# Design and Implementation of a Large Scale Tree-Based QR Decomposition Using a 3D Virtual Systolic Array and a Lightweight Runtime

Ichitaro Yamazaki*, Jakub Kurzak*, Piotr Luszczek*, Jack Dongarra*†‡
*University of Tennessee, Knoxville, TN 37996, USA
†Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
‡University of Manchester, Manchester, M13 9PL, UK

{iyamazak, kurzak, luszczek, dongarra}@eecs.utk.edu

***Abstract*—A systolic array provides an alternative computing paradigm to the von Neuman architecture. Though its hardware implementation has failed as a paradigm to design integrated circuits in the past, we are now discovering that the systolic array as a software virtualization layer can lead to an extremely scalable execution paradigm. To demonstrate this scalability, in this paper, we design and implement a 3D virtual systolic array to compute a tile QR decomposition of a tall-and-skinny dense matrix. Our implementation is based on a state-of-the-art algorithm that factorizes a panel based on a tree-reduction. Using a runtime developed as a part of the Parallel Ultra Light Systolic Array Runtime (PULSAR) project, we demonstrate on a Cray-XT5 machine how our virtual systolic array can be mapped to a large-scale machine and obtain excellent parallel performance. This is an important contribution since such a QR decomposition is used, for example, to compute a least squares solution of an overdetermined system, which arises in many scientific and engineering problems.**

***Keywords*-systolic array; QR decomposition; multithreading, message-passing, dataflow; runtime;**

## I. Introduction

Current trends in microprocessor technology suggest that emerging large-scale computers should have rapidly increasing floating point capacities due to the growing numbers of cores and of floating point units. On the other hand, their memory capacities are expected to grow at a much slower pace, eventually falling behind their arithmetic performance by an order of magnitude [1], [2], [3]. On such computers, it is unlikely that software without a strong scaling property can utilize the full computing capacity of the compute nodes, whose aggregated memory capacity can accomodate the memory required to run the software for solving a given problem.

The vast majority of modern computers are based on the von Neumann architecture where the execution of software is instruction-stream-driven by instruction counters. The systolic array provides an alternative computing paradigm that is composed of matrix-like rows of processing units, each of which is connected to a small number of nearest neighbors in a mesh-like topology. Stated in another way, a *systolic array* is a network of processors that compute and pass data through the system. On such a systolic array, the execution of an operation is message-driven or data-stream-driven by data counters and triggered by the arrival of a data object. Though the hardware implementations of such systolic arrays had been haunted by an array of problems, the systolic array becames very attractive as a parallel programming model on modern computers when it is implemented as a software layer like the Virtual Systolic Array (VSA) presented in [4]. Since this discovery, the concepts of 1D, 2D, and 3D systolic arrays as virtualized software designs have been combined with a distributed-memory dataflow runtime and delivered a wide range of scalability results, but with varying levels of achievable performance [5].

Our previous work [4] presented a 2D systolic array for a tile dense QR decomposition algorithm with its panel factorization based on a flat-tree reduction. Its strong scaling performance for square matrices was superior to that of the state-of-the-art software, and could potentially tackle the challenges of utilizing the increasing levels of concurrency on the emerging computers. To follow up this previous work, in this paper, we free ourselves from the constraint of a planar layout and present a 3D systolic array for a hierarchical-tree variant of QR decomposition [6], [7]. Albeit increasing the computational cost, this 3D systolic array allows us to exploit an increased level of parallelism that is difficult to exploit using the previous 2D systolic array, especially for a tall-and-skinny matrix. This is an important algorithm since such decomposition is used, for example, to solve an overdetermined least-squares problem that arises in many scientific and engineering problems.

For the hierarchical-tree based QR decomposition, the optimal match between the chosen reduction-tree and the underlying software and hardware layers is, for the most part, system-dependent. Such an optimal match could be found through experimentation, and thus may be guaranteed to scale even beyond the current breed of tightly coupled multicore supercomputers. In this paper, however, instead of looking for such an optimal match, we focus on a more generic tree (i.e., binary-tree on top of flat-trees) as our reduction tree. We then present its performance on a tightly-coupled system to demonstrate that this 3D systolic array to perform this particular tree-reduction can exploit the limited amount of parallelism during the tall-and-skinny QR decomposition, and obtain a superior scaling compared with the previous 2D systolic array.

Sections II and III motivate our current work and survey the related works, respectively. Sections IV and V present Parallel Ultra Light Systolic Array Runtime (PULSAR) and describe our 3D VSA for the QR decomposition, respectively. Section VI presents performance results to demonstrate the scalability of our implementation, and Section VII provides final remarks.

## II. Motivation for a Mixed Paradigm Implementation in a Weak Scaling Setting

This work is primarily motivated by the success of our previous software-defined systolic arrays [4]. As indicated by the reviewers, the scope of that research was well-defined and the experimental results were far beyond the accepted state-of-the-art. However, the often raised issue was the limited applicability along multiple conceptual axises: data distribution, scaling scenarios, virtual systolic architecture, memory management, and input problem types. The primary goal of this article is to address these concerns, and also to further discover the potential of this methodology to obtain an extreme scalability.

Our first goal – *making the software-defined systolic arrays more general* – has multiple aspects. In particular, it includes the development of an independent runtime that is fully decoupled from the user code and can only be accessed through established programming interfaces. By doing so, we allow for the reuse of the PULSAR runtime across multiple application domains and also introduce the possibility of replacing the runtime with another runtime implemented according to a different design principle. While general solutions tend to be flexible, they may also incur additional overheads. Hence, one of our objectives is to examine if broadening the scope of PULSAR retains the original scalability and performance metrics.

Another addition made in this article is the *weak scaling implementation and evaluation*. This partially stems from making the runtime more general, but is also naturally driven by the increases in the data sizes from the science domains that require models using least-squares optimization. Another aspect tested by the weak-scaling experiments is testing for arbitrary sizes of many parameters that describe the virtual systolic system, such as message counts and queues, dimension of the virtual array, communication buffer sizes, and so on. In fact, we discovered that in a strong scaling study, it is possible to exhaust the available local memory, which then precludes runs with data sets exceeding the offending problem size. Simply put, weak scaling allows the user to partition the data as well as the computation, which enables larger mathematical models to be evaluated.

We also enable a virtual systolic array of a true *3-dimensional structure*, which maps directly to the 3-nested loops of the QR algorithm. Such direct mapping lessens the burden of restructuring the code and allows an easy conceptualization of the virtual array structure without the need for any translation process. Such a process would be error-prone if performed manually, or require a substantial software infrastructure if done automatically. Furthermore, an added benefit of the proper handling of 3D structures is the flexibility of mapping of both data and computation onto the physical hardware, which often resembles 3D graphs.

The practical data sets, driven, by applications, are mostly based on *non-square matrix shapes*, which may provide a limited amount of parallelism. Hence, to exploit appropriate levels of parallelism, we require a tree-based panel reduction such that a large number of physical cores can be occupied with computations as soon as the data dependencies of the computations are resolved. Moreover, the runtime may need to adjust its workings to match the workload, especially when an algorithm has the distinct phases in the computation. For instance, the panel factorization phase of the QR decomposition is latency-sensitive, uses small messages, and does not contribute much to the computational load, hence precluding the use of the common techniques to overlap communication with computation. On the other hand, its update phase is computationally intensive and makes the right target for the overlapping and latency hiding techniques. The scalability concerns dictate that these two phases should not be run in sequence but rather intertwined together. This leads to the *mixed-paradigm programming* with a mix of latency-bound and throughput-oriented workloads. For a PULSAR user, these distinct phases can be separated by the use of distinct Virtual Data Processors (see Section IV), and the runtime mixes them appropriately for good performance while preserving the correctness of execution.

Finally, instead of enumerating and subsequently testing all possible tree variants [6], [7], our focus in this paper is to evaluate the virtual systolic runtime and its ability to handle a *mixed-paradigm* workload that may not map optimally onto the conceptual systolic array design. Moreover, the mixed-workload, combined with a tile algorithm, features the classical dataflow execution with the computational load of a single operation raised to offset the software overheads incurred by the data dependence tracking and data synchronization. Another methodological item in the mix is the heavy reliance on the light-weight reduction operations that do not enjoy the luxury of increased computational workload. This causes the natural flow of the algorithm to be directly exposed to the software overheads of the runtime, the system communication layer (an MPI implementation), and, finally, the hardware latency, each of which cannot be easily hidden by the pipeline mechanism inherent in the systolic design. A study of this mixing and the influence of these inefficiencies is one of our interests in this work.

## III. Related Work

The key publication by Kung and Leiserson [8] introduced systolic arrays, focusing on their *simple regular geometries and data paths* and introducing *pipelining as a general method for using these structures*. Provenance of systolic arrays could be found in array-like hardware structures that include iterative and processor arrays, and cellular automata.

Our previous work focused on lifting the conceptual design from the hardware level to the software level [4]. This form of virtualization offered a number of possibilities that could not have been matched easily, if at all, by hardware-only solutions. Our practical focus was on an extreme form of strong scaling of a square-matrix QR decomposition, where only a small amount of workload was assigned to a single virtual data processor. These virtual processors were then mapped to thousands of tightly coupled multicore CPUs. We provided the first implementation of a Virtual Systolic Array and validated the potential of the methodology for a large-scale dense linear algebra application.

### A. Runtime Support for Scheduling

The task-superscalar runtime environment, *QUeuing and Runtime for Kernels* (QUARK), provides a simple serial library for a superscalar execution on multicore processors [9]. QUARK provides the dynamic runtime layer for the PLASMA numerical library [10], [11] and has undergone substantial stress testing. QUARK has a number of features critical to the operation of a numerical software suite, such as error handling extensions and task cancellation capabilities. Recently, it was extended for use with GPU accelerators [12] and prototyped for a distributed memory implementation [13].

The OMPSs task-superscalar system developed at the Barcelona Supercomputer Center has variants that can target grids [14], the Cell B. E. processor [15], multicore processors [16], [17], and GPUs [18]. The best known variant is SMPSs, the multicore implementation, which is a compiler-based system that uses `#pragma` directives to annotate tasks that can be run in parallel and to decorate the data parameters with read/write usage information. The main thrust in OMPSs is to become part of the OpenMP standard.

The StarPU task-superscalar system was developed at INRIA Bordeaux [19] to explore task scheduling in a hybrid CPU/GPU environment. StarPU uses a history based adaptive scheduling technique to assign tasks to devices (it treats a GPU on par with an individual CPU core, potentially leading to a non-optimal scheduling).

Several other approaches exist that take a serial code and execute it on a parallel environment, such as Jade [20], [21] Cilk [22], OpenMP [23], Sequoia [24], SuperMatrix [25], Habanero [26], Hierarchical Place Trees [27], or the D-TEiP task library [28].

## IV. PULSAR Runtime

### A. Programming Model

PULSAR offers a straightforward programming model with a few simple abstractions to implement a systolic array as a software layer. The main abstraction in PULSAR is the *Virtual Systolic Array* (VSA), which is a set of *Virtual Data Processors* (VDPs) connected by *channels*. The VDP is a direct descendant of the *Processing Element* (PE) in the traditional systolic array nomenclature. Each VDP is uniquely identified by a *tuple* (a string of integers) and has a counter defining its life span. The VDP contains executable code, read-only global parameters, read/write persistent *local* variables, and a set of input and output channels (see Figure 1 for a schematic diagram). They communicate with each other by sending data *packets* through a channel.

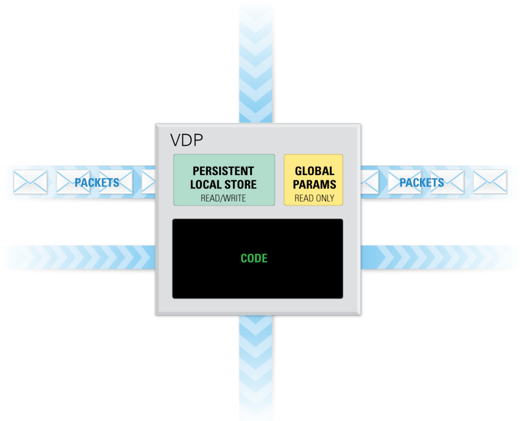


Figure 1. Schematic diagram of the Virtual Data Processor.

A channel is a static unidirectional connection between two VDPs. In principle, it is a *First In First Out* (FIFO) queue of packets, where the source VDP pushes packets to its output channel and the destination VDP pops packets from its input channel. On the corresponding VDP, each input or output channel has an assigned *slot* identified by a consecutive number starting at zero. When all input channels of a VDP contain a data packet, the VDP can be *fired* and can be scheduled for execution by the runtime. When fired, the VDP can pop packets from its input channels, invoke any computational kernels, and push packets to its output channels. At each firing, the VDP's counter is decremented, and when it reaches zero, the VDP is destroyed.

At anytime during its execution, the VDP can pop from or push to its input or output channels, respectively. Hence, there are two basic modes of operations to any data stream, either *read-process-write/read-modify-write* or *read-write-process* (*by-pass/pass-through*). PULSAR also provides options to disable a channel at its creation and to enable, disable, or destroy the channel during the execution of the VSA. These options allows us to dynamically reconfigure the network of VDPs, and will be used to implement the QR decomposition in Section V-C. The VDP can also send packets of different sizes, or create new packets, rather than popping them from an input channel. For example, we create packets to send the matrix transformations generated during the QR decomposition.

```
new vsa
for each vdp do
  new vdp
  for each channel do
    new channel
    vdp→insert(channel)
  end for
  vsa→insert(vdp)
end for
vsa→run()
delete vsa
```

Figure 2. VSA construction.

To implement an algorithm using PULSAR, we build a VSA by defining all the VDPs and their connections (see Figure 2 for an illustration). Each VDP has its own processing cycle, and can execute a sophisticated executable code that may invoke any computational kernel, access the read-only global parameters, read or write to locally persistent data storage, and pop or push packets from input channels or to output channels (see Figure 3 for an example).

```
for all inputs do
  read a packet
  [forward the packet]
end for
process packets
[create new packets]
for all outputs do
  write a packet
end for
```

Figure 3. VDP cycle.

Once the VSA is launched, the control is passed to the runtime, and the runtime propagates the data through the VSA and dynamically schedules VDPs for execution. To effectively utilize the physical core, we assign multiple VDPs to each thread. Then, each thread continuously sweeps its list of the VDPs in search of a ready VDP, i.e., the runtime can launch the VDPs when at least one packet is in each of its active input channels. By having multiple VDPs on each thread, the runtime can avoid the idling time of the core as long as at least one of the VDPs is ready. Hence, the last piece of information required from the user is the function that maps the VDPs to threads based on their tuples. This, in principle, is a many-to-one mapping that maps each VDP to a specific thread, but may map multiple VDPs to the same thread. Two scheduling schemes, *lazy* and *aggressive*, are available within the thread. The lazy scheme fires a ready VDP once and moves on to another VDP, while the aggressive scheme repeatedly fires the VDP as long as it is ready. In the next section, we discuss the implementation of the PULSAR runtime that decides which VDP should be fired for the execution according to its data availability.

### B. Runtime System

The *PULSAR Runtime* (PRT) is a lightweight layer that maps the abstractions described in the previous section to a distributed memory system with a multicore node architecture. The *Message Passing Interface* (MPI) is the mechanism used for the inter-node communication, while *POSIX Threads* (Pthreads) is used for the intra-node multithreading. Figure 4 shows a schematic diagram of the PRT structure. The VSA is executed by a collection of MPI processes, each of which lauches a number of *worker* threads and a *proxy* thread dedicated to handling inter-node communication. In our experiment, we ran PRT with one MPI process on each distributed memory compute node, and let the MPI process launch one thread on each of its physical cores, dedicating one of these threads as the proxy. However, other mappings are possible, such as having one MPI process on each socket of a node or launching multiple threads on each core (i.e., oversubscribing).

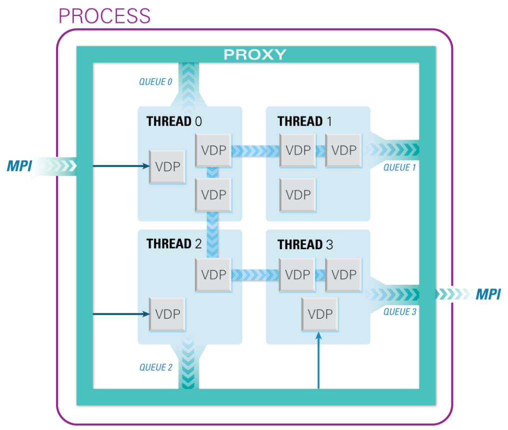


Figure 4. Schematic diagram of the PULSAR runtime.

An intra-node channel is a local channel within the same shared-memory node. It is a simple FIFO queue, connected to a source VDP on one end and to a destination VDP on the other (the mutual exclusion is enforced by a mutex). On the other hand, a non-local inter-node channel is connected to a local VDP on one end, while the other end is managed by the proxy. The proxy uses a single queue to manage all the incoming packets from the other nodes, while it uses one queue per worker thread for outgoing packets (see Figure 4 for the schematic diagram of the queues). Upon reception of an incoming packet from another node, the appropriate local channel is located and the packet is moved from the incoming queue to that channel. Sending an outgoing packet is handled by moving the packet from the output channel of the thread to the corresponding outgoing queue of the proxy, and posting an asynchronous send. At the completion of the send, the corresponding packet is removed from the outgoing queue.

The proxy follows the cycle of serving communication requests until all the communication queues are empty and all the VDPs are destroyed. Because of the simplicity of the underlying abstractions, the proxy only uses six MPI functions (i.e., `MPI_Isend`, `_Irecv`, `_Test`, `_Get_count`, `_Barrier`, and `_Cancel`) and spends most of its time cycling through the three functions, `MPI_Isend`, `_Irecv`, and `_Test`. Moreover, since the proxy is implemented as a separate process, it does not require the thread safety of the underlying MPI implementation.

In principle, the proxy implements the abstraction of VDP-to-VDP communication. The routing of packets to appropriate channels is accomplished by assigning consecutive numbers to all the channels connecting two VDPs. These numbers are placed in the MPI tag and combined with the sender rank to identify the destination channel on the receiving side. Since the channel numbering is applied independently for each pair of the nodes, the minimum guaranteed range of MPI tag values of 16K should be more than enough for the foreseeable future.

Overall, the PULSAR runtime provides the benefits of

- data-driven runtime scheduling,
- overlapping of communication and computation, based only on non-blocking messaging,
- zero-copy shared-memory communication by relying solely on aliasing of local data, and
- hiding the combined complexity of multithreading and message-passing.

At the same time, the runtime introduces minimal scheduling overheads, and assures portability by relying only on the small rudimentary set of MPI and Pthreads functions.

## V. Hierarchical QR using a 3D Systolic Array

### A. Algorithm

The Householder QR factorization [29] decomposes an $m$-by-$n$ matrix $A$ into a product of an orthogonal matrix $Q$ and an upper-triangular matrix $R$; i.e., $A = QR$. To compute such a factorization, the first step of a *column-based* algorithm zeroes out all the entries of the first column below the diagonal by applying a Householder transformation to the matrix $A$ from the left (i.e., $A := (I - \tau \mathbf{v}\mathbf{v}^T)A$, where $\tau$ is a scalar and $\mathbf{v}$ is a vector). This process is applied recursively to the trailing submatrix to transform the whole matrix $A$ into an upper-triangular form. To exploit the memory hierarchy or to reduce the communication latency on a respective shared or distributed memory computer, a *block* version of the algorithm divides the matrix $A$ into block columns. Then, the algorithm first transforms the first block column, *panel*, into an upper-triangular form and then applies the accumulated transformations to the trailing submatrix at once (i.e., $A := (I - VTV^T)A$, where $V$ is an $m$-by-$n_b$ matrix, $T$ is an $n_b$-by-$n_b$ upper-triangular matrix, and $n_b$ is the block size). LAPACK [30] and ScaLAPACK [31] implement this block algorithm for the shared and distributed memory computers, respectively.

The block algorithm exploits a high-level of parallelism during the trailing submatrix update, which dominates the computational cost of the factorization, delivering a good asymptotic scaling. However, the panel factorization accesses the matrix column-by-column. Hence, it is latency-bounded with limited parallelism, limiting the strong scaling of the factorization, especially for a tall-and-skinny matrix. To improve the data access and parallel performance of the block algorithm, a *tile* algorithm divides $A$ into square blocks, called *tiles*. The panel factorization then becomes a tree-reduction of the tiles in the panel (see Section V-B for a pseudocode). Compared to the block algorithm, this tile algorithm is not only cache-friendly since each tile is stored contiguously in memory, but it also exploits more fine-grained parallelism because the elimination of each tile can be immediately followed by the application of the corresponding updates of the tiles to the right. PLASMA [10] and DPLASMA [32] implements this on shared and distributed memory computers using QUARK [9] and PaRSEC [33] runtimes, respectively. Our first VSA implementation of the QR decomposition [4] is also based on a tile algorithm, obtaining a good strong-scaling for factorizing a square matrix on a distributed memory computer.

A *hierarchical* QR algorithm [6] aims to improve the performance of the tile algorithm, using a hierarchical tree for the panel factorization and further exploiting the parallelism and/or the hardware topology. The algorithm groups a set of tiles in the panel into a *domain* that can be factorized independently by a separate tree reduction, eliminating all the tiles except the top tile of each domain. After this first phase of the panel factorization, the top tiles of the domains are gathered together for the second phase of the panel factorization based on another tree reduction. As discussed in Section I, the optimal match between the reduction-tree and the underlying software and hardware layers is, for the most part, system-dependent. Instead of looking for such an optimal reduction-tree for a specific computer, in Sections V-B and V-C, we describe a more generic reduction tree and its 3D VSA implementation, respectively.

### B. Reduction Tree

For our first VSA implementation of the tile QR factorization [4], a flat-tree is used to factorize the entire panel, where the top tile of the panel is used to eliminate the remaining tiles of the panel in sequence. In this way, each off-diagonal tile is brought into the local memory only

```
for $j = 1, 2, \ldots, n_t$ do
  flat-reduction of domains
  for $i = j, j+h, \ldots, m_t$ do
    dgeqrt($A(i,j)$)
    for $\ell = j+1, j+2, \ldots, n_t$
      dormqr($A(i,j)$, $A(i,\ell)$)
    for $k = i+1, i+2, i+h-1$
      dtsqrt($A(i,j)$, $A(k,j)$)
      for $\ell = j+1, j+2, \ldots, n_t$
        dtsmqr($A(i,j)$, $A(k,j)$, $A(i,\ell)$, $A(k,\ell)$)
  end for
  binary-reduction of top tiles
  for each level of binary-tree
    for each pair $(i,k)$ of tree
      dttqrf($A(i,j)$, $A(k,j)$)
      for $\ell = j+1, j+2, \ldots, n_t$
        dttmqr($A(i,j)$, $A(k,j)$, $A(i,\ell)$, $A(k,\ell)$)
end for
```

Figure 5. Pseudocode of hierarchical QR (binary-tree on flat-trees).

once, while the top diagonal tile stays in the local memory throughout the panel factorization. Hence, the flat-tree can exploit the data locality and is especially suited for a shared-memory or tightly-coupled computer. Furthermore, it relies on standard LAPACK routines, for which optimized kernels may be available on a specific target computer. However, the tiles are eliminated in sequence, and the flat-tree can only exploit a limited amount of parallelism during the panel factorization. This limits the parallel performance of the QR decomposition, especially for a tall-skinny matrix.

To exploit more parallelism, our hierarchical QR implementation uses a binary-tree on top of flat-trees; namely, we perform the flat-tree reduction of the tiles in a domain, followed by the binary-tree reduction of the top tiles of the domains. In comparison to the flat-tree, this hierarchical tree maintains the data locality within each domain, while exploiting more parallelism, since the flat-tree reductions of the domains can be executed in parallel. Figure 5 shows a pseudocode of our implementation, where $A(i,j)$ is the $(i,j)$-th tile of $A$, $n_t$ and $m_t$ are the respective numbers of tiles in the column and row of $A$, $h$ is the number of tiles in each domain, and the required computational kernels are

- **dgeqrt(**$A(i,j)$**)** Performs QR factorization of $A(i,j)$. Places its $R$-factor and Householder reflectors in the upper and lower triangle parts of $A(i,j)$, respectively.
- **dormqr(**$A(i,j), A(i,\ell)$**)** Applies Householder reflectors computed by **dgeqrt(**$A(i,j)$**)** to $A(i,\ell)$ of the trailing submatrix.
- **dt[s|t]qrt(**$A(i,j), A(k,j)$**)** Performs incremental QR factorization of an off-diagonal tile by computing the QR factorization of two stacked tiles $A(i,j)$ and $A(k,j)$; where $A(i,j)$ is an already-factorized upper-triangular tile, and $A(k,j)$ is also in an upper-triangular form in **dttqrt**. Updates the $R$-factor in $A(i,j)$ and places Householder reflector coefficients in $A(k,j)$.
- **dt[s|t]mqr(**$A(i,j), A(k,j), A(i,\ell), A(k,\ell)$**)** Applies the Householder transformations computed by **dt[s|t]qrt(**$A(i,j)$**,** $A(k,j)$**)** to the two tiles $A(i,\ell)$ and $A(k,\ell)$ of the trailing submatrix.

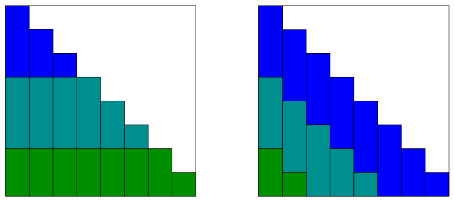
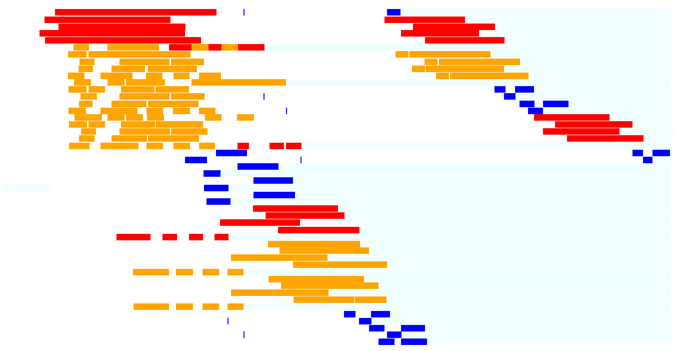

(a) Fixed domain boundary. (b) Shifted domain boundary.

Figure 6. Two strategies to break panels into domains


(a) Fixed domain boundary.

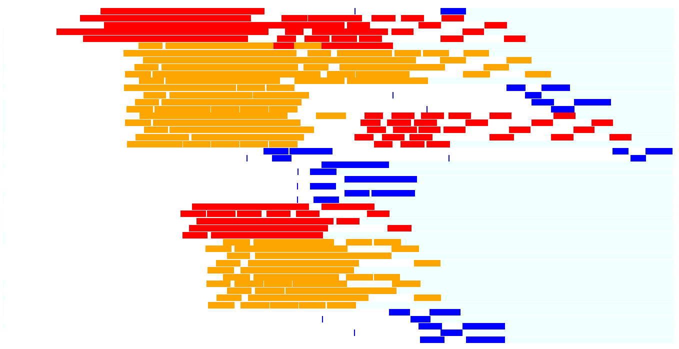

(b) Shifted domain boundary.

Figure 7. Execution traces using two strategies to break panels into domains (see Figure 6). The red, orange, and blue traces represent the flat-tree reduction of panels, the corresponding trailing submatrix updates in flat-trees, and binary-tree reductions, respectively. With the fixed domain boundary, during the second flat-tree reduction, only the first domain can overlap with the binary-tree reduction. On the other hand, with the shifted boundary, we see greater overlap of the flat-tree reductions.

While the first two kernels **dgeqrt** and **dormqr** are implemented in LAPACK, the rest of the kernels are specifically introduced for a tree-based QR factorization.

In our hierarchical QR implementation, all the domains own the same number of tiles, $h$, except the last domain which holds the remaining tiles. This essentially shifts the domain boundary by one tile for each panel factorization, and allows a better pipelining of the tasks in comparison to fixing the domain boundary throughout the entire QR factorization (see Figure 6 for illustration). Specifically, when the

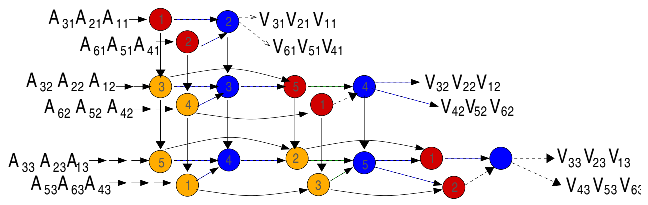


Figure 8. 3D Virtual Systolic Array for a hierarchical QR decomposition, where the circles and lines represent the VDPs and channels, respectively. The red and orange VDPs perform the flat-tree reduction, while the blue VDPs perform the binary-reduction. The vertical channels by-pass the transformation, while the dashed and solid horizontal channels pass the top and remaining tiles, respectively. The numbers within the circles denote the IDs of the threads assigned to the VDPs, where there are five available threads in this example.

domain boundary is fixed, for all the flat-tree reductions, the tiles in the same block row are used to eliminate the remaining tiles in the domain, except for the top domain. Since the top-tile goes through the binary reduction, the next flat-tree reduction cannot start until the corresponding level of the binary-tree reduction is completed (see Figure 7(a) for an execution trace). On the other hand, when the domain boundary is shifted, the current top-tile becomes the last-tile of a domain at the next flat-tree reduction. Hence, the next flat-tree reduction can start as soon as the corresponding tile is updated by the current flat-tree reduction, hence allowing a greater overlap between the tree-reductions (see Figure 7(b) for an execution trace).

## C. 3D Virtual Systolic Array

Figure 8 shows our 3D systolic array factorizing $A$ of 6-by-3 tiles with $h = 3$. At the top level of the VDPs (or right most in the figure), the red and orange VDPs are responsible for the flat-tree reductions of the first panel, where the red VDPs independently perform the flat-tree reductions of the domains, and the orange VDP applies the Householder transformation, generated by the corresponding red VDP, to the tiles in the same block rows. To exchange these Householder transformations, the vertical channel connects the VDPs that belong to the same domain. Although these Householder transformations are broadcasted through a chain, each VDP passes the transformation to the next VDP as soon as it receives the transformation and before it applies the transformation to the local tile. Because of this runtime feature to *by-pass* the packets, PRT can overlap the communication of the transformation between the VDPs with the application of the transformation on the VDPs.

At the next level of VDPs, the blue VDP performs one step of the binary-tree reduction. For this, the top tile of a domain is passed through the horizontal dashed channel from the flat-tree to the binary-tree, while the rest of the tiles are passed through the horizontal solid channels from the current flat-tree to the next flat-tree. Since the domain boundaries are shifted, the next flat-tree reduction can start as soon as the VDP receives a tile from the current flat-tree reduction; while in parallel, the blue VDPs perform the binary-reduction of the top tiles.

As described in Section IV, PRT schedules a task only when all the input channels contain packets. Hence, in order to overlap the flat- and binary-reductions, the dashed channel from the binary-tree to the flat-tree is initially deactivated. Only when the flat-tree finishes processing all the tiles except the last tile, will this channel from the binary-tree reduction be activated, and the VDP waits for the arrival of the last packet through the channel. Just like in the flat-tree reduction, at each level of the binary-reduction, each VDP is connected through the vertical channel that passes the Householder transformation. Again, these transformations are pushed to an output channel right away such that the communication and computation can overlap. Finally, after each binary-reduction of two top tiles, the second tile is passed right to the flat-tree such that the VDP can perform the last step of the flat-tree reduction.

To demonstrate how to implement VSA using PULSAR, Figure 9 shows our implementation of QR decomposition based on a flat-tree (domino QR), where the following PULSAR interfaces are used:

- **prt_vdp_new(tup, cnt, fnc, size_loc, nin, nout)** Create a VDP with the arguments specifying its tuple and initial counter, the name of subroutine to be executed, the size of the local store for the persistent storage, and the numbers of input and output channels.
- **prt_channel_new(size_pkt, tupin, chnin, tupout, chnout)** Create an output/input channel between two VDPs with the arguments specifying the maximum size of a packet to be sent/received, the tuple of the sender/receiver, the slot number of the output/input channel, the tuple of the receiver/sender, and the input/output channel slot at the receiver/sender.
- **prt_vdp_channel_insert(vdp, chn, in|out, slot)** Insert an input/output channel to a VDP.
- **prt_vsa_vdp_insert(vsa, vdp)** Insert a VDP to a VSA.

In Figure 9, the subroutine `vdp_factor` for calling `prt_vdp_new` factorizes the panel by first calling **dgeqrt** and then **dtsqrt**, while the subroutine `vdp_update` updates the trailing submatrix by first calling **dormqr** and then **dtsmqr**. Moreover, the structure `qr_local_t` stores the local variables (e.g., two tiles used for the reduction, and the required transformation) on each VDP, and `prt_tuple_new2`$(i, j)$ creates a tuple of an integer-pair $(i, j)$.

## D. Mapping VDPs to Threads

One of the benefits of using PULSAR is its flexibility to map the VDPs to the threads. This flexiblity allows, for instance, us to exploit the parallelism and/or to improve the data locality during the execution of a VSA. For our hierarchical QR implementation, we cyclically assign the available threads to the VDPs. Figure 8 illustrates this

```
for $i = 1, 2, \ldots, m_t$ do
  for $j = i, i+1, \ldots, n_t$ do
    // Create VDP.
    if $i == j$ then
      vdp = prt_vdp_new(prt_tuple_new2($i, j$), $n_t - i + 1$,
                       vdp_factor, sizeof(qr_local_t), 3, 3);
    else
      vdp = prt_vdp_new( prt_tuple_new2($i, j$), $n_t - i + 1$,
                       vdp_update, sizeof(qr_local_t), 3, 3);
    end if
    // input channel 1 (receive $A$)
    channel = prt_channel_new($n_b * n_b$*sizeof(double),
      prt_tuple_new2($i - 1, j$), 1, prt_tuple_new2($i, j$), 1);
    prt_vdp_channel_insert(vdp, channel, PrtInputChannel, 1);
    if $j > i$ then
      // input channel 2 (receive $V$)
      channel = prt_channel_new($n_b * n_b$*sizeof(double),
                       prt_tuple_new2($i, j - 1$), 2,
                       prt_tuple_new2($i, j$), 2);
      prt_vdp_channel_insert(vdp, channel, PrtInputChannel, 2);
      // input channel 2 (receive $T$)
      channel = prt_channel_new($i_b * n_b$*sizeof(double),
                       prt_tuple_new2($i, j - 1$), 3
                       prt_tuple_new2($i, j$), 3);
      prt_vdp_channel_insert(vdp, channel, PrtInputChannel, 3);
    end if
    if $i < m_t$ then
      // output channel 1 (send $A$)
      channel = prt_channel_new($n_b * n_b$*sizeof(double),
                       prt_tuple_new2($i, j$), 1,
                       prt_tuple_new2($i + 1, j$), 1);
      prt_vdp_channel_insert(vdp, channel, PrtOutputChannel, 1);
    end if
    if $j < n_t$ then
      // output channel 2 (send $V$)
      channel = prt_channel_new($n_b * n_b$*sizeof(double),
                       prt_tuple_new2($i, j$), 2,
                       prt_tuple_new2($i, j + 1$), 2);
      prt_vdp_channel_insert(vdp, channel, PrtOutputChannel, 2);
      // output channel 3 (send $T$)
      channel = prt_channel_new($i_b * n_b$*sizeof(double),
                       prt_tuple_new2($i, j$), 3,
                       prt_tuple_new2($i, j + 1$), 3);
      prt_vdp_channel_insert(vdp, channel, PrtOutputChannel, 3);
    end if
    prt_vsa_vdp_insert(vsa, vdp); // Insert the VDP.
  end for
end for
```

Figure 9. PULSAR Example Code to set up VSA of domino QR.

mapping where the five threads are cyclically assigned to the VDPs. In this illustration, we only have one level of the binary reduction. When we have multiple levels of the binary reduction, a parent VDP is assigned to the same thread as its first child. Hence, the number of threads assigned to a tree is at most the number of leaves. Since only one packet passes through each VDP, the child and parent VDPs cannot be executed in parallel, while this mapping exploits the data locality between the VDPs.

As discussed in Section IV-A, there are two scheduling schemes, *lazy* and *aggressive*, that decide which one of the ready VDPs is executed next within a thread. For our tree-based QR, the *lazy* scheduling scheme often obtained better core utilization than the *aggressive* scheme did. This is because the *lazy* scheme encourages the overlapping of the panel factorization with the trailing submatrix update through a mechanism commonly-known as *lookahead*. Hence, though our implementation does not explicitly take the mixed-paradigm into consideration, the execution of the program automatically takes advantage of the mixed-paradigm without much effort from the user. This is because we use different VDPs for the panel factorization and for the submatrix updates, and PRT dynamically schedules the VDPs based on the data availability.

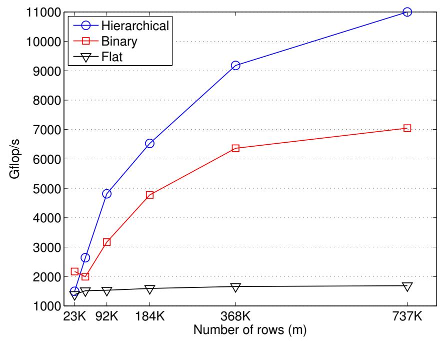


Figure 10. Asymptotic tree-based QR scaling ($n = 4,608$, 9K cores).

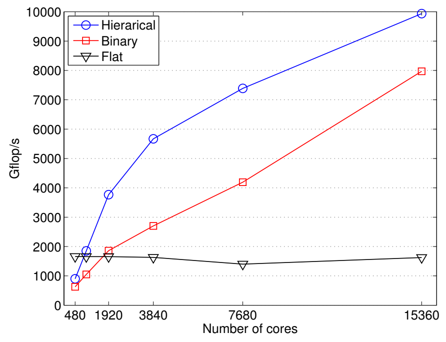


Figure 11. Strong scaling of tree-based QR at $(m, n) = (368640, 4608)$.

## VI. Performance Results

We conducted our experiments on the Cray XT5 system (Kraken) at the National Institute for Computational Sciences, operated by the University of Tennessee, Knoxville and hosted by the Oak Ridge National Laboratory. Each node of Kraken has two 2.6GHz six-core AMD Opteron processors and 16GB of main memory, and is connected by a Cray SeaStar2+ router. The code was compiled using the GNU 4.6.2 **gcc** compiler with **-O3** optimization flag, and linked to MKL 2011_sp1.8.273 and Cray MPICH2 5.3.5. All the experiments were ran in double-precision.

Our objective is to compare the performance of our VSA implementation of the tall-skinny QR decomposition using three different tree configurations: 1) binary-tree, 2) flat-tree, and 3) binary-tree on top of flat-trees. For the computational kernels listed in Section V-B, we used the **core_blas** kernels from the PLASMA package [10]. The kernel calls the optimized LAPACK subroutine if available, or otherwise, requires two tunable parameters: block size $n_b$ and inner-block size $i_b$. To compare the optimal performance using these three tree configurations, we run each configuration using two tile sizes of $n_b = 192$ and $240$, and the inner-block size of $i_b = 48$. Furthermore, for the binary-on-flat tree, we tested performance using each flat-tree to eliminate either 6 or 12 tiles (i.e., $h = 6$ or 12). Below, we report the best performance obtained using these setups.

For a square matrix, our flat-tree configuration obtains the performance that is equivalent to that of our first VSA implementation of the QR decomposition (domino QR) [4]. This domino QR not only obtained significant speedups over the ScaLAPACK implementation of the QR decomposition in the Cray LibSci package, but it also obtained the best performance among hierarchical QR factorizations implemented using another runtime system, PaRSEC [5]. In comparison to the domino QR, our flat-tree QR sends packets between the flat-trees through one level of the binary-tree. Hence, it requires one additional step of communication to send the top tiles between the two flat-trees, but this overhead was insignificant in overall performance.

We first study the asymptotic scaling of our implementation by increasing the number of rows of $A$, while fixing the number of columns at $n = 4,608$. This corresponds to increasing the number of data points, while fixing the number of unknowns in an overdetermined system of our interest. The figure clearly shows that for a tall-skinny matrix, the parallel performance of the flat-tree suffers due to the lack of parallelism that it can exploit. Even though the binary-tree can exploit more parallelism, it may suffer from the lack of data locality or the performance of the special kernels which may not be optimized on this computer. On the other hand, the binary-on-flat tree seems to balance the parallelism and data locality and obtained the best performance among these three configurations.

Next, in Figure 11, we show the strong scalability of the tree QR where the matrix dimension is fixed at 368640-by-4608. Again, the binary-on-flat (and binary) tree obtained much better parallel scaling than the flat-tree. Hence, these two reduction-trees have a better potential to utilize the large number of cores on the emerging computers (see Section I).

### A. *Comparison Against Established and Research Solvers*

The natural question to ask is how the results presented here compare with the existing solutions. We have done such comprehensive studies in [6], [7]. For completeness, here, we reiterate these previously reported findings in a concise fashion. The existing vendor or academic software still lacks in support for tall-skinny matrices. In particular, the performance of both Cray LibSci and open-source ScaLAPACK lag behind that of tree-based QR reductions by at least 3-fold, and could be as much as an order of magnitude slower. Specialized runtimes such as PaRSEC [33] can achieve comparable performance [7], but their performance is still slower by at least $10\%$ in a strong scaling scenario, and $20\%$ or more in a weak scaling regime. We have presented an extensive review and results from the various implementations, including the exact data analysis, in our prior papers [6], [7], and they still hold for the current PULSAR implementation presented here.

## VII. Conclusion

We used PULSAR to examine the potential of a virtual systolic array as a parallel programming model to obtain extreme scalability. We used a tree-based QR algorithm on a tall-and-skinny matrix as an example, and presented its performance on a Cray-XT5 system. Our performance results demonstrated that by using a more sophisticated tree, we can obtain much higher performance than that of a domino QR which was presented at this conference last year [4]. We are currently running larger-scale experiments to further investigate the scalability of the algorithm and to map other algorithms onto PULSAR. We also plan to compare the performance of the VSA using different runtimes. As we discussed in Section II, a different runtime is implemented according to a different design principle, and may perform better for a particular algorithm or on a particular hardware than other runtimes do. We will also conduct more detailed analysis of the effects of the VDP-to-thread mapping function on the performance of PRT.

## Software

The PULSAR software is freely available, and can be downloaded at http://icl.utk.edu/pulsar/. The package includes the *PULSAR Runtime* (PRT) and a handful of examples. Documentation is distributed with the package and also available online. The PULSAR license is the 3-Clause BSD-style permissive free software license (properly called "modified BSD"), which allows proprietary commercial use, and for the software released under the license to be incorporated into proprietary commercial products. With its version 1.0 release in August 2013, PULSAR currently supports classic multicore processors. Though we plan to support them in a future release, PULSAR does not yet provide a support for GPU accelerators or Xeon Phi co-processors.

## Acknowledgments

This work is supported by grant #SHF-1117062: "Parallel Unified Linear algebra with Systolic ARrays (PULSAR)" from the *National Science Foundation* (NSF). The authors would like to thank the *National Institute for Computational Sciences* (NICS) for a generous time allocation on the Kraken supercomputer.

## References

[1] P. Kogge , "Exascale computing study: Technology challenges in achieving exascale systems," DARPA Information Processing Techniques Office, Tech. Rep. 278, 2008.

[2] V. Sarkar , "Exascale software study: Software challenges in extreme scale systems," DARPA Information Processing Techniques Office, Tech. Rep. 159, 2008.

[3] J. Dongarra, P. Beckman *et al.*, "The international exascale software roadmap," *Int. J. High Perf. Comput. Applic.*, vol. 25, no. 1, 2011 (to appear).

[4] J. Kurzak, P. Luszczek, M. Gates, I. Yamazaki, and J. Dongarra, "Virtual systolic array for QR decomposition," in *IPDPS 2013, the 27th IPDPS*. Boston, Massachusetts, USA: IEEE Comp. Society Press, May 20-24 2013.

[5] G. Aupy, M. Faverge, Y. Robert, J. Kurzak, P. Luszczek, and J. Dongarra, "Implementing a systolic algorithm for QR factorization on multicore clusters with PaRSEC," in *PROPER'2013, the 6th Workshop on Productivity and Performance*. Springer Verlag, Aug. 2013, to be published in LNCS.

[6] J. Dongarra, M. Faverge, T. Herault, M. Jacquelin, J. Langou, and Y. Robert, "Hierarchical QR factorization algorithms for multi-core clusters," in *IPDPS'2012, the 26th IPDPS*. Shanghai, China: IEEE Comp. Society Press, May 21-25 2012.

[7] ——, "Hierarchical QR factorization algorithms for multi-core clusters," *Parallel Computing*, 2013.

[8] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proc.*. Society for Industrial and Applied Mathematics, 1978, pp. 256–282.

[9] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," Innovative Computing Laboratory, Univ. of Tenn., Tech. Rep., 2011.

[10] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan, "PLASMA Users Guide," Univ. of Tenn., Innovative Computing Laboratory, Tech. Rep., 2010.

[11] 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.*, vol. 24, no. 3, pp. 305–321, Mar. 2011.

[12] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "LU factorization with partial pivoting for a multicore system with accelerators," *IEEE Trans. Parallel Distrib. Syst.*, 2012.

[13] A. YarKhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, Univ. of Tenn., December 2012.

[14] R. M. Badia, J. Labarta, R. Sirvent, J. M. Perez, J. M. Cela, and R. Grima, "Programming grid applications with GRID Superscalar," *J. Grid Comput.*, vol. 1, no. 2, pp. 151–170, 2003.

[15] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, "CellSs: Making it easier to program the Cell Broadband Engine processor," *IBM J. Res. & Dev.*, vol. 51, no. 5, pp. 593–604, 2007.

[16] J. M. Pérez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Proc. of the 2008 IEEE Int. Conf. on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*. IEEE, 2008, pp. 142–151.

[17] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Perez, E. S. Quintana-Orti, and G. Quintana-Orti, "Parallelizing dense and banded linear algebra libraries using SMPSs," *Concurrency Computat. Pract. Exper.*, vol. 21, no. 18, pp. 2438–2456, 2009.

[18] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Proc. of the 15th Int. Euro-Par Conf. on Parallel Processing*. Springer-Verlag, 2009, pp. 851–862.

[19] C. Augonnet and R. Namyst, "A unified runtime system for heterogeneous multicore architectures," in *Proc. of the Euro-Par 2008 Workshops - Parallel Processing*, ser. Lecture Notes in Comp. Sci.. Las Palmas de Gran Canaria, Spain: Springer, August 2008, pp. 174–183.

[20] M. C. Rinard, D. J. Scales, and M. S. Lam, "Jade: a high-level, machine-independent language for parallel programming," *Comp.*, vol. 26, no. 6, pp. 28–38, 1993.

[21] M. C. Rinard and M. S. Lam, "The design, implementation, and evaluation of Jade," *ACM Trans. Programming Lang. Syst.*, vol. 20, no. 3, pp. 483–545, 1998.

[22] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, pp. 207–216, August 1995.

[23] L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *Computational Sci. Engineering, IEEE*, vol. 5, no. 1, pp. 46 –55, 1998.

[24] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the Memory Hierarchy," in *Proc. of the 2006 ACM/IEEE Conf. on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.

[25] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," in *Proc. of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '07. New York, NY, USA: ACM, 2007, pp. 116–125.

[26] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar, "The Habanero Multicore Software Research Project," in *Proc. of the 24th ACM SIGPLAN Conf. Companion on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '09. New York, NY, USA: ACM, 2009, pp. 735–736.

[27] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement," in *Proc. of the 22Nd Int. Conf. on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 172–187.

[28] A. Zafari, M. Tillenius, and E. Larsson, "Programming Models Based on Data Versioning for Dependency-aware Task-based Parallelisation," in *Computational Sci. and Engineering (CSE), 2012 IEEE 15th Int. Conf. on*, 2012, pp. 275–280.

[29] G. H. Golub and C. F. van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD: The Johns Hopkins Univ. Press, 1996.

[30] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*. Philadelphia, PA: SIAM, 1992.

[31] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997.

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

[33] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," in *Proc. of the Workshops of the 25th IPDPS*, 2011, pp. 1151–1158.