 14.04 OS (server edition).

For the distributed execution a 4-node cluster is used with a total of 128 cores (hardware threads). The nodes are connected using an off-the-shelf Gigabit Ethernet switch. Although the description of the distributed FREDDO implementation is outside the scope of this paper, we briefly describe two additional components that are needed for the distributed DDM execution:

- *Network Manager:* Each node is equipped with the Network Manager. This module is responsible for the communication with the other nodes of the system. It forwards/receives Updates and data packets to/from the other nodes. The Network Manager reserves another one core of the system. As such, a node is able to utilize up to 30 cores for computation.
- *Distributed Shared Memory (DSM):* A DSM is implemented across the nodes. Part or all of the main memory address space on each node is mapped to the Global Address Space (GAS) of the DSM.

The FREDDO framework and the LU source code were compiled with g++ 4.8.4. Table I depicts the average sequential execution time (in seconds) of five executions for each matrix size of the LU benchmark. The last two matrix sizes are very large and thus they are used only for the evaluation of the distributed execution. The execution time measurements were collected using the *gettimeofday* system call. Notice that for each matrix size we are using $32 \times 32$ tiles (i.e. B=32).

For the performance evaluation, all the experimental results

```
#include <freddo/dthreads.h>
using namespace ddm; // Use the ddm name-space

// DThread Objects
MultipleDThread *loop_1_thread, *diag_thread;
MultipleDThread2D *front_thread, *down_thread;
MultipleDThread3D *comb_thread;

// The code of the loop_1_thread DThread
void loop_1_thread_code(ContextArg kk) {
 diag_thread->update(kk);

 if (kk < N - 1) {
  front_thread->update({kk, kk+1}, {kk, N-1});
  down_thread->update({kk, kk+1}, {kk, N-1});
  comb_thread->update({kk, kk+1, kk+1},
                      {kk, N-1, N-1});
 }
}

// The code of the diag_thread DThread
void diag_thread_code(ContextArg kk) {
 diag(A[kk][kk]); // Execute Diag Operation

 if (kk < N - 1)) {
  front_thread->update({kk, kk+1}, {kk, N-1});
  down_thread->update({kk, kk+1}, {kk, N-1});
 }
}

// The code of the front_thread DThread
void front_thread_code(Context2DArg context) {
 auto kk = context.Outer, jj = context.Inner;

 // Execute Front Operation
 front(A[kk][kk], A[kk][jj]);
 comb_thread->update({kk, kk+1, jj},
                     {kk, N-1, jj});
}

// The code of the down_thread DThread
void down_thread_code(Context2DArg context) {
 auto kk = context.Outer, jj = context.Inner;

 // Execute Down Operation
 down(A[kk][kk], A[jj][kk]);
 comb_thread->update({kk, jj, kk+1},
                     {kk, jj, N-1});
}

// The code of the comb_thread DThread
void comb_thread_code(Context3DArg context) {
 auto kk = context.Outer, ii = context.Middle,
      jj = context.Inner;

 // Execute Comb Operation
 comb(A[ii][kk], A[kk][jj], A[ii][jj]);

 // Updates for the Next LU Iteration
 if (ii == kk + 1 && jj == kk + 1)
  diag_thread->update(kk+1);
 else if (ii == kk + 1)
  front_thread->update({ii, jj});
 else if (jj == kk + 1)
  down_thread->update({jj, ii});
 else
  comb_thread->update({kk+1, ii, jj});
}
```

Listing 2: The code of the LU's DThreads.

```
int main() {

 // Initializes the data (matrices, etc.)
 initializeData();

 // Initializes the FREDDO's execution environment
 init(NUM_OF_KERNELS);

 // Creates the DThread Objects
 loop_1_thread = new MultipleDThread(
     loop_1_thread_code, 1, N);
 diag_thread = new MultipleDThread(diag_thread_code
     , 2, N);
 front_thread = new MultipleDThread2D(
     front_thread_code, 3, N, N);
 down_thread = new MultipleDThread2D(
     down_thread_code, 3, N, N);
 comb_thread = new MultipleDThread3D(
     comb_thread_code, 4, N, N, N);

 // Updates resulting from data initialization
 loop_1_thread->update(0, N-1);
 diag_thread->update(0);
 front_thread->update({0, 1}, {0, N-1});
 down_thread->update({0, 1}, {0, N-1});
 comb_thread->update({0, 1, 1}, {0, N-1, N-1});

 run();  // Starts the DDM scheduling

 // Remove the DThreads
 delete loop_1_thread;
 delete diag_thread;
 delete front_thread;
 delete down_thread;
 delete comb_thread;

 // Stops the Kernels and releases the resources
 finalize();
}
```

Listing 3: The main program: initializations and Dependency Graph creation/execution.

are reported as average speedups. The average speedup of a certain configuration is defined by the following formula:

$$average\ speedup = \frac{average\ sequential\ execution\ time}{average\ parallel\ execution\ time},$$

where the *average parallel execution time* indicates the average of five parallel executions.

TABLE I: Average sequential execution time for each problem size (B=32).

| Matrix Size ($n \times n$) | Serial Time (sec) |
|---|---|
| $1024 \times 1024$ | 1.23 |
| $2048 \times 2048$ | 9.9 |
| $4096 \times 4096$ | 79.39 |
| $8192 \times 8192$ | 636.01 |
| $16384 \times 16384$ | 5138.01 |
| $32768 \times 32768$ | 41575.05 |

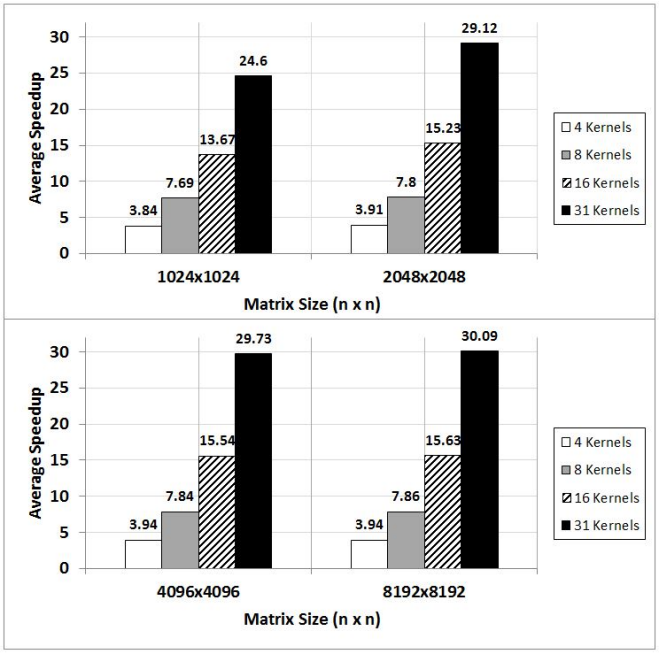


Fig. 4: LU Decomposition: Performance scalability for different number of computation cores (Kernels) and problem sizes.

### B. Performance Evaluation - Single Node Execution

In this work we performed a scalability study of the performance for the LU benchmark using the FREDDO framework. We have evaluated the performance of our framework on different number of Kernels and problem sizes (Figure 4). Four different Kernel configurations are used: 4, 8, 16 and 31.

The evaluation shows that FREDDO scales very well and achieves very good speedups, especially in the largest problem size ($8192 \times 8192$). Particularly, the LU benchmark achieves the following speedups: 3.94 out of 4, 7.86 out of 8, 15.63 out of 16 and 30.09 out of 31. This is justified by the fact that, as the benchmark's execution time increases, the parallelization overhead is amortized.

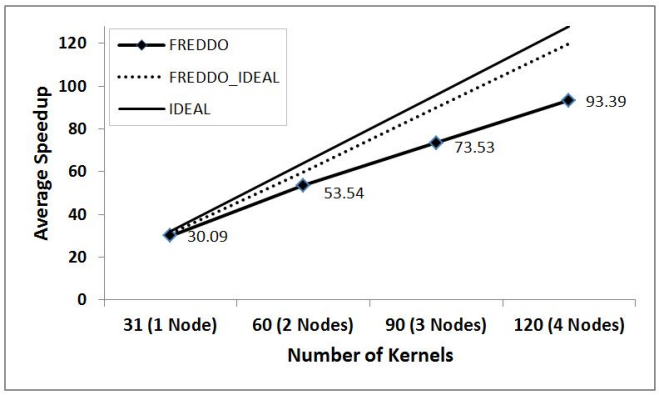


Fig. 5: Performance evaluation of the distributed LU implementation for Matrix Size=$8192 \times 8192$.

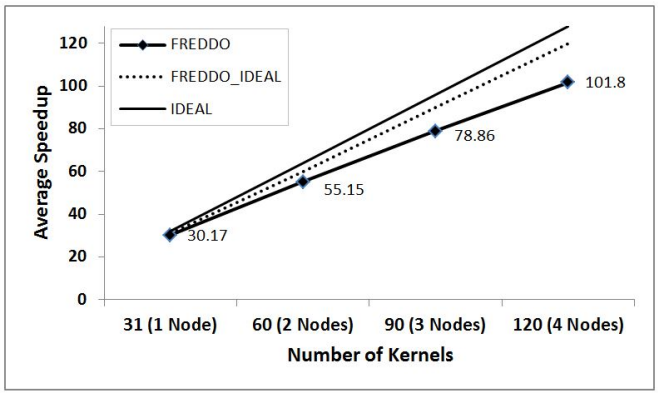


Fig. 6: Performance evaluation of the distributed LU implementation for Matrix Size=$16384 \times 16384$.

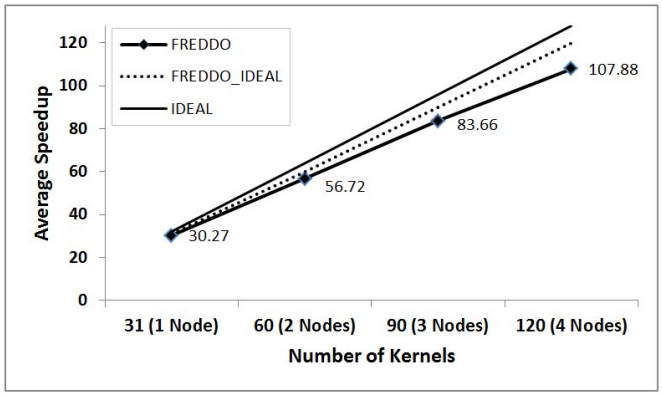


Fig. 7: Performance evaluation of the distributed LU implementation for Matrix Size=$32768 \times 32768$.

### C. Performance Evaluation - Distributed Execution

Figures 5 to 7 illustrate the performance evaluation of the FREDDO framework in our cluster for three large matrix sizes: $8192 \times 8192$, $16384 \times 16384$ and $32768 \times 32768$. In each figure, *IDEAL* indicates the maximum speedup that a parallel application is able to achieve based on the number of nodes of the system. *FREDDO_IDEAL* indicates the maximum speedup

that a parallel FREDDO application is able to achieve based on the number of nodes of the system. For instance, for a 4-node cluster, *IDEAL*=128 and *FREDDO_IDEAL*=120. This is because in the latter case we reserve two cores from each node for the TSU and the Network Manager.

According to our preliminary results, FREDDO scales well and achieves very good speedups. For the $8192 \times 8192$ matrix we achieved an average of 78% of the maximum possible speedup (*FREDDO_IDEAL*) for the 4-node configuration. For the same configuration, FREDDO achieves 84% and 90% of the maximum possible speedup, for the $16384 \times 16384$ and $32768 \times 32768$ matrix sizes, respectively. Thus, larger input sizes and possibly larger granularities (tiles) are needed for the system to scale due to the additional latencies introduced by the network data and synchronization messages transfer.

## V. Conclusions and Future Work

Dense linear algebra software libraries, such as LAPACK and ScaLAPACK, have shown limitations on multi-core architectures due to their inability to fully exploit thread-level parallelism. Several researchers proposed the use of tile algorithms and their execution using the data-flow/data-driven model of execution.

In this work we are using the FREDDO framework, an object-oriented software implementation of the Data-Driven Multithreading (DDM) model, to implement and evaluate the tile LU Decomposition benchmark. The presented algorithm has been evaluated on a 32-core server as well as on a 4-node multi-core cluster with a total of 128 cores. The evaluation shows that data-driven based models, like FREDDO, can scale very well and achieve very good speedups, on algorithms with high-complexity Dependency Graphs.

Future work will be focused on implementing more dense linear algebra algorithms, such as, the Cholesky factorization and the QR factorization. Additionally, we will compare our framework with other frameworks that are based on the data-flow/data-driven model, like, OmpSs, SWARM and DPLASMA.

## Acknowledgment

This work was partially funded by the IKYK foundation through a scholarship for George Matheou.

## References

[1] R. L. Graham, "The mpi 2.2 standard and the emerging mpi 3 standard," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2009, pp. 2–2.

[2] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," Nov. 2015. [Online]. Available: http://www.openmp.org/mp-documents/openmp-4.5.pdf

[3] G. Matheou, P. Evripidou, and C. Kyriacou, "Paradigm shift for exascale computing," in *Proceedings of the 3rd International Conference on Exascale Applications and Software*. University of Edinburgh, 2015, pp. 109–114.

[4] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Position paper: Using a codelet program execution model for exascale machines," in *EXADAPT Workshop*, 2011.

[5] P. Kogge, "Next-generation supercomputers," *IEEE Spectrum, February*, 2011.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' guide*. Siam, 1999, vol. 9.

[7] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra, "Comparative study of one-sided factorizations with multiple software packages on multi-core hardware," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 20.

[8] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra, "Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 3, pp. 305–321, 2012.

[9] L. W. Note, "Scheduling linear algebra operations on multicore processors."

[10] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, "Plasma users guide. parallel linear algebra software for multicore architectures," *Rapport technique, Innovative Computing Laboratory, University of Tennessee*, 2011.

[11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief *et al.*, "Distributed dense numerical linear algebra algorithms on massively parallel architectures: Dplasma," *Electrical Engineering and Comp. Sci. Dept., Univ. of Tenn., Tech. Rep. UT-CS-10-660*, 2010.

[12] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan *et al.*, "Design and scheduling of an algorithm-by-blocks for lu factorization on multithreaded architectures," 2007.

[13] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, Oct. 2006.

[14] P. Evripidou, "Thread synchronization unit (tsu): A building block for high performance computers," in *International Symposium on High Performance Computing*. Springer, 1997, pp. 107–118.

[15] K. Stavrou and et al, "TFlux: a portable platform for data-driven multithreading on commodity multicore systems." IEEE, Sep. 2008, pp. 25–34.

[16] S. Arandi and P. Evripidou, "Programming multi-core architectures using data-flow techniques," in *SAMOS-2010*. IEEE, 2010, pp. 152–161.

[17] ——, "DDM-VMc: the data-driven multithreading virtual machine for the cell processor," in *Proc. of the 6th Int. Conf. on High Performance and Embedded Architectures and Compilers*, 2011, pp. 25–34.

[18] G. Michael, S. Arandi, and P. Evripidou, "Data-flow concurrency on distributed multi-core systems," in *In Proceedings of the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA13)*, 2013.

[19] G. Matheou and P. Evripidou, "Verilog-based simulation of hardware support for data-flow concurrency on multicore systems," in *SAMOS XIII, 2013*. IEEE, 2013, pp. 280–287.

[20] ——, "Architectural support for data-driven execution," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 52:1–52:25, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2686874

[21] ——, "FREDDO: an efficient framework for runtime execution of data-driven objects," Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Tech. Rep. TR-16-1, January 2016. [Online]. Available: https://www.cs.ucy.ac.cy/docs/techreports/TR-16-1.pdf

[22] ——, "Freddo: an efficient framework for runtime execution of data-driven objects," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016, p. 265.

[23] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, "LU decomposition and its applications," *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, pp. 34–42, 1992.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

[25] Arvind and Gostelow, "The u-interpreter," *Computer*, vol. 15, no. 2, pp. 42–49, Feb. 1982.

[26] I. Watson and et al, "A prototype data flow computer with token labelling," in *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society, 1989, pp. 623–623.