Lightweight Superscalar Task Execution in Distributed Memory

Asim YarKhan
University of Tennessee, Knoxville
yarkhan@icl.utk.edu

Jack Dongarra
University of Tennessee, Knoxville
Oak Ridge National Laboratory
University of Manchester
dongarra@eecs.utk.edu

ABSTRACT

Arguably, we have yet to find a solution to the burden of multicore distributed programming facing domain scientists. This burden has been exacerbated by the increasing size of multicores, increasing the effect of any excess synchronization. To deal with these difficulties, numerical algorithms are re-engineered as sequences of interdependent tile-based tasks which can be executed by a dynamic runtime environment. We present a new runtime environment for distributed architectures which uses superscalar scheduling concepts. Tasks are inserted serially, and the runtime determines the dependencies dynamically and manages data movement transparently. QUARK-D (QUeuing and Runtime for Kernels on Distributed Memory) is shown to scale to $O(1000)$ cores for linear algebra algorithms and have competitive performance. The primary message of this research is that *scalable* and *competitive* performance can be achieved by a distributed-memory execution system using superscalar scheduling ideas where *serial code is the input and parallel execution correctness is guaranteed.*

1. INTRODUCTION

Multicore architectures with high core counts have come to dominate the world of high performance computing, from shared memory machines to the largest distributed memory clusters. The multicore route to increased performance has a simpler design and better power efficiency than the traditional approach of increasing processor frequencies. From a user's point of view, the availability of all this parallel power is welcome, but the ability to create applications that efficiently and effectively use this architecture is challenging. The complexities of using such architectures start at the level of the highly multicore machines, and any complexities are made much greater with the addition of the distributed memory clusters.

Traditional scientific libraries and development methodologies are finding it difficult to efficiently manage and use the high number of computational cores that have become available in these complex architectures. Even though alternative programming approaches exist for development on these many-core and distributed memory architectures, MPI remains a standard for achieving high performance in distributed memory machines. However, MPI is not designed to fully utilize these many-core architectures in a shared memory environment, so programming is complicated by the addition of a thread management systems such as Pthreads or OpenMP. As a result, developing efficient and scalable software for a complex, many-core, distributed-memory architecture remains a challenging and arduous task.

As a concrete example of a scientific software library, this work will use the PLASMA linear algebra library [1] as its driving application. Linear algebra algorithms are vital to many areas of scientific computing, and any improvement in the performance of these libraries can have a beneficial effect on a variety of fields. Research in linear algebra has reformulated algorithms as tasks acting on tiles of data, with data dependency relationships between the tasks [6]. This results in a task-based DAG for the reformulated algorithms, which can be executed via asynchronous data-driven execution paths analogous to dataflow execution.

In this work, we study the use of dynamic runtime environments executing data driven applications as a solution to programming multicore architectures. The goals of our runtime environments are productivity, scalability, and performance. We demonstrate productivity by defining a simple programming interface to express code. Our runtime environments are experimentally shown to be scalable and give competitive performance on large multicore and distributed memory machines.

Review of Tile-Based Algorithms It has been argued by Buttari et.al. [6] that to achieve high performance on highly parallel multicore architectures, algorithms will have to be expressed at a *fine-granularity* for improved local cache management, and with high *asynchronicity* to take advantage of the available cores and reduce synchronization points.

This tile approach consists of breaking the linear algebra algorithms into smaller tasks that operate on relatively small $nb \times nb$ tiles (or blocks) of cache-local consecutive data which are organized into block-columns The algorithms can then be restructured as tasks (which map to basic linear algebra operations) that act on tiles of the matrix. The data dependencies between these tasks forms a Directed Acyclic Graph (DAG) where nodes of the graph represent tasks and edges represent dependencies among the tasks. This approach is currently used in shared memory libraries, such as PLASMA from the University of Tennessee [1] and FLAME from the University of Texas at Austin [7]. The tile-based approach to linear algebra algorithms has been presented and discussed in [5], [11], [8], and [9].

The execution of tiled algorithms can be performed by using a runtime environment to asynchronously schedule the tasks in a way that dependencies are not violated. Optimally, we would like this asynchronous scheduling to result in an superscalar, out-of-order execution where the cost of slow, sequential tasks is hidden by parallel ones. This would be managed by having the sequential tasks start early, as soon as their dependencies are satisfied, while some of the parallel tasks (submatrix updates) from the previous iteration still remain to be performed and can be executed in parallel.

Tile QR factorization [1] The QR factorization implemented in LAPACK is a factorization of an $m \times n$ real matrix using the decomposition of A as $A = QR$, where Q is an $m \times m$ real orthogonal matrix and R is an $m \times n$ real upper triangular matrix. The QR factorization uses a series of elementary Householder matrices of the general form $H = I - \tau v v^T$, where v is a column reflector and τ is a scaling factor. -The tile QR algorithm produces essentially the same factorization as the LAPACK algorithm, but it differs in the Householder reflectors that are produced and the construction of the Q matrix. The algorithm is outlined in Algorithm 1. In order to restructure the QR algorithm as a tile algorithm, the dominant operation from the innermost loop needs to be different from the standard LAPACK implementation. In LAPACK, the dominant operation is the highly optimized `GEMM` and in the tile algorithm it is a new kernel operation `TSMQR`. The `TSMQR` operation, even though it is a matrix-matrix operation, has not been tuned and optimized to the extent of `GEMM`, so it reaches a lower percentage of peak performance on a CPU. For the purposes of thispaper, these operations can be viewed as black-box operations that operate on tiles of data.

Algorithm 1 Tile QR factorization algorithm

```
for $k = 1, 2$ to NT do
    DGEQRT($A_{k,k}, T_{k,k}$)
    for $n = k+1$ to NT do
        DORMQR($A_{kk}, T_{kk}, A_{kn}$)
    end for
    for $m = k+1$ to NT do
        DTSQRT($A_{kk}, A_{mk}, T_{mk}$)
        for $n = k+1$ to NT do
            DTSMQR($A_{kn}, A_{mn} A_{mk}, T_{mk}$)
        end for
    end for
end for
```

Fig. 1 shows the pseudocode for the tile QR factorization as an algorithm designer might view it. Tasks in this QR factorization depend on previous tasks if they use the same tiles of data. If these dependencies are used to relate the tasks, then a directed acyclic graph (DAG) is implicitly formed by the tasks (see Fig. 2).

Overview of Task-Superscalar Execution Task-superscalar execution takes a serial sequence of tasks as input and schedules them for execution in parallel, inferring the data dependencies between the tasks at runtime. The dependencies between the tasks are inferred through the resolution of data hazards: *Read after Write* (RaW), *Write after Read* (WaR) and *Write after Write* (WaW). These data dependencies are extracted from the serial code by having the user annotate the usage of the data when defining the tasks, noting whether the data is to be read and/or written.

Task-superscalar execution results in a asynchronous, data-driven execution that can be represented by a *Direct Acyclic Graph* (DAG), where the tasks are the nodes in the graph and the edges correspond to data movement between the tasks. Task-superscalar execution is a powerful tool for the productivity of the code writer. Since serial code is the input to the runtime system, and the parallel execution respects all the data hazards, the correctness of the serial code guarantees parallel correctness. Various approaches to superscalar task execution include [12], [10], [2] and [3].

[1]Note, because of space constraints, the details of tile Cholesky and other algorithms are not presented here. These can be found in [6] and related publications.

2. THE QUARK-D SOLUTION

In the work reported here, we have designed and implemented a prototype runtime environment for multicore distributed memory systems called QUARK-D (QUeuing And Runtime for Kernels on Distributed memory) [13].

Design Principles In designing QUARK-D a major desired feature was high productivity in writing applications. This productivity is enabled by having a simple serial API for adding tasks into the system. This API is then used in conjunction with a smart runtime environment that determines data dependencies, performs transparent communication, and schedules tasks. The user can provide additional information to tune the execution, but even in the absence of that information the runtime should make reasonable choices about where tasks execute and when data is moved. QUARK-D should make all runtime decisions using local knowledge, without requiring any global coordination. All runtime actions should proceed asynchronously without blocking for completion. A distributed data coherency protocol ensures that any copies of data are managed in order to decrease communication. A dynamic, non-blocking engine handles asynchronous communication.

Memory Model and Data Distribution QUARK-D uses a distributed memory model with the initial location of a data item referenced by a combination of a node and an address. The data that QUARK-D works on can be initialized and distributed by the user in several ways, but for linear algebra the data distribution is modeled after the block-cyclic distribution used by ScaLAPACK. This allows the problem size to be scaled with the size of the distributed machine.

Distributed Task Insertion API For each task that is inserted into QUARK-D, a function pointer specifies the function to be called when the dependencies are satisfied. Each argument for the function has flags that specify how that data was to be used; either as a static value, or an `INPUT/INOUT/OUTPUT` dependency. Each argument is provided with size information, which is needed for communication. Each argument needs a pointer to the data, the process identifier for the home of the data, and a process-specific key for the data. This key is required since the data item does not reside at every process;

```
for k = 0 ... TILES-1
  geqrt( $A^{rw}_{kk}$, $T^{w}_{kk}$ )
  for n = k+1..TILES-1
    unmqr( $A^{r}_{kk-low}$, $T^{r}_{kk}$, $A^{rw}_{kn}$ )
  for m = k+1..TILES-1
    tsqrt( $A^{rw}_{kk-up}$, $A^{rw}_{mk}$, $T^{rw}_{mk}$ )
    for n = k+1..TILES-1
      tsmqr( $A^{r}_{mk}$, $T^{r}_{mk}$, $A^{rw}_{kn}$, $A^{rw}_{mn}$ )
```

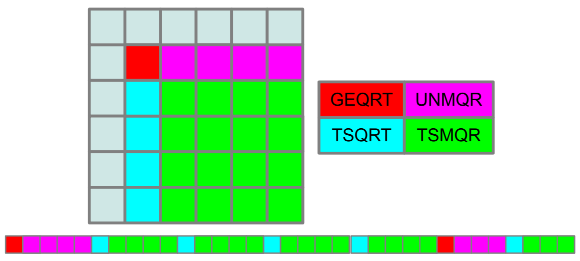


Figure 1: Pseudocode for the tile QR factorization, when acting on a matrix. The line at the bottom visualizes a sequence of tasks unrolled by the loops.

we do not have an easy common handle to use at every process.

```
QUARKD_Insert_Task(quark,*func,*funcflags,
   a_flags,size_a,*a,a_home,a_key,
   b_flags,size_b,*b,b_home,b_key,
   ..., 0 );
```

The task insertion API closely matches the shared memory API which was used in developing the PLASMA linear algebra library, allowing shared memory algorithms to be (almost) automatically extended to distributed memory platforms.

Execution Model and Runtime Decisions QUARK-D proceeds from serial task insertion operations, where every task is inserted into the runtime environment at each distributed memory process. As each task is inserted, information from the data parameters is used by each process to independently come to the same decision about which process is going to execute that task. By default, this decision about the executing process is based on which parameter is going to be written by the task, however, the decision can be overridden by the programmer.

Once the decision is made about which process is going to execute the task, each of its data parameters is examined. If the executing process does not have a valid copy of the data, then it inserts tasks to *receive* a valid copy. If another process is the current owner of the data, and the data-coherency shows that the executing task does not have a valid copy of the data, it inserts tasks to *send* that data. If the executing process is going to write the data item, then that process becomes the current owner of the data and all other copies are marked as invalid. All processes track the current owner and validity of the copies of the data parameter. After any *send* or *receive* tasks are inserted, the original task is inserted into the shared memory runtime of the executing process. The sequential insertion order of the tasks ensures that when the task finally executes, its data is already valid and available.

The dependency relationships from previous usage of the data create the implicit DAG based on data hazards: read-after-write (RAW), write-after-read (WAR), write-after-write (WAW). Any required data movement creates data transfer tasks which add new dependencies. The tile QR factorization was shown in pseudocode in Fig. 1. As an example of this execution model, the result of executing the QR factorization on multiple processes using the distributed memory algorithm is seen in Fig. 2.

Distributed Data Coherency When a data parameter is first seen by all processes during the serial insertion of tasks, the initial ownership and address are known, since they are provided by the task-insertion API. All non-owner processes can be assumed not to have copies of the data at this time. If the data is transferred to another process, and the task at the receiver is going to write the data, then the ownership of the data is transferred to the receiver process and any other copies are marked as invalid. If the data is only required for read, then the ownership stays with the sender, and the receiver is marked as having a valid copy. All processes track the movement of data ownership of current data items at all times, based on the information provided when each task is inserted.

This simple data coherency protocol enables us to minimize the transfer of data. Since information about valid copies of the data is available at all times, no unnecessary transfers are required. In order to reduce the footprint of this data coherency protocol, the information about data that was not recently used can be flushed at well structured, regular intervals. The data coherency protocol runs at task insertion time, not at the task execution time, so we use the serialization of task insertion to keep the distributed state of the data consistent.

Distributed Task Scheduling QUARK-D uses a mix of static and dynamic scheduling to assign tasks to the different processes and threads. The *static* scheduling refers to the fact that a task is scheduled for execution at a specific distributed memory process in a manner that is independent of the current state of the execution. The scheduling criteria have to be such that all the processes can independently come to the same scheduling decision. This is considered *static* because it is fixed at the time that the task is inserted. In general, this is accomplished by having all the nodes agree that a task will be scheduled at the home location of a specific data parameter though the algorithm designer can override this binding. Fixing the execution process at the time the task is inserted enables QUARK-D to avoid the complexities and coordination involved in distributed scheduling, distributed work balancing and data management.

The *dynamic* scheduling in QUARK-D occurs at the multi-threaded shared-memory level. Data locality is used to assign a task to a specific thread within the multi-threaded process. All the threads look for and execute tasks that are assigned to them. However, if there are no more tasks assigned to a thread, it will attempt to use work-stealing to obtain a task from another thread. This dynamic scheduling keeps the execution load balanced between the threads in a process.

Communication Engine In QUARK-D the communications are inferred from the data usage by tasks in conjunction with the current distribution of the data. For example, if a process is currently the owner of a piece of data, and that data is to be written by a task scheduled to be executed by another process, then the two processes

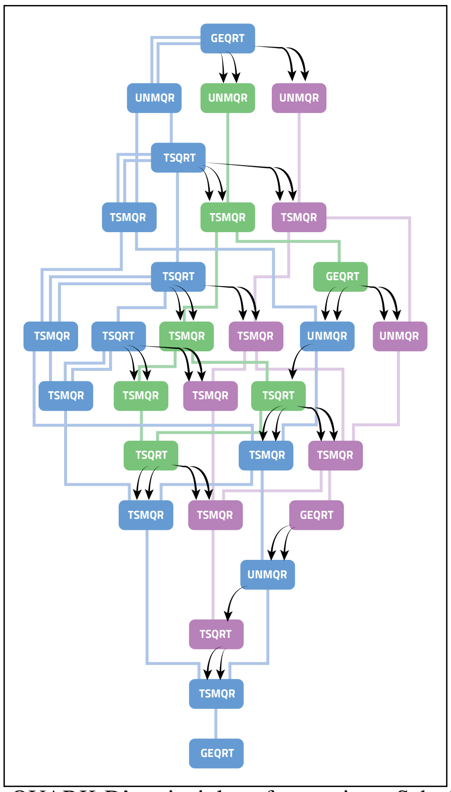


Figure 2: QUARK-D's principles of operation. Scheduling the DAG of the distributed memory QR factorization. Three distributed memory processes are running the factorization algorithm on a 3x3 tile matrix. One multi-threaded process runs all the blue tasks, another multi-threaded process runs the green tasks, and a third runs the purple tasks. Colored links show local task dependencies. Black arrows show inter-process communications.

independently and asynchronously insert tasks that manage the sending and receiving of that data. The key features of the QUARK-D communication engine are that it is dynamic, non-blocking and asynchronous, meaning that the engine will manage the transfer data as needed, the data transfers will not block the computational threads, and the data transfers are done using asynchronous techniques. The goal is to allow communication to overlap any computation that can be performed simultaneously. The communications are all point-to-point, from the task that is the current owner of the data to any tasks that need that data.

Window of Active Tasks In order to manage large problems, a fixed-size moving window of active tasks is used from the serialized task insertion. When the window is full, the master thread at each node switches to a computational mode. As tasks complete, additional tasks are allowed into the runtime system.

Serial Unrolling Bottleneck The productivity gain in QUARK-D is due to serial presentation of code. However, this creates a limitation in QUARK-D as the size of the distributed memory machine grows. Since fewer of the unrolled tasks are being executed at the local node, there is increasing overhead with respect to the computation work. We can compensate for some of this overhead by having larger tiles of data and thus fewer tasks and more local work. On the other hand, larger tiles would mean less available parallelism. The serial unrolling may eventually become a bottleneck for performance as the machine size keeps growing.

3. EXPERIMENTAL EVALUATION

Experiments are performed using the tile Cholesky and QR factorizations described in [6] and [13]. The Cholesky implementation is relatively simple and has few dependencies or constraints so the runtime can demonstrate the highest performance. The QR implementation is more complex, with more constraints in the DAG.

We perform weak scalability experiments where the quantity of work performed by a single core is kept constant and the matrix size is adjusted to reflect this. For our experiments with QUARK-D, we used two distributed memory clusters.

Small Cluster The *dancer* cluster is a 16 node machine, where each node has 2 Intel Xeon E5520 2.27 GHz quad-core processors for a total of 128 cores. The nodes are connected via Infiniband 20G and there is at least 8GB of memory per node. We used OpenMPI 1.5.5 compiled with gcc, and Intel MKL 11.1 math libraries.

Large Cluster The *Kraken* supercomputer at the Oak Ridge National Laboratory is a Cray XT5 machine with 9,408 compute nodes. Each node has two Istanbul 2.6 GHz six-core AMD Opteron processors, 16 GB of memory, and the nodes are connected through the SeaStar2+ interconnect. We used the PGI compilers with Cray MPI and the Cray LibSci math libraries. For our experiments, we used a small subset of the resources on Kraken.

On each platform we also performed the experiments using a high quality commercial numerical library that was appropriate for the platform. On the small cluster, QUARK-D is compared with the Intel MKL 11.1 ScaLAPACK implementation. On the large cluster, QUARK-D is compared with the Cray LibSci ScaLAPACK implementation. The ScaLAPACK implementations are executed using a process-per-core model with single threaded BLAS in each process.

Experimental results are also compared with the PARSEC/DPLASMA [3] linear algebra library on each platform. The PARSEC project is implementing a distributed memory DAG execution environment that uses compact parameterized DAG descriptions. The PARSEC runtime has been used to implement a subset of linear algebra applications in the DPLASMA package, which has been shown to be highly competitive with other specialized libraries and algorithm specific implementations [4]. Since PARSEC uses compact parameterized DAGs, it avoids a substantial part of the overheads in QUARK-D, so it is expected to give a higher performance than QUARK-D. The advantage that QUARK-D holds over PARSEC is in the productivity of writing serial code over the difficulty of generating compact DAG representations. It should be noted that the PARSEC developers are working on a compiler approach to simplify the generation of compact parameterized DAG descriptions.

```
#define A(m,n) ADDR(A),HOME(m,n),KEY(A,m,n)
#define T(m,n) ADDR(T),HOME(m,n),KEY(T,m,n)

void plasma_pdgeqrf(A, T,.) {
 for (k = 0; k < M; k++) {
  TASK_dgeqrt(quark,.,A(k,k),T(k,k));
  for (n = k+1; n < N; n++)
   TASK_dormqr(quark,..,A(k,k),T(k,k),A(k,n));
  for (m = k+1; m < M; m++) {
   TASK_dtsqrt(quark,.,A(k,k),A(m,k),T(m,k));
   for (n = k+1; n < N; n++)
    TASK_dtsmqr(quark,.,A(k,n),A(m,n),
                         A(m,k),T(m,k));  }}}
```

(a) The distributed memory tile QR factorization using QUARK-D. Each parameter's address, home process, and key is specified via the macro. The read/write usage of parameters is provided in a task wrapper.

```
void TASK_dgeqrt(
    Quark *quark,.,int m,int n,
    double *A,int A_home,key *A_key,
    double *T,int T_home,key *T_key )
{
 QUARKD_Insert_Task(quark,CORE_dgeqrt,...,
   VALUE,sizeof(int),&m,
   VALUE,sizeof(int),&n,
   INOUT|LOCALITY,sizeof(A),A,A_home,A_key,
   OUTPUT,sizeof(T),T,T_home,T_key,.,0);
}
```

(b) The DGEQRT task wrapper used in the QR factorization algorithm. This wrapper provides information about the read/write usage of the parameters and inserts the task into the QUARK-D runtime.

```
void CORE_dgeqrt(Quark *quark)
{
 int m,n,ib,lda,ldt;
 double *A,*T,*TAU,*WORK;
 quark_unpack_args_9(quark,m,n,ib,A,
    lda,T,ldt,TAU,WORK);
 CORE_dgeqrt(m,n,ib,A,lda,T,ldt,TAU,WORK);
}
```

(c) The DGEQRT task implementation is called by the QUARK-D runtime when all the dependency requirements have been met. Parameters are unpacked from the QUARK-D environment, and the (final) core routine provided by a library is called.

Figure 3: Tile-QR implementation using QUARK-D

QR factorization In order to demonstrate that the QUARK-D API increases the productivity of the user, we outline the implementation of the tile-QR factorization. Fig. 3a shows a refined version of the QR algorithm as it was implemented. This code closely matches the pseudocode in Fig. 1. The macro at the top shows how a data-tile reference is expanded to contain the address of the tile, the home process for the tile, and a key for referring to that tile. The fact that we were able to keep the code so close to the pseudocode shows that QUARK-D is achieving the productivity that was sought as one of the major goals of the project.

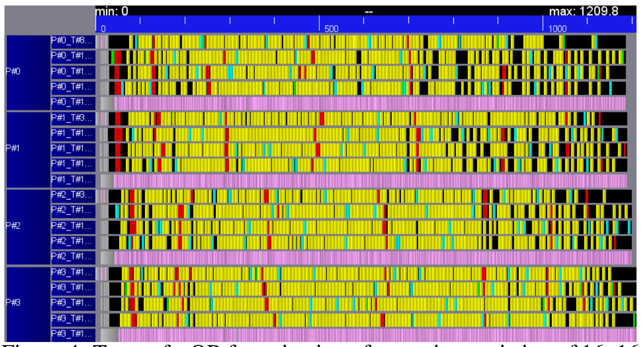
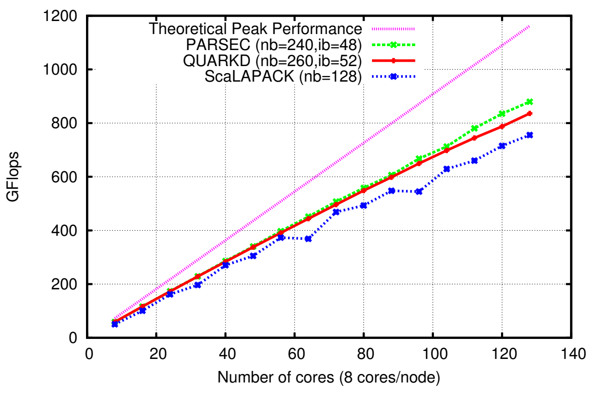
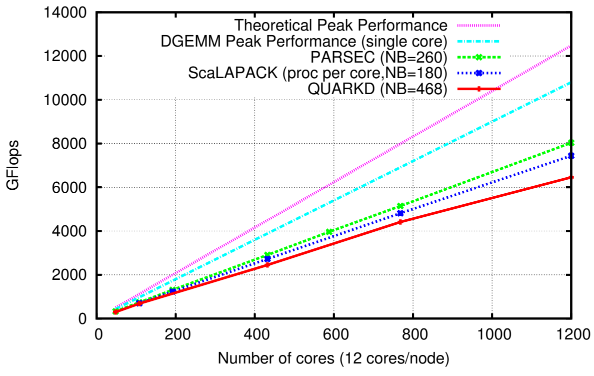


Figure 4: Trace of a QR factorization of a matrix consisting of 16x16 tiles on 4 (2x2) distributed memory nodes using 4 computational threads per node. An independent MPI communication thread is also maintained. Color coding: MPI (pink); GEQRT (green); TSMQR (yellow); TSQRT (cyan); UNMQR (red).



(a) Weak scaling performance of QR factorization on a small cluster. Factorizing a matrix (5000x5000/per core) on up to 16 distributed memory nodes with 8 cores per node.



(b) Weak scaling performance for QR factorization (DGEQRF) of a matrix (5000x5000/per core) on 1200 cores (100 distributed memory nodes with 12 cores per node).

Figure 5: Tile QR Experiments

Fig. 3b shows how the `DGEQRT` task is inserted into the runtime using a small wrapper routine. This wrapper provides the additional information required by the runtime system which is common to all calls to `DGEQRT`. Specifically, the information provided includes the usage (`INPUT`, `OUTPUT`, `INOUT`, `VALUE`) of the parameters, the sizes of the parameters, and additional hinting information provided by the programmer. Note that for each dependency parameter, a *home* node and a local *key* were provided from the calling routine. The task information is then stored in the QUARK-D runtime, where the execution of the task is held until all the data dependencies are satisfied. At that point, the task is ready to be scheduled for execution. Fig. 3c shows the function that is called by QUARK-D when the task is eventually executed. In this function, the parameters are extracted from the QUARK-D runtime, the arguments and dependencies are unpacked, and the serial core routine is called.

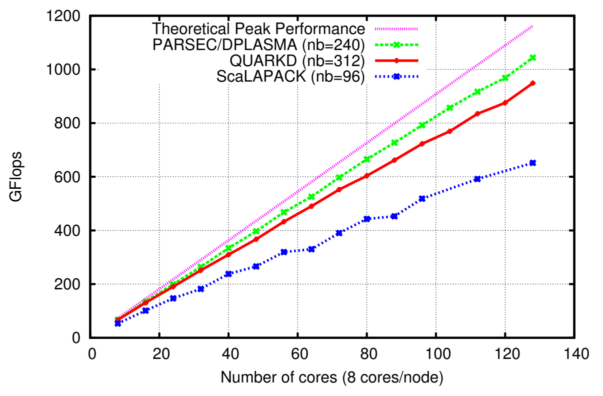
A example of the dependencies and execution of the tile QR algorithm was presented in the DAG in Fig 2. In Fig. 4 we see a trace of the execution of the QR factorization using 4 distributed memory nodes with 4 computational threads each. This trace shows the tasks keeping the cores busy with computation, with occasional gaps in the trace where the DAG does not have enough parallelism to keep the cores busy. The gaps in this trace are mostly because a small problem of 16×16 tiles is being traced and does not generate sufficient parallel work. Such gaps could occur anytime that the DAG does not provide sufficient parallelism and lookahead to hide the bottleneck tasks. The trace for a larger problem is not shown because it would become too congested. A separate communication thread is shown associated with each process. This thread is sharing one of the cores with a computational thread.

Tile QR Experiments Experimental results on the small cluster are given in Fig. 5a and show that QUARK is faster than MKL on this platform, and while performance is lower than PARSEC, it is still very competitive. Fig. 5b shows the weak scaling experiment on the large cluster using 1200 cores. In this experiment, we see that QUARK-D trails the Cray LibSCI implementation. PARSEC's performance on QR factorization exceeds the LibSCI performance, which validates the use of asynchronous data driven DAG execution. PARSEC and QUARK-D implement very similar algorithms, but QUARK-D has additional superscalar overheads that PARSEC does not have. The theoretical peak performance and the scaled single-core `GEMM` performance are shown to give a measure of how much of the peak is being achieved.

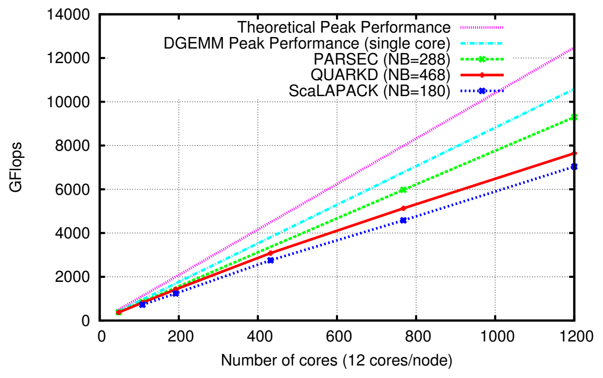
It is important to note that the tile QR implementations implemented by PARSEC and QUARK-D have `TSMQR` as the dominant operation in the innermost loop, whereas the commercial implementations have the much higher optimized `GEMM` as the dominant operation. In spite of that handicap, the QUARK-D and PARSEC implementations performed very well. It is likely that if the `TSMQR` kernel is optimized to the same point as the `GEMM` kernel, then the tile algorithms will show substantially increased performance.

Tile Cholesky Experiments The Cholesky factorization forms a relatively simple DAG structure with a single output dependency from each task. This means that data ownership does not need to be transferred to other processes, since data writes can always be done at the home process of the data. Copies of data will still be sent to other processors for reading.

In Fig. 6a experimental results on the small cluster compare Cholesky factorization performance using QUARK-D, Intel MKL, and PARSEC. The QUARK-D implementation scales better than the MKL implementation but not as well as the PARSEC implementation. The weak scaling performance on the large 1200 core cluster is shown in Fig. 6b. On this large cluster the QUARK-D implementation has better performance than the Cray LibSCI library. The PARSEC implementation reaches a higher performance than the other implementations. The main problem with PARSEC is productivity since writing compact DAG representations remains a difficult process. QUARK-D focuses on the productivity gained by writing serial style, loop based code and using superscalar execution. In spite of this ease of coding, QUARK is able to produce better performance than the commercial LibSCI implementation.



(a) Small cluster: Weak scaling performance of Cholesky factorization (DPOTRF) of a matrix (5000x5000/per core) on 16 distributed memory nodes with 8 cores per node.



(b) Large cluster: Weak scaling performance for Cholesky factorization (DPOTRF) of a matrix (5000x5000/per core) on 1200 cores (100 distributed memory nodes with 12 cores per node).

Figure 6: Tile Cholesky Experiments

4. SUMMARY AND CONCLUSIONS

We motivated this work by describing how improvements in distributed multicore architectures require asynchronous execution to properly take advantage of the available hardware. To this end, algorithms are being restructured as tasks with data dependencies that can be executed by a runtime environment. Using serial input and superscalar execution has a substantial positive impact on the productivity of the programmer, and we have presented our API and sample code to justify this.

We have designed and implemented QUARK-D, a runtime environment to schedule and execute task-based applications using superscalar techniques on distributed memory architectures. QUARK-D is designed for productivity, scalability and performance. To demonstrate productivity, algorithms from the PLASMA linear algebra library are executed using QUARK-D. The scalability and performance of QUARK-D is compared to that of commercial linear algebra libraries and to the PARSEC runtime environment. Experiments performed on 128 cores of a small cluster and 1200 cores of a large cluster show that QUARK-D can be scalable and have performance competitive with the commercial solutions.

This work shows that a runtime environment can achieve performance and scalability on distributed memory platforms while retaining the simplicity of a serial programming interface by using superscalar scheduling and execution, where serial code is the input and parallel execution correctness is guaranteed.

Acknowledgments This research was funded in part by grants from the National Science Foundation (NSF # OCI-1032815 and NSF # CCF-0811642) and industry support was provided by Microsoft. Thanks to the National Institute for Computational Science (NICS) for access to their computational resources.

5. REFERENCES

[1] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA Users Guide. Technical report, ICL, University of Tennessee, 2010.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops*, IPDPSW '11, pages 1432–1441, Washington, DC, USA, 2011. IEEE Computer Society.

[4] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[5] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The Impact of Multicore on Math Software. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 2007.

[6] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, Jan. 2009.

[7] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.*, 27:422–455, December 2001.

[8] J. Kurzak, A. Buttari, and J. Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Trans. Parallel Distrib. Syst.*, 19:1175–1186, September 2008.

[9] J. Kurzak and J. Dongarra. QR factorization for the Cell Broadband Engine. *Scientific Programming*, 17(1):31–42, 2009.

[10] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *Int. J. High Perf. Comput. Applic.*, 23(3):284–299, 2009. DOI: 10.1177/1094342009106195 .

[11] E. S. Quintana-Ortí and R. A. Van De Geijn. Updating an LU Factorization with Pivoting. *ACM Trans. Math. Softw.*, 35(2):11:1–11:16, July 2008.

[12] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, May 1998.

[13] A. YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. PhD thesis, University of Tennessee, December 2012.