A LIGHTWEIGHT RUN-TIME SUPPORT FOR FAST DENSE LINEAR ALGEBRA ON MULTI-CORE

Daniele Buono, Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli and Massimo Torquati
Department or Computer Science, University of Pisa
Largo B. Pontecorvo, 3, I-56127, Pisa, Italy
Email: {d.buono, marcod, dematteis, mencagli, torquati}@di.unipi.it

ABSTRACT
The work proposes ffMDF, a lightweight dynamic run-time support able to achieve high performance in the execution of dense linear algebra kernels on shared-cache multi-core. ffMDF implements a dynamic macro-dataflow interpreter processing DAG graphs generated on-the-fly out of standard numeric kernel code. The experimental results demonstrate that the performance obtained using ffMDF on both fine-grain and coarse-grain problems is comparable with or even better than that achieved by de-facto standard solutions (notably `PLASMA` library), which use separate run-time supports specifically optimised for different computational grains on modern multi-core.

KEY WORDS
Data-flow run-time, dense linear algebra, dynamic scheduling, multi-threading, multi-core.

1 Introduction

Dense linear algebra (DLA) kernels are representative of a large class of computations that are both highly demanding in terms of raw performance and used in a wide range of different applications. DLA kernels have been considered for parallelisation on any kind of parallel hardware. Lately, a lot of research effort is focusing on multi-/many-core platforms and heterogeneous systems [22, 6, 20].

The parallelisation techniques have been typically based on two distinct approaches operating at different levels: *pure vectorisation* or either *static* or *dynamic scheduling* of tasks composing Directed Acyclic Graphs (DAGs), representing the data dependencies within the kernels. Pure vectorisation techniques are suitable to parallelise DLA kernels on vector/SIMD like architectures, such as the GPUs or CPUs with SIMD instruction set extensions. Instead, static or dynamic scheduling of DAGs has been proven to be effective when targeting multi-core architectures, possibly in conjunction with the optimisations of computations associated with each DAG node, i.e. by exploiting the SIMD instruction set extensions of the micro-architecture at hand [17].

In the *static scheduling* approach, the correct execution of DAG nodes (*tasks*) is maintained by identifying a precise partial ordering, without the need to maintain complete data-dependency information. Instead, in the *dynamic scheduling* approach (used in ffMDF), the run-time support is in charge of scheduling tasks to the processing units as they become available (fireable), i.e. all input data-dependencies are satisfied. While static scheduling strategies need to be defined at hand for each DLA algorithm, a dynamic approach is more general in principle, and deserves special attention because it is potentially able to exploit the maximum parallelism of the considered DAG ensuring a balanced workload among processing units. However, this potential advantage has a trade-off in a higher implementation complexity. It is required to manage the on-the-fly generation of tasks, with their relative data dependencies, and to implement an efficient algorithm to explore the graph, searching for new tasks to assign to the available processing units.

The implementation of efficient DAGs schedulers is a critical problem especially for DLA kernels. Over the last years, several run-time frameworks have been developed providing support to the scheduling of computations represented as DAGs on multi-core (see Sect. 5). The `PLASMA` library [12, 2], introduced a new set of algorithms for DLA (named *tile algorithms* [10, 17]) in which parallelism is not bounded inside the `BLAS` kernel, but it can be described at a higher level modelling the computation as DAG of tasks. Due to the performance delivered by these new algorithms, `PLASMA` is considered the state-of-the-art for linear algebra on modern multi-core. In `PLASMA`, the execution of a tile algorithm is performed using two separate run-time supports: a static run-time based on static scheduling strategies suitable to efficiently execute fine-grain DAGs, and a run-time based on dynamic scheduling for coarse-grain problems. The selection of which run-time to use for a given algorithm is left to the user.

The work presented in this paper distinguishes from the previous ones by adopting a *Structured Parallel Programming* model approach [13, 16] while designing a run-time suitable to support the execution

This work has been partially supported by FP7 STREP ParaPhrase (`www.paraphrase-ict.eu`).

of DAGs from different applicative domains on multi-core architectures. The result is ffMDF, a lightweight dynamic run-time support for the efficient execution of DAGs generated *on-the-fly*. This design, based on the *macro-dataflow* model presented in Sect. 2 and using the synchronisation mechanisms presented in [4], makes it possible to achieve an efficient execution of *any* kind of DAGs, and in particular those resulting from the dependencies analysis, performed at run-time, of DLA algorithms.

The experiments prove that ffMDF is able to efficiently execute *both* coarse- and fine-grain DAGs, the latter typically solved using a *static scheduling* approach, where task assignment to processing units is predetermined at compile time to minimise run-time overheads [18].

The contribution of this paper is twofold:

1. the definition of a *generic* lightweight dynamic scheduling run-time support for the efficient execution of DAGs. The efficiency of the run-time has been proven on both fine- and coarse-grain DLA problems, demonstrating that *it is possible to build one single efficient run-time support* for any computational grains;

2. the implementation of the run-time provides a further yet meaningful scientific contribution. The run-time is built out of the structured composition of well-known parallel patterns *enabling the design and implementation with a quite moderate programming effort*. The structured design enabled general-purpose optimisations in the management of data dependencies and task scheduling and the elimination of lock-based operations on concurrent data structures.

The rest of this paper is organised as follows. Sect. 2 introduces the general macro data-flow model used to implement ffMDF. Sect. 3 discusses the overall design of ffMDF. Sect. 4 presents the results of a wide set of experiments validating the design and implementation choices and comparing ffMDF with de-facto standard solutions. Finally, Sect. 5 briefly discusses existing research works on dynamic scheduling and execution of DLA algorithms on parallel architectures, and Sec. 6 draws conclusions.

2 The Macro-Dataflow Model

Data flow execution is a well-known computing paradigm [23] extremely attractive for parallel processing. It consists in an asynchronous way to execute instructions based solely on the availability of their input arguments (pure data dependencies, i.e. *read-after-write*). The data flow model potentially makes it possible to achieve the highest throughput while executing a program, as parallelism is expressed at the finest granularity level (i.e. at the level of instructions). This is at the expense of a huge complexity in the development of the implementation model, in particular in the management of data dependencies and storage space (where operands and meta-data are maintained) and in the efficient detection and scheduling of the firable (ready to be executed) instructions. As a consequence, real hardware implementations of this paradigm usually provide poor scalability and performance [23] compared to the control-flow counterpart, and are mainly used to implement sub-portions of modern superscalar architectures as well as special-purpose machines for specific applicative domains (e.g. digital signal processing [21]).

With the emergence of highly parallel multi-core architectures, the problem of expressing fine grain parallelism through dataflow models has gained a renewed attention. Instead of expressing parallelism at the instruction level, portions of the sequential code having pure functional dependencies between input parameter and output results, are considered *macro-dataflow* (MDF) instructions. The resulting MDF program is therefore represented as a graph whose nodes are computational kernels and arcs *read-after-write* dependencies. The instructions interpreter (i.e. the run-time support) is in charge of scheduling fireable instructions and managing data dependencies as fast as possible to avoid introducing new computational bottlenecks. To this end, the following three points represent fundamental aspects for an efficient implementation of a MDF interpreter in particular for fine-grain computations:

Construction of the task graph (DAG): in the general case the task graph could be very large; its generation time can affect significantly the computation time, and the memory required to store the entire graph may be extremely large. To alleviate these issues, a widely used solution consists in generating the graph during the computation, such that only a "window" of the graph is maintained in memory. This also allows to overlap tasks computation with the graph generation.

Handling task dependencies: two main operations on the graph need to be properly optimised: i) update dependencies after the completion of previously scheduled tasks; ii) determine ready (fireable) tasks to be executed. These operations should be done as fast as possible and *in parallel* with tasks computation, to avoid affecting the performance.

Scheduling of fireable tasks: a task having all input dependencies resolved (fireable) may be selected by the interpreter for execution. This selection needs to be performed in a smart way considering that two main optimisations can be applied in this phase when targeting multi-core architectures: i) *locality optimi-*

sation in order to better exploit cache level hierarchies, and ii) *parallelism optimisation* in order to maintain the number of ready tasks as big as possible during the execution to prevent stalls. The first optimisation leverages on the fact that graph arcs represents data dependencies, so that if task i unlocks execution of task j, then they share at least one of the dependencies. Executing task j on the same processor that ran task i (as soon as possible), increases the probability that the common data reside in the cache hierarchy. The second optimisation leverages on the fact that a graph node with an higher degree of output arcs, might unlock a larger number of tasks. Following this principle, the scheduling policy should select with higher priority among all fireable tasks those ones that have the higher number of outgoing edges. This can be accomplished by ordering fireable tasks with respect to the degree of the related node on the graph.

Finally, to reduce memory consumption, in-place computation is generally used on shared memory platforms instead of the classic dataflow approach. To this end, dataflow graphs have to be enriched by additional anti-dependencies (*write-after-read*) between tasks for removing the need of costly copies of the original data structure. This at the price of a possible lower parallelism between MDF instructions.

3 Designing a lightweight dynamic run-time

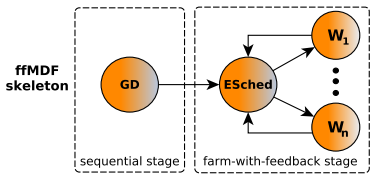
3.1 Skeleton–based design

A MDF interpreter may be implemented as a two-stage pipeline: a sequential stage called Graph Descriptor (GD), which defines and executes the user algorithm that eventually produces the instructions for building the DAG, and a parallel stage that generates (a portion) of the DAG using a suitable data representation and then executes in parallel the resulting DAG nodes as soon as their input dependencies are satisfied. The partial results of the computation are typically produced in output as a stream of tasks or stored in the shared memory by updating a data structure.

The parallel stage is logically composed by 2 concurrent entities: a task scheduler (called ESched) and a set of replicated workers (Ws) which are the real interpreter of the MDF instructions. The ESched receives in a non-deterministic way, both new instructions coming from the GD and also completed tasks coming from the set of workers. It generates dynamically the graph nodes during the computation upon receiving instructions from GD. From the sequential order in which tasks are generated by the GD, the ESched computes a *partial ordering* that ensures computation correctness, and, by evaluating data dependencies among tasks, it adds the corresponding node/edge to the DAG structure. A completed task coming from one of the workers may either activate new tasks ready to be scheduled for execution, or trigger the termination condition. The resulting skeleton structure (called ffMDF) is sketched in Fig. 1 (upper left hand side). Using the proposed skeleton, all the run-time support overhead is bound in the two sequential stages GD and ESched. While this approach may in principle limit the scalability with a large amount of graph nodes, the high number of cores in current and foreseen multi-core platforms, and the careful design of the ESched, makes this pattern a good candidate to remove much of the overhead that limits the performance of the dynamic scheduling in other frameworks.

The implementation of the ffMDF skeleton has been developed using FastFlow [1, 9], a skeleton-based programming framework. FastFlow is a structured parallel programming environment implemented in C++ on top of POSIX threads [5, 3]. It provides the user with streaming parallel patterns/skeletons like *sequential*, *pipeline* and *task-farm*, which can be composed arbitrarily. The task-farm pattern can be instantiated in different ways. We used the one that allows to customise the task scheduling policy, and to completely programm the Emitter thread, which performs pre-processing of input tasks and their scheduling towards a pool of sequential workers. The workers (i.e. the FastFlow sequential stages), compute the results and route them back to the Emitter thread using a feedback channel. Figure 1 (left hand side) shows the FastFlow code needed to build the 2-stage pipeline of the ffMDF skeleton. The task-farm Emitter thread implements the ESched modules.

The programmer that uses the ffMDF pattern, is only required to express the operations that compose the DAG. As an example, in Figure 1 (right) it is shown the simplified code needed to instantiate and run the Cholesky factorisation algorithm using the ffMDF pattern. The programmer has to select one of the available scheduling policy *SchedP*, and to provide a function pointer (the Algo function in the figure) which describes the algorithm that will be executed by the GD stage of the run-time. In the Algo function, the **AddTask** procedure (implemented by the ffMDF runtime) is used to define the operations composing the graph. It requires, a function pointer to the real kernel code (i.e. low level `PLASMA` wrappers or LAPACK wrappers), the number and the list of parameters used by the function. For each parameter the user has to specify its *size* and its *mode*. The mode specifies if the parameter is used in INPUT or OUTPUT, in such a way that the corresponding graph dependencies may be built. A special mode, VALUE, is required for those parameters that are directly evaluated inside the GD and do not concur to the DAG creation. All tasks generated via the **AddTask** func-




```
ff :: ff_farm<SchedP> farm; // farm instance
std :: vector<ff_node *> V;  // workers vector
for(int i=0;i<nworkers;++i) V.push_back(new W);
farm.add_workers(V); // adds workers to the farm
farm.add_emitter(new ESched(farm.getlb()));// scheduler
farm.wrap_around(); // adds the feedback channel
ff :: ff_pipeline  pipe(false, QUEUE_SIZE); // pipeline
pipe.add_stage(new GD(Algo,Parameters)); // adds 1st stage
pipe.add_stage(&farm);                   // adds 2nd stage
```

```
int main() {
  ff_MDF<SchedP> pchol(Algo,Parameters,nworkers);
  pchol.run_and_wait_end();
}
void Algo(void* params){
  for(int k=0;k<tiles;k++) {
    for(int n=0;n<k;n++)
      AddTask(CHERK,11,CblasRowMajor,VALUE,sizeof(int)
          ,...);
    AddTask(CPOTF2,5,...,A[k][k],OUTPUT,sizeof(A[k][k]),...);
    for(int m=k+1;m<tiles;m++) {
      for(int n=0;n<k;n++)
        AddTask(CGEMM,14,...,A[k][n],INPUT,sizeof(A[k][n])
            ,...);
      AddTask(CTRSM,12,...,A[k][k],INPUT,sizeof(A[k][k]),...);
    }
  }
}
```

Figure 1: Left): ffMDF implementation skeleton using FastFlow. Right): Example code needed to describe and execute the block Cholesky factorisation algorithm using the ffMDF pattern.

tion are packed and sent to the ESched thread, which creates the corresponding DAG node.

In the ffMDF skeleton only the ESched thread works on the DAG, so that neither critical sections nor cache invalidations overhead is spent for updating the graph structure. This way, we also remove the run-time overhead for managing concurrent data structures in the worker threads. Nonetheless, the Worker threads have to obtain tasks from the ESched following a Producer–Consumer pattern. The lock-free Single-Producer Single-Consumer queue implementing a shared-memory data-flow channel available in the FastFlow task-farm pattern, allowed us to bound such communication time to few tenths of clock cycles [4]. Furthermore, as we will see in the experimental section, the ESched thread is not a real bottleneck of the system, since we were able to bound all the preprocessing and scheduling activities for a single input task in few hundreds of clock cycles.

Since the memory required to store the entire DAG may be huge, we decided to mantain in main memory only a "window" of the graph structure in order to limit the amount of memory used. When the number of generated graph nodes reaches a predetermined threshold (that can be tuned by the user), the channel from GD to ESched is temporarily disabled; this way the graph generation is halted and the ESched handles only completed tasks coming from the workers. When the number of available instructions falls below the threshold, the channel from the GD node is re-enabled.

The pseudo-code of the ESched and of the generic worker are shown in Algorithm 1 and 2, respectively. We omitted the GD pseudo-code since it only executes the user-defined function ("Algo" in the example code of Fig. 1).

Algorithm 1: ESched pseudo-code

```
while !ComputationEnded do
    Receive a task t
    if t is from one of the workers then
        Update dependencies
        Remove t from the DAG
        foreach available worker w do
            t ← Select a ready task
            Send t to w
        end
        if size(DAG)<threshold then
            Enable input from GD
        end
    else
        Calculate t dependencies
        Add t to the DAG
        if size(DAG)>threshold then
            Disable input from GD
        end
    end
end
```

3.2 Task scheduling policies

Worker threads receive tasks through an *on-demand* protocol, i.e. tasks are scheduled upon request. This policy ensures a very good work load balancing without using more complex and costly work-stealing techniques. When multiple fireable tasks are available, they are enqueued in a local buffer by the ESched. When the worker completes the computation on a task, a notification is sent back to ESched, so that it can update the graph with the new dependencies, and the computed node is removed from the graph freeing memory space. Using a centralised entity for the task scheduling does not limit the optimisations discussed

in Section 2, on the contrary this allows the implementation of simpler and efficient algorithms without incurring in the extra complexity and overheads of concurrent implementations.

The (currently) available ffMDF scheduling policies (SchedP) are the following ones:

SIMPLE (S): the tasks to be executed are selected on a FIFO order basis: the first task becoming *fireable* is the first one executed; this is considered the basic scheduling strategy. It entirely relies on the FastFlow task-farm support, so that it basically came "for free".

LOCALITY FIFO (LF): a locality-oriented scheduling, implemented by using multiple ready queues, one per worker thread. Tasks that become fireable after the completion of a given task executed by the worker i, are inserted in a ready queue associated to the worker i. Tasks scheduled to the worker i are extracted in FIFO ordering by the ready queue i. When the queue is empty, tasks are stolen from other workers queues, implementing a kind of centralized work stealing strategy.

LOCALITY LIFO (LL): another locality-oriented scheduling that works exactly as the LF policy, with the only exception that tasks are extracted from the ready queues in a LIFO order (i.e. the last inserted task is the first to be extracted), possibly guaranteeing even more cache locality than the previous case.

PARALLELISM (P): a parallelism-oriented scheduling, in which ready tasks are kept in a single queue. The first task to be executed is the ready task with the higher number of forward dependencies in the DAG following the concept expressed in Section 2. This policy has been implemented by using a single priority queue.

LOCALITY PARALLELISM (LP): a mix of parallelism and locality-oriented scheduling policies, in which we ensure locality by using a queue per worker thread, as in the LL and LF policies, and parallelism by extracting from the queues using the priority mechanism of the P policy.

Although more complex policies can be added, we tried to keep them as simple as possible in order to avoid the case in which the ESched stage is the main bottleneck of the ffMDF pattern when fine grain DAGs are executed.

Algorithm 2: Worker pseudo-code

```
while !ComputationEnded do
    Get a task from ESched
    Compute the task
    Send the completed task to ESched
end
```

4 Experiments

In this section we validate the implementation of the ffMDF run-time using the LU and Cholesky factorisation algorithms (hereinafter CHOL) on a dense matrix of single precision complex elements. Three platforms are used in the evaluation: *SB*) a 16-core machine with 2 CPUs eight-core 2-way hyperthreading Intel Sandy Bridge Xeon E5-2650 2.0GHz with 20MB L3 cache and 32GB of RAM; *NH*) a 32-core machine with 4 CPUs eight-core 2-way hyperthreading Intel Nehalem Xeon E7-4820 2.0GHz with 18MB L3 cache and 64GB of RAM; and *MC*) a 24-core machine with 2 CPUs twelve-core AMD Magny-Cours Opteron 6176 2.3GHz with 12MB L3 cache shared by two groups of 6 cores, no hyperthreading support and 32GB of RAM. For the sake of conciseness, since the SB and NH machines deliver similar qualitative results, we mainly report the results obtained on the SB machine. The three servers run the same Linux x86_64 distribution.

In the experiments we consider different matrix sizes to evaluate the run-time in different conditions of task granularity and total number of tasks. We use relatively large matrices of 4096×4096, 8192×8192 and 16384×16384 single precision complex numbers, which are a sub-set of the problem size considered in similar existing work [11, 10, 17]. We also compare our implementation using smaller matrices of 512×512 and 1024×1024 elements. These sizes represent a challenging scenario for a dynamic run-time support, since we are forced to use smaller blocks to extract enough parallelism. The resulting graph is a fine grain DAG. In general, fine grain DLA DAGs are those graphs whose nodes represent a computation equivalent to few thousand (1-10) of floating point instructions.

A very important parameter of block based algorithms is the block size, which affects both the number of tasks in the graph and the amount of parallelism. Decreasing too much the block size produces two negative effects on the parallel execution time: having smaller blocks imply that i) the run-time support is more frequently called therefore its overhead may eventually affect the performance, and ii) the low level kernel sequential execution is less efficient. A careful study of the trade-off between speedup and se-

quential time is required to obtain the best parallel execution time. In this section, except when stated otherwise, we use the block size that enables the best execution time on the considered platform.

4.1 Scheduling strategies and overall speedup

We start with a brief analysis of the different scheduling policies for the ffMDF run-time described in Sec. 3, using relatively small matrix size (the same results are obtained for larger matrices) considering the LU factorisation (see Fig 2-left). The experimental results match our expectations: the `SIMPLE` scheduling is the worst of the set, while `LP` performs better than the others since it implements both locality and parallelism optimisations. Given the optimal results of `LP`, we use this policy for all the other tests.

Then, we measured the average time the ESched takes to fetch a task from one of the input queues and to execute the Algorithm 1. We obtain an average computation time in clock cycles of ∼1200. Taking into account that for the considered algorithms, when using very small blocks of 32×32 the average computation time per block is at least one order of magnitude higher, the ESched thread is not a bottleneck of the system for any reasonable block size.

In Fig. 2-right, it is sketched the execution time and the speedup obtained on the SB machine for the 2 algorithms when considering large matrices (8192×8192). The ffMDF implementation is able to obtain very good time decrease and almost ideal speedup for both algorithms up to 16 workers threads (18 threads used in total). The same test for smaller matrix (1024(64)–not reported here for lack of space) exhibits good execution time decrease but not the maximum speedup possible (8.5 and 5.4 for LU and CHOL, respectively). This is mainly due to the lack of independent tasks that can be executed concurrently for the considered block size.

4.2 ffMDF vs `PLASMA` library

We compare the ffMDF run-time against the `PLASMA` library (2.5.0b1). The block Cholesky factorisation implemented uses the left-looking variant of the algorithm. The same version is used in the `PLASMA` library when the static run-time is selected; the dynamic run-time uses the right-looking version [17]. For the LU factorisation, the algorithm used is the same.

The concurrency degrees of the ffMDF and `PLASMA` run-time are different, the ffMDF run-time uses two extra threads: one (GD) for DAG's instruction generation, and one (ESched) for graph building and task scheduling purposes. The number of working threads (workers), i.e. threads that perform low level kernel operations, are the same in both versions. For the sake of readability, in the following graphs is reported the number of working threads rather than the actual concurrency degree. In order to present a fair comparison, Table 1 shows the best times achievable for the various considered implementations. For the low level kernel, we used the `PLASMA` wrappers to the BLAS and LAPACK functions. `PLASMA` has been compiled against the Intel MKL library (version 11.0).

In Fig. 3, we compare the ffMDF with the two `PLASMA` run-time versions, varying the block size on the SB platform. `PLASMA-S` and `PLASMA-D` are the static and dynamic run-time, respectively. For this test, we selected small matrices so that the different run-time support overhead becomes more evident due to the finer grain computation. All tests were performed using 16 worker threads. As expected, `PLASMA-S` and `PLASMA-D` have very different behaviour depending on the block size. `PLASMA-S` confirms its low run-time overhead, being able to deliver good results using blocks with a size unacceptable for the `PLASMA-D` run-time. Conversely, the `PLASMA-D` version works well only with blocks larger than 128 that, given the small size of the matrices, provide too few concurrent tasks to guarantee good performances with 16 cores. The ffMDF implementation obtains very good results start-

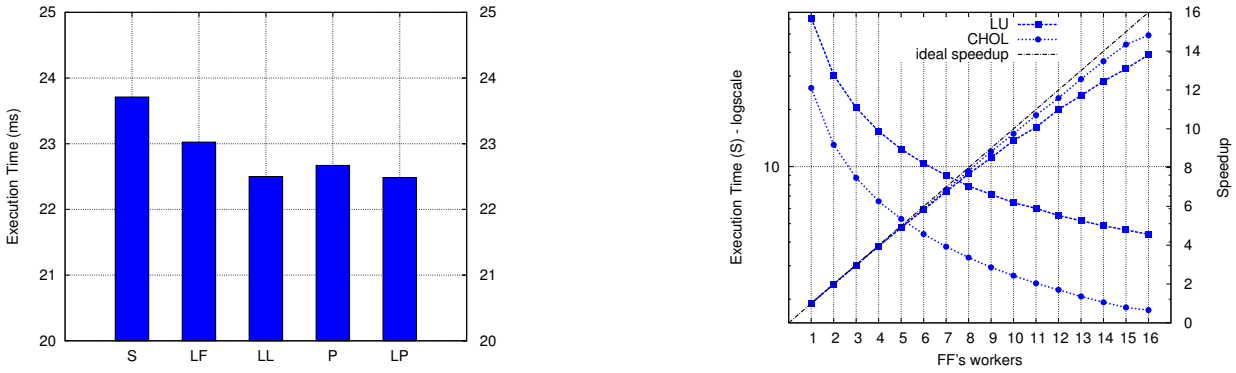


Figure 2: ffMDF on the SB platform. Left): LU execution time for different scheduling policies (size 1024(64)). Right): Execution time and speedup for LU and CHOL, 8192(512) and 8192(256), respectively.

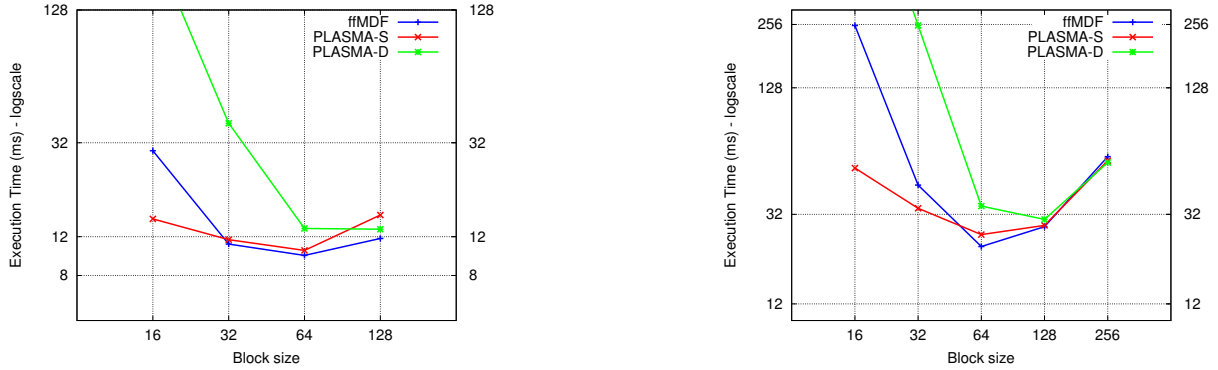


Figure 3: ffMDF vs PLASMA varying the block size, LU algorithm. Left): 512×512 matrix. Right): 1024×1024 matrix.

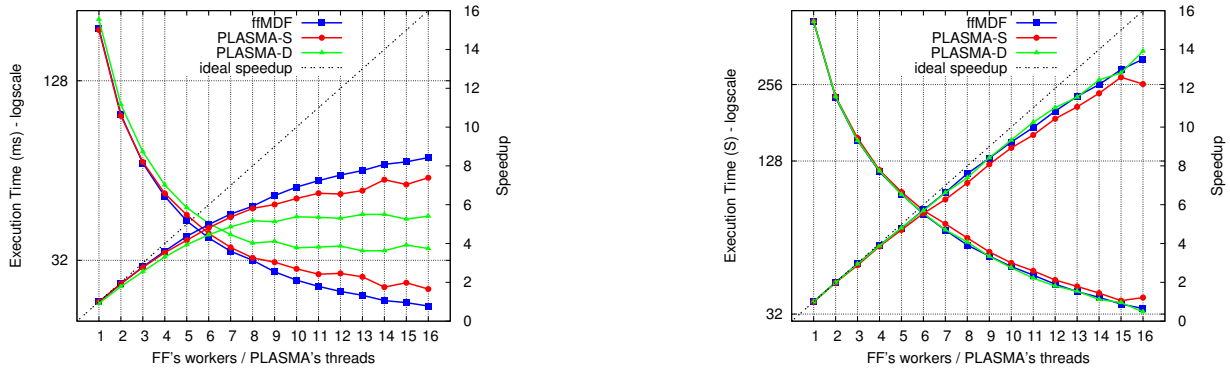
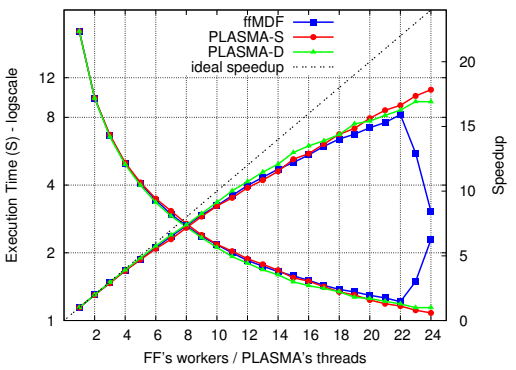


Figure 4: ffMDF vs PLASMA small vs big matrices(blocks), LU algorithm. Left): 1024(64). Right): 16384(1024).

ing with block sizes of 32×32; for smaller blocks, the difference with PLASMA-S is mainly due to the overhead introduced by the dynamic scheduling for the many small tasks of the graph.

This behaviour is confirmed with the experiment reported in Fig. 4, where bigger matrices are analysed varying the parallelism degree. The graph on the left-hand side shows that the ffMDF run-time obtains the best performance with less than 16 workers, obtaining a speedup of ~ 8.5 and an overall gain of $+12\%$ and $+34\%$ w.r.t. the PLASMA-S and the PLASMA-D, respectively. Figure 4-right shows that when larger matrices are considered the PLASMA library performance is inverted considering the two run-time versions: the dynamic implementation works better than the static one. The ffMDF implementation obtains almost the same performance of the PLASMA-D version up to 15 worker threads and then spots a performance penalty of -3% (32.64s vs 33.62s). Although not reported here due to space constraints, with smaller matrix (8192×8192 elements) the performance penalty measured of the ffMDF run-time drops to less than 1%. The small performance degradation is mainly due to the extra thread used in the ffMDF run-time w.r.t. to PLASMA-D. In particular with 16 worker threads, we have both the GD and the ESched threads mapped on cores hosting also a worker thread (each thread pinned on a separate HW context of the 2-way hyperthreading CPU support). When the computation granularity of the task is high (several hundreds of thousands of clock cycles), the "noise" introduced by the extra thread mapped on the same physical core of the ESched and GD threads is not negligible though very small. In this scenario, however, the performance of the application is highly dependent on the HW threading implementation offered by the platform, in fact, while such "noise" is bounded in a small range (less than 3%) on the SB machine, the same test (not reported here for lack of space) executed on the NH machine (32-core 2-way hyperthreading) using 32 worker threads, produces opposite results: the ffMDF takes 38.5s whereas the PLASMA-D implementation takes 41.6s with a gain of $\sim 7\%$ (see also Fig. 5-right for the case 8192(256)).

When hyperthreading support is not available on the platform, we expect an important performance drop by using multiprogramming for large matrices. We estimated such overhead by running the LU factorisation algorithm on the MC platform. The obtained results are sketched in Fig. 5- Left. The performance up to 22 worker threads, spots the same trend as in the SB and NH platforms since we have 1 thread per physical core. With more worker threads the performance dramatically decreases due to the high con-



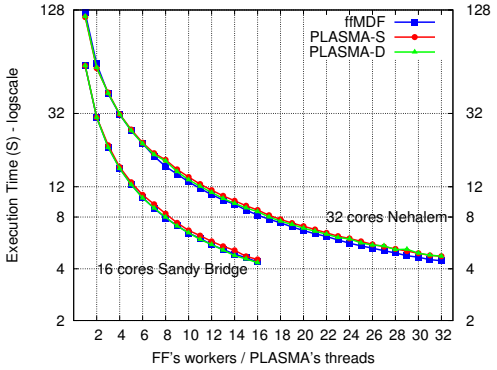
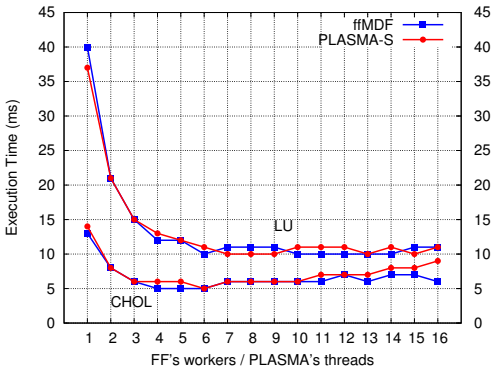


Figure 5: Left): ffMDF vs PLASMA on MC platform, LU algorithm 4096(256). Right): Comparing performance on SB and NH platforms 8192(512) on SB and 8192(256) on NH.



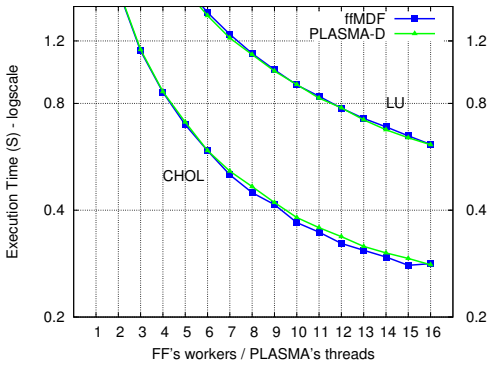


Figure 6: ffMDF vs PLASMA, LU and CHOL algorithms on SB platform. Left): 512(32). Right): 4096(256).

tention produced by the non-blocking run-time used in the implementation of the ffMDF. This behaviour is particularly critical for big matrices where, without hyperthreading, we loose 2 cores whereas for small matrices, where the maximum performance is obtained with fewer worker threads, the performance of the ffMDF run-time is almost the same of the PLASMA static run-time. The measured penalty on this platform for big matrices is $\sim 2/24$ ($\sim$8%). On architecture with hyperthreading support the non-blocking run-time used in the FastFlow framework for implementing efficient synchronisations, does not impair performance for 2 main reasons: i) the context switch overhead between the 2 threads is negligible, and ii) busy-waiting instructions and other instructions of the second thread (the Worker) are pipelined together in the core's functional units, [14].

In Fig. 5-right it is reported the execution time on the NH and SB platforms. It is worth pointing out that the minimum service time obtained on the NH platform using 32 worker threads is almost the same of the one obtained on the SB platform using 16 worker threads (4.45s vs 4.39). The Intel MKL library on the SB platform exploits the AVX instruction set which uses SIMD registers of 256 bits instead of the 128 bits register used in the NH platform. The execution time on the NH platform is almost twice the time obtained on the SB platform proving that: i) the AVX implementation is quite good and is able to double the performance for this kind of applications, and ii) the ESched thread does not introduce significant delay also in the presence of relatively high parallelism degrees.

Finally, in Fig. 6 we compare both LU and CHOL algorithms on the SB platform considering *the best* PLASMA implementation for the 2 cases: PLASMA-S for the case 512×512 and PLASMA-D for the case 4096×4096. The performance obtained in both cases by the ffMDF implementation is better than or comparable with the one obtained by the best implementation of the PLASMA library. This confirms our initial claim: it is possible to implement a dynamic run-time support able to obtain optimal performance for any DAG granularity on modern shared-cache multi-core.

As final summary, in Table 1 are reported the best times for ffMDF, PLASMA-S and PLASMA-D obtained in executing the LU and CHOL benchmarks over the SB platform. The last column shows the percentage gain of the ffMDF implementation over the best one of the PLASMA library.

Bench.	Size	ffMDF	PLASMA-S	PLASMA-D	% G.
LU	512(64)	**8.69**	9.30	9.95	6.5
	1024(64)	**22.48**	25.63	34.43	12.3
	4096(256)	612.89	613.44	**611.69**	-0.2
	8192(512)	4391.24	4531.45	**4335.92**	-1.3
	16384(1024)	33622	37148	**32641**	-2.9
CHOL	512(32)	**5.43**	5.98	10.37	9.2
	1024(64)	**12.02**	15.04	19.19	20
	4096(256)	279.77	**278.98**	280.81	-0.28
	8192(256)	1751.3	1781.39	**1738.19**	-0.75
	16384(512)	13516	13185	**13136**	-2.8

Table 1: Best times (in *ms*) obtained in executing the benchmarks on the SB platform

5 Related Work

A large research effort has been devoted to studying efficient run-time supports for the execution of parallel programs expressed as DAG graphs [15, 20, 6, 24, 8].

In *Cilk* [15], tasks represent procedures performed by spawning a set of threads for each recursive call. Load balancing is achieved through a randomised work stealing technique and is based on a classic fork-and-join pattern. This makes *Cilk* particularly good for highly recursive code and not well suited to extract parallelism from a data dependencies DAG.

SMP Superscalar [20] (SMPss) is a parallel programming environment based on compile time annotations. SMPss and Cilk share some common points. The programmer is responsible for explicitly identifying tasks, and load balancing is achieved using a cache-aware work stealing technique optimising data locality. Compared to Cilk, SMPss handles the computation of arbitrary DAGs which are automatically constructed from an abstract representation of tasks and their input/output dependencies.

PaRSEC [8] is another scheduling environment for graphs of tasks. Differently from the others, it targets clusters of multicore, therefore handling data exchange among the machines by means of communication mechanisms such as MPI. Another distinctive point of PaRSEC is the use of a symbolic, problem-size-independent representation of the DAG, that relieves the authors from the run-time creation of the task graph. It has been used in DPLASMA [7], the distributed version of `PLASMA`.

StarPU [6] is focused on handling accelerators such as GPUs. Graph tasks are scheduled by its run-time support on both the CPU and various accelerators, provided the programmer has given a task implementation for each architecture.

QUARK is the run-time support for the dynamic scheduling execution of the `PLASMA` library [19]. In QUARK, the application algorithm is expressed by means of a sequence of tasks each one composed of the function to be computed and of parameters list. The construction of the DAG is performed sequentially by a (*master*) thread during the computation, maintaining only a graph "window" to bound memory usage with very large graphs. The master schedules tasks towards a pool of worker threads each one having an input queue. Work-load balance is obtained using a work stealing algorithm. Once a task is executed, the dependency graph is updated either by the master or by the workers themselves. Locality on data usage is preserved by scheduling tasks that use the same data on the same worker. A task can also be characterised by a priority, so a task with higher priority will be executed earlier. Both these optimisations require the intervention of the user that has to specify the hints for preserving locality on a given task parameter or the priority of the task.

6 Conclusion

This paper presents and assesses ffMDF, a lightweight dynamic run-time support for efficient execution of DLA on modern shared-cache multi-core platforms. Two important DLA algorithms have been studied: LU and Cholesky factorisation. For both of them we conduct an analysis of the performance when both small and big matrices are taken into account.

The performance obtained is compared with those achieved by using the efficient, and widely used `PLASMA` numerical library [2]. `PLASMA` has been designed to provide the best performance in the execution of DLA algorithms on multi-core platforms by using two separate run-time variants: a static run-time optimised for small problem size, and a dynamic run-time optimized for large problem size. The results show that: i) for large matrices, ffMDF is able to obtain comparable performance of the `PLASMA` dynamic run-time with a maximum performance penalty, in the worst case, less than 3% on machine with hyperthreading support (and 8% on the tested machine without hyperthreading), instead, for small matrices, ffMDF is able to obtain better performance, up to more than 10% and 30% with respect to the static and dynamic `PLASMA` run-time, respectively.

As future work, we intend to deeply analyse which are the performance contributions provided by the low level non-blocking synchronisation mechanisms used to implement ffMDF. Furthermore, in order to extend the work for targeting heterogeneous many-core platforms, we plan to investigate other skeleton implementations for the dynamic run-time.

References

[1] *FastFlow website*, 2013. `http://mc-fastflow.sourceforge.net/`.

[2] *PLASMA library website*, 2013. `http://icl.cs.utk.edu/plasma/`.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating code on multi-cores with fastflow. In E. Jeannot, R. Namyst, and J. Roman, editors, *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, aug 2011. Springer.

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 662–673. Springer, aug 2012.

[5] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. Wiley, 2013.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, Feb. 2011.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1432–1441, 2011.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. Dague: A generic distributed {DAG} engine for high performance computing. *Parallel Computing*, 38(12):37 – 51, 2012.

[9] D. Buono, M. Danelutto, S. Lametti, and M. Torquati. Parallel patterns for general purpose many-core. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 131–139, 2013.

[10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurr. Comput. : Pract. Exper.*, 20(13):1573–1590, Sept. 2008.

[11] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35:38–53, January 2009.

[12] T. A. C. Center. Plasma users guide, parallel linear algebra software for multicore architectures, version 2.3, November 2012.

[13] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[14] T. De Matteis, F. Luporini, G. Mencagli, and M. Vanneschi. Evaluation of architectural supports for fine-grained synchronization mechanisms. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, 2013.

[15] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[16] H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software–Practice & Experience*, 40(12):1135–1160, Nov. 2010.

[17] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurr. Comput. : Pract. Exper.*, 24(3):305–321, Mar. 2011.

[18] J. Kurzak, A. Buttari, and J. Dongarra. Solving systems of linear equations on the cell processor using cholesky factorization. *IEEE Trans. Parallel Distrib. Syst.*, 19:1175–1186, September 2008.

[19] J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. Technical Report 220, LAPACK Working Note, June 2009.

[20] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151, 29 2008-Oct. 1.

[21] S. Ritz, P. Matthias, and M. Heinrich. High level software synthesis for signal processing systems. In *Proceedings of the Intl. Conf. on Application-Specific Array Processors*, pages 679–693. Prentice Hall, IEEE Computer Society, 1992.

[22] F. Song and J. Dongarra. A scalable framework for heterogeneous gpu-based clusters. In *Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 91–100, New York, NY, USA, 2012. ACM.

[23] A. H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, Dec. 1986.

[24] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.