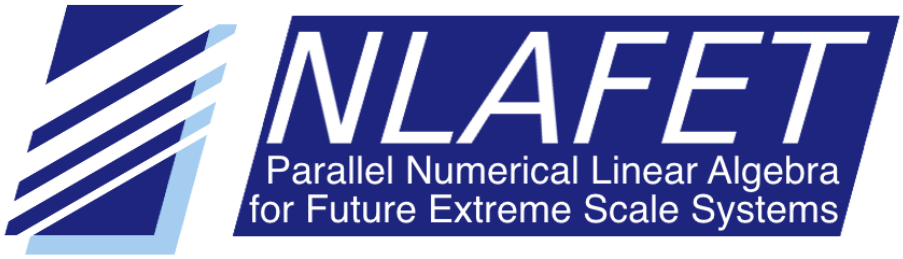

H2020–FETHPC–2014: GA 671633

D6.2

Novel Methods for Static and Dynamic Scheduling

October 2017

Document information

Scheduled delivery	2017-10-31
Actual delivery	2017-10-31
Version	1.0
Responsible partner	UNIMAN

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2017-10-25	Mawussi Zounon	Version	1.0	Feedback from reviewers added.
2017-07-10	Mawussi Zounon	Draft	0.1	Initial version of document produced.

Author(s)

Negin Bagherpour (UNIMAN)
Sven Hammarling (UNIMAN)
Jack Dongarra (UNIMAN)
Mawussi Zounon (UNIMAN)

Internal reviewers

Stojce Nakov (STFC)
Angelika Schwarz (UMU)

Contributors

Neil Walton (UNIMAN)
Thomas McSweeney (UNIMAN)

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

1 Introduction

The *Description of Action* document states for Deliverable D6.2:

> *D6.2: "Evaluation of existing and novel methods for static and dynamic scheduling in various types of HPC systems. Includes documentation of algorithms and prototypes developed."*

This deliverable is in the context of Task–6.1 (Scheduling and Runtime Systems).

The ongoing challenge for high-performance computing (HPC) is the design of systems capable of exascale performance (i.e., 10^{18} floating point operations per second [exaFLOP/s]). To achieve such an ambitious goal with additional power and energy consumption constraints, one must increase parallelism at all levels, with a corresponding and unprecedented increase in heterogeneity. This heterogeneity will go beyond traditional HPC systems—which currently consist of mostly homogeneous, massively parallel nodes—with the addition of accelerators (graphics processing units [GPUs]) and coprocessors (Intel Xeon Phis). In the exascale era, the computing nodes themselves are expected to be very heterogeneous, with lower power consumption nodes based on ARM chips or IBM's reduced instruction set computer (RISC) architecture, regular nodes based on x86 chips (with additional GPU accelerators and Xeon Phi coprocessors thrown into the mix), and very specialized nodes for specific applications. To fully exploit such complex and highly heterogeneous systems, novel scheduling strategies and runtime systems should be investigated to enable scientific and engineering task-based applications to fully utilize these systems.

The primary goal of this deliverable is to evaluate existing scheduling strategies and investigate novel approaches in the context of numerical linear algebra libraries, which are the building blocks of many scientific and engineering simulations. To cope with the permanent change in HPC architectures, the current trend in the numerical linear algebra community is to employ task-based programming models to guarantee the portability and scalability of software. The underlying idea is to rethink each application as a set of tasks to be executed with respect to data dependencies and constraints imposed by the application. Both the data dependency tracking and the task execution are then delegated to a task-based runtime system, which serves as an interface between user applications and the hardware. The effectiveness of runtime systems depends mainly on the scheduling strategies and policies. Consequently, it is critical to devote more energy to evaluating different scheduling methods and analyzing their affinity with high-performance linear algebra libraries.

To assess the performance of existing scheduling strategies in the context of extreme-scale linear algebra libraries, we developed a new library called PlaStar (Section 2.5). This library implements task-based dense linear algebra algorithms, with the flexibility to choose between different scheduling strategies thanks to the StarPU [3] runtime system on which the PlaStar library is based.

The remainder of this work is organized as follows. In Section 2, we discuss the underlying principles of PlaStar and other task-based dense linear algebra libraries that are used for evaluating the scheduling strategy. Section 3 contains a discussion of different static and dynamic runtime systems on various HPC systems. Section 4 outlines promising ideas for novel scheduling strategies. Experimental results are reported in Section 5, and Section 6 contains our concluding remarks.

2 Linear Algebra Libraries and Runtime Systems

The primary use of high-performance linear algebra software in scientific applications is in solving linear systems. Whether one needs to solve a system arising from the discretization of a partial differential equation, fit a model to some data in a least-squares sense, or solve a KKT system in an optimization problem, there will undoubtedly be linear systems to be solved. As applications have evolved, these systems have become larger, necessitating the use of high-performance linear algebra routines to solve them in a reasonable amount of time. Usually, such dense systems are solved using Cholesky, LU, QR, or symmetric-indefinite factorizations. More recently, a few high-performance linear algebra software libraries have been developed to cope with the aforementioned changes in modern HPC systems.

With the advent of multicore architectures, for example, the Linear Algebra PACKage (LAPACK) was introduced to effectively exploit cache-based memory architectures. However, its memory layout—which consists of the subdivision of matrices in block columns—and the fork-join parallelism model lead to a serious performance penalty on modern, massively parallel architectures. To address the limitations of LAPACK-style routines, a new generation of numerical linear algebra libraries has been designed using the novel task-based programming paradigm. Such libraries rely heavily on task-based runtime systems and represent a very rich context for exploring new scheduling strategies. In this section, we present some of the task-based linear algebra libraries and their associated runtime systems.

2.1 The PLASMA Library

The Parallel Linear Algebra for Multicore Architectures (PLASMA) package is a dense linear algebra library designed to fully exploit modern many/multicore shared memory computers. PLASMA implements efficient parallel task-based variants of level-3 Basic Linear Algebra Subprograms (BLAS) and LAPACK routines. Roughly, PLASMA routines consist of three main steps. First, each dense input matrix, initially stored in column-major form, is converted to tile layout. This approach offers many advantages. Small tiles are more likely to fit into the cache memory, which can reduce cache and translation lookaside buffer (TLB) misses, increase opportunities for data prefetching, and—consequently—improve overall performance. Then, at the computation step, each task operates at tile granularity. It is here that efficient runtime systems and scheduling strategies are required to schedule each task to the appropriate computing unit. At the last step, the output matrices are converted back to column-major data layout.

The first version of PLASMA [2] was based on a runtime system called QUeueing And Runtime for Kernels (QUARK) [30], developed by Assim et al. QUARK was developed primarily to serve as a runtime support system for the PLASMA library. It allowed task-based applications to execute tasks asynchronously in many/multicore shared memory architectures. In addition, QUARK permitted both static and dynamic task scheduling.

With the introduction of the task-based paradigm in OpenMP 3.0, followed by the dataflow-based task scheduling provided in the OpenMP 4.0 standard, the QUARK runtime system in PLASMA was replaced with OpenMP [31]. PLASMA's current version, PLASMA 17[1], is now based on OpenMP, with performance comparable to the QUARK-based PLASMA.

[1] `https://bitbucket.org/icl/plasma`

2.2 The DPLASMA Library

Although it is actually a separate library, the Distributed Parallel Linear Algebra for Multicore Architectures (DPLASMA) [7] package can be considered as an extension of PLASMA for use on heterogeneous and distributed-memory architectures. Like PLASMA, DPLASMA is based on tile algorithms, but it relies on a completely different runtime system and task-based programming model. DPLASMA uses PaRSEC [8]; the key difference between ParSEC and other runtimes is the task-based programming model.

Task-based programming models can be divided into two main classes: (1) sequential task flow (STF) models and (2) parametrized task graph (PTG) models. In the STF model, task are inserted sequentially, data dependencies are detected using data access information, and each node unrolls the directed acyclic graph (DAG). Consequently, each node has access to the entire DAG. STF is the most common model; for example, it is implemented by both QUARK and StarPU [3], (Section 2.4). On the other hand, in the PTG model, tasks and their associated dependencies are expressed symbolically. Each node exploits this symbolic representation to extract the portion of the DAG relevant to the tasks it has to execute. This model is implemented by the Parallel Runtime Scheduling and Execution Controller (PaRSEC), which is the runtime system behind DPLASMA.

PaRSEC provides high-performance, architecture-aware features to schedule tasks in both shared and distributed memory architectures and also supports accelerators and coprocessors.

2.3 The MAGMA Library

The MAGMA library [27] is an implementation of LAPACK routines for heterogeneous manycore systems with GPUs. MAGMA is limited to a single hybrid CPU/GPU node but can support multiple GPUs. Unlike PLASMA, DPLASMA, and Chameleon—which are based on tile algorithms with a lot of support from runtime systems for dynamic task scheduling—MAGMA is much closer to the LAPACK parallelism style with 1-D block cyclic data distribution and static scheduling. More details on MAGMA scheduling strategies are discussed in Section 3.

2.4 The Chameleon Library

The Chameleon library is also based on the PLASMA tile algorithm style. The main contribution of the Chameleon library is that it provides a unified interface, developed in the Matrices Over Runtime Systems at Exascale (MORSE) project [1], to define data dependencies and task submission. Chameleon's task management abstract layer permits different low-level runtime systems. Put differently, Chameleon enables a user to choose the appropriate runtime system for scheduling a given linear algebra tile algorithm. So far, Chameleon supports QUARK, PaRSEC, and StarPU.

2.5 Prototype Algorithm

In order to assess different scheduling strategies, we developed a new task based dense linear algebra library called PlaStar. The PLASMA on top of StarPU runtime system (PlaStar) is another variant of the PLASMA library that relies on the StarPU runtime system instead of OpenMP. StarPU is a fully featured runtime system initially designed

for heterogeneous multicore architectures and recently extended for use in distributed-memory systems. PlaStar was initiated in the context of the Parallel Numerical Linear Algebra for Extreme Scale Systems (NLAFET) project and is currently under active development.

The OpenMP-based PLASMA library demonstrates very competitive performance with the highly optimized Intel Math Kernel Library (MKL); however, it is limited to shared-memory architectures. The goal of the PlaStar library is twofold. First, it mains at designing a task-based dense linear algebra solver that gives to users the flexibility exploit different scheduling strategies. Second, it mains at taking the algorithm design effort invested in PLASMA to the next level: heterogeneous and distributed systems. The current version of PlaStar is already fully functional in shared-memory systems. While a few routines are already extended to distributed systems, we are actively working to integrate the cuBLAS and Matrix Algebra on GPU and Multicore Architectures (MAGMA see Section 2.3) kernels into PlaStar to add support for GPUs. In the long term, this library will serve as a framework for the implementation of fault-tolerant algorithms presented in the deliverable D6.6.

We chose to use StarPU in our project because of its maturity and because it implements most state-of-the-art scheduling strategies. For these reasons, StarPU is the most suitable choice for a fair comparison of different scheduling strategies. In addition, it is an appropriate framework for investigating novel scheduling strategies, because it allows users to implement their own custom scheduling policies. The PlaStar library for the evaluation of different scheduling strategies is available on the NLAFET GitHub project in the repository NLAFET/PlaStar that can be found at https://github.com/NLAFET/PlaStar.

3 Scheduling Strategies for Various HPC Systems

The ultimate goal of any scheduling strategy is to achieve an optimal mapping of computation resources to tasks, given some optimization targets. The main target in HPC applications is the minimization of the overall execution time, but minimization of power and energy consumption might also be desirable. While the target seems clearly defined, there are a large number of research papers that have explored a wide spectrum of scheduling strategies and policies [11]. This large variation in scheduling strategies boils down to the fact that the perfect scheduling strategy for one class of problems may be completely useless for another. However, even though each problem may require a specific scheduling policy, some generic strategies have proven successful in many cases.

3.1 Generic Scheduling Strategies

Random Assume we have a system of n workers and an overall performance of P, that each worker $i \in [1, n]$ has a performance of P_i, and let $r_i = \frac{P_i}{P}$ denote the relative performance of each worker. The random scheduling strategy consists of assigning $r_i\%$ of the work load to each worker, i. Basically, a worker with a high processing speed is assigned more work. The efficiency of this strategy depends strongly on the accuracy of the performance model. The main drawback is the lack of a dynamic load balancing scheme to correct potential errors in the estimation of the performance of each worker and of the whole system.

Greedy/Eager All the tasks ready to be executed are stored in a single queue that is shared by all workers. Each worker pop a task from the centralized queue as soon as it becomes idle. The main drawback of this strategy is that tasks are scheduled very late, which prevents the possibility of data prefetching, data reuse, and data transfer and computations overlapping. Nevertheless, this strategy optimizes load balancing on homogeneous systems.

Work Stealing Work stealing is a decentralized strategy with a local queue of tasks for each worker. All tasks assigned to a worker are added to a queue. An idle worker can steal a task from the most loaded worker. This strategy has an efficient load balancing policy but requires checking each worker's workload before stealing a task.

Local Work Stealing A decentralized strategy similar to the work stealing strategy. The difference is that, when becoming idle, a worker steals a task from the queue of a neighboring worker, rather than the most loaded worker.

These generic scheduling strategies may be very efficient on homogeneous systems where all the workers have the same processing speed. But the design of a high-performance scheduler for heterogeneous systems requires considering the processing speed of every worker and estimating the completion time of each task on each of the workers. Scheduling strategies exploiting this knowledge are discussed in the following section.

3.2 Scheduling Strategies Based on Performance Models

The main idea behind scheduling strategies based on performance models is to estimate the cost of scheduling a given task on each worker to help the scheduler minimize the application's execution cost. This cost can include the execution time, memory consumption, and/or power and energy consumption. To this end, one may exploit theoretical or empirical methods in this strategy. Theoretical approaches involve considering the peak processing time of each worker (CPU core or GPU), the bandwidth and latency of the available memory, and the time to solution and memory complexity of each task. Based on this information, the scheduler can approximate the completion time of each task for each worker and make the appropriate scheduling decisions to reduce the overall execution time. Unfortunately, on many systems, peak performance estimates can differ significantly from the actual performance seen during execution. This discrepancy/gap is often closed by improving the performance model itself by measuring the real performance parameters of the system and the execution time of each task. Depending on the application, an accurate performance model may facilitate the design of a static scheduling strategy that has comparable performance to a dynamic strategy.

3.2.1 Dynamic Scheduling

Once the performance model is obtained, it can be exploited to minimize the overall completion time of each task. This approach is related to dynamic programming or dynamic optimization. The reduction of the application's overall execution time is achieved by scheduling a task to a worker whose current task is expected to complete early. This strategy denoted heterogeneous earliest finish time (HEFT) is implemented by StarPU. The scheduler records the tasks already scheduled for each worker; the scheduler can then estimate the date of availability for each worker and deduce the date of completion for

the current task to schedule on each worker. During the execution, the estimation errors are corrected by updating the performance model with the real termination data of completed tasks. This policy can also be extended to consider the data transfer between the CPU and the GPUs, as implemented in StarPU as the deque model data aware (DMDA) policy.

3.2.2 Static Scheduling

Some dense linear algebra applications have predictable compute patterns, which is a prerequisite for successful static scheduling. In addition, the huge gap between modern multicore CPUs and advanced accelerators/GPUs and coprocessors simplifies scheduling decisions for some well known dense linear algebra applications. As a consequence, a static scheduling strategy based on the hardware configuration and software stack of the system, as well as performance analysis and profiling of the target routine, may be very competitive compared to dynamic scheduling strategies.

Depending on the arithmetic intensity and data volume of a given application, it may be divided into two blocks: (1) a GPU-friendly block and (2) a CPU-friendly block. GPU-friendly routines are characterized by a high arithmetic intensity and a large amount of data (e.g., large dense matrix–matrix multiplication kernels). On the other hand, CPU-friendly routines consist of very complex algorithms that require sophisticated low-level optimizations like branch prediction, resolution of data dependence hazards, and efficient management of buffer cache/memory requests.

With this in mind, designing a scheduling strategy for hybrid CPU/GPU architectures can now be reduced to the identification of the CPU-friendly blocks and the GPU-friendly blocks that compose a given application. The MAGMA library, for example, uses this approach, and all MAGMA routines are based on static scheduling strategies [13, 14]. Roughly, matrix factorization algorithms consist of a recursion of a panel factorization followed by an update to the trailing matrix. Since the panel factorization is CPU friendly and the trailing matrix update is floating-point intense (i.e., GPU friendly), MAGMA starts the factorization by moving the whole matrix to the GPU and then uses the CPU cores for the panel factorization.

3.3 Hybrid Dynamic/Static Scheduling Strategies

Static scheduling strategies are efficient for dense linear algebra kernels on hybrid CPU/GPU architectures, but they reveal their limitations when using complex applications and/or distributed-memory systems with many GPUs. A reasonable compromise may consist of combining these static scheduling strategies with efficient dynamic scheduling strategies, as investigated by Suraj Kumar et al. [17].

3.3.1 The Case of Heterogeneous Multicore/Accelerator Architectures

Since some tasks may be labeled as CPU friendly and others labeled as GPU friendly, it seems like an attractive option to simply statically schedule a given task to either CPU cores or GPUs, depending on the task's affinity. Unfortunately, this strategy may lead to severe resource starvation. For example, when all tasks involving GPU-friendly routines are finished, the GPUs remain idle while other tasks that were statically scheduled to the CPU may still be waiting in the queue to be executed. In those cases, dynamic scheduling,

based on task and resource affinity, can be an alternative. The idea behind the affinity-based dynamic scheduling strategy is to go beyond the dichotomy that connects tasks to a single resource. Instead, each task has a list of resource affinities (i.e., starting with the types of hardware on which the task performs best). When a worker becomes idle, the next task is selected based on its affinity for the available (idle) hardware, relative to the other remaining tasks [18].

In [5], Oliver Beaumont et al. provided theoretical analysis of the affinity-based strategy's performance in the context of dense linear algebra on a heterogeneous CPU/GPU architecture. They also pointed out a serious performance issue related to this strategy. For example, when all CPU-friendly tasks are finished, an idle CPU can be assigned to a GPU-friendly task, although a GPU may become idle shortly after. They improved the scheduler using a strategy called "spoliation." Basically, when a task is assigned to a resource on which it will be very slow, it can be canceled and reassigned when a more appropriate worker becomes idle. The decision is not automatic; it depends on the estimated completion time and how far the non-appropriate resource has gone in the computation. That said, this approach has shown very promising results, and the strategy has been extended to several classes of heterogeneous workers [4].

3.3.2 Distributed-Memory Systems

The increasing gap between the high computation peak and the stagnating communication speed of modern distributed-memory HPC systems makes the interconnect bandwidth a critical resource that must be used carefully. Consequently, inter-node dynamic load balancing strategies may fail to reduce the application completion time and even lead to a performance penalty. The consequence for scheduling is that dynamic scheduling should be limited to the node level, while efficient static scheduling strategies must be employed across nodes to reduce communication costs caused by dynamic work balancing.

In distributed dense linear algebra, for example, it is common to use 2-D block cyclic data distribution of matrices across nodes, as promoted by the Scalable Linear Algebra PACKage (ScaLAPACK) [28]. This data distribution is exploited in libraries like DPLASMA and Chameleon to statically schedule tasks to nodes based on ownership of the data required by the task (i.e., each node will be assigned a set of tasks that have input and/or output dependencies on its data. After this first step of static scheduling, various dynamic scheduling strategies are then used to assign tasks to CPU cores and GPUs within each node.

3.4 Multilevel Scheduling Strategies

The new generation of dense linear algebra libraries (e.g., PLASMA, DPLASMA, PlaStar, and Chameleon) owe their high efficiency, to some extent, to the tile memory layout. The tile layout offers more room for parallelism, and dividing large matrices into small tiles enables us to exploit cache memory more efficiently. To fit into caches, tile sizes are optimized at core granularity. This strategy has demonstrated its efficiency on multicore architectures but seems limited on massively parallel, heterogeneous multicore machines with GPUs. A tile size optimized for CPU cores will be extremely penalizing for GPUs, since GPUs require large matrices to deliver reasonable performance. In some cases, a compromise on the tile size may decrease the performance of both the CPU and the GPU, thereby degrading the system's overall performance.

Alternatively, one can design a multilevel scheduling strategy. The main idea here is to enable different levels of task granularity and use an appropriate scheduler at each level. At the highest level, large matrix tasks can be scheduled to the multicore CPU and to the GPU. The GPU will then be more likely to process the large matrix task directly, while the multicore CPU may divide the tasks in a cache-friendly manner and use another scheduler to assign these tasks to the available CPU cores. A variant of the multilevel strategy has been investigated in [9] by Cojean et al. To improve the load balance between CPU cores and GPUs, they grouped multiple CPU cores together to form virtual resources. For multilevel task scheduling, they extended the StarPU runtime system to simulate an external runtime system that works at the aggregated resource level and an inner runtime system that operates within the virtual resources.

A more general version of the multilevel scheduling calledn "hierarchical DAG," was proposed in [29] and implemented in the PaRSEC framework for distributed hybrid environments. While the resource aggregation approach tends to limit the heterogeneity in different processing units by aggregating processing units to create more balanced virtual resources, the hierarchical DAG adopts another approach. The hierarchical DAG allows tiles of different sizes to coexist in the same runtime system, and—depending on the availability of resources—a large task can be dynamically subdivided into a finer-granularity inner DAG. The authors demonstrated that, in the context of dense linear algebra, the PaRSEC version based on the hierarchical DAG strategy (h-PaRSEC), has a significant performance gain on distributed heterogeneous nodes compared to the standard ParSEC implementation.

4 Towards Novel Scheduling Strategies

Despite a large collection of literature on scheduling strategies, the increasing heterogeneity of modern HPC processing units means that the need for novel scheduling approaches is greater than ever. We envision adapting scheduling strategies that are already being used successfully in other fields to extreme-scale dense linear algebra computations. Below, we present some of the scheduling strategies that we believe have the potential for these future heterogeneous HPC systems.

These approaches are divided into three primary methods: (1) distributed scheduling, (2) myopic scheduling, and (3) approximate-dynamic programming. The aim is to provide a theoretical analysis of these algorithms and to pursue an agenda for research and implementation. Note that this analysis will go beyond the scope of this deliverable thanks to a collaboration with experts in statistics and queue theory at the University of Manchester.

4.1 Distributed Scheduling

Here, the aim is to study how congestion avoidance can be implemented through the communication of appropriate "prices." For instance, in the case of the transmission control protocol (TCP), this shadow price consists of the packet drop probability [15], which is used to guide several TCP connections toward an optimal operating point [16]. These primal-dual approaches have been developed to optimize task assignment with heterogeneous server pools in service systems and in virtual machine placement [24, 12, 25]. We believe that further investigation of these approaches will help substantially in

designing efficient scheduling strategies for heterogeneous and distributed dense linear algebra kernels where communication represents a bottleneck.

4.2 Myopic Scheduling

We will analyze how system throughput can be optimized by leveraging information on the system's current state to appropriately schedule tasks. One class of algorithms, called "BackPressure/MaxWeight," employs information about the queue size to provide stability properties. This method was first employed for scheduling in wireless ad-hoc networks [26] and was then included in the development of input-queued network switch design [20]. This approach is currently considered in fork-join networks (or in DAGs in HPC applications) [23], and similar approaches were developed in the context of the MapReduce system.

In homogeneous systems, other heuristics are commonly considered. Specifically, prioritizing servers with low loads while keeping a low message passing overhead. An initial proposal was the "power-of-d" choices paradigm, where two random servers are chosen and then jobs are routed to the server with the shorter queue [21]. This approach is successfully deployed in hash function design, Google used it for virtual machine placement, and Spark used power-of-1.1 choices. An alternative approach is to prioritize idle servers that pull jobs when idle; this method was implemented in Microsoft's cloud servers [19]. Other approaches involve replicating jobs to reduce the (expected) make span.

4.3 Approximate Dynamic Programming

The above approaches do not allow for the level of forward planning required for HPC applications. Task assignment and scheduling is closely linked to planning along the critical path of a DAG. Of course, the critical path can be solved through dynamic programming (through Bellman-Ford, in particular); however, given the potential stochastic effects from unknowns, it could be worth considering larger stochastic dynamic programs (i.e., Markov decision processes). Although the full solution of these may not be computationally traceable, more modern approximate dynamic programming approaches might be applicable. In fact, we will investigate the use of neuro-dynamic programming approaches [6], where the Markov-decision processes' Q-function is approximated by a neural network. This was successfully employed more recently in the context of reinforcement learning of games [22]. Further, some distributed approaches can be implemented and parallelized [10].

Thus, the primary goal is to better understand how reinforcement learning can be applied to provide better estimation and scheduling by generalizing the current critical path analysis approaches that are currently used, for instance, in HEFT and in existing runtime systems.

5 Experimental Results

5.1 Experimental Setups

Our experiments are designed to evaluate the efficiency of different scheduling strategies for solving dense linear algebra problems on various HPC systems. Note that the comparison of dense linear algebra algorithms or libraries is out of the scope of this work, and we

rely on these libraries only as a framework to assess the potential of different scheduling approaches for exploiting computational resources at full efficiency.

As described below, for our experiments, we considered three different multicore architectures with two different types of NVIDIA GPUs.

Intel Xeon Haswell For Haswell, we employed a 10-core Intel Xeon E5-2650 v3 CPU, with a base core clock frequency of 2.30 GHz, running on a dual-socket, non-uniform memory access (NUMA) node (20 CPU cores per node). In double precision, its theoretical peak performance is 736 gigaFLOP/s for the 20 cores, with 32 GB of total main memory (dynamic random access memory [DRAM]). The libraries were compiled using GNU Compiler Collection (GCC) 7.1.0 and linked to MKL 17.2. Memory was allocated on the two NUMA nodes in a round-robin fashion using `numctl` commands (i.e., `numactl -interleave=all`).

Intel Xeon Broadwell For Broadwell, we employed a 10-core Intel Xeon E5-2690 v4 CPU, with a base core clock frequency of 2.6 GHz (3.5 GHz in Turbo mode), running in a dual-socket configuration (28 CPU cores per node). Each core achieves up to 16 FLOPs per cycle in double-precision arithmetic. Consequently, in double precision, the theoretical peak performance of the whole node (all 28 cores) is 1.20 teraFLOP/s when set in base frequency mode. The libraries were compiled using GCC 6.3 and linked to MKL 2017. Memory was allocated on the two NUMA nodes in a round-robin fashion.

Intel Xeon Phi Coprocessor We also performed experiments on the new self-hosted Intel Xeon Phi coprocessor, code named Knights Landing (KNL). This 68-core 7250 KNL chip can achieve up to 3.05 teraFLOP/s in double precision. The KNL has two types of memory: (1) a large DDR4 memory with a low bandwidth (115.2 GB/s) and (2) a smaller 16 GB MCDRAM providing up to 490 GB/s of sustained bandwidth. We configured the KNL in a flat mode, making the whole 16 GB MCDRAM available to be allocated directly by the application. During the experiments, all of the memory allocations were restricted to the 16 GB high-bandwidth memory using `numactl -membind=1`, where "`1`" is the ID of the MCDRAM node. The libraries were compiled with GCC 7.0.1 and linked to MKL 2017.2.174.

NVIDIA K40c GPU NIVDIA's K40c is based on the Kepler GPU architecture and consists of 2,880 CUDA cores. With 12 GB memory providing up to 288 GB/s of bandwidth, the K40c GPU can achieve a peak performance of 1.43 teraFLOP/s in double precision. The kernels are executed on this GPU using CUDA 8.

NVIDIA P100 GPU NVIDIA's P100 is based on the Pascal GPU architecture and consists of 2,880 CUDA cores. The P100 GPU can achieve a peak performance of 4.67 teraFLOP/s in double precision (i.e., it is three times faster than the K40c GPU). The kernels are executed on this GPU using CUDA 8.

5.2 Homogeneous Multicore Architectures and KNL

This experiment is based on the PlaStar library with the primary goal of comparing current state-of-the-art dynamic scheduling strategies. We selected two PLASMA algorithms

(the tile QR factorization algorithm and the tile Cholesky algorithm) to assess the efficiency of four dynamic scheduling strategies. These strategies, which are described in more detail in Section 3, are: random, greedy (eager), work stealing (ws), and local work stealing (lws). The results for Haswell are shown in Figure 1, the results for Broadwell are shown in Figure 2, and the results for KNL are shown in Figure 3.

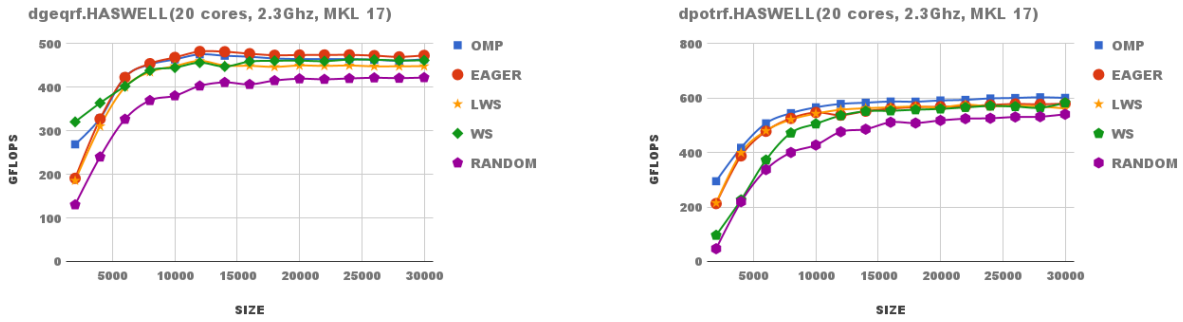

Figure 1: Efficiency of different scheduling strategies on Intel Haswell, a homogeneous multicore architecture with 20 cores, for the tile QR (left) and Cholesky (right) factorizations.

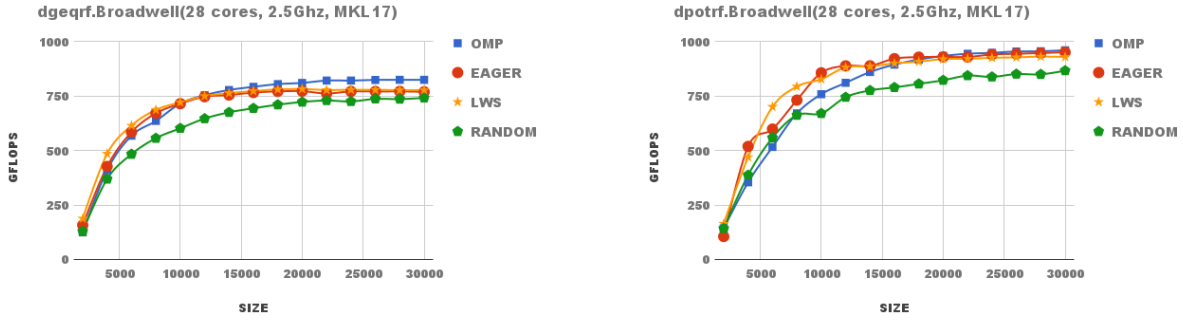

Figure 2: Efficiency of different scheduling strategies on Intel Broadwell, a homogeneous multicore architecture with 28 cores, for the tile QR (left) and Cholesky (right) factorizations.

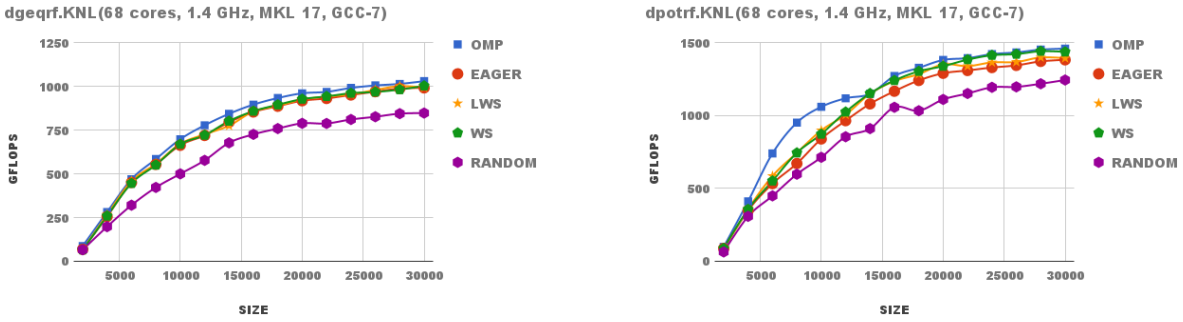

Figure 3: Efficiency of different scheduling strategies on Intel KNL, a homogeneous Xeon Phi processor with 68 cores, for the tile QR (left) and Cholesky (right) factorizations.

Compared to the results obtained using the OpenMP runtime system and scheduler (OMP), most of the scheduling strategies considered here (except for the random strategy)

are very competitive. The three schedulers that performed well are designed to enable dynamic load balancing. While the random scheduler initially assigns tasks to workers depending on their processing speed, it does not have a mechanism to correct unbalanced workloads. This suggests that, on homogeneous architectures, dynamic load balancing is a key feature to consider in the design of efficient schedulers. To some extent, this observation confirms our intuition. On homogeneous architectures, since all workers have the same processing speed, no performance profile is required to assign a task to workers. An alternative mechanism would be to reduce the execution time by distributing the workload equally.

Note that we omitted the results of our experiment for the working stealing (ws) strategy on the Broadwell architecture (Figure 2). The performance was so poor that we believe it might be a bug. Although we did not report the results here, we are actively investigating the issue.

5.3 Heterogeneous Multicore Architectures

Unlike for homogeneous systems, schedulers for heterogeneous architectures require an accurate performance profile of the underlying compute system assign tasks to workers based on a task's affinity. In this section we compare the results of static scheduling strategies and performance model–based dynamic scheduling strategies. We employed the MAGMA library to serve as a framework for static scheduling strategies and we used the DPLASMA library to assess performance with profile-based dynamic scheduling.

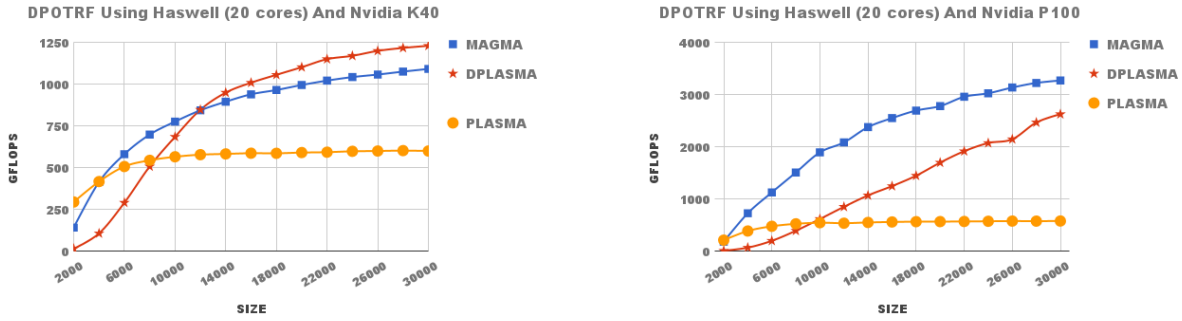

Figure 4: Comparison of the static and dynamic scheduling strategies on a heterogeneous architecture consisting of an Intel Haswell CPU (20 cores) and either an NVIDIA K40c GPU (left) or an NVIDIA P100 GPU (right). PLASMA uses a dynamic scheduling strategy but only exploits the CPU cores. DPLASMA implements a dynamic scheduling strategy for the CPU cores and the GPU. MAGMA uses a static scheduling strategy to exploit the CPU cores and the GPU. A Cholesky factorization was used to facilitate this comparison.

In the results reported in Figure 4, the heterogeneous system consists of two 10-core Intel Haswell CPUs (20 total cores) with either one NVIDIA K40c GPU (left) or one NVIDIA P100 GPU (right). For very small matrices, the PLASMA library, which is limited to CPU cores only, provides better performance. This is likely because PLASMA is not burdened with the data movement overhead associated with offloading tasks to the GPU, which occurs in MAGMA and DPLASMA. However, once the matrix sizes began to increase, PLASMA stagnated quickly while DPLASMA and MAGMA maintained their performance. Note that the static scheduling strategy implemented by MAGMA

focuses on maximizing the use of the GPU at the cost of starving some CPU cores, while the DPLASMA scheduler is designed to keep all of the workers busy. Consequently, on the K40c GPU, MAGMA is limited to the GPU performance (1.29 teraFLOP/s) and is outperformed by DPLASMA, which makes good use of the CPU cores *and* the GPU. However, owing to the huge performance gap between the two 10-core Haswell CPUs (0.7 teraFLOP/s) and the NVIDIA P100 GPU (4.67 teraFLOP/s), the static MAGMA strategy of maximizing the GPU exploitation proved to be more beneficial overall. In fact, because of the data movement cost and high ratio of the GPU performance over the CPU performance, it seems reasonable to starve the CPU cores to minimize the overall execution time.

The big lesson from this experiment is that, on heterogeneous systems, the scheduler should consider the heterogeneity level—in terms of performance—of the resources before deciding on which scheduling policy to use. In the case of two 10-core Haswell CPUs (0.7 teraFLOP/s) and an NVIDIA K40 GPU (1.29 teraFLOP/s), where the GPU is almost 2× faster than the CPU, a dynamic scheduling based on task and resource affinity may be an attractive option. In addition, the cost of offloading data lowers the real attainable performance of the GPU, which reduces the performance gap between a CPU + GPU solution and a CPU-only solution. On the other hand, since the NVIDIA P100 (4.67 teraFLOP/s) is almost 7× faster than two 10-core Haswell CPUs (0.7 teraFLOP/s), even when including the data movement cost, it would be beneficial to investigate a static scheduling strategy where almost all of the workload is moved to the GPU, with a few CPU-friendly tasks assigned to the CPU.

5.4 Heterogeneous Multi-GPU Architectures

In this section, we provide further results comparing static and dynamic scheduling strategies by extending our experiments to include multiple GPUs. We again used two 10-core Haswell CPUs (20 cores total); however, this time we add either two or three NVIDIA K40 GPUs to the mix. Using multiple GPUs does present new challenges. First, how do we maintain workload balancing between the GPUs? Second, how do we keep efficiently exploiting the CPU cores when the performance gap between the CPU and the GPU is low?

Once again, the results displayed in Figure 5 confirm that, when the GPU and CPU are close in terms of performance, the performance profile–based scheduling strategies (used in DPLASMA) have a distinct advantage over the static scheduling strategies (used in MAGMA). The performance achieved by MAGMA with both two and three K40 GPUs showed that it did not go beyond the peak performance of the GPUs, while DPLASMA exceeds the peak performance of the GPUs by efficiently exploiting the CPU cores using effective dynamic scheduling policies.

6 Conclusions

The significant increase in both the complexity and heterogeneity of modern and emerging HPC systems makes a strong case for novel static and dynamic scheduling strategies to efficiently match applications to HPC resources. The contribution of this work to the design of novel scheduling strategies is twofold. First, we evaluated many existing scheduling strategies in the context of extreme-scale dense linear algebra libraries. Second, we proposed guidance for designing novel scheduling strategies based on lessons learned

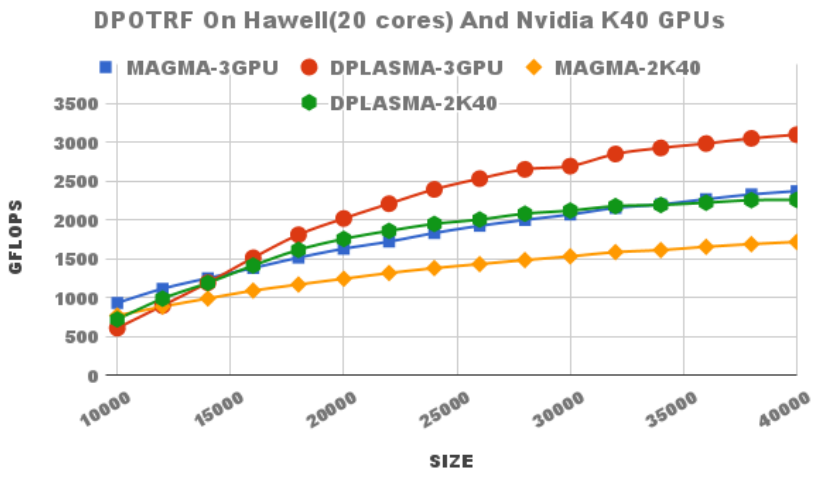

Figure 5: Comparison of static and dynamic scheduling strategies on a heterogeneous architecture consisting of two 10-core Intel Haswell CPUs (20 cores) and NVIDIA K40 GPUs for a Cholesky factorization. DPLASMA implements a dynamic scheduling strategy for the CPU cores and the GPUs. MAGMA, by contrast, uses a static scheduling strategy to exploit the CPU cores and the GPUs.

from a performance analysis of current state-of-the-art scheduling strategies on various HPC systems.

To assess the efficiency of scheduling strategies in homogeneous multicore and KNL architectures, we developed a new dense linear algebra library by designing PLASMA tile algorithms on top of the StarPU runtime system. From our experimental results on various systems, we noticed that dynamic load balancing is the key feature to consider in the design of scheduling strategies for homogeneous HPC systems.

For the evaluation of static and dynamic scheduling strategies on heterogeneous architectures, we used existing dense linear algebra solvers, namely the MAGMA and DPLASMA libraries. One notable observation from our assessment of scheduling strategies on heterogeneous architectures is that the efficiency of the scheduler is highly sensitive to the degree of heterogeneity of the resources. When there is a small difference in the processing speed of the resources, using a dynamic scheduling strategy based on task and worker affinity is an attractive option, as demonstrated by the performance of the DPLASMA library. However, on heterogeneous systems exhibiting a very high performance gap between the resources and a very high data movement cost, we found that maximizing the use of the faster resources—even at the risk of starving the slower processing units—is the key principle to consider in the design of an efficient scheduler.

We are currently working on new scheduling strategies in collaboration with experts in statistics and queuing theory. We believe that this collaboration on the rigorous theoretical analysis and design of new scheduling strategies will lead to the prototyping of efficient schedulers—initially in the StarPU runtime system framework and eventually for other task-based runtime systems.

References

[1] Emmanuel Agullo, George Bosilca, Berenger Bramas, Cedric Castagnede, Olivier Coulaud, Eric Darve, Jack Dongarra, Mathieu Faverge, Nathalie Furmento, Luc Giraud, et al. Matrices over runtime systems at exascale. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1330–1331. IEEE, 2012.

[2] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, and Hatem Ltaief. Plasma users guide. Technical report.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Euro-Par 2009 Parallel Processing*, pages 863–874, 2009.

[4] Olivier Beaumont, Terry Cojean, Lionel Eyraud-Dubois, Abdou Guermouche, and Suraj Kumar. Scheduling of linear algebra kernels on multiple heterogeneous resources. In *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, pages 321–330. IEEE, 2016.

[5] Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. Approximation proofs of a fast and efficient list scheduling algorithm for task-based runtime systems on multicores and gpus. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2017.

[6] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1995.

[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. 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. IEEE, 2011.

[8] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[9] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P. A. Wacrenier. *Resource Aggregation for Task-Based Cholesky Factorization on Top of Heterogeneous Machines*, pages 56–68. Springer International Publishing, Cham, 2017.

[10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[11] Hesham El-Rewini, Theodore G Lewis, and Hesham H Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., 1994.

[12] Yang Guo, Sasha Stolyar, and Anwar Walid. Shadow-routing based dynamic algorithms for virtual machine placement in a network cloud. *IEEE Transactions on Cloud Computing*, 2015.

[13] Azzam Haidar, Chongxiao Cao, Asim Yarkhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 491–500. IEEE, 2014.

[14] Azzam Haidar, Asim YarKhan, Chongxiao Cao, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Flexible linear algebra development and scheduling with cholesky factorization. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 861–864. IEEE, 2015.

[15] Van Jacobson, Robert Braden, and David Borman. Tcp extensions for high performance. 1992.

[16] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998.

[17] Suraj Kumar. *Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources*. PhD thesis, Université de Bordeaux, 2017.

[18] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1):259–271, 1990.

[19] Yi Lu, Qiaomin Xie, Gabriel Kliot, Alan Geller, Jim Larus, and Albert Greenberg. Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services. In *The 29th International Symposium on Computer Performance, Modeling, Measurements and Evaluation*. Elsevier, October 2011.

[20] Nick McKeown, Adisak Mekkittikul, Venkat Anantharam, and Jean Walrand. Achieving 100% throughput in an input-queued switch. *IEEE Transactions on Communications*, 47(8):1260–1267, 1999.

[21] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[23] Ramtin Pedarsani, Jean Walrand, and Yuan Zhong. Scheduling tasks with precedence constraints on multiple servers. In *Communication, Control, and Computing (Allerton), 2014 52nd Annual Allerton Conference on*, pages 1196–1203. IEEE, 2014.

[24] Alexander L Stolyar. Maximizing queueing network utility subject to stability: Greedy primal-dual algorithm. *Queueing Systems*, 50(4):401–457, 2005.

[25] Alexander L Stolyar and Tolga Tezcan. Control of systems with flexible multi-server pools: a shadow routing approach. *Queueing Systems*, 66(1):1–51, 2010.

[26] Leandros Tassiulas and Anthony Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE transactions on automatic control*, 37(12):1936–1948, 1992.

[27] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 36(5):232–240, 2010.

[28] D Walker. Lapack working note 95 scalapack: A portable linear algebra library for distributed memory computers-design issues and performance.

[29] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 156–165. IEEE, 2015.

[30] Asim Yarkhan, Jakub Kurzak, and Jack Dongarra. Quark users' guide.

[31] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the PLASMA numerical library to the OpenMP standard. *International Journal of Parallel Programming*, 45(3):1–22, 2016.